Senior Capstone 2009–2010

Applied Cryptography and Information Security

Project Advisor: Prof. Nelly Fazio

A Secure Communication System for Sensor Networks

Tolulope Abayomi Stanis Laus Billy Edwin Guzman
Irippuge Milinda Perera

May 10, 2010

The City College of CUNY

Abstract

The benefits of the Global Positioning System (GPS) network have been enormous and far-reaching. The system is in use on all forms of transportation and personal navigation devices. Due to the limitations of the GPS, however, there is a need for a system that can be used for more localized positioning in environments that are not outdoors and in direct line-of-sight to the satellites, which are requirements for the GPS devices. The SunSPOT Local Positioning System is designed to address this need. Positioning is implemented using triangulation methods similar to the methods used in the GPS system. Instead of satellites in outer space, however, it uses Sun Microsystems Small Programmable Object Technology (SunSPOT) devices, which are wireless sensors with built-in radios. The devices will be used to program a local positioning system that will help in emergency situations. The system will be used as a locator for emergency personnel when they are inside buildings where they may get injured and unable to call for help. This system requires a secure network that will prevent intruders from stealing information and from inputting false information to the network. To construct a secure network, a Certification Authority is set up to certify the validity of the public keys generated by the devices. A secure broadcast channel is established using the Elliptic Curve Diffie-Hellman Key Exchange protocol. Messages are then encrypted (using the Advanced Encryption standard (AES) algorithm) and authenticated (using

a Message Authentication Code (MAC) algorithm). These algorithms are implemented to uphold the security and integrity of data in the network.

Table of Contents

List of Figures

1 Executive Summary

The problem of maintaining security, confidentiality and integrity of information is one with a long history. Over the years, cryptographic protocols have been developed to help encrypt and verify information. These protocols have applications in many scenarios, such as in everyday credit card transactions and cell phone communications. In emergency scenarios, it is critical to be able to send information without fear of data corruption, or eavesdropping from a third party. The software presented in this report provides a method to communicate in a secure and closed environment while sending and receiving electronic sensor data from multiple parties especially during an emergency.

1.1 Project Goals

This project's primary goal was to design a system for real time communication of secure data between wireless SunSPOT sensor devices. In particular, the aim of the project was to transmit secure sensor data, including location information in a network comprised of roaming wireless SunSPOTs, fixed SunSPOTs and a hosted application running on a server, collecting and analyzing the data transmitted by the wireless sensor devices during an emergency rescue operation.

1.2 Group Work

During preliminary research and preparation for the project, it was determined that the project would have four main aspects: Security, Location, a running application with a Graphic User Interface and a method to integrate all aspects into one cohesive system. The four aspects were divided between the members of the team. Work involved doing research on existing implementations of location tracking, the theory behind existing cryptographic methods and available libraries for synchronous encrypted communications.

1.3 Hardware

The hardware used in this project is the Small Programmable Object Technology (SPOT) device from Sun Microsystems (SunSPOT). These are wireless devices with sensors for light, temperature and orientation using a built-in accelerometer. In addition, the devices are programmable using the Java language which can run on the Squawk VM, the embedded Java virtual machine on the SunSPOT. The device is also able to support Mesh Networking, Multi-hop Over The Air programming, and a software library with high grade elliptic curve cryptography (ECC).

Additional hardware considered for the project included the Texas Instruments C2431 System-on-Chip (SoC) Solution for ZigBee/IEEE 802.15.4 Wireless Sensor Network with Location Engine. This chip was researched as a way to provide more accurate location information to the SunSPOTs.

2 Introduction

2.1 Objective

The desire to keep information secure, confidential and intact has been an integral part of the development of civilizations from the beginnings of time. Modern science and mathematics have provided avenues to achieve that goal with the rise of Cryptography. With the current methods of encryption, we have numerous avenues for achieving security, confidentiality and integrity of almost any kind of information. The types of information and circumstances vary widely, from passwords for logging into websites, to encrypted communications between governments, to sustaining an ongoing secure channel of communication in a closed environment as is the case in this project. The proposed use case for this project is as a secure locator beacon for firemen during a fire-fighting operation and possibly other emergency personnel in similar situations. Devices would be attached to personnel going into a building. As the firemen go about their duties, it would provide data in a continues, secure stream about the environment and the location of the fireman back to a monitoring center. Where necessary, such as in the case of an injury, the devices can communicate the location of the fireman back to the monitoring computer system.

