Effects of Delay Variation of IEEE 802.11 on Synchronization in Ad Hoc Networks

Philip Sitton

Certificate of Approval:

Effects of Delay Variation of IEEE 802.11 on Synchronization in Ad Hoc Networks

Philip Sitton

A Thesis

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Master of Science

Auburn, Alabama
May 10, 2007

Effects of Delay Variation of IEEE 802.11 on Synchronization in Ad Hoc Networks

Philip Sitton

Signature of Author

Date of Graduation

Vita

Philip Sitton, the son of Jeffrey and Wanda Sitton, was born on April 26, 1982, in Birmingham, Alabama. He graduated from Hayden High School in 2000. He then attended the University of Alabama, where he graduated with a Bachelor of Science degree after double-majoring in Computer Science and Mathematics. In August 2004, he began his graduate work in the Computer Science and Software Engineering of the Samuel Ginn College of Engineering at Auburn University.

Thesis Abstract

Effects of Delay Variation of IEEE 802.11 on Synchronization in Ad Hoc Networks

Philip Sitton

Master of Science, May 10, 2007
(B.S., University of Alabama, 2000)

47 Typed Pages

Directed by Saad Biaz

The 802.11 MAC protocol, extensively used in wireless networks, handles the problem of collisions over a shared medium in part by avoiding them through the use of a random backoff counter drawn from an exponentially increasing range. This stochastic process, along with the threat of collisions and the need to share the medium, cause random packets to face significantly higher delays than others. This work explores these variations in delay, notes the effect on end to end delay variation, and records time synchronization data over modestly-sized 802.11 ad hoc networks.

Acknowledgments

I would like to thank my advisor, Dr. Saad Biaz, for his insight and patience while undertaking this work. I am also grateful for the contributions my committee members Dr. David Umphress and Dr. Alvin Lim have made toward this thesis and my education.

The Department of Education's GAANN Fellowship was a great help in my research efforts.

Finally, I would like to thank my parents, as well as my friends and other family members, for their encouragement and unfailing support.

Style manual or journal used Journal of Approximation Theory (together with the style known as "aums"). Bibliograpy follows van Leunen's *A Handbook for Scholars*.

Computer software used The document preparation package TeX (specifically LaTeX) together with the departmental style-file `aums.sty`.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Wireless technology, which has already achieved widespread use in modern computational environments, is becoming ever more pervasive as new applications for wireless communication are found. One area which has seen many advancements in recent years in this new environment is ad hoc networking [1], in which a network of computers is set up using only wireless links. These networks are highly flexible and free from the infrastructure requirements of traditional networks. Because they are much more dynamic, however, these networks face unique challenges in providing similar functionality as that achievable using wires.

One measure often taken to help provide suitable performance in wireless environments is the use of the IEEE 802.11 MAC protocol. 802.11 has debatably become the *de facto* standard MAC protocol for a wide variety of wireless applications, and is often the protocol of choice when setting up a mobile ad hoc network (MANET), a mesh network, or a wireless sensor network (WSN) which may consist of hundreds or even thousands of communicating nodes. 802.11 handles communication across a shared medium, along with the issues of the hidden and exposed terminal problems, via its distributed coordination function (DCF), which minimizes the effect of collisions on a network via the use of a randomized backoff counter. Regardless, performance in terms of bandwidth and average delay still lag behind what would be easily obtainable in a more conventional, wired environment.

A challenging area of research that has arisen due to these new paradigms is the development of appropriate security schemes for a wireless environment. Typical security schemes may place too much emphasis on expensive computation, require a large number of messages each time secure communication is initiated, rely on a trusted third party, or hold some other requirement making them unsuitable for a number of ad hoc environments.

Whether deployed in a wireless ad hoc environment or a more conventional one, a secure communication protocol must be prepared to face the threat of replay attacks in which an unauthorized third party attempts to initiate a fraudulent transaction by re-sending messages previously accepted by the protocol in hope the receiver will be unable to differentiate this replayed message from an authentic one. Typical safeguards against these kinds of attacks include the use of timestamps (an encrypted timestamp will cause replayed messages to fail authorization, since the decrypted replayed message will visibly be badly out of sync) and nonces (requiring a shared list of secret numbers to be used as session keys).

Of chief concern to synchronization efforts is the issue of uncertainty in expected packet delivery times. As many security protocols simply estimate the one-way packet delivery time to be half the round-trip packet delivery time, errors will be introduced in any distributed synchronization scheme in most real networks, where a synchronization packet will likely experience larger delays during one half of its trip than the other. These errors will grow larger along with the variation between delivery times, so it is best for synchronization purposes that the two halves of the trip match up well chronologically.

1.2 Goals and Methodology

The purpose of this research is to address the feasibility of application-level security schemes requiring clock synchronization in ad hoc networks. More specifically, it explores the hindrances to synchronization in a wireless environment and provides a quantitative analysis of these effects on a handful of security protocols. To this aim, synchronization experiments are conducted on a number of randomly distributed simulated wireless networks, while the findings will be used to report the degrees of success that may be expected using a number of proposed security protocols in such an environment.

Due to the stochastic nature of the 802.11 DCF, along with the standard difficulties in reliably broadcasting across a shared medium, the uncertainty in expected packet delivery times between any two given nodes may be relatively high when compared to a similar situation in a wired network. Much of the focus of these experiments is to determine the extent of the damage this randomness does to synchronization networks in ad hoc situations. For a real ad hoc network, even more factors will need to be addressed, such as energy consumption (many wireless devices operate without connection to an electric power infrastructure and thus rely mainly on batteries as a source of power), but these issues are not the focus of this work.

1.3 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 covers previous work in several of these areas, including security, and wireless technology (particularly the IEEE 802.11 standard), to the extent that is applicable to this research. Chapter 3 will cover issues in clock synchronization and the difficulty inherent in synchronizing over ad hoc networks using the 801.11 protocol. Chapter 4 discusses the setup and environment of our experiments on clock synchronization, while chapter 5 will contain information regarding the results of our experiments, along with expected degrees of success of various synchronization-critical security protocols expected on the networks examined. Chapter 6 presents overall conclusions.

Chapter 2

Literature Review

2.1 Wireless and 802.11

As is often the case in schemes that feature multiple senders attempting to communicate over a shared medium, collisions may occur when using 802.11 that render some transmitted packet indecipherable and thus unusable. Reliability is attained through retransmission: when collisions occur, the colliding packets will be retransmitted. Should a packet experience many collisions in a row, it will be retransmitted until it is successfully transmitted without collision.

Scheduling of retransmissions in 802.11 is determined by a process known as the distributed coordination function (DCF), in which attempts to avoid collisions are made via sensing the medium for transmissions and the use of random backoff counters.

