December 1, 2010

Broadcast Authentication for Wireless Devices

An Android Implementation

Pranjal Kumar Singh (IIT2008008)
Divij Vaidya (IIT2008077)

Broadcast Authentication for Wireless Devices

An Android Implementation

Divij Vaidya

IIT2008077

B.Tech. V Sem

Pranjal Kumar Singh

IIT2008008

B.Tech. V Sem

Dr. Shekhar Verma

Project Supervisor

Table of Contents

Goal of the Project

This project is aimed at the implementation of the μTesla Broadcast Authentication Protocol on the Android™ Platform.

Motivation behind the Project

The idea behind the project work is to introduce an energy-efficient Broadcast Authentication Protocol for use on wireless devices.

However, this project does not involve encryption of the data being transferred, as it is assumed that the application of this protocol would not need confidential data to be broadcasted. In such applications, the data is required to be coming from a valid and authentic source, and as such, secrecy in the data is not required. We explore these applications as follows.

There are many applications of broadcast authentication on wireless devices, for example-

- Disaster Management: It can be used to broadcast critical information to concerned (or affected) people about any upcoming threat.
- Coordination in an Organization: It can be used to make important announcements to concerned people about any upcoming work (such as an emergency meeting), and can serve as a quick way to coordinate between people.
- Hospital Management: It can be used for various coordination related activities in a Hospital.

The inspiration for use of μTesla came from its origin in Wireless Sensor Networks[i].

Wireless Sensor Networks

Wireless Sensor Network (WSN) is a low-powered wireless network formed by inexpensive devices that are battery-powered with limited computing resources. A WSN consists of many Wireless Sensor Devices (WSD) that are deployed in the field to sense physical phenomena such as humidity, temperature, vibration, light, etc. The Wireless

Sensor Devices form point-to-point low-powered wireless links to each other for communication.

A WSN network has security provisions to achieve data confidentiality or privacy, data integrity, data authentication, availability, and data freshness[ii]. Data confidentiality ensures only authorized parties can access the data. Data integrity ensures only authorized parties can modify the data and the data is not altered during transmission. Data authentication ensures the data is really sent by the claimed sender instead of fabricated by someone else. Availability ensures reliable delivery of data against denial of service attacks. Data freshness ensures the data is current and fresh (i.e., is not replayed by an adversary.)

SPINS is a security protocol for Wireless Sensor Networks, that achieves the above mentioned security parameters. SPINS has two components: SNEP (Sensor Network Encryption Protocol) and µTesla. SNEP provides security functions such as encryption, two-party data authentication, replay protection, freshness, and integrity. µTesla is similar to Tesla[iii] to provide broadcast authentication. However, µTesla has been modified to reduce overhead from Tesla, which is designed for a regular network, to run on the resource-restricted WSN devices.

Essentially, in a TESLA scheme, the sender would send the receiver the key 'k' to authenticate a packet that was sent in the past. SPINS network architecture assumes that devices only communicate with a super base-station. Thus, the network topology formed is a star network. The types of communication in a star network are: device to base-station, base-station to device, and broadcast from base-station. SNEP would provide security protection for the device to base-station, and base-station to device communication while µTesla would protect the base-station broadcast communication to devices. SPINS uses symmetric cryptography because of the belief that public key cryptography is too intensive for the resource-limited WSN nodes. SPINS was never fully implemented and was never released by TinyOS.

μTESLA: Broadcast Authentication Protocol[iv]

μTesla is a part of SPINS protocol that provides authentication for data broadcast. It is a modified version of TESLA protocol.

μTesla is implemented to solve the following inadequacies of TESLA in sensor networks:

- TESLA authenticates the initial packet with a digital signature, which is too expensive for our sensor nodes. μTesla uses only symmetric mechanisms.
- Disclosing a key in each packet requires too much energy for sending and receiving. μTesla discloses the key once per epoch.
- It is expensive to store a one-way key chain in a sensor node. μTesla restricts the number of authenticated senders.

The Android™ Platform

Android is a mobile operating system developed by Google and is based upon a modified version of the Linux kernel. It was initially developed by Android Inc. (a firm purchased by Google) and later positioned in the Open Handset Alliance.