This project explored creating a secure environment in ideal circumstances where wireless sensor data including the sensor's location relative to other devices, the surrounding temperature and light readings, and the orientation are transmitted to a monitoring system. Where necessary, this information would also be transmitted between devices. To achieve this, certain elements would need to be in place: A device to collect sensor data and communicate it, a way to secure that communication and a system to monitor that data.

The devices chosen for this project, the SunSPOTs, were ideal for this project because of the features that come built into the devices, their small size and the relative ease of programming using the Java language.

To communicate with the server, the devices have built in WiFi radios and networking libraries which allow for creating ad-hoc and mesh networks between the devices without much configuration.

2.2 Purpose and Scope

The purpose of the project is to investigate creating secure wireless communications in a closed environment, in an ad-hoc manner, where real-time monitoring of data is mission critical. Under ideal circumstances, any kind of data can be transmitted. The scope of the project was limited to emergency rescue operations performed by firemen, where sending sensor and location data in real-time is critical. If successful, the scope of the project can be expanded for many more applications.

3 Background Knowledge and Theoretical Concepts

Implementing security and location required some research and a survey of already existing implementations on the market and the possibilities of adapting or enhancing those solutions to our project.

3.1 Security

Wireless sensor networks are a new breed of networks systems which are used in a variety of realms, such as battlefield surveillance, machine health monitoring, home automation, inventory control, and rescue situations [15]. The main goal of our project was to design and implement a cryptographically safe communication protocol to be used in these sensor networks. Next, we developed a system in order to demonstrate how the aforesaid protocol could be used for communication between the sensors in fire-rescue situations.

The main goals of the security protocol that was designed were as follows,

1. Low power consumption
2. Ability to expand/shrink the sensor network dynamically with minimal cost
3. Confidentiality of the communicated data
4. Integrity of the communicated data

Given the ad-hoc nature of the sensor networks, asymmetric cryptography appears to be the perfect solution. However, since the sensors are usually characterized by limited power supplies, low power consumption by the security protocol is paramount [11]. Thus, public key cryptography is too computationally expensive to be employed. Therefore, the protocol that was designed for this project has a hybrid nature such that asymmetric cryptography is used for key exchange and symmetric cryptography is used for secure communication of data.

The hybrid cryptographic protocol [10] that was designed for this project can be compared to the TLS (Transport Layer Security) protocol which is used to provide security for communication over the internet. One of the differences in our project from TLS is that we require every sensor node to obtain a certificate from the certificate authority before any communication can take place. The certificate authority is a trusted third party that can create certificates for sensors [15]. Once the certificate is created, any sensor with the correct verification key of the certificate authority can check the validity of the certificate. This greatly minimizes the cost of adding new sensors to an existing sensor network. Similar to the TLS protocol our design has two main components, namely,

1. Handshake process
2. Secure communication of data

We examine each component thoroughly in the next few sections.

3.1.1 Handshake

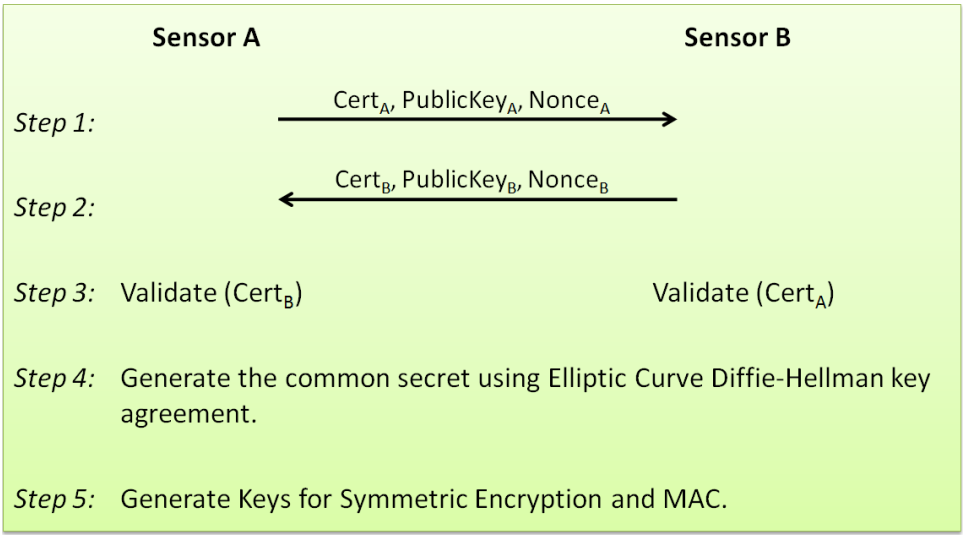


Figure 1: Handshake Process

During the handshake process the two participating sensors agree on various parameters which are required to establish a secure channel between them. The handshake is a five step process as pictured in Figure 1.