802.11 DCF is described and modeled by Bianchi in [2], and a simplified explanation of its functionality is as follows: when a node must send a packet, it first chooses a uniformly random integer (known as the backoff time) within the range $[0, w_1 - 1]$ with w_1 set to a system parameter representing the minimum contention window. If the chosen number is not 0, the node will sense the medium, and decrement the backoff counter while no concurrent transmissions are sensed. Should a transmission be detected, the node will cease to decrement its backoff counter until the medium is sensed to be no longer busy. When the counter reaches 0, the node begins transmission of the packet as soon as it senses the medium is idle.

Should a collision occur, the node repeats the process, selecting its backoff counter now from the range $[0, w_2 - 1]$, where $w_2 = {w_1}^2$. If another collision occurs, the backoff counter is then selected from the range $[0, w_3 - 1]$ with $w_3 = {w_2}^2$, and so on. The range increases exponentially for every consecutive collision until it reaches a size w_{max} representing the maximum contention window size. For each consecutive collision afterward, the backoff counter will be selected from the range $[0, w_{max} - 1]$ [2].

Upon the successful transmission of a packet, if the node has another packet to send it will choose its backoff counter in a similar manner as the previous packet's initial backoff counter, beginning again with a random selection from the range $[0, w_1 - 1]$. It then behaves as before, choosing backoff counters from a range that grows exponentially for each consecutive collision [2].

The scheme's stochastic nature and the exponential growth of the backoff selection range illustrate two points: it is impossible to accurately predict the single-hop transmission delay of an individual packet (because the backoff counter is randomly selected), and packets that face multiple collisions and are not lucky enough to consistently select small backoff counters may experience delays exponentially larger than those encountered by other packets (because successive backoff counter selections can increase exponentially).

For queuing delays (assuming a first in, first out queuing scheme), the potential for wide variations is again increased, as a queued packet must first wait for each packet preceding it to be transmitted according to the DCF scheme before its own transmission can begin.

A similar situation occurs when a packet must traverse multiple hops in a wireless environment, as the DCF scheme will cause it to face random transmission and queuing delays at each hop, indicating that wide variations in one-way trip times and round trip times may be expected. Since the probability of a collision is equivalent to the probability that two clients within the receiver's listening range will attempt to make overlapping transmissions, it increases as the number of clients in the receiver's range with data to send increases.

For a node n_0 that will immediately send a packet to a receiver r, which is in transmission range of m other nodes, let p_i be the probability that client n_i will attempt to send a packet before it receives notification that n_0 is attempting to transmit, or already has begun a transmission of which n_0 is unaware. The probability p that a collision will occur, then, is:

$$\prod_{i=1}^{m}(1-p_i).$$

Thus the probability that there will be a collision at node n_0 is expected to be larger in situations where multiple neighbors have packets ready to transmit.

In a fully ad-hoc, densely populated wireless network, such as a WSN, in which a receiver may be within range of many transmitters and traffic load may be high, collisions will likely occur more frequently than in sparser wireless networks with less traffic. Nodes with traffic to send will also more frequently need to halt their backoff counters while other nodes transmit. Thus, greater variations in one-way and round trip time should be expected than in sparse networks with low traffic loads.

The effect of these variations in delay on end-to-end synchronization schemes is obvious: nodes can inform each other of their local time, but accuracy of the receiving node's estimate is limited due to the variation in delay experienced by synchronization packets.

2.2 Mobile Ad-hoc Networks

Much study has been undertaken regarding the use of wireless technologies to facilitate networking in nontraditional and even mobile environments without a cable infrastructure. The use of wireless transmitters and protocols is widespread today, with many applications, and may become even more ubiquitous in the future.

While many applications have been found for wired-cum-wireless situations, in which a wireless access point provides the “last hop” for a client on an otherwise traditionally wired system, research has also been devoted to entirely wireless, or “ad-hoc” networks [1]. These Mobile Ad-hoc Networks (MANETs) have distinct requirements for efficient operation, as they typically rely on transmission over a shared medium using omnidirectional antennae, as opposed to networks using Ethernet cable.

Power consumption is a chief concern as the systems making up these networks often rely on limited-supply batteries for their power. MANETs typically must also deal with dynamic topologies, as the wireless nature of stations provides opportunities for mobility that may remove stations from others' direct communication range and place them within others. Many routing schemes and energy conservation have been proposed for effective communication on MANETs, and new research is still being done in these areas.

Wireless sensor networks (WSNs) [23] are a subset of MANETs in which many of the stations are actually small, inexpensive sensors with wireless capabilities. These sensors must typically operate under extremely tight energy constraints, must deal with random and possibly extremely dynamic topologies, and may form networks numbering into the hundreds or even thousands. Common applications for sensor networks include event detection [23] and environment monitoring [22].

2.3 Security Protocols

2.3.1 Kerberos

Kerberos [5, 6] is a private key authentication service for open systems developed at the Massachusetts Institute of Technology and based upon the Needham-Schroeder authentication model [7]. Since its inception in 1987, Kerberos has been adopted for widespread use in real systems, and was even selected as the default authentication package for Windows 2000 and Windows Server 2003. Encryption in Kerberos is based upon the Data Encryption Standard (DES) [8].

Authentication in Kerberos, as described by Steiner et al. [5], relies upon an authentication server which knows of every client within its domain, as well as the personal keys each client will use for encrypting and decrypting authentication messages. To begin the authentication process, a client will generate a message containing its name (or some other appropriate identifier) and the name of the ticket-granting server (the purpose of this service will be explained later), then send it to the authentication server in plaintext. When the authentication server receives this message, it checks the client's reported name to determine whether it knows of this client. If it does, the server will send the client a message encrypted with the client's secret key, generated by applying a function to the client's password.

This message, when unencrypted, will provide the client with two important pieces of information: first, a randomly-generated session key for the client to use while communicating with TGS, and second, a "ticket" encrypted with the TGS's secret key. This ticket will provide the TGS with the client's name, the TGS's name, the time at which the ticket was created, a lifetime for which the ticket is valid, the client's IP information, and a copy of the same randomly-generated session key provided to the client. For a system x, this notation will be used: n_x is its name information, IP_x is its address, K_x is its private key, $K_{x,y}$ is a session key to be used while communicating with a system y, and $K_x(data)$ is some information $data$ encrypted by the private key K_x. For a client c the aforementioned ticket can be represented by

$$T_{c,TGS} = n_{TGS}, n_c, IP_c, timestamp, lifetime, K_{c,TGS}$$

and the message the authentication server provides the client is represented by

$$K_c(K_{c,TGS}, K_{TGS}(T_{c,TGS})).$$

Although an impostor may send the authentication server enough information to receive such a message, assuming it does not know the client's password it cannot obtain the private key necessary to extract any useful information from such a message [5].