Work till Mid-Semester

- Read about the various aspects (including security) of Wireless Sensor Networks[v].
- Read about various security concerns and the proposed security protocols that were presented in the last decade[vi].
- Selected μTesla as the target protocol to be implemented.
- Selected Android OS as the target platform.
- Analyzed the source code of `wpa_supplicant` to find out an optimal way to implement the μTesla protocol.

Description of Complete Work

μTesla has multiple phases: Sender setup, sending authenticated packets, bootstrapping new receivers, and authenticating packets. We implemented μTESLA for the case where the base station broadcasts authenticated information and the clients verify that information. Our model consists of one broadcasting server and multiple Clients.

Implementation Specifications

Java language has been used to code the protocol since Android provides support for Java only. Hence Java.net package is used for socket programming. The data transfer is being achieved through the UDP. Multicast sockets are used to receive and send data. It is assumed (and can be safely assumed) that the clocks of the mobile devices running this application are synchronized from the central time server in the Communication Network Tower. The maximum size of data that can fit in a packet has been taken as 1000 bytes (but can be easily changed, up to the theoretical limit of a UDP Packet size).

Sender Setup

The sender first generates a sequence of secret keys (or key chain). To generate the one-way key chain of length n, the sender chooses the last key K_n randomly, and generates the remaining values by successively applying a one-way function F (e.g. a cryptographic hash function such as MD5):$K_j = F(K_{j+1})$. Because F is a one-way function, anybody can compute forward, e.g. compute K_0 ,..., K_j givenK_{j+1},but nobody can compute backward, e.g. computeK_{j+1} given only K_0 ,...,K_j , due to the one-way generator function. This is similar to the S/Key one-time password system[vii]. After preparing the keys, the message to be transferred is taken from the user by a suitable input method, and packets are prepared. Some extra information is sent with the packets which helps the Client verify the packet and add proper numbering to it. A Message Authentication Code (MAC) is also sent with each packet for this purpose.

In our implementation of Sender setup, the following tasks are performed:

Generation of Keys

Keys are generated through MD5(Message Digest Algorithm 5), which is a one-way hash function. A large random string is used to compute the initial key, and then N keys are computed and stored in an Array `Key`. The implementation of MD5 is available in the `java.security` package.

The number of keys to be generated is defined by the constant MAX_KEYS. Keys are generated and used in opposite order, to maintain the use of one-way function, i.e. The initial key is at `Key[MAX_KEYS-1]` and the last generated key at `Key[0]`. The Keys are used from the beginning, so that `Key[i+1]` cannot be computed from `Key[i]`.

Division of Packets

The input string is divided into substrings of length 1000 bytes, and stored in a String Array.

Computing the MAC

Later in this report, we will see that one key is used to authenticate several packets (stored in PACKETS_PER_INTERVAL). Hence, one key is associated with a number of packets.

The following algorithm is used to compute the MAC:

1. Append the Key associated with the packet string to the string.
2. Compute a hash value of the string with the following hash function (computed through integer arithmetic only):

$$hash(s) = \sum_{i=0}^{n-1} s[i] \times 3^{n-1-i}$$

3. Compute the MD5 checksum of the given hash.

We perform these operations to avoid computing the MD5 checksum on the data directly, as it is computationally very expensive.

Preparation of Packets

The packets are structured in the following format:

```
<num>1</num><mac>5053e842688a93bbfbf133c4c0a7d85f</mac><data>Thi
sisthedatainthepacket.</data>
```

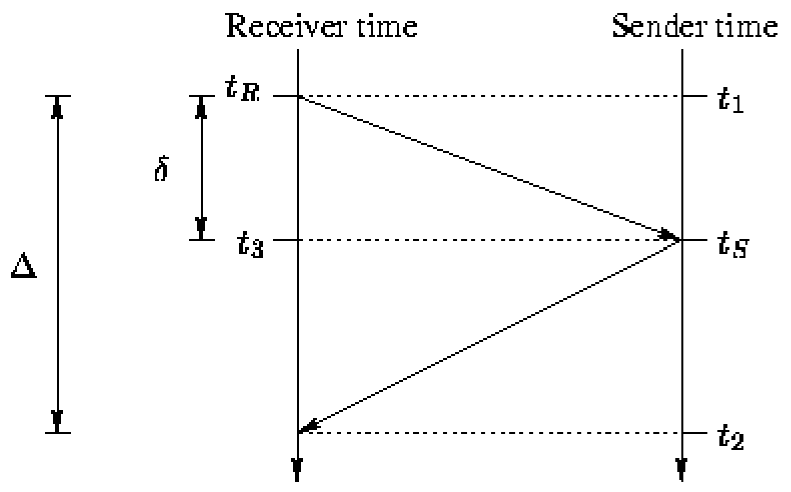
Please note that this type of structuring has been done for clarity. If we aim at reducing the packet overhead, the same information can be reduced by a factor of 4.