During the step 1 of the handshake, the sensor requesting the secure channel (sensor A) sends its certificate, the data signed in the certificate, and the nonce to the second sensor (sensor B). The nonce is a random number which is used in the common secret generation (step 4) to ensure that old communications cannot be reused in reply attacks.

At step 2, the requested sensor (sensor B) sends its certificate and the data signed in the certificate to the requesting sensor (sensor A). The requested sensor (sensor B) also sends its own nonce with this reply.

At the third step of the process, both sensors validate the certificates that they have received by using the verification key of the certificate authority. If the verification is successful, the handshake process continues. Otherwise, the handshake process halts and the secure channel creation terminates.

Reaching the fourth step of the process implies that both parties have the verified certificates of each other. Now a common secret (Diffie-Hellman key) is generated by using the

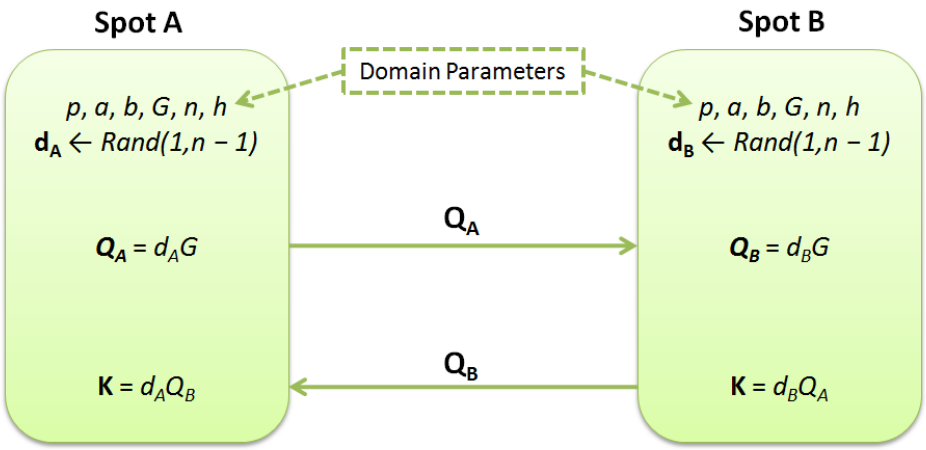


Figure 2: Elliptic Curve Diffie-Hellman Key Agreement

Elliptic Curve Diffie-Hellman key agreement [4] as shown in Figure 2. Before this common secret is generated, both parties must have agreed on all the elements that define the elliptic curve. These elements are also called the domain parameters of the scheme. The field is determined by the prime p, and the elliptic curve is defined by the constants a and b. The cyclic subgroup is defined by the base point G, which is of order n (a prime number). The number h is the cofactor. d_A and d_B, which are randomly selected integers in the interval [1, n – 1], are the private keys. Q_A and Q_B, which are generated by multiplying the corresponding private key and the base point G, are the public keys. Finally, the common secret K is found by multiplying the private key of one entity and the public key of the other entity.

Diffie-Hellman Key :	K
Public keys (Q):	$Q_X \| Q_Y$ (where Q_X and Q_Y are lexicographically ordered Q_A and Q_B)
Nonce (n):	$n_X \| n_Y$ (where n_X and n_Y are lexicographically ordered n_A and n_B)
Extended Diffie-Hellman Key :	$K \| Q_X \| Q_Y \| n_X \| n_Y$

Figure 3: Extending the Common Secret

After creating this common secret K, the lexicographically ordered public keys and the nonces of the two participating sensors are concatenated to K as shown in Figure 3. The main reason for adding the public keys to the common secret is to bind the generated secret to the public keys of the sensors. As mentioned previously, the nonces are added to avoid reply attacks.