Once the client has received the ticket authorizing it to communicate with the TGS, it may use this ticket to gather other tickets authorizing it to use various services available on the network. Since the TGS is itself providing such a service, however, it is more straightforward to cover the general scheme by which a client may request a service under the assumption it has an appropriate ticket and session key.

To request a service from a server s, the client c will prepare an authenticator of the form

$$A_{c,s} = K_{c,s}(n_c, IP_c, timestamp)$$

and send it to the server, along with its ticket $T_{c,s}$ and any other information the server may require. The server may then decrypt the ticket using its private key, use the session key included in the ticket to decrypt the authenticator, and check that the information in the ticket matches that included in the authenticator. If it does not, the client will not be authenticated with that server and the server will respond in an appropriate manner (for example, it will likely refuse to provide the client with the requested service). Otherwise, the client will be authenticated with that server and it may provide the requested service. In the case of the TGS, the client would additionally provide the name of the server for which it is requesting a ticket, and after authentication the TGS would provide the client with a ticket and session key appropriate for communicating with said server [5].

Once the client has obtained a ticket and session key to prove its identity and communicate with a server, it may (or may not) request the server prove its identity as well. If it does, the server will send back a message, encrypted by the session key, containing a value of $timestamp + 1$, with $timestamp$ being the time sent in the client's authenticator. Such a message demonstrates that the server was able to decrypt the ticket, to obtain the session key, which shows that the server knows its correct private key, effectively proving its identity. The client and server may now communicate using messages encrypted by their shared session key with a reasonable degree of security that each entity is authentic [5].

Timestamping in Kerberos

In a private-key authentication scheme that does not protect against replay attacks, a malicious entity may disguise itself as a client c by storing messages previously used by c to communicate with servers, then re-sending them at a later interval. Thanks to the use of private keys to encrypt and decrypt messages, the attacker will still be unable to understand the information it is sending and receiving, because it does not know c's private key. However, the re-use of old messages could theoretically allow it to repeat entire interactions with the server, thus causing the server to react in unintended and unauthorized ways. Kerberos and some other authentication schemes prevent this type of attack via the timestamps included in many of the encrypted messages. Timestamps thwart replay attacks by providing a method to ensure that a received message is not simply a copy of a previous message.

In order to use timestamps, Kerberos (and other security protocols) selects an upper bound for the synchronization of an acceptable message. When a system receives a message, it checks the timestamp, comparing it to the local clock. If the timestamp in the message is too far in the future or the past (according to the upper bound for synchronization), authentication will fail and the message will not be accepted. This prevents malicious users from storing old messages and sending them at much later dates, as the timestamps they contain will fall outside the acceptable range. Further, since attackers do not know the private key of the system they are impersonating, they will be unable to undertake the decryption and encryption upon the stored message necessary to insert their own, more recent timestamp.

Another route an attacker may choose in a replay attack is to replay stored messages quickly, before the upper bound on time allotted for synchronization errors has passed. In doing so, the replayed packets may reach their destination with valid timestamps. The possibility of this method, if it is to be diverted, forces the attacked system to keep records of all previously received messages containing still-valid timestamps, and to check all incoming messages with valid timestamps against these records. Malicious messages with valid timestamps will be detected as duplicates of stored messages, and may not be authenticated. It is up to the application to determine how to handle such messages; for example, they may simply be discarded.

In determining an acceptable upper bound for synchronization errors of timestamps in a Kerberos system, a few considerations make themselves immediately apparent. Obviously, lower bounds will require tighter synchronization throughout the network. A boundary set too low for the degree of synchronization achievable will result in legitimate clients being unable to authenticate their messages. On the other hand, higher bounds will necessitate that systems store more old messages to thwart replay attacks. In a hypothetical distribution of Kerberos with an infinitely high bound on synchronization, every previous message will need to be stored indefinitely, or else an attacker with a very old stored message will have an opportunity for a successful replay attack. The need to store all previous messages containing valid timestamps may prove troublesome for memory-constrained systems and devices if the synchronization bound is set too high.

It is stated in [5] that Kerberos assumes that clocks are synchronized within a range of several minutes. A common default window is 5 minutes, according to [6].

2.3.2 TESLA

Timed Efficient Stream Loss-tolerant Authentication (TESLA) [9, 10], is a protocol used to authenticate multicast messages in potentially lossy networking environments. TESLA addresses a fundamental problem of naively using shared keys in a multicast environment: that because each recipient will have access to the shared key, a recipient may then generate fraudulent messages using this key and send them to other members of the multicast group. Ariadne [11] and Secure Efficient Ad hoc Distance vector routing protocol

(SEAD) [12] may both take advantage of TESLA to provide secure routing in wireless ad hoc networks. TESLA requires loosely synchronized clocks in order to properly operate, and is suitable for use in encrypting streaming multimedia data.

TESLA authenticates multicast messages using pseudorandom functions (PRFs) and message authentication codes (MACs). Perrig et al. [9] describe TESLA by presenting a series of increasingly complex protocols. The simplest version of TESLA could be described as so: to begin the process, the sender S selects a random key $k_{initial}$, then applies a PRF F to $k_{initial}$ to obtain a new key $k_{initial-1}$ (i.e. $k_{initial-1} = F(k_{initial})$). S then computes $k_{initial-2} = F(k_{initial-1})$, and so on, until it has an adequate chain of keys. One of these keys will be inserted into each packet in the following way (subsequent packets shall be denoted by incrementally larger subscripts): if packet P_{i-1} contains k_{i-2}, P_i will contain k_{i-1} and P_{i+1} will contain k_i [9].

In addition, each packet will contain a MAC. Let D_i be the entirety of P_i, excluding its MAC. Given an eponymous function MAC and a pseudorandom function F', a MAC for packet P_i will be of the form $MAC(F'(k_i), D_i)$ For a packet P_i containing data M_i,

$$P_i = D_i, MAC(F'(k_i), D_i)$$

while D_i may be represented by

$$D_i = M_i, k_{i-1}$$

[9].

Note that this scheme is tolerant of packet losses. For example, if P_i and P_{i+3} reach a receiver R, but P_{i+1} and P_{i+2} are lost due to congestion or other networking problems, R

can still authorize P_i. Using the k_{i+2} revealed in P_{i+3}, k_i can be computed by $F(F(k_{i+2}))$. R will then check that $F(k_i)$ matches the value for k_{i-1} in P_i. After it is found to match k_i may then be used to check the value of P_i's MAC, and assuming the computed value is the same P_i will be authenticated [9].

