IMPLEMENTATION OF KEY DISTRIBUTION SCHEMES ON REAL SENSOR NETWORK NODES

by Hüseyin Ergin

Submitted to the Graduate School of Sabancı University
in partial fulfillment of the requirements for the degree of
Master of Science

Sabancı University

February, 2011

IMPLEMENTATION OF KEY DISTRIBUTION SCHEMES ON REAL SENSOR NETWORK NODES

APPROVED BY:

Assoc. Prof. Dr. Albert Levi (Thesis Supervisor)	.....
Assist. Prof. Ahmet Onat	.....
Assist. Prof. Cemal Yılmaz	.....
Assoc. Prof. Kemalettin Erbatur	.....
Assoc. Prof. Özgür Gürbüz	.....

DATE OF APPROVAL:.....

IMPLEMENTATION OF KEY DISTRIBUTION SCHEMES ON REAL SENSOR NETWORK NODES

Hüseyin Ergin

CS, Master's Thesis, 2011

Thesis Supervisor:Albert Levi

Keywords: Sensor Network Security, Key Distribution, TelosB Motes

Abstract

Wireless Sensor Network is a network type that consists of small sensor devices. The communication between these devices must be secured in case of an attack. Sensor devices have to share a secret key for secure communication. There are several key distribution schemes for wireless sensor networks in the literature. The most common key distribution scheme is the basic scheme which is proposed by Eschenauer and Gligor. Basic scheme has three phases; *Key Predistribution*, *Shared Key Discovery* and *Path-key Establishment*. Ergun proposed an alternative phase to *Path-key Establishment*, called *Key Transfer* phase. To the best of our knowledge, there is no real node implementation of the basic scheme. In this thesis, we implemented all three phases of the basic scheme and Ergun's Key Transfer phase on a real sensor device. We use TelosB devices, which have 10kB RAM, 1 MB flash memory, a microcontroller and RF interface. We design flowcharts for each phase, create packet structures, implement in NesC programming language and test the implementation. We analyze the results using processing time, code space and memory usage metrics. We show that *Key Transfer* phase is more efficient than *Path-key Establishment* phase.

ANAHTAR DAĞITIM ŞEMALARININ GERÇEK DUYARGA AĞI AYGITLARINDA GERÇEKLENMESİ

Hüseyin Ergin
CS, Yüksek Lisans Tezi, 2011
Tez Danışmanı: Albert Levi

Anahtar Kelimeler: Duyarga Ağları Güvenliği, Anahtar Dağıtımı, TelosB Aygıtları

Özet

Kablosuz Duyarga Ağları, içerisinde küçük duyarga aygıtları barındıran bir ağ tipidir. Bu aygıtlar arasındaki haberleşme bir saldırı olma ihtimaline karşılık güvenli yapılmalıdır. Duyarga aygıtları güvenli haberleşme için gizli anahtarlar paylaşırlar. Literatürde bir çok anahtar dağıtım şeması vardır. Bunlardan en bilineni Eschenauer ve Gligor'un sunduğu basit şemadır. Basit şemanın üç evresi vardır: *Anahtar Öndağıtım*, *Ortak Anahtar Keşfetme* ve *Yol Anahtarı Kurma*. Ergun, *Anahtar Transferi* adında, *Yol Anahtarı Kurmaya* alternatif bir evre önermiştir. Bildiğimiz kadarıyla, literatürde basit şemanın gerçek aygıtlar üzerinde gerçeklenme çalışması yoktur. Bu tezde basit şemanın üç evresini ve Ergun'un *Anahtar Transferi* evresini gerçek bir duyarga aygıtında gerçekledik. Bunun için 10 kB RAM, 1 MB flash bellek, mikroişlemci ve RF arayüzü olan TelosB isimli aygıtları kullandık. Her evreyi tasarladık, paket yapılarını oluşturduk, NesC programlama dilinde kodladık ve gerçeklemeyi test ettik. Sonuçları işlem zamanı, kod boyutu ve bellek kullanım oranı metriklerini kullanarak analiz ettik. *Anahtar Transferi* evresinin *Yol Anahtarı Kurma* evresinden daha verimli olduğunu gösterdik.

To my dearest, Tülay

Acknowledgements

I would like to thank my thesis supervisor, Albert Levi, for his precious support and help throughout the project and my master of science education, also for admitting me to the project at the beginning.

I especially thank my dearest, Tülay Sarıkaya, for her great mental support at each step of my thesis.

I also thank my FENS 2001 Lab friends; Sami Dirik, Erman Pattuk, Erdal Mutlu, Burcu Özçelik, Cengiz Örencik, Emre Kaplan, Emine Dumlu, Barış Altop, İsmail Fatih Yıldırım, Onur Durahim and all other residents.

I also thank Ahmet Onat, Cemal Yılmaz, Kemalettin Erbatur and Özgür Gürbüz for being my jury and giving me from their valuable time.

I also thank Scientific and Technological Research Council of Turkey (TÜBİTAK) for funding me by BİDEB scholarship during my education.

Contents

List of Figures

List of Tables

1 Introduction

Wireless Sensor Network (WSN) is a network type which consists of small sensing devices [1][2]. These devices are connected through RF interface and they usually use batteries as the source of energy. Moreover, there are a small microcontroller and sensing interfaces such as temperature, sound, vibration, pressure and humidity.

The most common picture of wireless sensor network is shown in Figure 1. All nodes (from now on we will call each device as node) are supposed to be connected to each other and one gateway sensor node performs the communication between these nodes and the operator. Gateway sensor node is usually more powerful than the other nodes but it may also be the same.

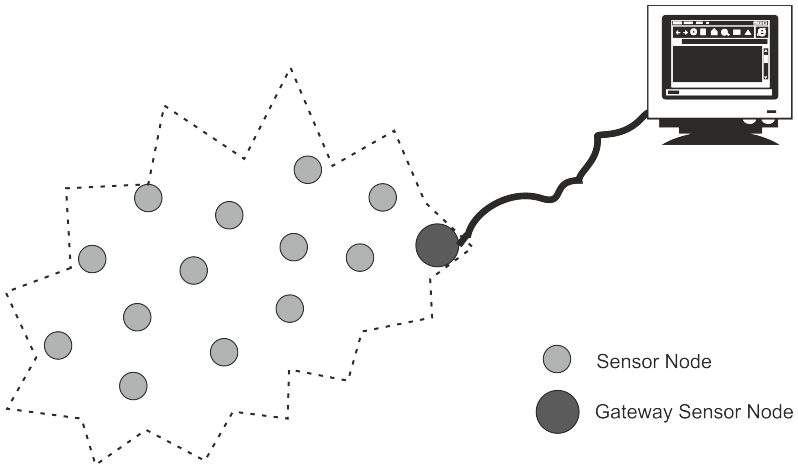

Figure 1: Wireless Sensor Network

Sensor networks can be used in many areas such as military applications, healthcare applications, habitat monitoring etc. The application of a sensor network is usually done in the following way:

- Each node is programmed and the nodes are deployed in the environment. This environment can be a forest, a military zone or just a home to collect information.
- The gateway sensor node is triggered by a computer or an operator and it broadcasts command to other nodes.
- All nodes start to work and do the sensing tasks. Moreover they send the necessary data to the operator with the help of gateway sensor node and other nodes.

In case of an attack, the messages between nodes must be encrypted. For these kinds of scenarios, each node must share a secret key with other nodes so that it can send the messages by encrypting with the related secret key of its neighbors. Since the power consumption of nodes is limited, strong asymmetric cryptographic algorithms such as RSA [3] are not suitable. For this reason, we use symmetric cryptography [4] in which both parties use the same key for encryption and decryption. Also power consumption of symmetric cryptography is much lower than asymmetric cryptography.

Delivering these secure keys over the air to each node is not a feasible way because of sniffing attacks. Thus, these keys must be pre-distributed to each node while programming them. Giving the same key to each node for cryptography purposes is not suitable, because if a node is compromised, all the network becomes insecure. In order to improve the security of the network, the nodes must encrypt or decrypt their communication with different keys. At this point a new problem takes place. The nodes have limited memory capacity. Therefore they will not have enough memory for sharing separate keys for all the nodes in the network.

In order to overcome these difficulties, Eschenauer and Gligor [5] proposed a random key predistribution scheme. It is also called the *basic scheme* in the literature. In this scheme, a large key pool is created and a limited subset of this pool is distributed to each node while programming. Then a three-step method is carried out. At the end, neighboring nodes establish secure links with shared keys with an acceptable probability. More detailed explanation of the *basic scheme* will be given in Section 2.2. Later Ergun [15] proposed an improvement for the last (third) phase of the *basic scheme*. This improvement will be explained in Section 2.2.2.

In this thesis, we implement the *basic scheme* in real sensor nodes called TelosB in Figure 8. To best of our knowledge, there are no real node implementation of this basic scheme in the literature. Thus there are no processing timing, memory usage or code space analysis about the performance of the scheme in real nodes. Same thing can be said for Ergun's improvement. Thus we implement Ergun's scheme on real nodes as well. Our implementation includes both node side and user interface side developments. We also comparatively analyze memory usage and processing times of these schemes.

2 Background Information

In this section, we give some background information about cryptography and key distribution in wireless sensor networks.

2.1 Background Information on Cryptography

Cryptography is the science of secret writing. There are currently two main approaches in cryptography. These are symmetric cryptography and asymmetric cryptography. In symmetric cryptography, each party uses the same key for encryption/decryption purposes. In asymmetric cryptography, each party has a key pair. In our implementation we use symmetric cryptography.

Symmetric cryptosystems are preferred in sensor networks because its encryption/decryption speed is much faster than asymmetric cryptography. In this thesis, we use AES (Advanced Encryption Standart) [6] as the symmetric cryptosystem.

2.1.1 Symmetric Cryptosystems

In symmetric cryptosystems, the same key is used for both encryption and decryption purposes. In this scheme, both parties share the same secret or cryptographic key as in Figure 2 and with the help of this key, they create a secure link between each other. The most common symmetric cryptosystems are AES [6], Blowfish [7] and 3DES [8].

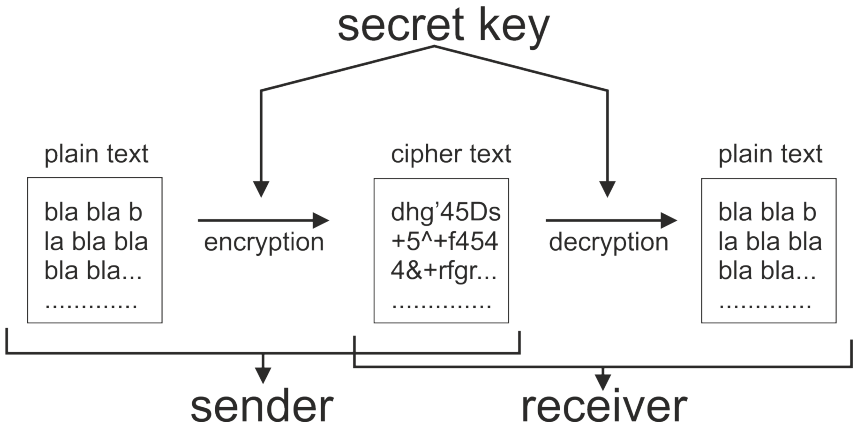

Figure 2: Symmetric Cryptography

2.1.2 AES

AES [6] stands for Advanced Encryption Standart. AES is a standart symmetric cryptosystem. Actually it is the name of the standart only. Rijndael is selected as the standart algorithm by NIST (National Institute of Standarts and Technology of US) in 2001. In 1998, NIST selected 15 candidates for AES. In 1999 five algorithms were chosen to be extensively analyzed. These are MARS from IBM, RC6 by RSA, Rijndael by two Belgian developers Joan Daemen and Vincent Rijmen, Serpent by Ross Andersen, Eli Biham and Lars Knudsen, Twofish by Bruce Schneier. These five algorithms are tested for speed and reliability, encryption/decryption speed, key and algorithm setup time and resistance to attacks. At the end Rijndael was selected in 2001 as AES.