Bootstrapping the Receivers

Now that the packets have been prepared for sending, the Server needs to do some additional work for Time Synchronization between the Sender and the Receiver. Since the traffic conditions of Mobile Networks are quite unpredictable, the round trip time of the packets needs to be calculated before sending. Also, the receivers need to be sent some information about the data transfer that is going to take place, so that we can achieve loose time synchronization.

Time Synchronization

μTesla does not need the strong time synchronization properties that sophisticated time synchronization protocols provide[viii,ix,x], but only requires loose time synchronization, and that the receiver knows an upper bound on the sender's local time. We denote the real difference between the sender and the receiver's time with δ. In loose time synchronization, the receiver does not need to know the exact δ but only an upper bound on it, Δ, which we also refer to as the *maximum time synchronization error*.



The figure shows direct time synchronization between the sender and the receiver. The receiver issues a time synchronization request at time t_R, at which time the sender's clock is at time t_1. The sender responds to the request at its local time t_S. In μTesla, the receiver is only interested in an upper bound on the sender's time. When the receiver has its current time t_r, it computes the upper bound on the current sender's time as $t_s \leq t_r - t_R + t_S$. The real synchronization error after this protocol is δ. The receiver, however, does not know the propagation delay of the time synchronization request packet, so it must assume that the time synchronization error is Δ (or the full round-trip time (RTT)).

We calculate the full Round Trip time as follows:

1. The Server broadcasts its timestamp packet `<tss>timestamp</tss>`.
2. On receiving this packet, the client responds with its own timestamp packet `<tsc>timestamp</tsc>`.
3. Server calculates and stores the RTT from this information for multiple clients.
4. Server calculates Δ as,

$$\Delta = \max(\mathrm{RTT}[\,])$$

Sending Bootstrapping information

Other bootstrapping information is calculated, such as the time interval, total transfer time and the time of start of transmission by the server.

The Time Interval,

$$\mathrm{T_{int}} = \Delta \times (\mathrm{PACKETS\_PER\_INTERVAL} + 1)$$

Total Transfer Time,

$$T_{total} = \left(\left\lfloor \frac{\mathrm{NUM\_PACKETS}}{3} \right\rfloor + DELAY + 1\right) \times T_{int}$$

A bootstrapping packet is prepared by the server as:

`<bootstrap> Δ :`T_{int}`:`T_{total}`:`T_{start}`</bootstrap>`

Here, T_{start} is the time at which the Server will start sending data.

Sending Data and Keys

After sending the Bootstrap information, the Server waits for the time T_{start} and then starts sending the data packets, at the rate of PACKETS_PER_INTERVAL per time interval.

The Keys are sent after a particular delay of intervals stored in the constant DELAY. Instead of sending the keys right at the beginning of an interval, they are sent somewhere in the middle of the interval (for some added delay in disclosing the keys).

The Server thread quits after sending all the data and key packets.

Authenticating Broadcast Packets

The security issues addressed by our protocol are:

1. Sender Authentication
2. Data Authentication
3. Data Freshness
4. Non-repudiation

Packet Arrival

When a receiver receives the packets with the MAC, it needs to ensure that the packet could not have been spoofed by an adversary. The threat is that the adversary already knows the disclosed key of a time interval and so it could forge the packet since it knows the key used to compute the MAC. Hence the receiver needs to be sure that the sender did not disclose the key yet which corresponds to an incoming packet, which implies that no adversary could have forged the contents. This is called the security condition, which receivers check for all incoming packets. Therefore the sender and receivers need to be loosely time synchronized and the receivers need to know the key disclosure schedule. If the incoming packet satisfies the security condition, the receiver stores the packet (it can verify it only once the corresponding key is disclosed). If the security condition is violated (the packet had an unusually long delay), the receiver needs to drop the packet, since an adversary might have altered it.