This scheme is also relatively secure: because F and F' are pseudorandom functions, there is reasonable doubt that an attacker armed with the knowledge of P_i but lacking limitless computing resources will be able to quickly compute a k_i that provides correct values for $k_{i-1} = F(k_i)$ or $MAC(F'(k_i), D_i)$. However, timing does play an important role. If an attacker is able to study the contents of P_{i+1} before P_i reaches R, said attacker will have access to the value of k_i. If this information is used to produce a fraudulent packet which is then sent it to R, the security of the protocol is compromised and R may authenticate the malicious packet [9].

One sure way to prevent such scenarios is to have R discard any packet P_i not received before S sends out P_{i+1}. The rationale is that once P_{i+1} is released into the network, the value of k_i becomes insecure, and a speedy attacker (or one with some degree of control over the traffic in the network) will be able to feed R fraudulent information. In order to apply this policy, R will need to know S's schedule relative to the local clock, and for this reason TESLA requires a degree of clock synchronization. More specifically, in addition to S's schedule R must know some upper bound δ on the difference between S's clock and its own. This δ may be on the order of seconds. Armed with this knowledge, R may safely accept P_i as long as it arrives earlier than δ time units before S's scheduled time to send P_{i+1} [9].

The simplified version of TESLA described above has several drawbacks. First, because the key for a packet P_i is revealed in P_{i+1}, S must wait until R receives P_i before it can send the next packet, which is clearly not an optimal design for networks in which propagation delays will cause S to wait excessive amounts of time before sending multiple packets. This problem may be solved easily enough by, in essence, supplying the key for P_i in some later P_{i+d}, where d is not constrained to a value of 1. Note that this will set $F(k_i) = k_{i-d}$, meaning there will need to be at least d separate random keys which F will be applied to multiple times when S starts up. Also, S will need to announce its value for d at start-up [9].

Next, the assumption that R knows the schedule by which S wants to send its packets has relegated S to sending packets within very strict (and likely constant) timing guidelines. This requirement is relaxed by allowing S to manage keys in a manner based on intervals, rather than on packet numbers. Rather than associate key k_i with a packet P_i, it will be associated with an interval i, and will serve to authenticate all packets sent during said interval. This does introduce a new value, i, to the packet format which is now represented by

$$P_j = M_j, i, k_{i-d}, MAC(F'(k_i), M_j).$$

Also, at start-up S will need to inform its receivers of the first interval's starting time and the length of each interval, which may be represented by the symbols T_0 and T_Δ respectively [9].

For R, checking that packets arrive before their significant keys are leaked to the network is only slightly more complicated now that S has been freed from a strict schedule.

At any given time t, R can determine the interval i by computing

$$i = \lfloor \frac{t - T_0}{T_\Delta} \rfloor.$$

Recall that S's clock is at most δ units ahead of R's. For a packet P_j arriving at time t_j, the largest interval S may currently be in is

$$i' = \lfloor \frac{t_j + \delta = T_0}{T_\Delta} \rfloor.$$

If $i' < i + d$ then the key must not yet have been released in any subsequent packets, and P_j will not need to be discarded [9].

If δ_{tMax} is the maximum synchronization error to be handled, and d_{NMax} is the maximum delay across the network to be handled, d may be calculated as

$$d = \lceil \frac{\delta_{tMax} + d_{NMax}}{T\Delta} \rceil.$$

Larger values for δ_{tMax} and d_{NMax} will allow distant receivers, slow receivers, and those with large synchronization errors to properly authenticate packets rather than simply discarding them. These large values will, however, cause receivers who obtain packets quickly and have small synchronization errors to wait longer than necessary to authenticate previously received packets [9].

TESLA uses an interval-based scheme similar to that stated above, but also allows S to solve the tradeoff between slow and fast receivers by using multiple authentication chains suited to receivers with a range of capabilities. For example, S could include two authentication chains in its packets: one suited to fast receivers, and another suitable for slow receivers. A given receiver R will not need to drop a packet P_j as long as a single authentication chain remains valid. P_j may be authenticated with the fastest (i.e. the smallest value for d) authentication chain not possibly compromised by the time it reaches R. The use of multiple authentication chains adds extra overhead to the protocol, as more authentication information will need to be included in each packet [9].

Chapter 3
Clock Synchronization

While some networked applications require only that causality of events be recorded (which can be handled using vector clocks), it is often preferred or necessary to synchronize physical clocks. For example, the two security protocols discussed earlier, Kerberos and TESLA, both make use of synchronized clocks. The maximal degree of synchronization (that is, the minimal amount of skew) reliably achievable is bounded by the uncertainty regarding the timeliness of the information processors are able to provide one another. Effectively, this means a network's synchronization is bounded by the uncertainty regarding packet delivery times. For networks in which packets are reliably transmitted and networking delays are bounded, algorithms exist which provide a provable degree of synchronization. For networks in which packets are not reliably transmitted, delays are unbounded, or both (many real networks fall into this category), errors in synchronization techniques that rely on packet transport are also technically unbounded; however, in practice a number of methods may be employed to synchronize clocks within satisfactory bounds.

3.1 The Welch-Lynch Clock Synchronization Algorithm

The Welch-Lynch clock synchronization algorithm [13, 14], developed at MIT, is able to provide a provable degree of clock synchronization for non-faulty systems in a computer network (under certain conditions) in which less than one third of the systems are compromised. This protocol assumes the network provides reliable packet delivery with uncertainty in packet delivery times bounded by some small ϵ. Further, the assumption is made that all clocks are initially synchronized, allowing the protocol to simply maintain an acceptable degree of synchronization in the face of clock drift; however, it may be modified to achieve synchronization in an initially unsynchronized network, then begin regular operation to prevent drifting clocks from moving too far out of line with the others in the network.

Operation of the Welch-Lynch synchronization algorithm follows a relatively simple message passing scheme. To describe it, some symbols need to be defined. Let n be the number of processes within a network to be synchronized, δ the expected packet delivery delay, and ϵ the uncertainty in a packet's delivery delay (if a packet is sent at a time t, it will be successfully delivered within the range $[t + \delta - \epsilon, t + \delta + \epsilon]$). A process p's local time L_p is computed by $L_p = Ph_p + CORR_p$ where Ph_p is the hardware clock for the system on which the process is located and $CORR_p$ is a local variable added to the hardware clock to yield the local time. The value of $CORR_p$ may be adjusted over iterations of the protocol to account for rates of drift in Ph_p. The maximum allowable number of faulty processes will be denoted f. The algorithm is not designed to handle values of f larger than $\lfloor \frac{n+1}{3} \rfloor$ [13].