AES has a fixed block size of 128 bits and also has a key size of 128, 192 or 256 bits. As AES is symmetric, it uses the same key for both encryption and decryption. Currently, AES is known to be resistent against cryptanalytic attacks [9].

2.2 Key Distribution for Sensor Networks

Sensor nodes have limited memory, limited processing power and limited battery. Establishing secure communication between the nodes is very difficult because of these limited resources. For these reasons, the most suitable cryptosystem for these nodes is symmetric cryptosystem. Symmetric cryptography requires the nodes to be loaded with pairwise keys for their neighboring nodes. There are several key distribution schemes proposed [10][11][12][13][14]. Among these, Eschenauer and Gligor [5]'s basic scheme in the most important one. Basic scheme has three phases. Later Ergun [15] proposed another phase, called *Key Transfer*, to be used as an alternative to the third phase of the basic scheme.

2.2.1 Eschenauer and Gligor's Basic Scheme

In 2002, Eschenauer and Gligor [5] proposed a key distribution scheme for wireless sensor networks. In this scheme, keys are randomly distributed to each node before deployment. After that, the nodes are deployed over the field. Then the nodes try to establish secure communication links between each other.

Basic scheme has three phases. First phase is *Key Predistribution*. In this phase a

large key pool of random keys are created. In this pool, keys are stored with their index values. From this key pool, a random subset size of *key chain size* is selected and loaded into each node before deployment. *Key Predistribution* phase is shown in Figure 3.

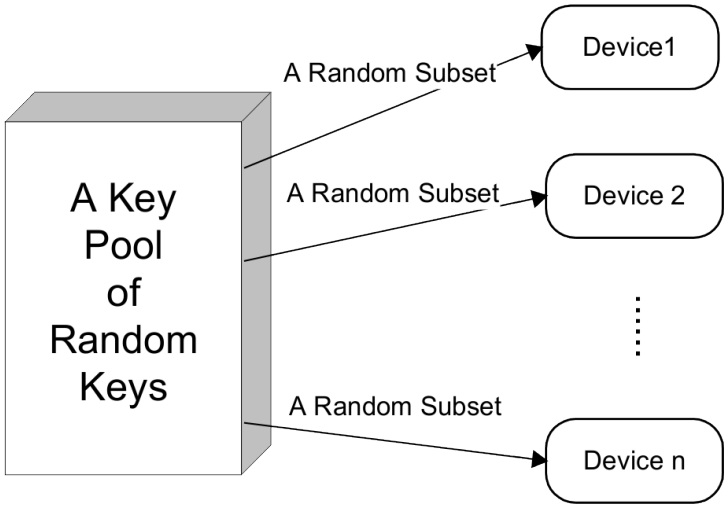

Figure 3: Key Predistribution Phase

Second phase is *Shared Key Discovery*. Each node has now a small subset (which is called *key chain* from now on) of the key pool and will start to communicate with neighbors (the nodes within the communication range) to find out a shared key with each neighbor. Communication occurs in two steps. In the first step, all nodes broadcast their key indices as in Figure 4. In the second step, other nodes receiving those index values check whether they have this index in their own key chains or not. Since all the nodes have a subset from the same key pool, they will share a key with a certain probability.

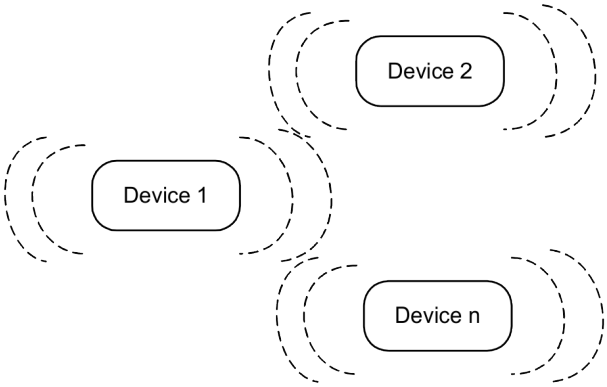

Figure 4: Devices are broadcasting their key indices

At the end of *Shared Key Discovery*, the status of the sensor network is a semi-

connected graph, which is called *Key Sharing Graph*. An example of *Key Sharing Graph* is shown in Figure 5. In this graph, each black dot represents a node in sensor network. Each line represents two nodes' sharing a pairwise secret key and a dotted line represents two nodes' not sharing a pairwise secret key even though they are neighbors.

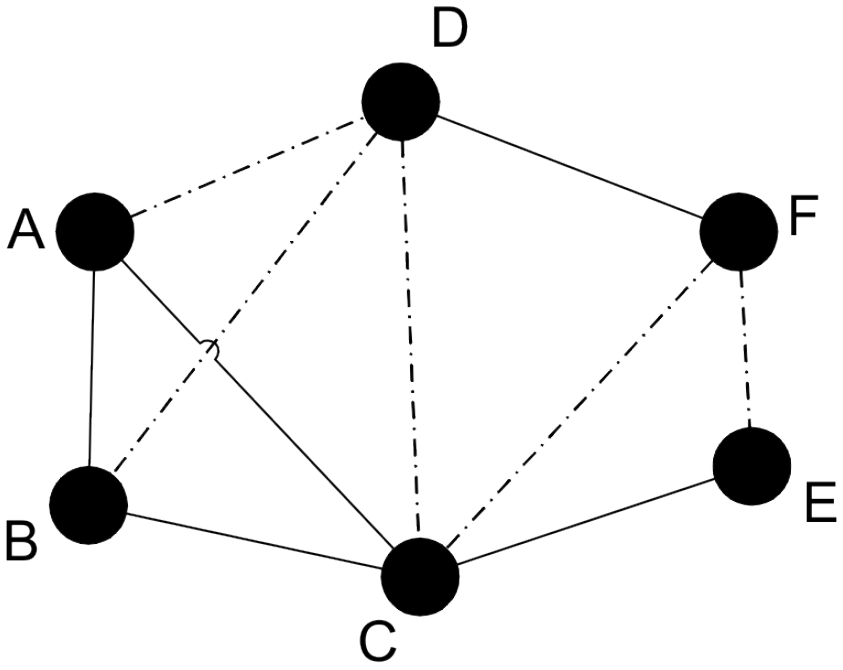

Figure 5: An example of Key Sharing Graph

After *Shared Key Discovery* finished, the third and the last phase, *Path-key Establishment*, begins. Some nodes may not have a shared key with their neighbors at the end of *Shared Key Discovery*. In *Path-key Establishment* phase, each node sends a request to its secure neighbors asking for help to establish a secure link for each of its unsecure neighbors. If the secure neighbor also has a secure link with the same unsecure neighbor, it creates a random path-key and sends it to both ends in two encrypted messages. In this way, insecure neighbors establish a secure link.

An example of *Path-key Establishment* is shown in Figure 6. In this figure, device 1 and 2 are two neighbors that do not have a common key in their key chains. On the other hand, device 3 has a secure link with both device 1 and 2. In *Path-key Establishment* phase, device 3 generates a random path-key and sends it encrypted to both device 1 and 2.

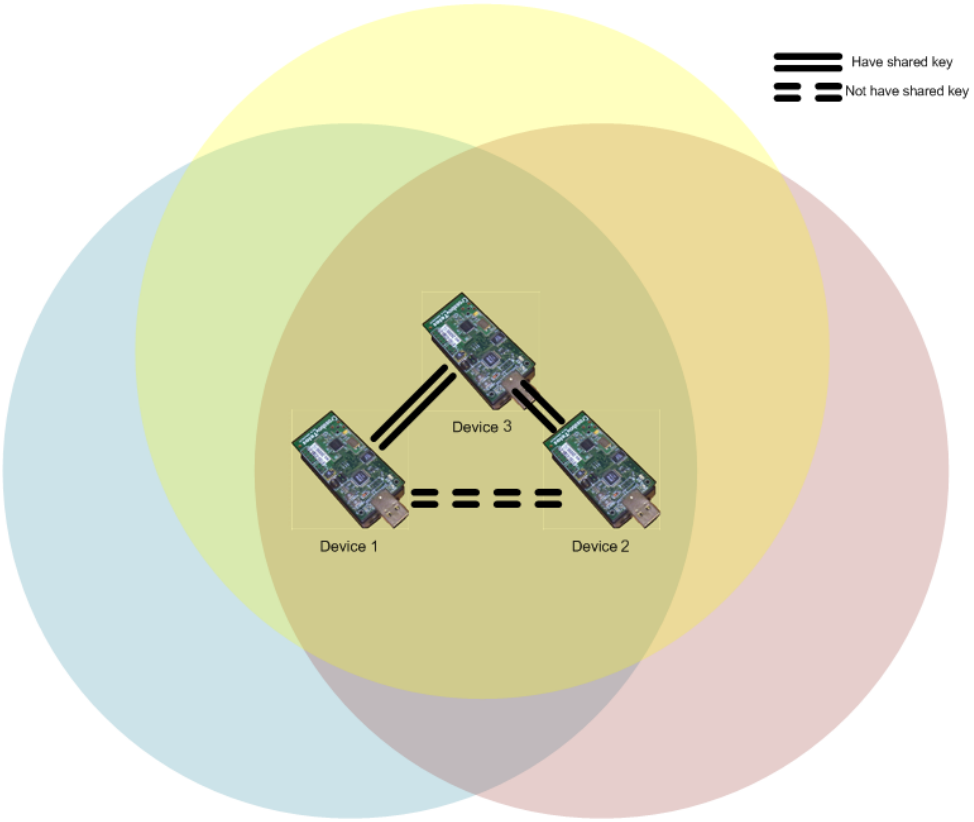

Figure 6: An example case for Path-key Establishment

The complexity of the *Path-key Establishment* per node is calculated as follows. Each node searches all neighbors to understand whether they share a key or not. To do so, for each neighbor, it searches the memory area for the keys that it shares with the secure neighbors. Thus, the complexity is:

$$\mathcal{O}(n \cdot s) \tag{1}$$

where n is the total number of neighbors of a node and s is the number of secure neighbors.

Since

$$n = s + u \tag{2}$$

where u is the total number of unsecure neighbors, we can write this complexity as:

$$\mathcal{O}(n \cdot (n - u)) = \mathcal{O}(n^2 - n \cdot u) \tag{3}$$

Eschenauer and Gligor [5] only proposed an abstract scheme for key distribution. Implementations of the scheme may differ according to encryption algorithm, key size, path-key design and so on.

The performance of the key distribution schemes are analyzed using connectivity and resiliency metrics as described below.

- **Local Connectivity:** The probability that any two neighboring sensor nodes having a common key in their key chains with which they can establish a secure link. Local connectivity of the network can be calculated using the following equation in [5].

$$p' = 1 - \frac{((P-k)!)^2}{(P-2k)!P!} \tag{4}$$

  Where *p'* is *local connectivity*, P is *key pool size*, k is *key chain size*.

- **Global Connectivity:** Number of nodes in the largest connected subgraph in *Key Sharing Graph* over total number of nodes.

- **Resiliency:** The number of safe links after a number of nodes (so the keys) are captured by the attacker.

2.2.2 Key Transfer Phase as an Alternative to Path-key Establishment

In 2010, Ergun [15] proposed a novel phase that can be used in place of the third phase in basic scheme. In Ergun's scheme, the first two phases of the basic scheme remain the same. A third phase called *Key Transfer* is added instead of *Path-key Establishment*. In this phase after *Shared Key Discovery*, if a node A doesn't have a shared key with a neighbor C, node A makes a search process in the key chains of its secure neighbors and try to find a shared key between itself and node C. After a key is found, node A requests this key from that neighbor to have a secure link to node C.