Security Condition is checked by verifying that when the packet is received by the receiver, the corresponding key of that interval has not been disclosed by the server. This is done as follows.

Let the time of receiver at which the packet is received is t_r, the key disclosure interval d, t_s is the time at server when t_r is the time at receiver, T_{int} is the interval length, Δ is the time delay, T_{start} is the starting time of transfer.

Given that,

$$t_s - t_r \leq \Delta$$

$$t_s \leq \Delta + t_r$$

$$\therefore \quad S = \lfloor ((\mathrm{t_r} + \Delta) - \mathrm{T_{start}})/\mathrm{T_{int}} \rfloor$$

$$\& \; R = \lfloor ((\mathrm{t_r} - \Delta) - \mathrm{T_{start}})/\mathrm{T_{int}} \rfloor$$

To satisfy security condition:

$$S \leq R + d$$

$$\Rightarrow \quad \boxed{\lfloor ((\mathrm{t_r} + \Delta) - \mathrm{T_{start}})/\mathrm{T_{int}} \rfloor \leq \lfloor ((\mathrm{t_r} - \Delta) - \mathrm{T_{start}})/\mathrm{T_{int}} \rfloor + d}$$

Where:

t_r=Time at which packet is received at client

t_s=Server time at t_r

S= Interval of server at time t_r

R=Interval of client at time t_r

T_{start} = Starting time of the transmission

T_{int}=Interval size

Δ=Time delay due to loose time synchronization

d=key disclosure delay

If the packet arriving at t_r satisfies this condition then the packet is said to satisfy the security condition. This prevents the Man in the Middle attacks as well as provides Data Authentication.

Key Verification

As soon as the node receives a key K_j of a previous time interval, it authenticates the key by checking that it matches the seed key it knows K_{seed}, using a small number of applications of the one-way function $F: K_{seed} = F^{j-seed}(K_j)$. If the check is successful, the new key K_j is authentic and the receiver can authenticate all packets that were sent within the time intervals number j. The keys are verified using the `verifyKey()` routine.

The security of this protocol is dependent on the loose time synchronization and the security condition. Given that security condition is satisfied and the verified key is used to verify the packets, this protocol provides data freshness, user authentication, data authentication and Non-repudiation.

Termination of Transfer

The bootstrapping packet contains the `maxtransfer` time which is estimated by the server as the maximum total time that the transfer can take place. Thus, at `t=(maxTransferTime+Δ)` the code at the client side finishes execution. This is done to ensure that the client side does not keep on waiting indefinitely for the packets to arrive.

Message Rebuilding

Once the whole transfer is complete, the message is build using the message sequence number embedded in each message. This is done in the `showmessage()` routine.

Android Development

This protocol was ported on android OS. The application developed had the following development components:

1. **Application GUI:** The GUI for the application was done using the Android API and the layout design was prepared using XML. The layout is a nested structure of Different Layout elements.
2. **Back End:** The back end was coded in JAVA which included multiple threads for clients and server, UDP datagram transmission using socket programming etc.
3. **Integration:** The Back end was integrated with the GUI using Android API. It involved creation of Activities, handlers, intends and listeners for various GUI elements.

Implementation Issues

- Δ, the time delay achieved due to loose time synchronization may vary a lot in different networks. Hence, this delay is calculated using the maximum of RTT taken from all the clients.
- The Android simulator does not support Wi-Fi. Solution to this problem lies in simulating the network locally using different thread for clients and the server. Hence, a single Android application was created for both the server and the client but running different threads.
- Android applications on a simulator run in an isolated environment from the system. Hence, additional changes have to be done in the `Android.manifest.xml` file to grant it permissions to access the system resource such as the system clock and network access.
- The simulation varies locally depending on the platform. In windows, the transfer using the UDP packets take place much more slowly as compared to Linux environment.

- The protocol is susceptible to many attacks such as DoS (Denial of Service) attacks since the protocol is developed to provide broadcast authentication only.
- Implementation of an efficient algorithm to compute MAC had to be devised as the processing on the client side has to be minimal.

Result and Analysis

The protocol works as expected on both Linux and the Android Simulator.

Test Cases for the Protocol

The following tests were run on the protocol for data transfer:

Test-1:

Data Size: 6000 bytes

No. of Packets: 6

Result: All packets successfully transferred and verified.