It is assumed that local clocks are initially synchronized within some β when the protocol begins operation. A nonfaulty process p, upon reaching a time T_0, will broadcast a message to all processes in the network, then wait long enough to receive all messages sent them by other nonfaulty processes at their respective local T_0's. The local times at which p receives these message are recorded. A fault-tolerant average AVG_p of these values is recorded, and an adjustment ADJ_p to $CORR_p$ is computed by $ADJ_p = T_0 + \delta - AVG_p$. The $CORR_p$ value is then updated as the sum of its current value and ADJ_p. This process repeats iteratively after p judges that some period of time P has passed [13].

The fault-tolerant averaging method prevents byzantine errors from having a detrimental effect on the network's synchronization by removing the highest third and the lowest third from the times recorded for receipt of a certain period's packets before computing a value. Different computations may be used to return an "average" for the node, such as taking either the median or mean of the times remaining after discarding the possibly faulty recordings. If the mean is used, the algorithm approaches a synchronization bound of 2ϵ. The algorithm will need to repeat periodically to maintain synchronization in case the clocks drift [13].

In case the assumption cannot be made that local clocks are initially synchronized, the algorithm provides a method for the clocks to reach synchronization before commencing with normal operation. Like the previously described algorithm, it operates in rounds. To begin, a nonfaulty p will broadcast its local time to the other processes in the network. p waits for an interval guaranteed long enough to receive clock readings from the other processes, then computes a correction to its local clock using the abovementioned fault-tolerant averaging method of choice. Process p will then wait long enough to ensure that other processes have calculated their own fault-tolerant averages before it broadcasts a $READY$ message, although if p receives $f + 1$ $READY$ messages it will cease to wait and broadcast its own. After receiving $n - f$ $READY$ messages, p updates its clock according to the computed adjustment and begins the next round by broadcasting its local time. Like the algorithm for maintenance of synchronization, taking the mean of recorded values in the averaging method yields a synchronization approaching 2ϵ [13].

After enough rounds have been completed to achieve a desirable level of synchronization, the algorithm may then begin to operate in the usual manner, maintaining closeness of clock values in case of drift. While switching between the two methods, each process will broadcast for one round without updating its local time. This ensures that on the second round all nonfaulty processes will broadcast, and the algorithm may then proceed [13].

3.2 The Averaging Algorithm

For a network in which clocks do not drift, errors are bounded and packet delivery is reliable, once an algorithm has successfully synchronized a network within a wanted bound it may safely cease operation. The averaging algorithm presented in [19, 20] requires only one iteration to reach its best state of synchronization. For a completely connected network, this results in a skew of less than $u(1 - 1/n)$, with u the uncertainty in the time it will take a packet to be transported from one processor to another and n the total number of processors.

This algorithm behaves as so: for a processor p_i, let HC_i be its hardware clock (which, although it does not drift, the processor cannot modify), adj_i be the adjustment to HC_i which the algorithm will compute, and AC_i be the adjusted clock, such that $HC_i + adj_i = AC_i$. Let d be the maximum delay a packet may experience traversing the network. For simplicity's sake, assume $u = d$, such that a packet may experience a delay ranging from 0 to u. When processor p_i starts the algorithm, it sends a packet containing the reading of HC_i to each processor in the network (except for itself). When p_i receives a packet containing the hardware clock reading HC_j for some other processor j, it will perform a calculation $HC_j + u/2 - HC_i$, then storing the result, perhaps in an array as $diff_i[j]$. Note that $|HC_i + diff_i[j] - HC_j| \leq u/2$. Once p_i has calculated a value for every other processor in the network, it sets the value of adj_i to the sum of all the $diff_i$ divided by n. In this way AC_i becomes an estimate of the average hardware clock value in the network. After all processors have computed their adjusted clock values, they should be within $u(1 - 1/n)$ of each other [19]. Figure 3.1 presents this algorithm in pseudocode format.

```
$diff_i[i] \leftarrow 0$;
**foreach** $p_{l \neq i}$ **do**
    send $HC_i$ to $p_l$
**end**
**if** *A packet is received from any* $p_j$ **then**
    $diff_i[j] \leftarrow HC_j + u_{ij}/2 - HC_i$
**end**
**if** *A packet has been received from all* $p_{k \neq i}$ **then**
    $adj_i \leftarrow 1/n \sum_{k=0}^{n-1} diff_i[k]$
**end**
```

Figure 3.1: Averaging Algorithm; code for processor p_i, $0 \leq i \leq n - 1$.

The ability to achieve closely synchronized clocks within one round and the simplicity of the scheme are strengths of the averaging algorithm for clock synchronization. However, the scheme is not fault tolerant, it requires *a priori* knowledge of the delay and uncertainty packets experience along the network, the number of packets that must be sent out with respect to the number of processors in the network grows on the order of n^2, and extra iterations are required in case of losses. These issues can make the algorithm impractical for use in real networks.

3.3 Offset Delay Estimation Method

The offset delay estimation method described in [15, 16] is a widely used, simple procedure for estimating the delay experienced by packets from systems attempting to synchronize clocks. Notably, a similar scheme is employed by the Network Time Protocol (NTP)

[17], a hierarchical synchronization protocol that has achieved widespread use on the Internet. Using this scheme, a system estimates the packet delivery delay between itself and the system it is attempting to synchronize with using a simple averaging scheme.

Consider systems A and B that wish to synchronize with each other. A will record its current local time, T_3, in a packet which it sends to B. Upon receipt of the packet, B inserts its local time, T_1, then inserts its new time T_2 when returning the packet to A. A records the time T_4 upon receiving the packet. Figure 3.2 illustrates this sequence.

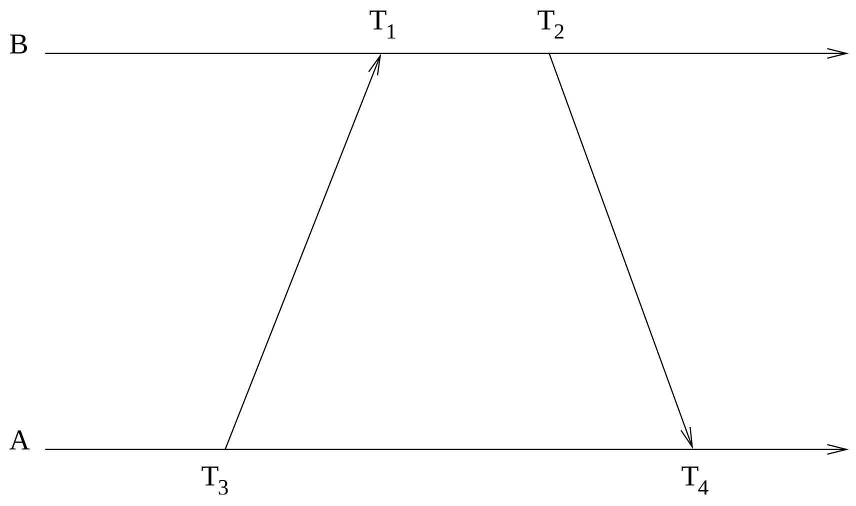

Figure 3.2: Time Values in the offset delay estimation method

For $a = T_1 - T_3$ and $b = T_2 - T_4$, the round trip delay δ experienced by the packet can be expressed by $\delta = a - b$. An estimation of the clock offset θ between the two systems can then be estimated by $\theta = T_2 + \frac{a-b}{2} - T_4 = \frac{a+b}{2}$.