An example of the *Key Transfer* is shown in Figure 7. In this figure, node A is in communication range of both node B and node C. It shares a secret key with node B but does not share a secret key with node C. Node B is not in the communication range of node C but they are sharing a secret key. In this case, node A has key chains of both node B and node C. It searches node B's key chain to find the shared secret key between

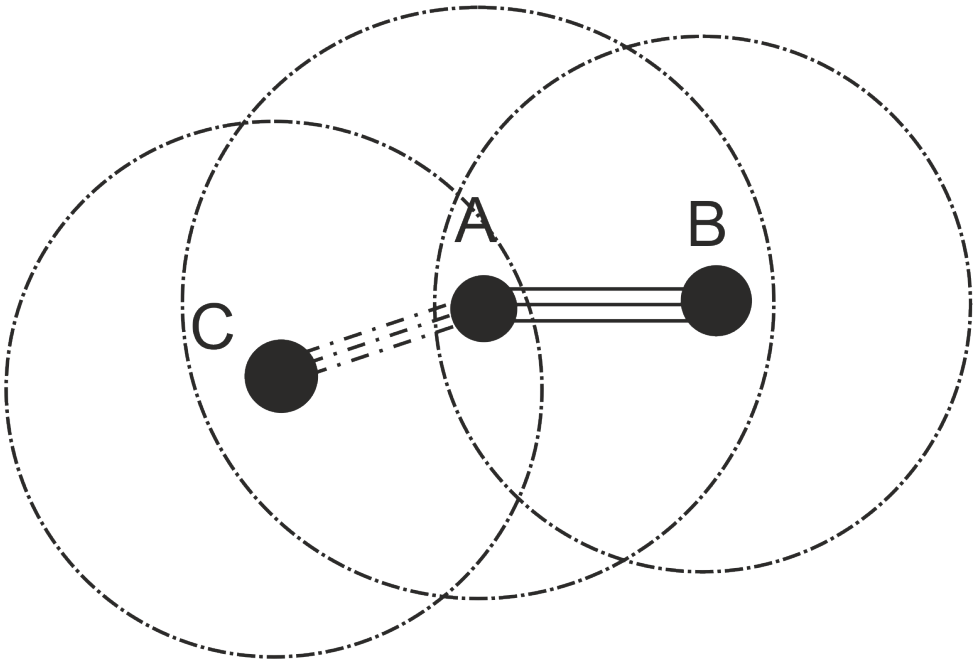

Figure 7: An example case for Key Transfer

node B and node C. After it finds the secret key, it will request that key from node B. In this way, node A will have a secure link with node C.

The complexity of the *Key Transfer* phase per node is calculated as follows. Each node searches the key chains of each unsecure neighbor to find the shared key between unsecure neighbors. Binary search is used in the search process. Thus, the complexity is:

$$\mathcal{O}(u \cdot log(k)) \tag{5}$$

where u is the total number of unsecure neighbors and k is *key chain* size.

2.3 Hardware

In this section the hardware that is used in our thesis is explained in details.

2.3.1 TelosB

We use TelosB [16] sensor devices (motes) from XBow company in our implementations. An example of TelosB is shown in Figure 8.

These devices are open source, low-power wireless sensor devices designed for research community. TinyOS [17] runs on these devices, an open source operating system for small

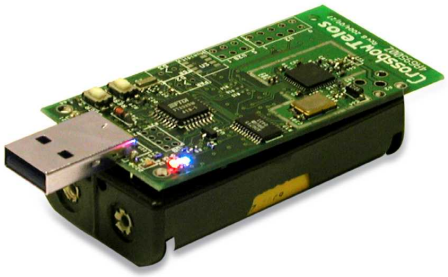

Figure 8: TelosB sensor device with RF interface and battery

sensor devices. These devices can easily be programmed via USB port.

Some information from the datasheet [16] can be found in Table 1:

Table 1: The specification of our sensor device: TelosB

IEEE 802.15.4 compliant RF transceiver
2.4 to 2.485 GHz ISM band
250 kbps data rate
Integrated onboard antenna
8 MHz TI MSP430 microcontroller with 10 kB RAM
1 MB external flash for data logging
Programming and data collection via USB
Sensor suite including integrated light, temperature and humidity sensor
Runs TinyOS 1.1.11 or higher

3 Our Design and Implementation

In this section, we give the details of our design and milestones about the implementation including testbed, flowcharts, data structures and control panel, which is designed for controlling the node environment.

First, we give some information about the testbed. Then, the flowchart of each step in the basic scheme is given. Basic scheme implementation has three phases: *i) Key Predistribution, ii) Shared Key Discovery, iii) Path-key Establishment.* Ergun's [15] *Key Transfer* phase has its own flowchart instead of the third phase of basic scheme. After that, data structures in the implementation are given. This part includes structures, linked lists and some important parameters about the implementation. Finally, we introduce the control panel of our system and its features.

3.1 Our Testbed

We have three TelosB devices which can be programmed via USB. Each device is connected to computer while testing the implementation and programmed by using the control panel that will be explained in Section 3.7. One of the devices is selected as gateway sensor node and all commands are broadcasted to the network by using this device. The flow of the commands is shown in Figure 9. A picture of our testbed is shown in Figure 10.

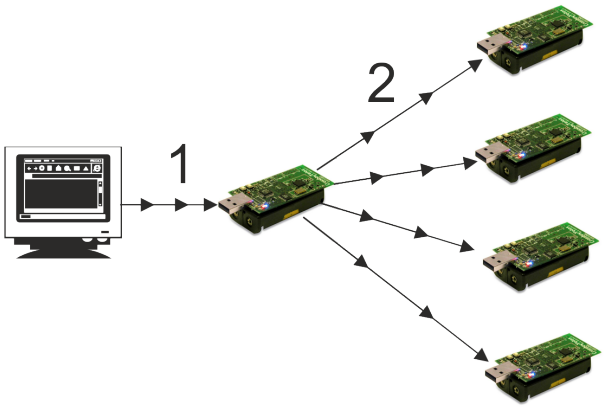

Figure 9: The flow of commands

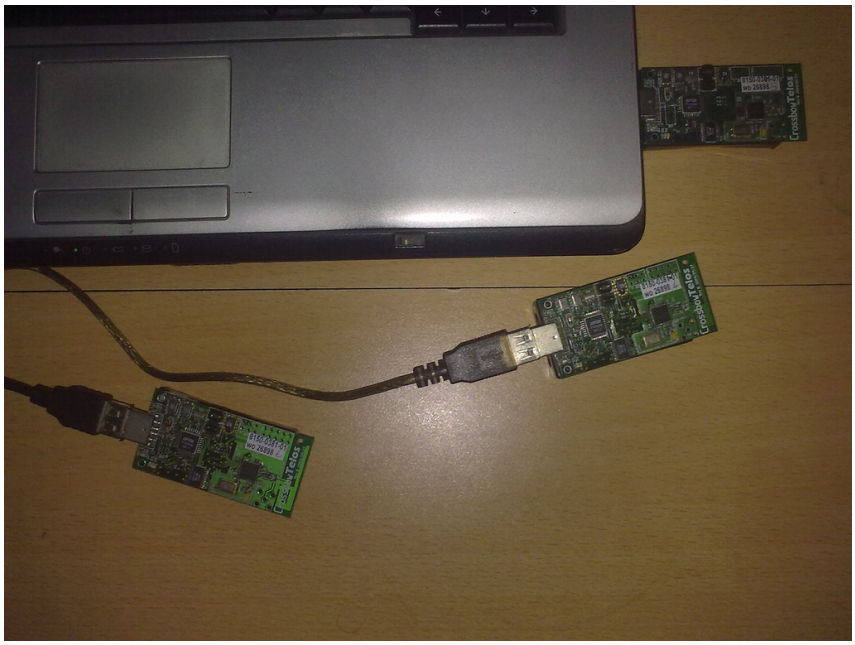

Figure 10: A snapshot from our testbed

3.2 Design of Key Predistribution Phase

In this phase, a random key pool, whose size is denoted with *key pool size*, is created using the control panel. The random keys are generated by Java's SecureRandom class. The keys are stored in a text file with their indices and written as 16 short integers each 8-bit.

Then a small subset of these keys, which is the *key chain*, are sent to the nodes. The size of the *key chain* is defined as *key chain size*. A packet which consists of the index and the key is generated and sent to node one by one. An example is shown in Figure 11.

As soon as the node receives this packet, it hashes the index value to calculate the appropriate memory address to save that key. The address is calculated and the key is saved to memory with its index. The keys are not sequentially saved to memory. They are stored as a linked list. The first incoming key's address is the head of the linked list and each key is connected with a next address field at the end of the entry. An illustration of linked list will be shown in Figure 24 of Section 3.6.

After all keys are sent and saved to memory, *Key Predistribution* phase is finished. Flowchart of this phase is shown in Figure 12.

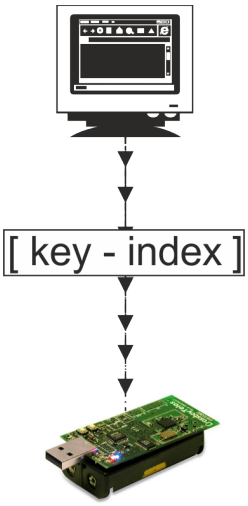

Figure 11: Key packets are being sent to nodes

3.3 Design of Shared Key Discovery Phase

In this phase, each node tries to find which keys are common with its neighboring nodes. This is done in two steps. First, the nodes broadcast the indices of the *key chain* in a packet and neighboring nodes receive and save them. Then, each node compares the received index values with the indices of its *key chain*.

In the first step, each node reads its index values from the memory one by one and it puts these indices in a packet. Each packet has a configurable size. In a packet more than one index values are sent. After a packet is filled with index values, it is broadcasted. The packet is shown in Figure 13. Flowchart of this step is given in Figure 14.

Other nodes receive the broadcast packets and they add the index values to an array with the related node numbers. Moreover, each device holds a neighbor list which is used in other phases (*Path-key Establishment* and *Key Transfer*). In this neighbor list, each device is initially marked as "not have shared key". The process continues for all the received packets and for all index values in those packets. At the end of this step, we have an array of index values with related node numbers and a neighbor list. The flowchart of reception is shown in Figure 15.