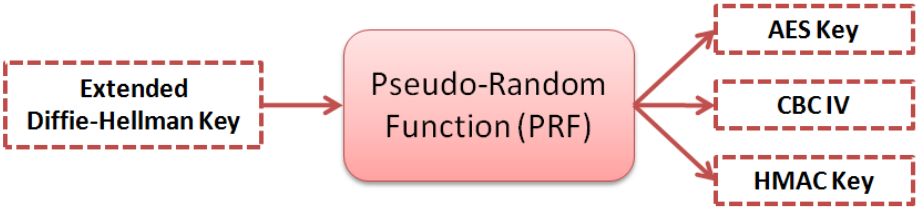


Figure 4: Black Box Representation of the Key Generator

At the last stage of the handshake process, the keys required for the symmetric encryption scheme and the MAC (Message Authentication Code) are generated sequentially. The secret generating function could be explained as a pseudo-random function which accepts the extended Diffie-Hellman key as the input, and outputs the required keys as shown in Figure 4.

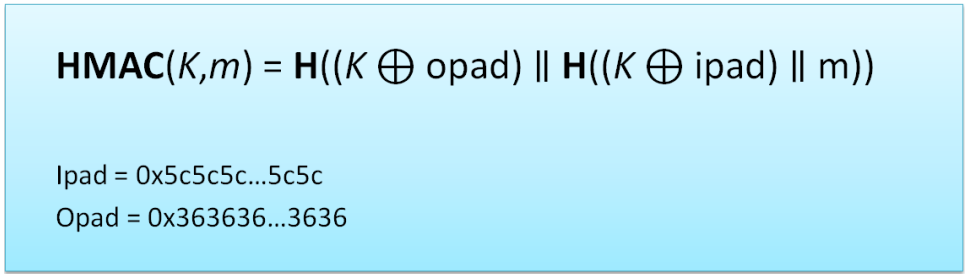


Figure 5: HMAC Function

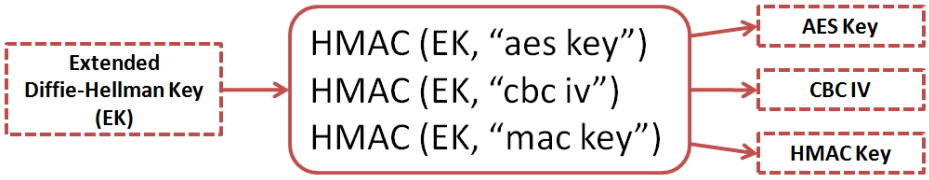


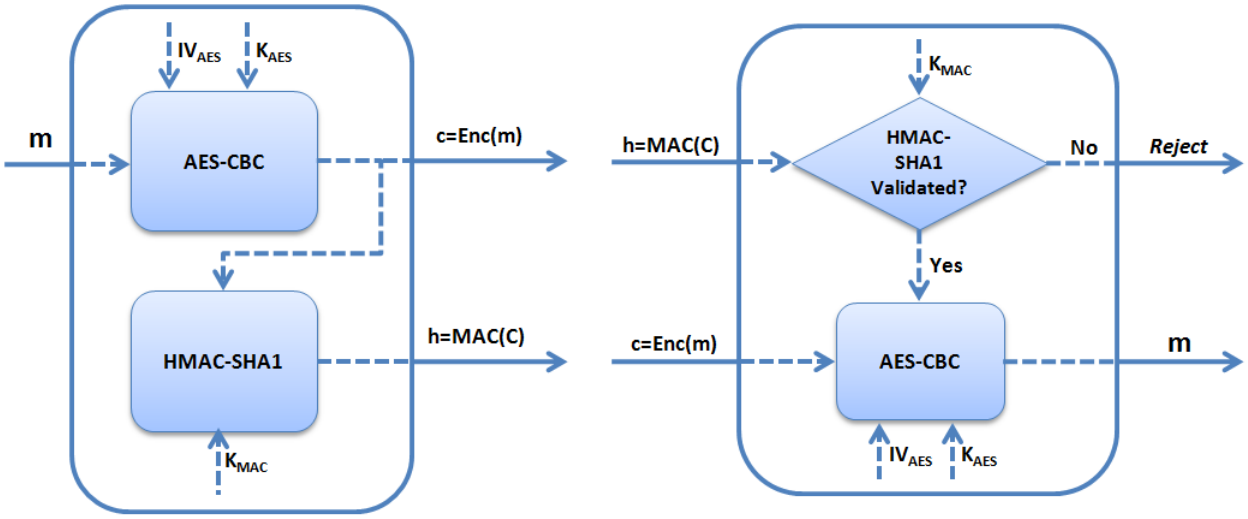
Figure 6: Implementation of the Key Generator

The implementation of the pseudo-random function used in this protocol was the HMAC, which is a Hash based MAC. As shown in Figure 5, the HMAC function takes in the key(k) and the message(m) as arguments and outputs the result of two hashes composed in a special way [1]. Figure 6 demonstrates how this HMAC-based pseudo-random function is used to generate keys.