Test-2:

Data Size: 11902 bytes

No. of Packets: 12

Result: All packets successfully transferred and verified.

Sample Attack attempt at the Protocol

An attacker agent was created in Java, and it was used to attempt a Man-in-the-middle attack on the Receiver. The advantage given to the attacker was that, it was supposed to know all the algorithms used to compute the Key and the MAC.

So, the attacker operated on the following algorithm:

1. Capture a data packet from the Server.
2. Capture the corresponding Key from the Server, when it is released.

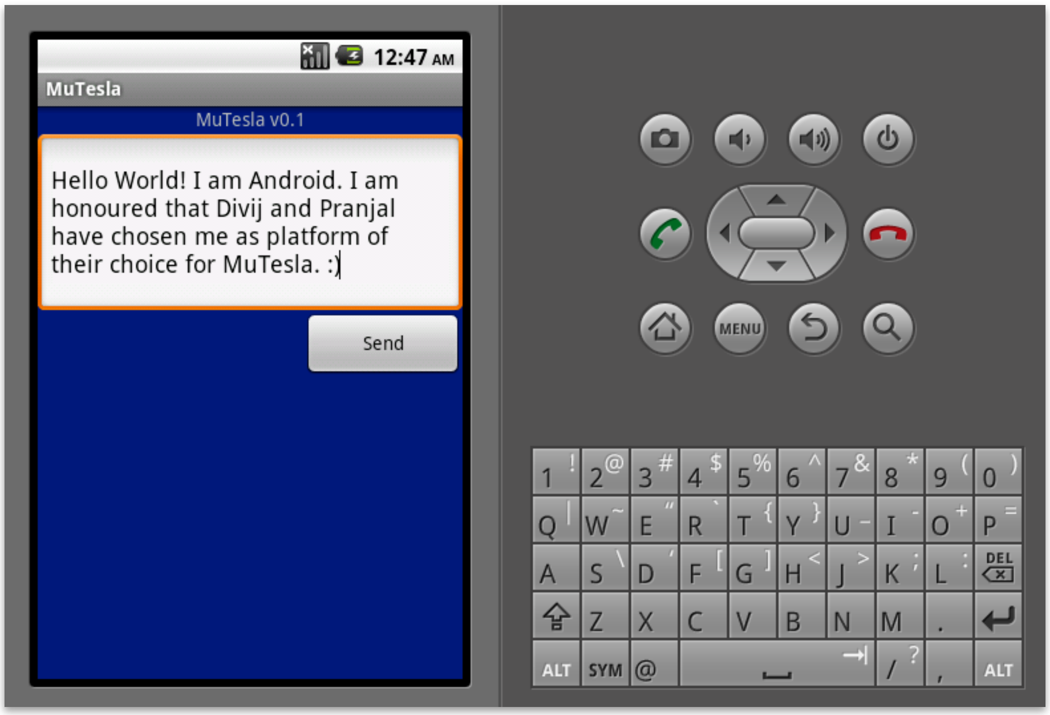
3. Change the data in the packet, and replace its MAC with a new computed MAC, that is valid.
4. Send the new "compromised" packet to the receivers.

RESULT: All the receivers actively rejected the packet sent by the attacker, as it arrived unusually late, which means the security condition for the packet failed.