The second is performed to compare the received index values with the index values of the node and to identify which keys are in common. In this step, each entry in the array of index values of other nodes will be processed. Each index is read from the array and

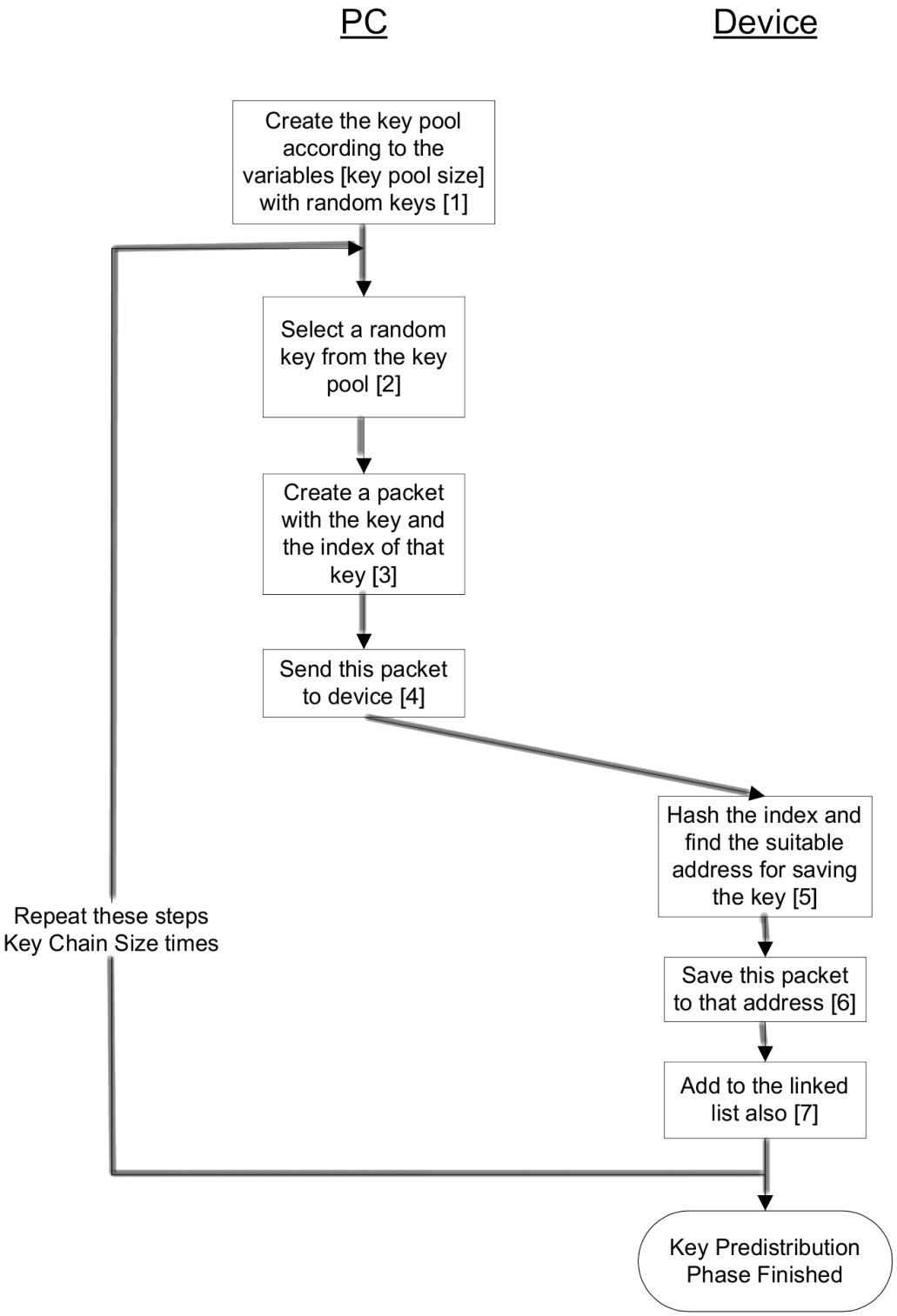

Figure 12: Flowchart of Key Predistribution Phase

hashed to calculate the necessary memory address. After the address is calculated, the entry in that address is checked for a match with the current index. If there is a match,

source device	index 1	index 2	.....	index 10

Figure 13: Index packet that is broadcasted to other nodes

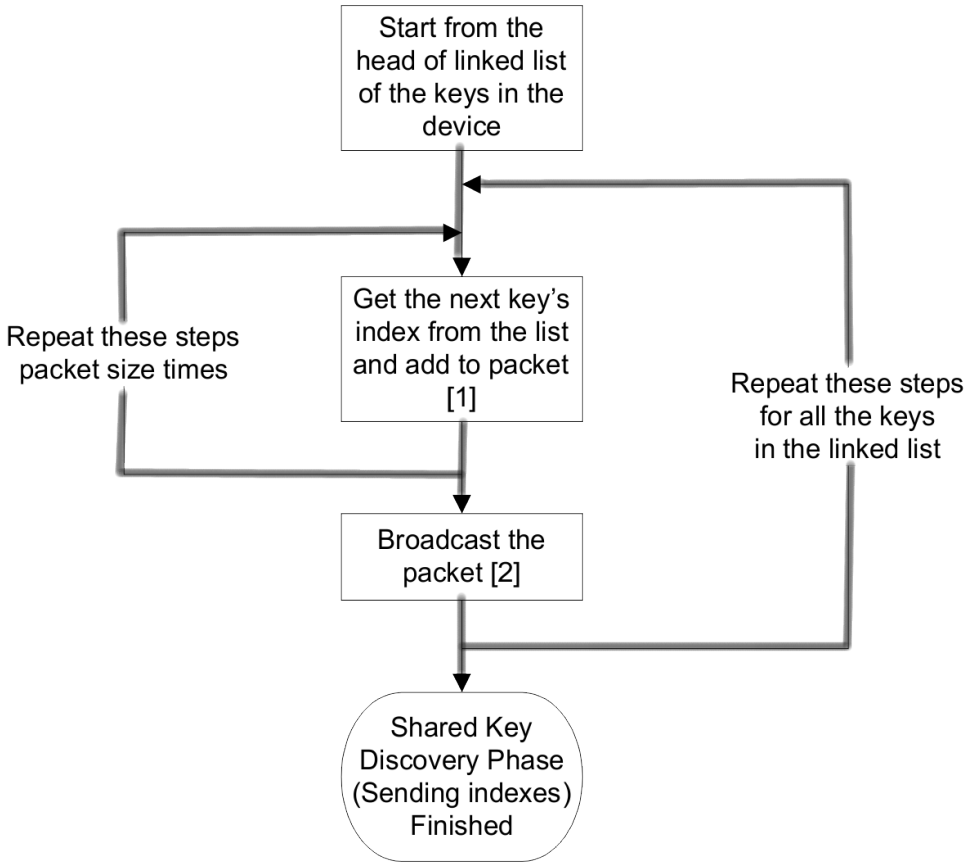

Figure 14: Flowchart of Shared Key Discovery Phase (Sending Indices)

the index will be saved in another special memory location with the node number and the node is marked as "has shared key". This special memory location is used to hold the matches of the neighboring nodes and the shared indices with them. The steps are shown in Figure 16.

After the second step, *Shared Key Discovery* phase is finished. Now each node has a list of shared keys between itself and neighboring nodes. Moreover, it also has a list of neighbors and a flag for each neighbor, indicating whether they have a shared key or not.

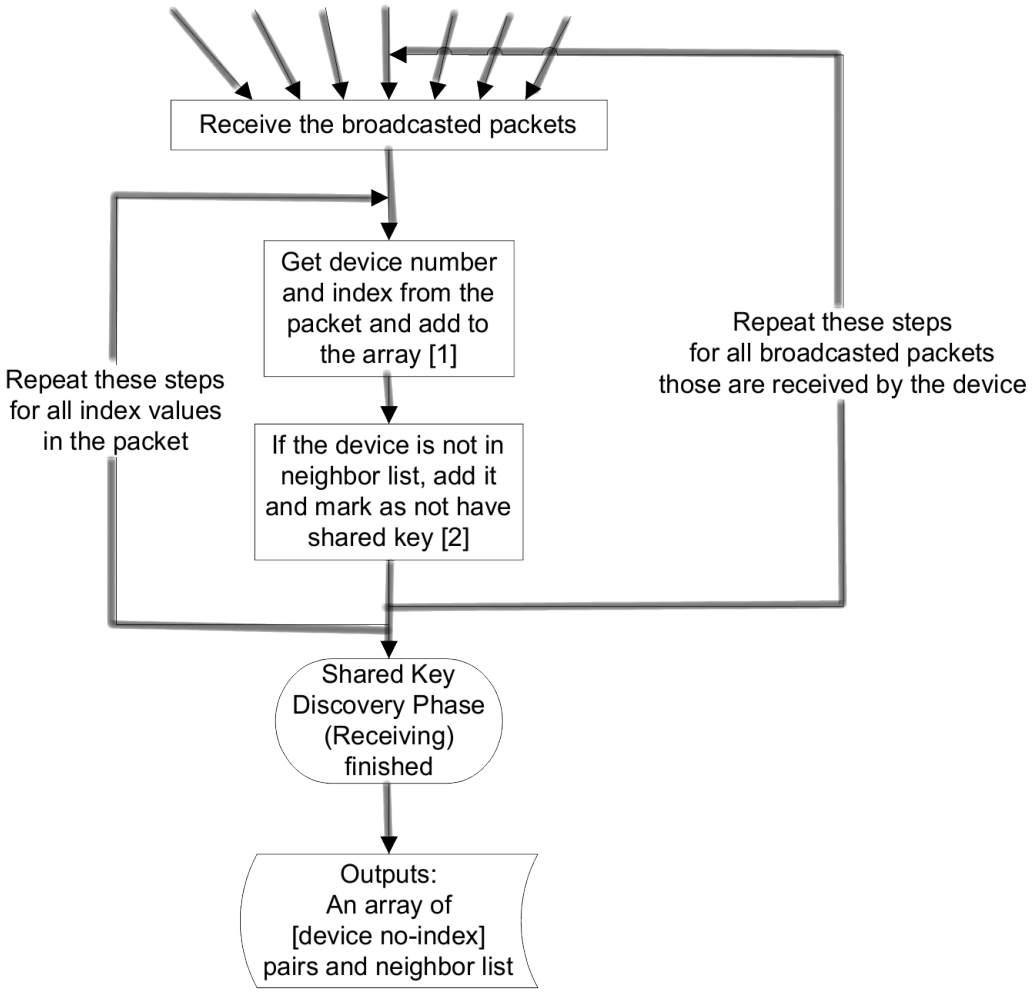

Figure 15: Flowchart of Shared Key Discovery Phase (Reception)

3.4 Design of Path-key Establishment Phase

At the end of first two phases, nodes share keys with some neighbor nodes. However, some neighboring nodes may not share a key and can not communicate securely. This case is depicted in Figure 6. To handle such cases, a third phase called *Path-key Establishment Phase* is employed in basic scheme.

In this phase, a node has a neighbor list and in this list there is a flag for each neighbor indicating whether they are sharing a key or not. The node first scans the neighbor list and find the neighbors with no shared key. Then, for each insecure neighbor found, the node makes a request to its secure neighbors. From now on, our current node will be called as *request sender*. The node that responds to the request of *request sender* will be called as *request handler*. The insecure neighbor will be called as *third device*.