3.4 Application Layer Synchronization in Ad Hoc Networks

Typical ad hoc networks face many impediments to efficient and precise clock synchronization from the application layer. Compared to wired networks, IEEE 802.11 is lossy and faces more contention due to its operation over a shared medium. The DCF randomizes transmission times (increasing uncertainty in the delay) and may cause unlucky packets to wait an unnecessarily long amount of time. Further, unlike theoretical networks with completely reliable transmission and known bounds on the uncertainty of transmission, an everyday ad hoc network without access to global information may drop packets and offers no method for nodes to gauge the uncertainty along links with complete accuracy.

Implementations of algorithms like the Welch-Lynch synchronization algorithm or the averaging algorithm face problems on ad hoc networks. Most obviously, the lack of global knowledge, such as a bound on uncertainty, provides a difficulty for several of the calculations that call upon such a value. Packet losses and congestion may prevent the Welch-Lynch algorithm from being able to compute average values (since no more than $\frac{1}{3}$ of the processes may be faulty, and a process from which a packet was not received is considered faulty). Since the averaging algorithm requires that all packets from all processes be received before it calculates the average, the effects of a dropped packet should be obvious: at least one process will not be able to gather enough information to calculate the average, and will thus not be able to make the adjustment, so its clock will very likely be far out of sync with respect to the others.

The issue of unknown bounds on the uncertainty in packet delivery time may be partially countered using the offset delay estimation method. Instead of volunteering timing information to the other processes in the network, a process can request that information instead, and only reply with its timing information upon receipt of a request. In this manner, the receiver of the timing information may subtract the time its request was sent from the time the reply was received, learning the range of time during which the reply must have been sent. This range may be as large as twice the network's actual uncertainty in packet delivery times, as two packets have traversed the network in this amount of time. Another drawback of implementing the offset delay estimation method is that it doubles the amount of packets that need to be sent in a single iteration.

Since the averaging algorithm will only work if a process receives information from every other process in the network, dropped packets will prevent it from working in a single iteration. To solve this problem, the algorithm may be run for multiple iterations, with the values from old iterations retained assuming they do not become less valid over time (in the case of clock drift, allowances will need to be made). Once a process has received a packet from every other process, it may calculate its adjustment, regardless of the iterations in which its individual packets of timing information were received.

Chapter 4

Simulation Environment

This chapter addresses the environment and design methodology behind the simulations used for this thesis.

4.1 General Notes

All results were gathered using the NS [21] simulation engine from the University of Southern California's Information Sciences Institute (ISI). NS is built using C++ and Tcl/Tk, is freely extensible, and can be used to simulate a wide variety of networking conditions, including many wireless topologies and protocols. Furthermore, the NS simulation engine has been used for a wide variety of research projects, making it a natural choice of simulation environments for this thesis.

These experiments were simulated using ns-2 version 2.31, and made use of a modified version of the "ping" application made available by Marc Greis in [18]. Modifications to the application were made to allow synchronizing clocks to input their local times when sending and receiving packets, making it possible for simulated wireless nodes to gauge each others' local clocks in a manner similar to the offset delay estimation method described above. This modified application was dubbed "ping2."

All experiments were conducted on connected (i.e. non-partitioned) "chain" topologies, in which n nodes are placed in a straight line, with $n-1$ necessary hops between the two outermost nodes. This form of topology was chosen because a tight bound of $1/2u(n-1)$ is known for the closeness of synchronization available on these networks [19].

4.2 Synchronization Experiments

A set of NS simulations were devised to test the effectiveness of and explore the difficulties encountered by application level synchronization schemes in 802.11 networks. These simulations involve were also conducted on chain topologies ranging from sizes 3 to 20.

Synchronization was obtained using a modified version of the averaging synchronization algorithm. Each node in a given the simulated network is initially assigned a random offset that, when added to the system-wide NS time, will serve as the node's local time. This offset is a uniformly distributed real number of seconds ranging from 0 to 604800 (the number of seconds in a week in the Gregorian calendar). Note that since the system wide clock increases over simulated time, the local clock will increment in the correct manner even though this offset is a constant value. Since these offsets are selected randomly the clocks in all likelihood initially be far out of sync.

The synchronization algorithm acts in rounds. At a specified simulator time, the first round begins and each node will send out a ping2 packet requesting timing information from each of the other nodes in the network. Upon receipt of such a request from any other node, a node will input its local time and return the packet. Each node, after sending ping2 packets to all others, will wait for a specified amount of time estimated to be long enough to receive all replies not lost to congestion or other unfavorable network conditions. This waiting period was arbitrarily chosen to be 170 seconds. New rounds begin every 180 seconds, and the simulated environments sends packets through the network, as if running the algorithm, for 30 minutes (1800 seconds) before measurements of synchronization are taken in order to ensure routing information is established before synchronization and measurements of uncertainty begin.

After receiving timing information from another node, this implementation of the averaging algorithm calls for a number of local measurements to be taken. First, the round trip time (the time elapsed between sending out the request and receiving the reply) is determined, to be used in the calculation of the average delay and total uncertainty between the pair of nodes. The one-way trip delay is measured as half the average of all round-trip delays between the pair in question, and the uncertainty between the pair is the difference between the maximum and minimum recorded values for one-half the round trip delay. These figures are then used for the averaging algorithm, just as in networks in which the uncertainty is bounded and use of the offset delay estimation method is not necessary. Figure 4.1 displays some of the calculations used in implementing the averaging algorithm.

```
foreach timing request answered do
    $t_{roundtrip} \leftarrow t_{received} - t_{sent}$;
    $counter \leftarrow counter + 1$;
    $t_{oneway} \leftarrow 0.5 t_{roundtrip}$;
    $t_{delay} \leftarrow \frac{counter-1}{counter} t_{delay} + \frac{1}{counter} t_{oneway}$;
    $t_{max} \leftarrow max(t_{max}, t_{oneway})$;
    $t_{min} \leftarrow min(t_{min}, toneway)$;
    $uncertainty \leftarrow t_{max} - t_{min}$;
end
```

Figure 4.1: Necessary calculations for averaging algorithm implementation

To capture measurements of the uncertainty of synchronization, it seems insufficient to simply stop synchronizing at an arbitrary point in time and record the greatest difference between two clocks in the network: it may happen at any point that the networked nodes are highly synchronized by pure chance, far beyond the limits of a particular synchronization algorithm to approach via its own merit. To obtain more reliable measurements of synchronization errors, then, the nodes were allowed to synchronize for an extra hour, with the worst synchronization of the final 20 synchronization periods becoming the recorded value.