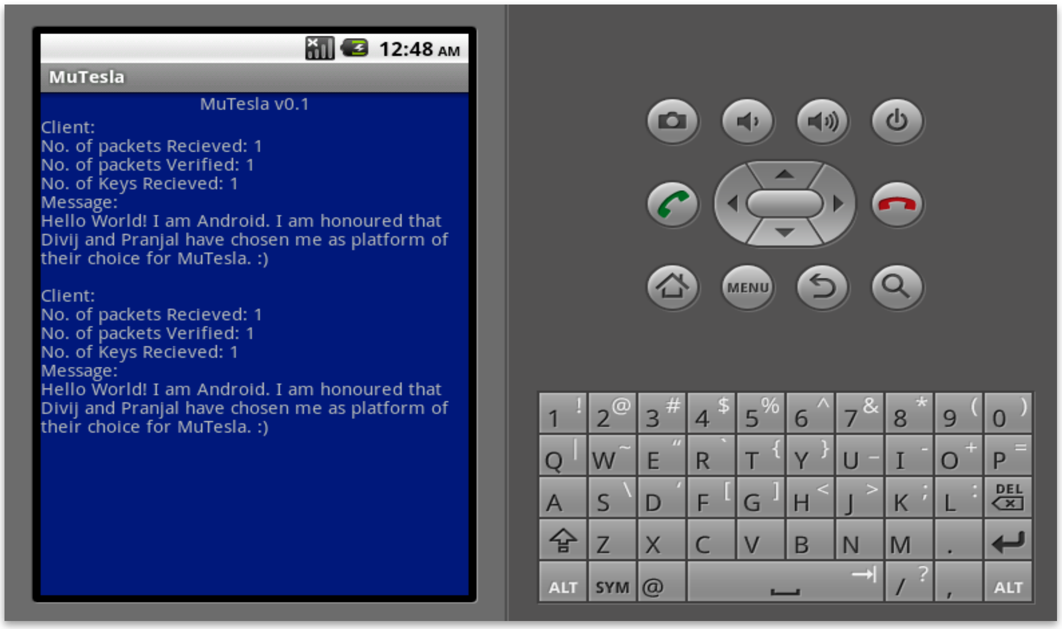
Rejection of Valid Packets

Sometimes, due to any unusually long delay in packet arrival at the client side, from the server, the packets may be rejected. This delay is caused by the limitations of the UDP Protocol, and in such a case, the data needs to be re-sent. The protocol doesn't compromise with the security condition and any late packets will be rejected, because they might have been spoofed by an adversary.

Screenshots of the Android Application



Hello World! I am Android. I am honoured that Divij and Pranjal have chosen me as platform of their choice for MuTesla. :)



```
12:48 AM
MuTesla
          MuTesla v0.1
Client:
No. of packets Recieved: 1
No. of packets Verified: 1
No. of Keys Recieved: 1
Message:
Hello World! I am Android. I am honoured that
Divij and Pranjal have chosen me as platform of
their choice for MuTesla. :)

Client:
No. of packets Recieved: 1
No. of packets Verified: 1
No. of Keys Recieved: 1
Message:
Hello World! I am Android. I am honoured that
Divij and Pranjal have chosen me as platform of
their choice for MuTesla. :)
```

Conclusion

The μTesla Broadcast Authentication Protocol was successfully ported and tested on the Android™ Platform. It is the first attempt anyone has made at porting this protocol to mobile operating systems.

Since our primitives are solely based on fast symmetric key mechanisms for authentication, and use no asymmetric algorithms, our building blocks are applicable to a wide variety of device configurations. The computation costs of symmetric key mechanisms are low.

Beyond this Implementation

This implementation of the μTesla protocol was to provide an energy efficient protocol for use on Mobile Devices. Since it is purely a Java based implementation, it can also be ported to other Operating Systems that support Java based applications, with minimum hassle. The implementation presented in this project work is highly customizable, and can be used as a base for making a practical Authentication System.

REFERENCES

[i]Mayank Saraogi, "Security In Wireless Sensor Networks", 2005.

[ii] Yang Xiao. Security in Sensor Networks. Florida: Auerbach Publication, 2007. Page 239.

[iii] A. Perrig, R. Canetti, J.D. Tygar, Dawn Song. "The TESLA Broadcast Authentication Protocol."

[iv] A. Perrig, R. Szewczyk, V. Wen, D. Culler, J.D. Tygar. "SPINS: Security Protocols for Sensor Networks." Proceedings of Seventh Annual International Conference on Mobile Computing and Networks (MOBICOM 2001), July 2001.

[v] D. Boyle, T. Newe, "Security Protocols for use with Wireless Sensor Networks: A Survey of Security Architecture", 2007.

[vi] Jinat Rehana, "Security of Wireless Sensor Network", Helsinki University of Technology, 2009.

[vii] Neil M. Haller. The S/KEY one-time password system. In ISOC, 1994.

[viii] L. Lamport and P. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52-78, 1985

[ix] D. Mills. Network Time Protocol (version 3) specification, implementation and analysis. Internet Request for Comment RFC 1305, Internet Engineering Task Force, March 1992.

[x] B. Simons, J. Lundelius-Welch, and N. Lynch. An overview of clock synchronization. In B. Simons and A. Spector, editors, *Fault-Tolerant Distributed Computing*, number 448 in LNCS, pages 84-96, 1990.