3.1.2 Secure Communication of Data

After the handshake process has been completed, both sensors have the required keys to instantiate the secure channel. Let's examine how symmetric cryptography is used to communicate supposedly sensitive data through this newly established secure channel. The main two requirements that should be satisfied in the secure communication of data are:

- Confidentiality of the communicated data
- Integrity of the communicated data



(a) Encryption (b) Decryption

Figure 7: Encryption/Decryption Functions

Confidentiality of the data is enforced by encryption with the AES (Advanced Encryption Standard) block cipher [7] in CBC (Cipher-block Chaining) mode [5]. In the CBC mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted and the first plaintext block is XORed with an initialization vector. This process ensures that the encryption results in pseudo-randomness. Since only the sensor that participated in the handshake process has the valid key and the CBC initialization vector, no adversary could decrypt the data. This, however, does not imply that an adversary cannot tamper with the encrypted data even though he/she would not be able to obtain any information from it. This is why we must also maintain the integrity of the encrypted data by using some mechanism. The mechanism used in this protocol is the HMAC. Figure 7a gives a high-level, black box representation of the overall mechanism.

The decryption mechanism is symmetric to the encryption since the decrypting keys are the same as the keys used in the encryption. However, as shown in Figure 7b, the verification of the message authenticating code is done before the actual decryption. If the verification fails the decryption process rejects the encrypted message. This ensures that the integrity of the encrypted message is kept intact.

3.2 Location

3.2.1 Triangulation

In trigonometry and geometry, triangulation is the process of determining the location of a point by measuring angles to the point from two known points at the end of a fixed baseline. The coordinate of the point can then be calculated by fixing it as the third point of triangle with two known angles and one known side, as illustrated in Figure 8.

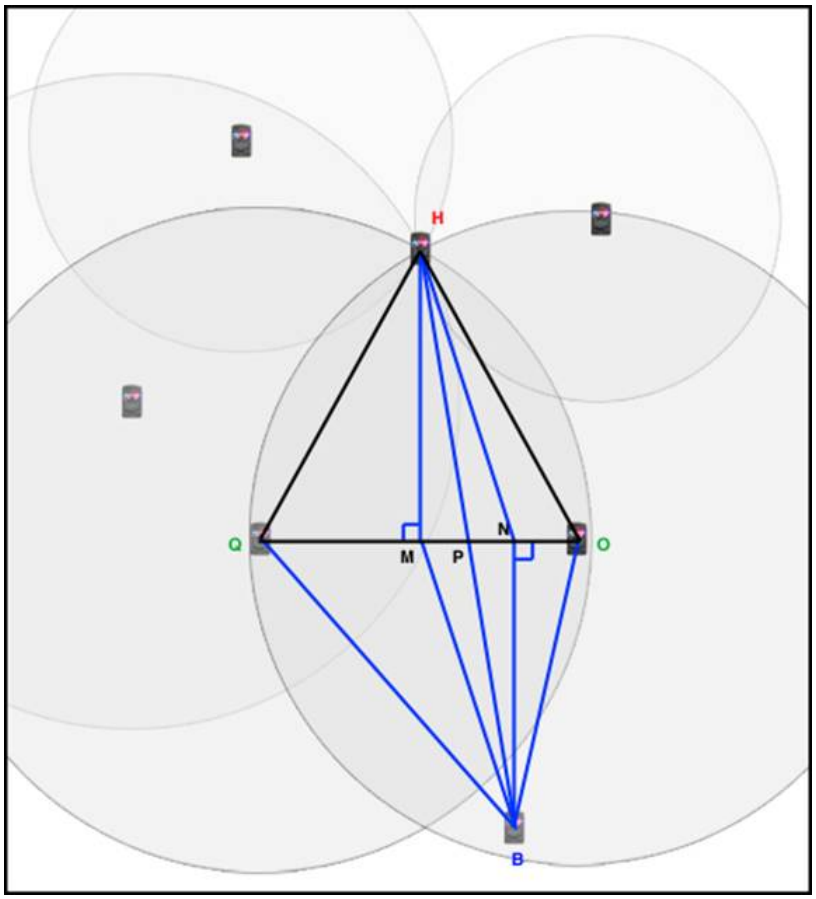


Figure 8: The ideal scenario for Triangulation

In an ideal scenario, we have two stationary SPOTs **(observers)**, one moving SPOT