Chapter 5
Simulation Results

This chapter presents the results of the simulations run to gauge degrees of synchronization. Synchronization achievable using the modified averaging algorithm over 802.11 was measured across "chain" topologies of sizes ranging from 3 nodes up to 20.

Results indicated that for chain topologies, the averaging algorithm is quite capable of performing within its upper bound of $\frac{1}{2}u(n-1)$. Since this bound amounts to half the sum of the individual uncertainties across the links of the network (in other words, $u(n-1)$ is equivalent to the maximum uncertainty from one side of the network to the other), it is easy to compare the synchronization favorably. For example, for the 3 node topology the synchronization achieved, 11 ms, was less than $\frac{1}{5}$ the maximal uncertainty displayed across the network. With 4 nodes in the chain, the synchronization of 26.3 ms was less than $\frac{1}{9}$ the maximal variance of 254 ms. Overall results are compiled in Figure 5.1 below:

Topology size	Synchronization	Variance	Ratio
3	11.0 ms	58.3 ms	18.9%
4	26.3 ms	254.1 ms	10.4%
5	39.1 ms	625.7 ms	6.2%
6	74.0 ms	1086.5 ms	6.8%
7	123.2 ms	1793.0 ms	6.9%
8	195.4 ms	2393.9 ms	8.1%
9	308.4 ms	3022.8 ms	10.2%
10	320.3 ms	3626.8 ms	8.8%
11	471.2 ms	4486.9 ms	10.5%
12	457.5 ms	5611.9 ms	8.2%
13	431.1 ms	6577.6 ms	6.6%
14	561.6 ms	8210.0 ms	6.8%
15	650.0 ms	9041.7 ms	7.2%
16	654.5 ms	10213.8 ms	6.4%
17	544.5 ms	11744.9 ms	4.6%
18	703.7 ms	12994.7 ms	5.4%
19	653.6 ms	14536.4 ms	4.5%
20	668.6 ms	16255.8 ms	4.1%

Figure 5.1: Reported synchronization and variance across chain topologies of varying sizes

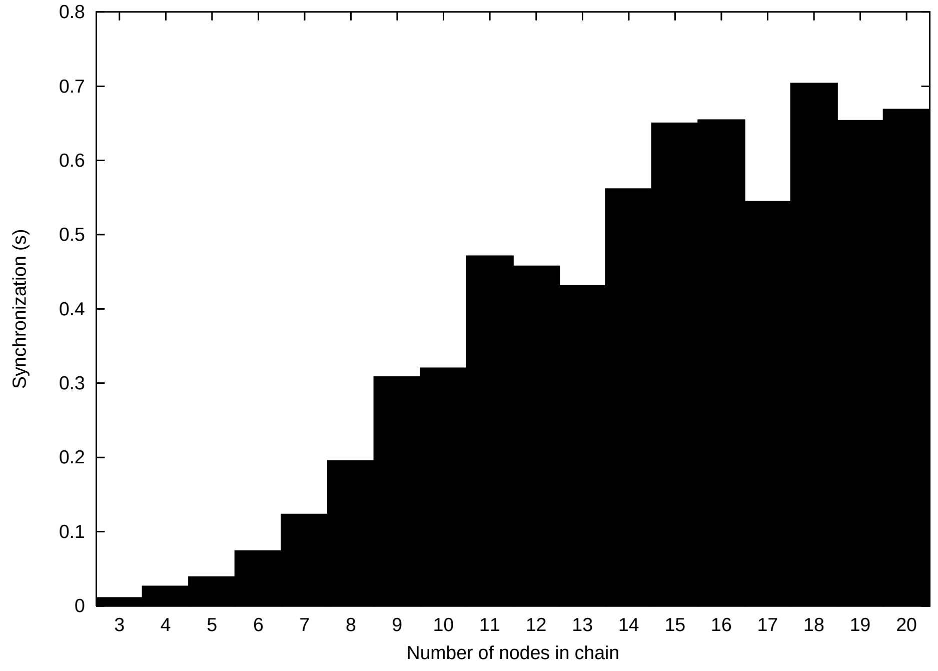

Figure 5.2: Synchronization error at a selection of network sizes

Growth rates for synchronization error and maximum uncertainty with respect to the network size are presented in Figure 5.2 and Figure 5.3 respectively. Of particular interest is the matter in which the synchronization error grows erratically, but on average at a much slower rate than the maximum uncertainty. Also note that the maximum uncertainty grows at a superlinear rate with respect to network size.

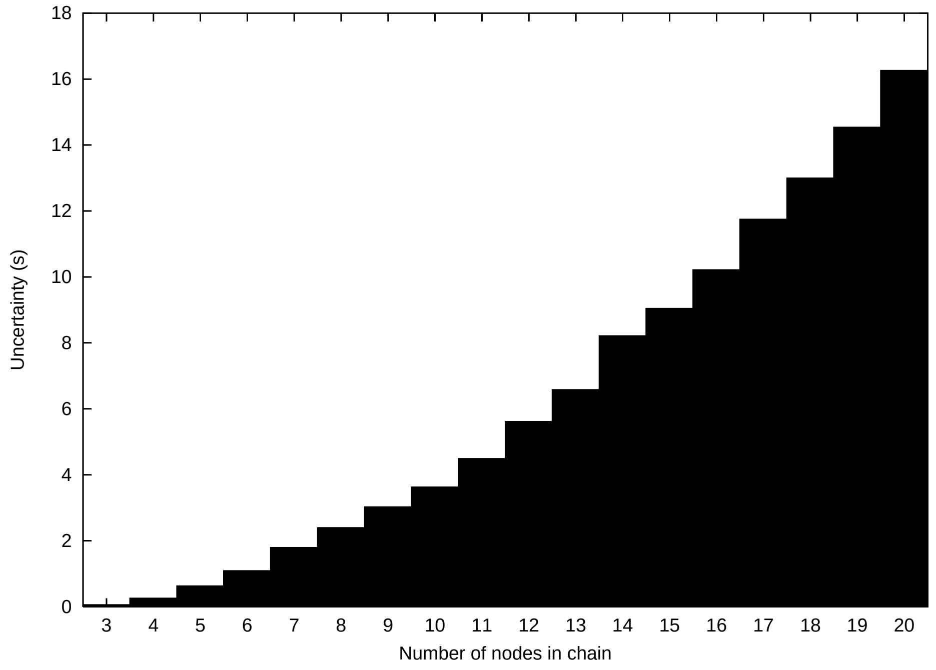

Figure 5.3: Maximum uncertainty at a selection of network sizes

Topology size	Kerberos	TESLA (1 sec)	TESLA (0.1 sec)
3	Yes	Yes	Yes
4	Yes	Yes	Yes
5	Yes	Yes	Yes
6	Yes	Yes	Yes
7	Yes	Yes	No
8	Yes	Yes	No
9	Yes	Yes	No
10	Yes	Yes	No
11	Yes	Yes	No
12	Yes	Yes	No
13	Yes	Yes	No
14	Yes	Yes	No
15	Yes	Yes	No
16	Yes	Yes	No
17	Yes	Yes	No
18	Yes	Yes	No
19	Yes	Yes	No
20	Yes	Yes	No