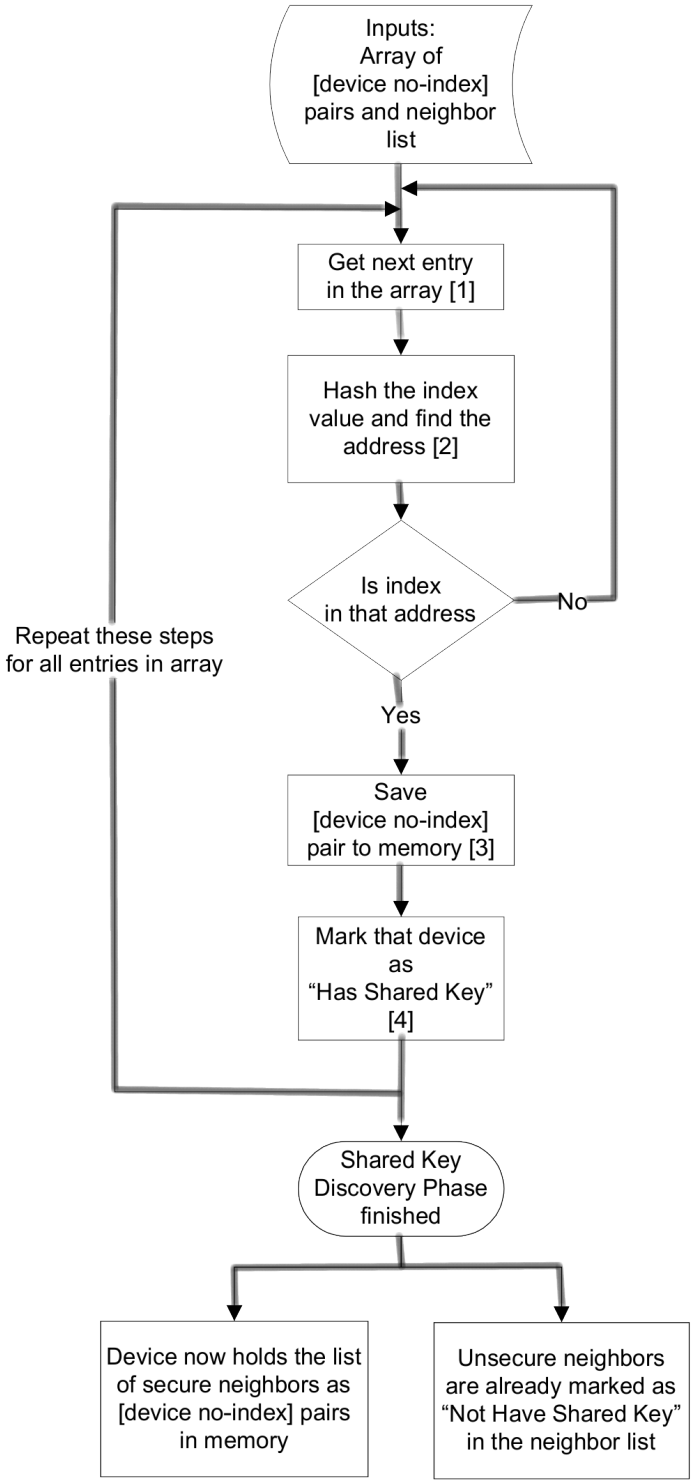

Figure 16: Flowchart of Shared Key Discovery Phase (Check)

In our scenario, all three nodes are neighbors and *request sender* and *third device* do not share a key, but, *request handler* shares keys with both of them.

After *request sender* makes a request for a *third device*, it waits for a configurable

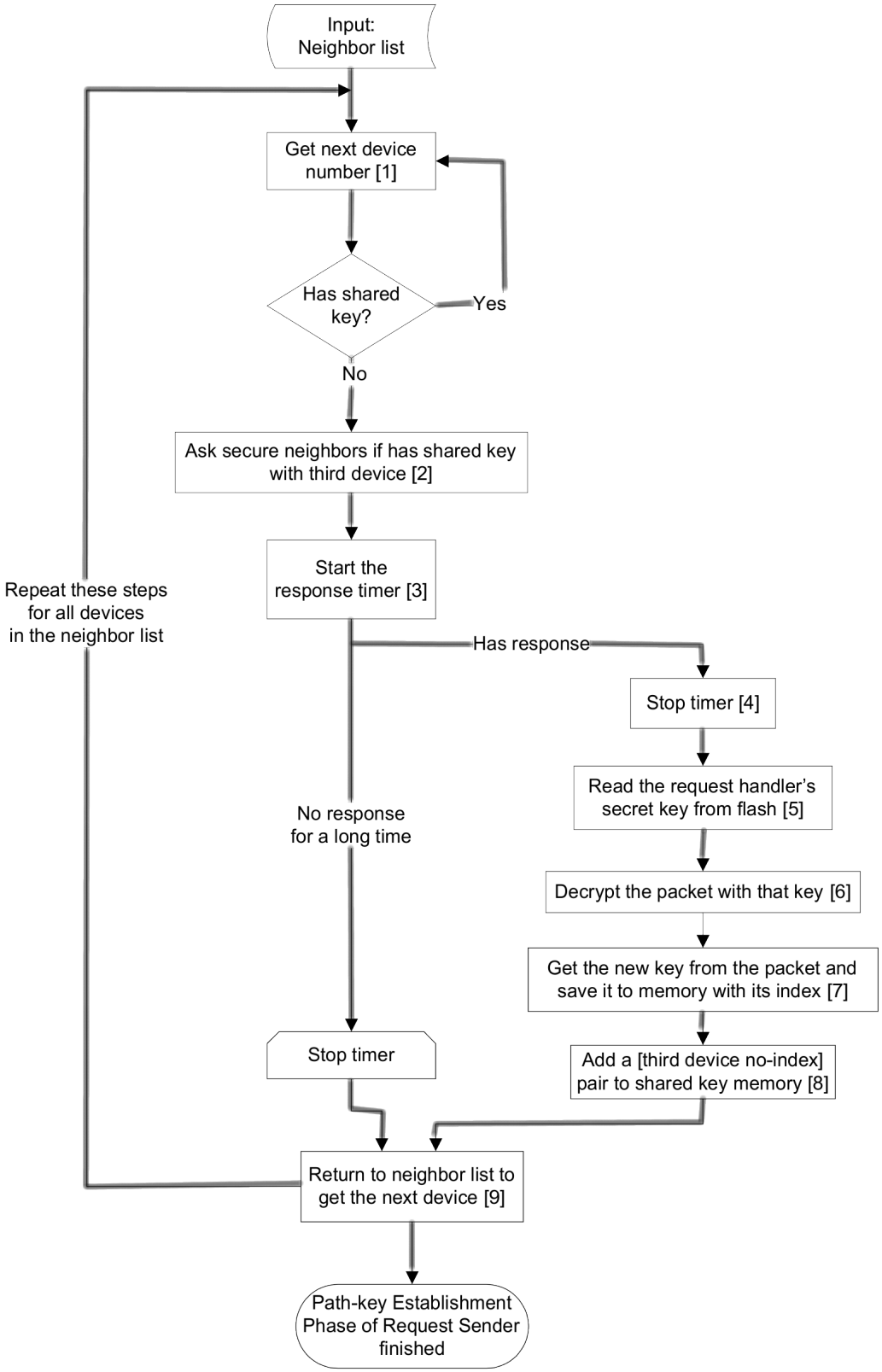

Figure 17: Flowchart of Path-key Establishment Phase (Request Sender)

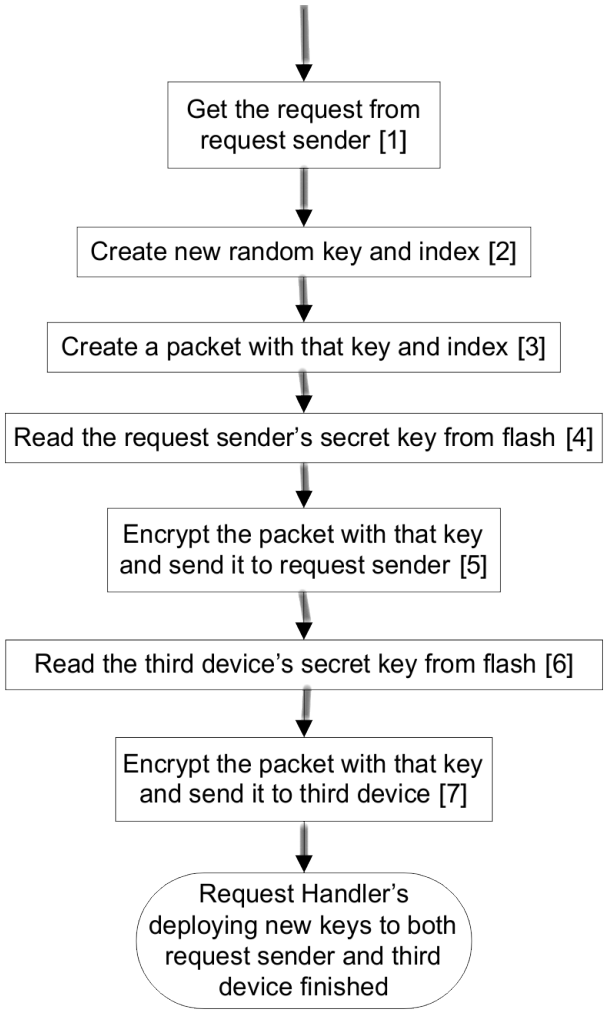

Figure 18: Flowchart of Path-key Establishment Phase (Request Handler)

amount of time for getting a response. If it does not receive a response from any neighbor nodes, it continues with another *third device* and sends a request for this device also. A potential *request handler* first checks whether it has a shared key with the *third device* or not. If it has a shared key with the *third device*, it first generates a random key and a random index. Then it creates a packet as shown in Figure 19 and it sends the packet to both *request sender* and *third device*. Since *request handler* shares keys with both nodes, it reads the secret keys of each node from flash memory and encrypts the packets with these keys before sending. In this way, the transferof the newly created key is secured. When *request sender* receives this packet, first it reads the secret key that is shared with *request handler* to decrypt the packet. After decrypting the packet, it gets the newly created key and its index to be stored in the memory. It also saves this new shared key

within the special area of the memory that stores the shared indices with the nodes. This memory is called as *shared index memory*.

request handler device no	newly created index	newly created key

Figure 19: The packet of newly created key and index

Request handler also sends the packet to *third device* to notify this node about the newly shared key and its index with the *request sender*. After *third device* receives this packet, it first reads the secret key that is shared with *request handler* from the memory to decrypt the packet. After it decrypts the packet, it performs the same process as *request sender*.

Now all three nodes share at least one pairwise secret key in between. The same procedures are continuously performed until each node shares at least one key with its neighbors. The flowcharts of all three nodes, *request sender*, *request handler* and *third device* are shown in Figures 17, 18 and 20, respectively.

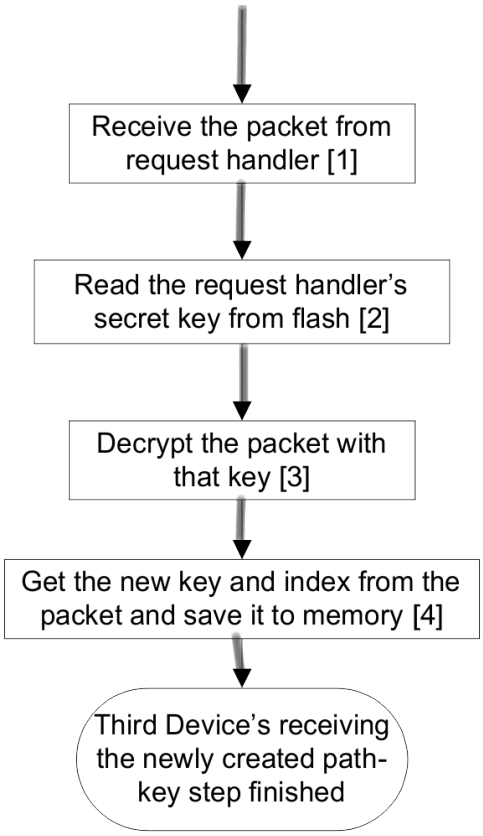

Figure 20: Flowchart of Path-key Establishment Phase (Third Device)

3.5 Design of Key Transfer Phase

Key Transfer phase is an alternative to *Path-key Establishment* phase for creating a secure link between two unsecure neighbors. The same name conventions that are used for *Path-key Establishment* phase are kept for explaining this phase. *Request Sender* is the node which sends key transfer requests. *Request Handler* is the node which sends responses to the *request sender*. *Third Device* is the node, which does not share a key with *request sender*, even though they are neighbors.

At the end of first two phases, *request sender* has a neighbor list and an array of indices from each neighbor in the communication range. *Request sender* scans the neighbor list and finds the unsecure neighbors that they do not share a key. Unsecure neighbor, in our case is *third device*. The *request sender* has already obtained key chains of each neighbor at reception step of *Shared Key Discovery* phase. After *request sender* identifies that it does not have a shared key with *third device*, it searches the indices of *third device* in the index arrays of other secure neighbors. The search process is done using binary search.

The main logic in this phase is that *request handler* and *third device* are already sharing a key. We simply request this key from *request handler*, save it to *request sender* and notify *third device*.