(roamer), and one **base-station**. In our scenario, Q and O are the observers, H is the roamer, and B is the base-station. The following qualities are known and pre-calculated by the observer SPOTs:

- The distance between the two observers (OQ)
- The distances between the two observers and the roamer (HQ, HO)
- The distances between the two observers and the base-station (BQ, BO)

The goal is to approximate the distance and direction of the roamer relative to the base-station and consequently, relative to each observer as well). With these known conditions, the following algorithm was proposed:

Step 1: Finding angle HQO

The main goal here is to approximate the angle between edges HQ and $OQ(HQO)$ or the angle between edges HO and $OQ(HOQ)$. Since OQ, HQ, and HO are known, we can calculate the following:

1. $HM^2 = HQ^2 - MQ^2$
2. $HO^2 = HM^2 + (OQ - MQ)^2$
   $HO^2 = HQ^2 - MQ^2 + (OQ - MQ)^2$
3. $MQ = \frac{(HQ^2+OQ^2-HO^2)}{2OQ}$
4. $HQO = \arccos(\frac{MQ}{HQ})$

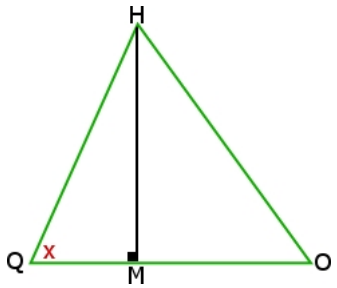


Figure 9: Triangulation: Finding the first half of the angle

Step 2: Finding angle BQO

The main goal here is to approximate the angle between edges BQ and $OQ(BQO)$ or the angle between edges BO and $OQ(BOQ)$. Since OQ, BO, and BQ are known, we can calculate the following:

1. $BN^2 = BO^2 - NO^2$
2. $BQ^2 = BN^2 + (OQ - NO)^2$
   $BQ^2 = BO^2 - NO^2 + (OQ - NO)^2$
3. $NO = \frac{(BO^2 + OQ^2 - BQ^2)}{2OQ}$
4. $HQO = \arccos\left(\frac{OQ - NO}{BQ}\right)$

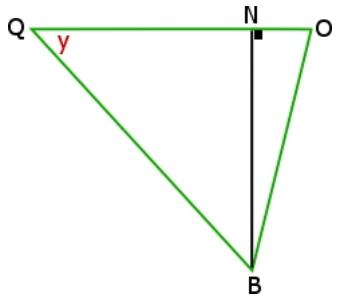


Figure 10: Triangulation: Finding the second half of the needed angle

Step 3: Finding BH - the distance between the roamer and the base-station

Now that we know HQ, BQ, and the angle between them ($BQH = HQO + BQO$), we can easily find BH by applying the Law of Cosines as such:

$$BH^2 = BQ^2 - 2 \cdot HQ \cdot BQ \cdot \cos(BOH)$$

Step 4: Finding the direction (using the angles QBH or OBH)

The angle QBH can be calculated by applying the Law of Cosines as follows:

$$\begin{aligned} QH^2 &= BH^2 + BQ^2 - 2 \cdot BH \cdot BQ \cdot \cos(QBH) \\ QBH &= \arccos\left(\frac{BH^2 + BQ^2 - HQ^2}{2 \cdot BQ \cdot BH}\right) \end{aligned}$$

3.2.2 Distance-Bounding

Another option for localization is the Distance-Bounding protocol, first proposed by Brands and Chaum [3]. This protocol actually deals more with the security part of the project than

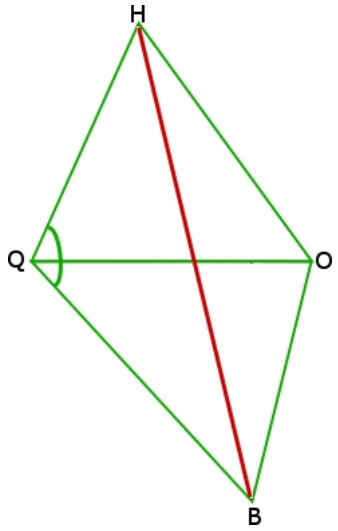


Figure 11: Triangulation: Finding distance between roamer and base-station

localization, but the team considered this option crucial in hiding the location of the roamer from malicious attacks in our scenario of a sensor network.