Figure 5.4: Security protocol compatibility with simulated networks

As mentioned in Chapter 2, some security protocols, such as Kerberos and TESLA, require a degree of clock synchronization in order to function properly. While the degree of synchronization these protocols require is adjustable, default values were used to estimate their performance on wireless networks using this synchronization scheme.

The most common value for required synchronization in Kerberos is 5 minutes. All of the simulations tested achieved that level of synchronization easily. In the simulations in [11], TESLA operated under the assumption that synchronization within the network would remain within a default bound of 0.1 seconds. This was only the case for a handful of the simulations tested here; the topologies of size 6 or smaller fulfilled this requirement. If the assumed synchronization error in the network is raised to 1 full second, however, TESLA will operate on every topology tested. These results are presented in Figure 5.4.

Chapter 6
Conclusions

Large variations in one-way trip and round-trip times can negatively impact the performance of many protocols, most notably those with a need for high degrees of clock synchronization. Also, as is shown via simulation, 802.11 is subject to variations in delay posing a hindrance to delay estimation and thus synchronization even in small networks. In addition, the large variation in delays occurring in 802.11 networks has a negative impact on application layer clock synchronization that can build up quickly as network size increases. The variation is bad enough in topologies of only 7 nodes with little traffic to prevent security protocols (TESLA) from working properly, even though our simulated algorithm was able to remain well below the theoretical synchronization error bound. This problem is exacerbated over multiple hops, or in dense networks with high traffic loads, as the packet may experience more collisions or traffic, causing the DCF to (generally) wait longer, as evidenced by the superlinear growth of the variation with respect to network size.

In deployment of 802.11 networks, care must be taken regarding acceptable degrees of variations in communication time. Careless deployment of protocols requiring high degrees of clock synchronization or low degrees of variation in multiple-hop trip delays will achieve poor results.

Bibliography

[1] M. Frodigh, P. Johansson, and P. Larsson, "Wireless ad hoc networking–The art of networking without a network," Ericsson Review 4, 2000.

[2] G. Bianchi, "Performance Analysis of the IEEE 802.11 Distributed Coordination Function," IEEE Journal on Selected Areas in Communications, Vol. 18, pp. 535-547, March 2000.

[3] C. E. Perkins and E. M. Royer, "Ad hoc on-demand distance vector routing," Proceedings of the 2nd IEEE workshop on mobile computing systems and applications, pp. 90-100 February 1999.

[4] C. Perkins, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers," SIGCOMM '94 Conference on Communication Architectures, Protocols, and Applications, pp. 234-24

[5] J. Steiner, C. Neuman, and J. Schiller, "An authentication service for open network systems," Proceedings of the USENIX Winter Conference, pp. 191-202, February 1988. http://citeseer.ist.psu.edu/steiner88authentication.html

[6] B. C. Neuman and T. Tso, "Kerberos: An Authentication Service for Computer Networks," IEEE communications 32, pp. 33-38, September 1994.

[7] R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," Communications of the ACM, 21(12), pp. 993-999, December 1978.

[8] D. Coppersmith, "The Data Encryption Standard (DES) and its Strength Against Attacks," IBM Journal of Research and Development, Vol. 38, No. 3, May 1994.

[9] A. Perrig, R. Canetti, J. Tygar, and D. X. Song, "Efficient Authentication and Signing of Multicast Streams over Lossy Channels," IEEE Symposium on Security and Privacy, May 2000.

[10] A. Perrig, R. Canetti, J. Tygar, and D. X. Song, "Efficient and Secure Source Authentication for Multicast," Network and Distributed System Security Symposium, NDSS '01, pp. 35-46, February 2001.

[11] Y. C. Hu, A. Perrig, and D. B. Johnson, "Ariadne: A secure on-demand routing protocol for ad hoc networks," The 8th ACM International Conference on Mobile Computing and Networking, September 2002.

[12] Yih-Chun Hu, David B. Johnson, and Adrian Perrig "SEAD: Secure Efficient Distance Vector Routing for Mobile Wireless Ad Hoc Networks," Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2002), June 2002.

[13] J. Lundelius and N. Lynch, "A New Fault-Tolerant Algorithm for Clock Synchronization," Information and Computation 77, pp. 1-36, April 1988.

[14] B. Dutertre, "The Welch-Lynch Clock Synchronization Algorithm," Technical Report 747, Department of Computer Science, Queen Mary and Westfield College, March 1998.

[15] F. Cristian, "Probabilistic Clock Synchronization," Distributed Computing 3, pp. 146-158, 1989

[16] B. Sundararaman, U. Buy, and A. D. Kshemkalyani, "Clock Synchronization for Wireless Sensor Networks: A Survey," Ad Hoc Networks 3 (3), pp. 281-383, 2005.

[17] D. L. Mills, "Internet Time Synchronization: the Network Time Protocol," IEEE Transactions on Communications, Vol 39, no 10, pp. 1482-1493, October 1991.

[18] M. Greis, "Tutorial for the Network Simulator "ns,"" http://www.isi.edu/nsnam/ns/tutorial/

[19] S. Biaz and J. Welch, "Closed Form Bounds for Clock Synchronization Under Simple Uncertainty Assumptions," Information Processing Letters, Vol 80, pp. 151-157, December 2001.

[20] H. Attiya and J. Welch, "Distributed Computing: Fundamentals, Simulations and Advanced Topics," McGraw Hill, 1998.

[21] "The Network Simulator - ns-2," http://www.isi.edu/nsnam/ns/

[22] A. Mainwaring, J. Polastre, R. Szewczyk, and D. Culler, "Wireless Sensor Networks for Habitat Monitoring," WSNA'02, Atlanta, Georgia, September 2002. http://citeseer.ist.psu.edu/mainwaring02wireless.html

[23] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless Sensor Networks: A Survey," Computer Networks, 38(4), pp. 393-422, March 2002. http://citeseer.ist.psu.edu/akyildiz02wireless.html