After *request sender* searches the array of other neighbors to find a match with the *third device*, it finds the shared key between *request handler* and *third device*. Then it makes a request to get that key from the *request handler*.

The *request handler* receives the request and it reads the requested key from the flash memory. After it reads the key from the memory, it prepares a packet, encrypts the packet with the key that it shares with the *request sender* and send the packet to *request sender*.

Then the *request sender* receives the packet from the *request handler* and decrypts the packet using the key that it shares with *request handler*. It gets the new key from the packet and save it to memory with its index. It also saves the index of that key to *shared index memory* and notify the *third device* about the match.

Third device receives this notification and adds *request sender* and that index to *shared index memory*. From now on, the *request sender* and the *third device* share a key and establish a secure link. The flowcharts of all three nodes, *request sender*, *request handler*

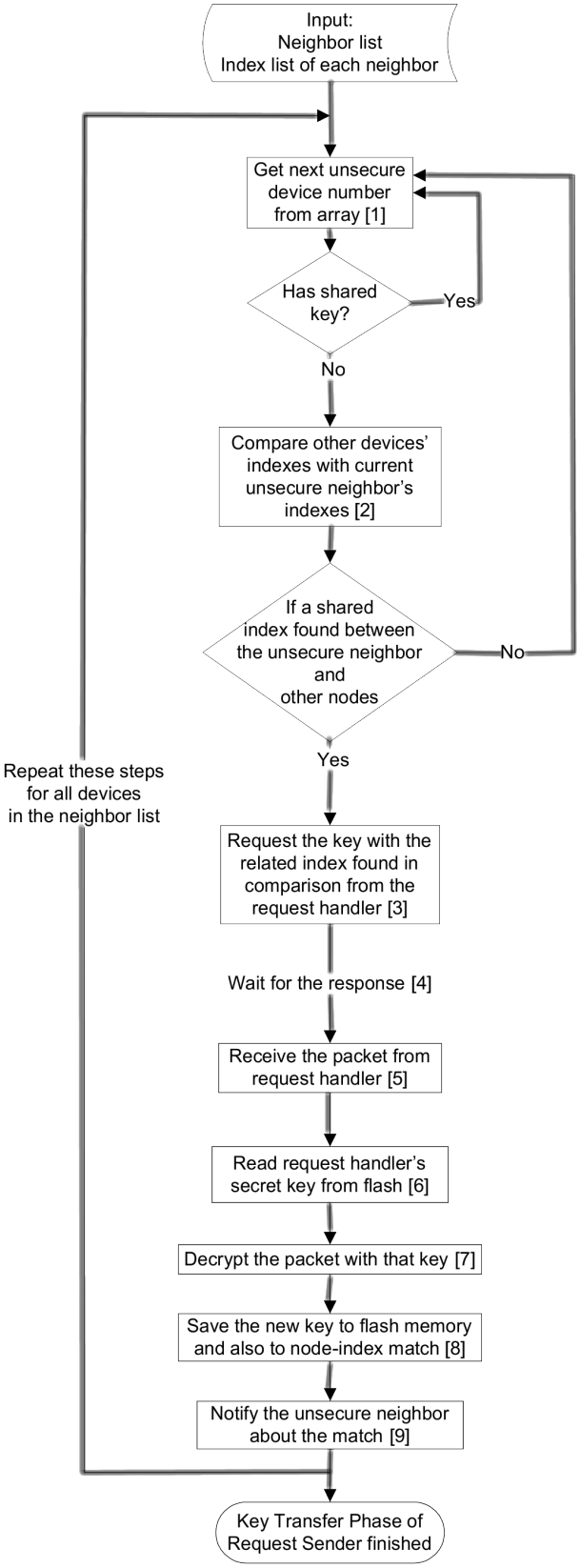

Figure 21: Flowchart of Key Transfer Phase (Request Sender)

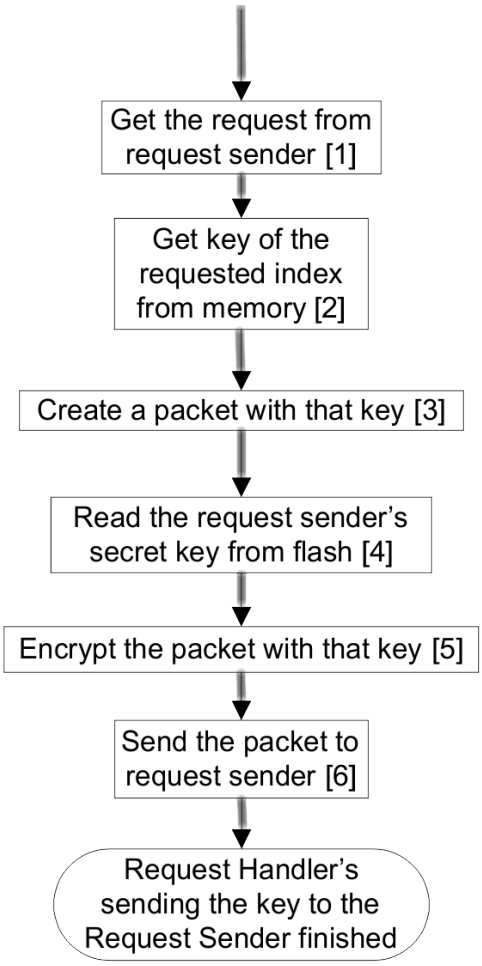

Figure 22: Flowchart of Key Transfer Phase (Request Handler)

and *third device* are shown in Figures 21, 22 and 23, respectively.

3.6 Data Structures Used In The Implementation

There are various data structures we use during implementation. Firstly, we use a linked list to save the keys and indices to flash memory. The linked list structure helped to make the check process in *Shared Key Discovery* phase faster.

The address, in which we save each key and its index, is calculated by using a hash function. Hash function gets index value as input and returns the address to be saved. It is basically doing a modulo operation according to the slots in flash memory. With hash function, checking whether an index is shared or not is done O(1) time.

They key packets are stored as a linked list in flash memory. This is illustrated in Figure 24. Packet 1 is received from the computer and saved to flash as the head of the

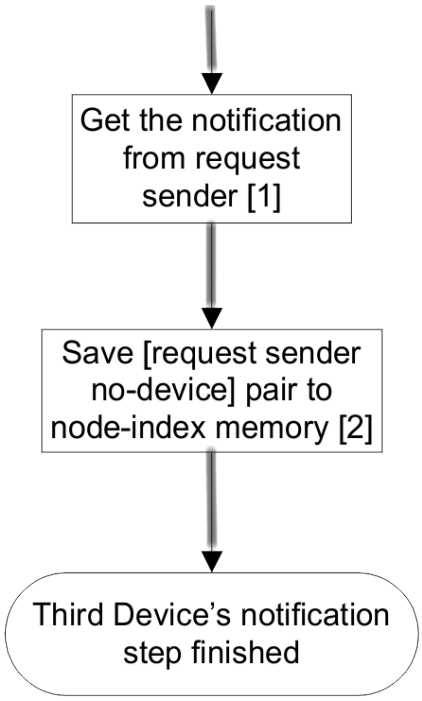

Figure 23: Flowchart of Key Transfer Phase (Third Device)

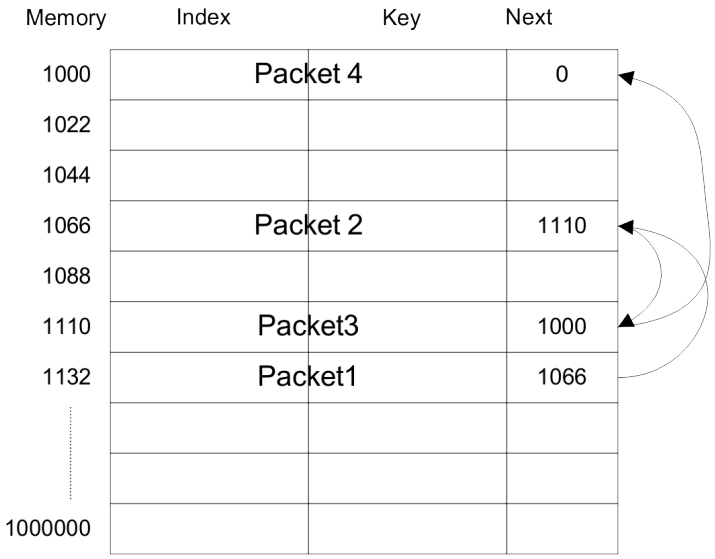

Figure 24: Sample flash memory after keys are loaded

linked list. Then, packet 2 is received and added to the linked list as second element. At the same time, the next address part of the first entry is modified to show the second entry's address. Moreover, when other key packets are received, they are added to the linked list one by one. The last element of the linked list has 0 in its next address part.

We also employ a neighbor list to keep track of which nodes are in communication

neighbor	has shared key
6	true
8	false
13	true
23	true

Figure 25: Neighbor list to keep track of nodes in communication range

range within the current node and a flag to determine whether they are sharing a key or not. This list is filled in *Shared Key Discovery* phase in Section 3.3. An example is depicted in Figure 25.

3.7 The Features of the Control Panel

In order to control the phases and the nodes, we develop a control panel that is written in Java language. This control panel supports some functionalities using TinyOS [17] libraries.

Control panel has basicly four parts: Key Pool, Connection, Mote Control, Basic Scheme. A screenshot can be seen in Figure 26.

Key Pool: This part is used to set variables (key pool size, key chain size), to create the key pool and to write to a text file. An example can be seen in Figure 27. In this screenshot first column is the index of the key and the rest is 128 bit keys written as 16 short integer blocks.

Connection: Each device is connected to the computer through a unique COM port. In this part, the connection parameters are set and connections are made. Parameters are the connection type (serial), COM port number and the type of the device (in our case *telosb*).

Mote Control: This part controls the mote. It includes formatting the flash memory of the node, reading a specific memory location, sending the randomly selected keys to nodes, reading from memory within a specific range. We can find which keys