In a sensor network, distance estimation between two nodes (in our case, observer SPOTs) plays crucial part in setting up and maintaining the integrity of the sensor network. One node is able to localize itself, if and only if, it can learn its own distance from three or more nodes in the network. One of the most accurate measures of distance estimation is the time of flight of the signal between two nodes. For example, node A bounces a signal off of node B and measure the round-trip time it took for the signal to travel from node A to node B, and back to node A. It seems like a simple enough method, except in the scenario where hostile attackers need to be taken into account.

In the Distance-Bounding protocol, the node that tries to localize itself is called a verifier and the node that acts as the reference point in bouncing off the signal is called a prover. In the simplest scenario, if a prover is of the malicious nature, it can pretend to be closer or further away from the verifier than it actually is, therefore disrupting the integrity of the network. This is very dangerous, especially in our emergency situation scenario.

An obvious defense against such an attack is to have the prover authenticate its response signal. The problem with this method is that the time it takes to authenticate the signal might exponentially add to the total time, which would be significantly lower if it was not authenticated (only the travel time). A possible solution to this is to have the prover send two responses: fast, unauthenticated response and the slower, authenticated response. This however, is still subject to hostile attacks that can easily prevent the authenticated response from reaching the verifier, replacing the response with its own "authenticated" response. The Distance-Bounding protocol offers a solution to this problem.

The basic idea of a distance-bounding protocol consists of a single-bit challenge and

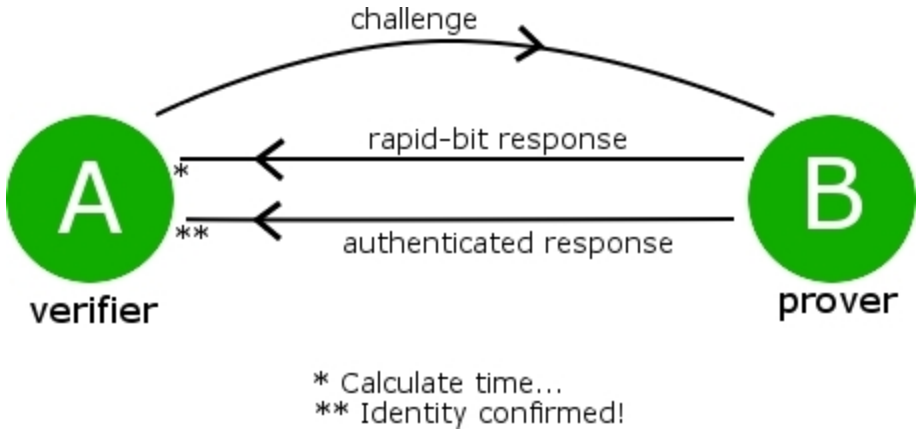


Figure 12: Distance-Bounding Protocol

rapid single-bit response. Each bit of the prover (*the rapid single-bit response*) is to be sent out immediately after receiving a bit from the verifier (*the single-bit challenge*) and then the prover follows up with sending the authenticated response, as illustrated in Figure 12. The verifier uses the elapsed time between sending its challenge and receiving the prover's response to measure the distance.

3.2.3 Received Signal Strength Indicator (RSSI)

Our goal was to attempt to approximate the distance and the direction of the SunSPOT that is requesting help relative to the base-station SunSPOT and other observer SunSPOTs.

As previously described, being able to determine position in this project is of particular importance, since the application of our project involves use in situations where localization is crucial. When a rescuer is in need of help, knowing where he is in a building becomes a top priority.

As a part of the project, research was done on additional methods to implement positioning. The team reviewed how positioning was previously implemented by others and the necessary knowledge needed to implement it. The SunSPOT API and reports by Sun Lab personnel were a useful starting point in learning about Radio Signal Strength Indication on the SunSPOTs.

3.2.4 RSSI and Link Quality

One of the first steps taken was to look at the `RadioStrength` demo that was included in the SunSPOT SDK. The demo showed how to use the API to get the RSSI and link quality. Another feature of the demo was to show the signal strength between two SunSPOTs by using the LEDs. All the lights were on when two SunSPOTs were next to each other and

the lights faded as they grew farther apart in distance.

According to the SunSPOT API [12]:

1. Received Signal Strength Indicator (RSSI) measures the strength (power) of the signal for the packet. It ranges from +60 (strong) to -60 (weak). To convert it to decibels relative to 1 mW (= 0 dBm[1]) we subtract 45 from it, e.g. for an RSSI of -20 the RF input power is approximately -65 dBm. It is worth noting that RSSI values have high variance and low entropy. What this means is that the values fluctuate from high to low due to reflections, absorptions, refractions, and other environmental influences.