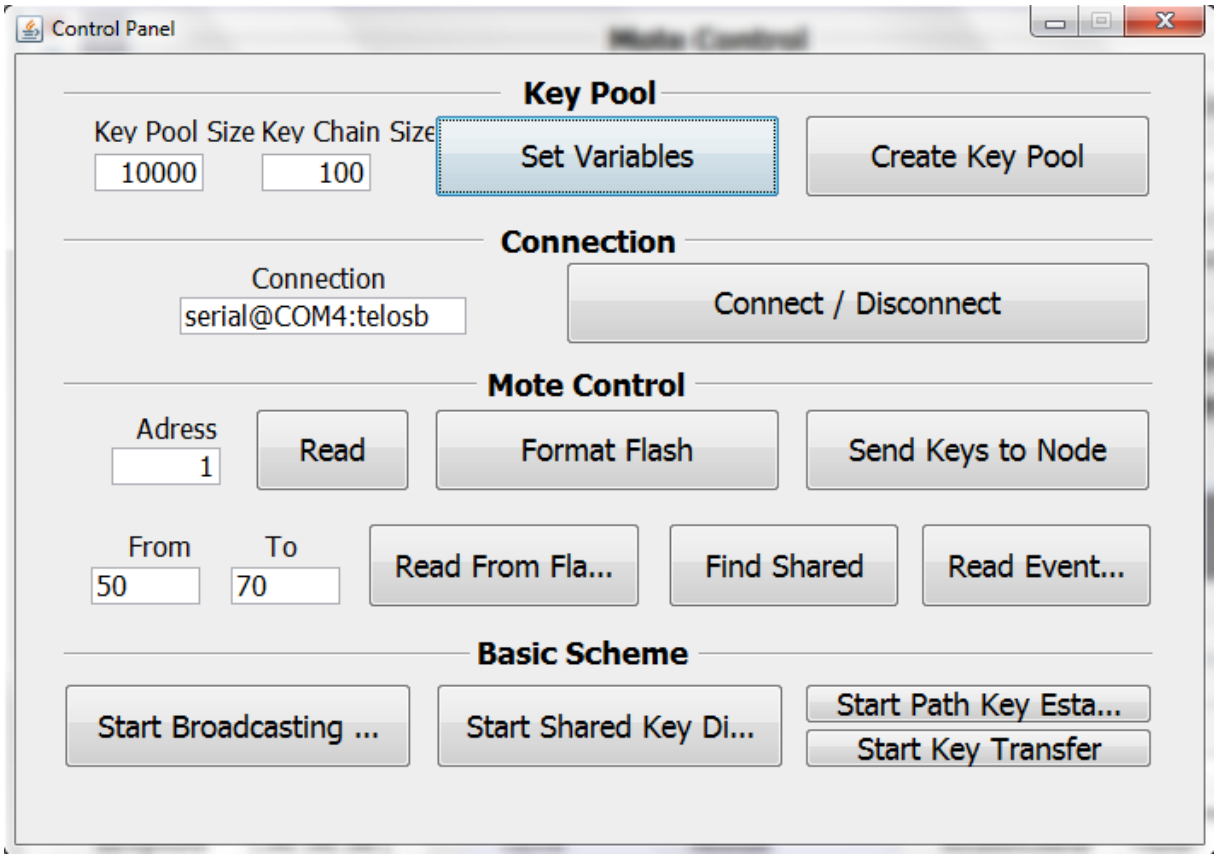

Figure 26: A screenshot of the Control Panel

```
3 106 -127 -106 -100 -47 91 38 35 -44 -117 -112 -118 64 -36 -38 -118
4 98 -96 104 -69 51 -43 21 -71 -7 114 96 78 53 -79 59 -95
5 116 -112 -14 -33 5 -104 -98 38 -36 -42 -82 -83 102 -39 -96 82
6 120 123 -29 -109 -16 -9 125 -104 -110 23 90 -19 -22 -110 17 -96
7 126 76 -95 -56 -121 -49 99 -11 38 9 -73 -80 -22 -124 -52 4
8 -44 76 83 55 -31 0 -43 55 -105 -38 -16 66 122 57 110 -18
9 -18 65 -25 20 95 -123 55 -126 -86 -13 -124 -20 10 92 -28 51
10 110 -28 86 25 -81 -51 107 -56 83 59 45 90 -42 -10 33 70
11 -11 -14 62 6 -59 25 65 -40 55 -16 41 -61 -73 -57 -121 -92
```

Figure 27: An example from the keys in the key pool

are shared between which nodes. Also event log, which is a necessary structure to calculate time between events, is read from the RAM.

Basic Scheme: This part includes the start commands of each phase. First two phases are common and *Key Transfer* phase is an alternative to the *Path-key Establishment* phase.

4 Performance Evaluation

In this chapter, we give the performance metrics and performance results of our implementation. We use two scenarios for performance evaluation. We use different values for the parameters *key pool size*, which defines the size of the randomly created keys, and *key chain size*, which defines the size of the subset that is selected for deploying to each node. These parameters define the local connectivity of the network [5]. Way of calculating local connectivity is explained in Section 2.2.1. The details of two scenarios are given in Table 2.

Table 2: The details of our scenarios

	Scenario 1	Scenario 2
Key Pool Size (P)	15000	60000
Key Chain Size (k)	100	200
Local Connectivity (p')	0,489	0,488

We have three devices in our testbed. These parameters are set to satisfy the case that node A shares a secret key with both node B and node C but node B and node C do not share key in between. In this way, *Path-key Establishment* and *Key Transfer* phases perform.

For the sake of accuracy, each scenario is run 10 times and the average values are reported.

4.1 Performance Metrics

We use three performance metrics. First one is *code space*. *Code space* is the memory used for storing binaries after compile. We give the *code space* for both of the scenarios.

Second metric is *processing time*. The processing time values of *Key Predistribution*, *Shared Key Discovery*, *Path-key Establishment* and *Key Transfer* phases will be given seperately. Moreover, the processing times of the substeps of these phases are also detailed. Processing time measurements are performed in milliseconds.

Third metric is *memory usage* for the keys and their indices. The predistributed keys must be kept in memory for communicating securely with the neighbors. For this purpose, our device has 1 MB flash memory. The usage ratios of this flash memory are given for

both scenarios.

4.2 Analysis of Scenario 1

In scenario 1, the key chain size of 100 keys and a key pool size of 15000 keys. This corresponds to a local connectivity value of 0,489.

Firstly, our testbed is in stable mode as shown in Figure 10. The keys are predistributed to each node. Then, "Start Sending Packets" command is given to the network. The nodes turn on the the blue leds while sending and receiving the key packets as shown in Figure 28.

After this step, reception of the index packets end and the yellow led of each node is turned on as shown in Figure 29.

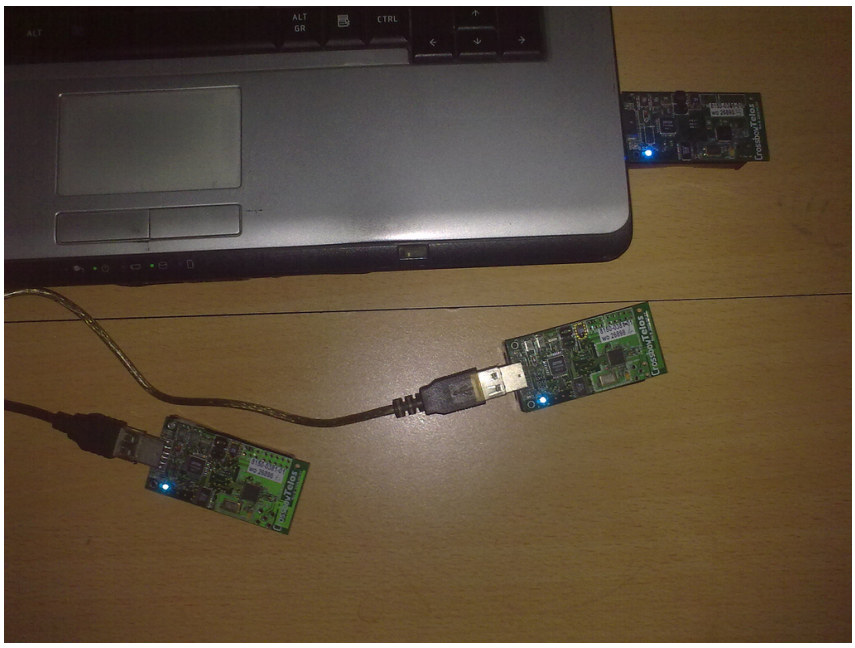

Figure 28: The nodes are sending/receiving index packets (blue led on)

Then, indices are checked by sending "Start Shared Key Discovery" command to the network and the red led of each node is turned on as shown in Figure 30. At this point, one of the two phases, *Path-key Establishment* or *Key Transfer* can be selected as a third phase. After all phases are finished, the nodes turn into stable mode again and turn off all leds as in Figure 10.

Code space for Scenario 1 is 29674 bytes. *Key chain size* and *key pool size* are not

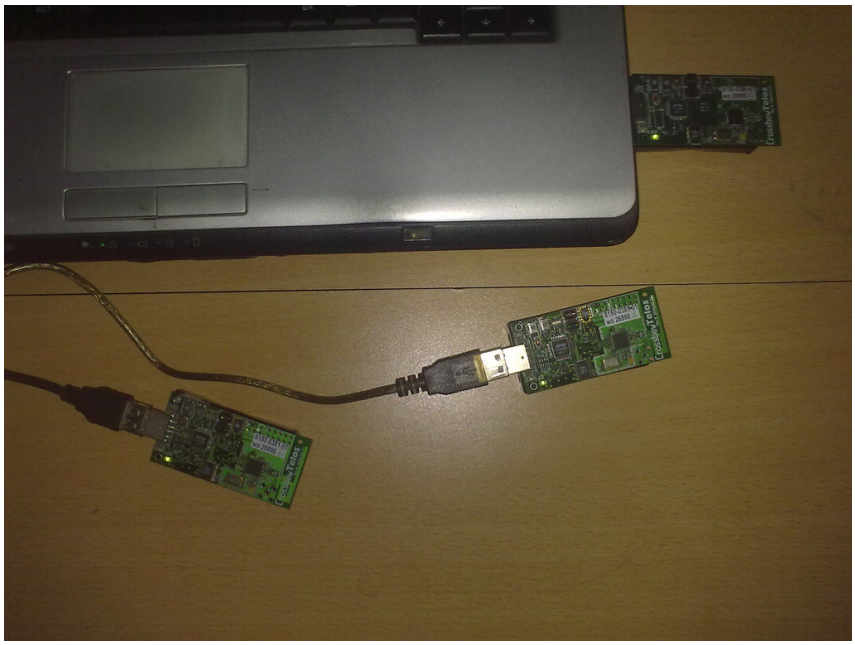

Figure 29: The nodes finish reception (yellow led on)

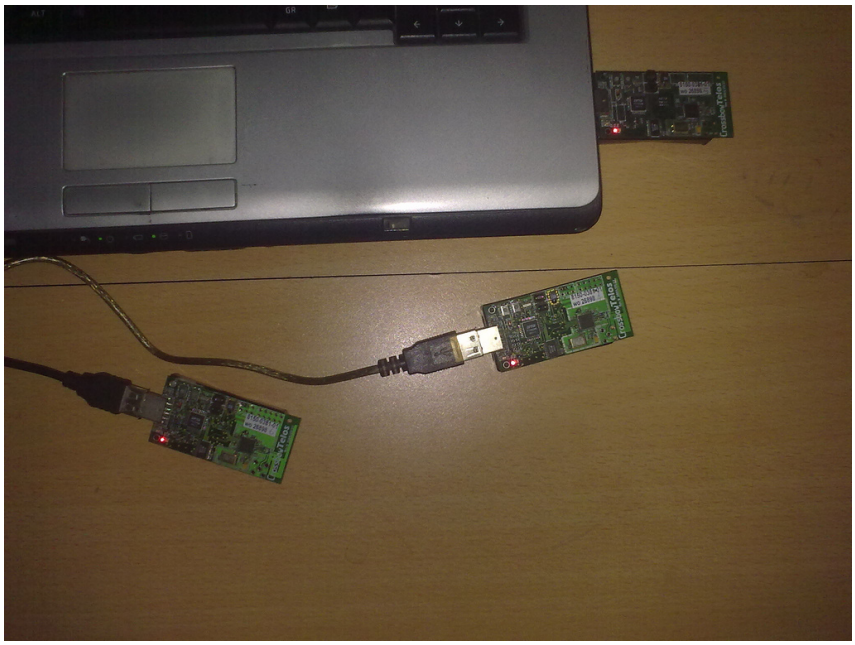

Figure 30: Shared Key Discovery is finished (red led on)

considered in the code space metrics. They will be considered in memory usage metric later.

The measured processing times for the steps in each phases are given in Table 3. In party column, the figure numbers are included to refer to the corresponding flowchart.

In the rows of related columns, the numbers in brackets are the step numbers in the corresponding flowchart figure. Number of runs column indicates how many times that step runs in the node.

The processing time of the *Key Predistribution* phase is 2064 ms. In this phase, the keys are sent to the nodes one by one via computer. A node receives a key packet and save it to memory in 20,64 ms. This step is repeated *key chain size* times, which is 100 in this scenario.

In *Shared Key Discovery* phase, each node has an array of other neighbors' indices and it compares these values with its own indices. A comparison operation is a hash function calculation and a read operation from the memory that take a total of 1,84 ms. This step is repeated for all entries in the index array, which is neighbor count times *key chain size*. In our case, this value is 2 x 100 = 200. Thus, this phase takes 369,7 ms.

In *Path-key Establishment* phase, there are three parties, which were mentioned earlier in Section 3.4. The total processing time of this phase is 201,7 ms. Reading a secret key of a node from the memory is an important step of this phase. In *request sender*, this takes 5,1 ms. In *request handler*, it takes 6,7 and 7,3 ms. In *third device*, it takes 5,1 ms. This means that reading a key from the memory takes 6,05 ms on the average. Another important processing time measurement is encryption and decryption times per packet. From Table 3, we can see that on the average these two operations take 2 ms each.

In *Key Transfer* phase, we almost obtain the same results for reading a key from memory and encryption/decryption. The most important step of *Key Transfer* phase is the search function. The search function takes 102,23 ms for the node. This step is repeated the number of insecure neighbor times; this is 1 in our testbed. The total processing time of this phase is 168,73 ms.

The most important result that we obtain from the total results is that *Key Transfer* phase is faster than the *Path-key Establishment* phase. This is not a surprising result, because *Key Transfer* phase reduces the processing times in the *third device* and *request handler*.

We also analyzed the memory requirements for the key and index storage in the nodes. Index is 2 bytes, key is 16 bytes and next address field is 4 bytes as shown in Figure 24. That means one entry is 22 bytes. Since the *key chain size* in this scenario is 100, total

Table 3: The details of scenario 1

Phase	Processing party / Corresponding flowchart figure numbers	Steps / Step numbers in the corresponding flowchart	Processing time for one run	Number of runs	Total processing time	Phase Total	
Key Predistribution	All nodes / (Figure 12)	Reading key packet from computer and saving to memory (steps 5,6,7)	20,64	100	2.064,00	2.064,00	
Shared Key Discovery	All nodes / (Figure 16)	Get next index and check memory (steps 1,2)	1,84	Neighbor x 100	369,70	369,70	
Path-key Establishment	Request Sender (Figure 17)	Get next neighbor and check if has a shared key (step 1)	52,70	Neighbor	105,50	166,20	201,70
		Path-key request is sent and it is being handled	47,20	1	47,20		
		Get request handler's secret key from flash (step 5)	5,10	1	5,10		
		Decrypt the packet with request handler's key (step 6)	2,20	1	2,20		
		Add the new key to memory (steps 7,8)	6,20	1	6,20		
	Request Handler (Figure 18)	Search memory if has a shared key with requested node	4,40	1	4,40	21,70	
		Get sender's secret key from flash (step 4)	6,20	1	6,20		
		Encrypt packet with sender's key (step 5)	2,00	1	2,00		
		Get third node's secret key from flash (step 6)	7,30	1	7,30		
		Encrypt packet with third node's secret key (step 7)	1,80	1	1,80		
	Third Device (Figure 20)	Get request handler's secret key from flash (step 2)	5,10	1	5,10	13,80	
		Decrypt packet with request handler's secret key (step 3)	2,40	1	2,40		
		Add the new key to memory (step 4)	6,30	1	6,30		
Key Transfer	Request Sender (Figure 21)	Search array and find a match (step 2)	102,23	Insecure Neighbor	102,23	154,49	168,73
		Key transfer request is sent and it is being handled (steps 3,4)	38,14	1	38,14		
		Get request handler's secret key from flash (step 6)	5,57	1	5,57		
		Decrypt the packet with request handler's key (step 7)	2,42	1	2,42		
		Add the new key to memory (step 8)	6,13	1	6,13		
	Request Handler (Figure 22)	Get the requested key of index from flash (step 2)	3,10	1	3,10	10,67	
		Get sender's secret key from flash (step 4)	5,57	1	5,57		
		Encrypt packet with sender's key (step 5)	2,00	1	2,00		
	Third Device (Figure 23)	Save the match to memory (step 2)	3,57	1	3,57	3,57	

key storage is 22 x 100 = 2200 bytes. This data is stored in plain memory. The flash memory capacity is 1 MB, that means the usage ratio is 2200 bytes / 1 MB = 0,209%. Besides, RAM is used for neighbor list, a received index array of 1000 entries and the variables used in the implementation. 8264 bytes of the 10 kB RAM is used.

4.3 Analysis of Scenario 2

In scenario 2, the key chain size of 200 keys and a key pool size of 60000 keys. This corresponds to a local connectivity value of 0,488. The code space is the same as the first scenario, because *key chain size* effects only the flash memory usage.

Since there are 200 keys in the *key chain* and each entry is 22 bytes, that means the used flash memory is 22 x 200 = 4400 bytes. Thus the usage ratio is 4400 bytes / 1 MB = 0.4189%. RAM usage is the same and 8264 bytes. *Code space* is also the same, because there is no change in the implementation of device side.

The processing time performance of Scenario 2 is given in Table 4. As can be seen from this table, the processing times of the *Key Predistribution* and *Shared Key Discovery* phases almost double as compared to Scenario 1. The total processing times of these two phases are proportional to the *key chain size*. Since the *key chain size* is two times more than Scenario 1, the processing times also double.

The operations of the *Path-key Establishment* phase are not so dependent on the *key chain size* since index matching is performed via hashing and comparison. As a matter of fact, when we compare Table 3 and Table 4, we do not see a significant difference in the total processing times of the *Path-key Establishment* phases in both of the scenarios.

On the other hand, the processing time of *Key Transfer* phase depends on the *key chain size*. However, since we employ binary search here, the increase in the *key chain size* does not affect the total processing time significantly and *Key Transfer* phase is still faster than *Path-key Establishment*.

Table 4: The details of scenario 2

Phase	Processing party / Corresponding flowchart figure numbers	Steps / Step numbers in the corresponding flowchart	Processing time for one run	Number of runs	Total processing time	Phase Total	
Key Predistribution	All nodes / (Figure 12)	Reading key packet from computer and saving to memory (steps 5,6,7)	20,66	200	4.132,00	4.132,00	
Shared Key Discovery	All nodes / (Figure 16)	Get next index and check memory (steps 1,2)	1,87	Neighbor x 200	749,90	749,90	
Path-key Establishment	Request Sender (Figure 17)	Get next neighbor and check if has a shared key (step 1)	53,10	Neighbor	106,20	167,66	204,74
		Path-key request is sent and it is being handled	48,21	1	48,21		
		Get request handler's secret key from flash (step 5)	5,43	1	5,43		
		Decrypt the packet with request handler's key (step 6)	2,12	1	2,12		
		Add the new key to memory (steps 7,8)	5,70	1	5,70		
	Request Handler (Figure 18)	Search memory if has a shared key with requested node	4,60	1	4,60	22,83	
		Get sender's secret key from flash (step 4)	6,80	1	6,80		
		Encrypt packet with sender's key (step 5)	2,20	1	2,20		
		Get third node's secret key from flash (step 6)	7,13	1	7,13		
		Encrypt packet with third node's secret key (step 7)	2,10	1	2,10		
	Third Device (Figure 20)	Get request handler's secret key from flash (step 2)	5,80	1	5,80	14,25	
		Decrypt packet with request handler's secret key (step 3)	2,35	1	2,35		
		Add the new key to memory (step 4)	6,10	1	6,10		
Key Transfer	Request Sender (Figure 21)	Search array and find a match (step 2)	105,60	Insecure Neighbor	105,60	160,66	175,83
		Key transfer request is sent and it is being handled (steps 3,4)	40,20	1	40,20		
		Get request handler's secret key from flash (step 6)	6,23	1	6,23		
		Decrypt the packet with request handler's key (step 7)	2,33	1	2,33		
		Add the new key to memory (step 8)	6,30	1	6,30		
	Request Handler (Figure 22)	Get the requested key of index from flash (step 2)	3,40	1	3,40	11,60	
		Get sender's secret key from flash (step 4)	6,10	1	6,10		
		Encrypt packet with sender's key (step 5)	2,10	1	2,10		
	Third Device (Figure 23)	Save the match to memory (step 2)	3,70	1	3,70	3,57	

5 Conclusions

In this thesis, we have implemented Eschenauer and Gligor [5]'s basic scheme's three phases (*Key Predistribution*, *Shared Key Discovery*, *Path-key Establishment*) and also the alternative of *Path-key Establishment* phase, which is proposed by Ergun [15], called *Key Transfer*, in real sensor network nodes.

We used two scenarios for testing and performance evaluation. We used TelosB [16] motes in our testbed. We have selected key pool size of 15000 and key chain size of 100 for the first scenario and key pool size of 60000 and key chain size of 200 for second scenario.

We have analyzed the results of each scenario according to code space and memory usage metrics. We have also measured the processing times of the phases in details.

Our results show that when there two neighbors, *Shared Key Discovery* is completed in less than one second. Another important conclusion that we reach is that Ergun's *Key Transfer* phase is 14,11% faster than its alternative *Path-key Establishment* phase.

References

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam and E. Cayirci, Communications Magazine, IEEE, Vol. 40, No. 8., 2002, pp. 102-114.

[2] J. Yick, B. Mukherje and D. Ghosal, Wireless Sensor Network Survey, Computer Networks, Vol. 52, No. 12., 2008, pp. 2292-2330.

[3] R. L. Rivest, A. Shamir and L. Adleman, A Method for Obtaining Digital Signatures and Public Key Crypto-systems, Commun. ACM, Vol. 21, No. 2., 1978, pp. 120-126.

[4] W. Stallings, Cryptography and Network Security, Third Edition, Pearson Education, 2003

[5] L. Eschenauer and V. D. Gligor, A key-management scheme for distributed sensor networks, In CCS '02: Proceedings of the 9th ACM conference on Computer and communications security (2002), pp. 41-47.

[6] J. Daemen and V. Rijmen, The Design of Rijndael: AES - The Advanced Encryption Standard., Springer, 2002

[7] B. Schneier, Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish), In Fast Software Encryption, Cambridge Security Workshop, Ross J. Anderson (Ed.), Springer-Verlag, London, UK, pp. 191-204, 1993.

[8] NIST Special Publication 800-67 Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher, http://csrc.nist.gov/publications/nistpubs/800-67/SP800-67.pdf

[9] J. Kelsey, S. Lucks, B. Schneier, M. Stay, D. Wagner and D. Whiting, Improved Cryptanalysis of Rijndael, Fast Software Encryption, 2000, pp. 213-230.

[10] S. Hussain, F. Kausar, and A. Masood, An efficient key distribution scheme for heterogeneous sensor networks, In Proceedings of the 2007 International Conference on Wireless Communications and Mobile Computing (IWCMC '07), 2007, pp. 388-392.

[11] D. Liu, P. Ning, and W. Du, Group-based key pre-distribution in wireless sensor networks, In Proceedings of the 4th ACM Workshop on Wireless Security (WiSe '05), 2005, pp. 11-20.

[12] J. Hwang and Y. Kim, Revisiting Random Key Pre-distribution Schemes for Wireless Sensor Networks, In SASN '04: Proceedings of the 2nd ACM Workshop on Security of Ad Hoc and Sensor Networks, 2004, pp. 43-52.

[13] W. Du, J. Deng, Y.S.Han, S. Chen and P.K. Varshney, A Key Pre-Distribution Scheme Using Deployment Knowledge for Wireless Sensor Networks, INFOCOM, Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies, Vol. 1, 2004 pp. 586-597.

[14] S. Choi and H. Youn, An Efficient Key Predistribution Scheme for Secure Distributed Sensor Networks, Embedded and Ubiquitous Computing, LNCS 3823, 2005, pp. 1088-1097.

[15] M. Ergun, Resilient and Highly Connected Key Predistribution Schemes for Wireless Sensor Networks, MSc Thesis, Sabanci University, 2010.

[16] http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html?download=152

[17] http://www.tinyos.net