2. Link Quality Indicator (LQI) is a characterization of the quality of a received packet. Its value is computed from the correlation value. The LQI ranges from 0 (bad) to 255 (good).

3.2.5 Accelerometer

Another method explored for localization was the use of the SunSPOT's accelerometer. The SunSPOTs are equipped with the LIS3L02AQ Accelerometer, which is a low-power accelerometer that measures changes in speed in three-axis [6]. Figure 13 shows the axes that the SunSPOTs are calibrated to.

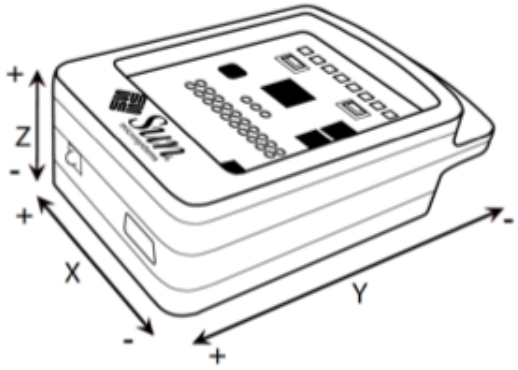


Figure 13: The X, Y, and Z axes of the SunSPOT's Accelerometer

With the SunSPOT API, accelerometer readings for the X, Y, and Z-axes were easy to obtain, but determining positioning was harder. An accelerometer measures the change of speed of an object over time. From basic physics, we know that acceleration is the change of speed and speed is the change in position of an object. To obtain the position on said object, one can perform integration on acceleration and then integration again on speed. This can be done on the SunSPOTs by adding all the samples taken from the readings. The problem

[1]dBm: is the power ratio in decibels relative to one milliwatt.

encountered here is that sampling should be done infinitely fast or else the measurement will be off [8].

It is also worth noting that SunSPOTs constantly measure the gravitation force while it is at rest and therefore, if the SunSPOT is free-falling, it will read a gravitational force of 0. The effects of gravity can be ignored and subtracted from the total accelerometer readings.

3.2.6 Radio

Apart from the aforementioned approaches, we also tried to obtain location readings via the SunSPOT's own radio. The radio equipped is a 2.6GHz radio with a built-in antenna in the SunSPOT, IEEE 802.15.4 compliant (standard for low-rate wireless networks), and able to transmit packets at 250kbits/second.

As was discussed in one of the Sun's personnel's blog [9], calculating distance using radio is tricky because we need an accurate reading of direction, a known distance to a stationary object, and as low interference/noise from objects in the environment.

3.2.7 Weighted-Centroid Localization

The Weighted-Centroid Localization (WCL) [2] algorithm was another option that we considered, making a total of three algorithms that can be mixed-and-matched and implemented to triangulate. This algorithm showed the most promise in obtaining our goal.

Let's look at how the basic Centroid Localization (CL) algorithm works. Say we have n stationary nodes (**observers**): in a primitive version of the algorithm, all stationary nodes will send their position $B_j(x, y)$ and the sensor node (**roamer**) will calculate its own position $P_i'(x, y)$ using the summation of all the positions from the stationary nodes.

$$P_i'(x, y) = \frac{1}{n} \sum_{j=1}^{n} B_j(x, y)$$

$B_j(x, y)$ is the position transmitted by a stationary node at position (x, y).

$P_i\prime(x, y)$ is the position transmitted by a sensor node at position (x, y) without weights.

While the CL algorithm makes use of taking the average of the nodes' coordinates, the WCL algorithm results in a more accurate reading for determining distance and position.

$$P_i''(x,y) = \frac{\sum_{j=1}^{n}(w_{ij}B_j(x,y))}{\sum_{j=1}^{n} w_{ij}}$$

$P_i''(x,y)$ is the position transmitted by a sensor node at position (x,y) with weights.

w_{ij} is the weight of a particular stationary node.

As described by the formula above, the WCL algorithm takes into account w_{ij} which is a function that depends on the distance and characteristics of the sensor node's receivers. The weights of each stationary node changes according to the environment setting. The weight function w_{ij} is inversely proportional to the distance ij from that stationary node to the sensor node. The distance is raised to the power of g to marginally obtain better readings; longer distances weight lower.

$$w_{ij} = \frac{1}{(d_{ij})^g}$$

d_{ij} = distance between beacon B_j and sensor node P_i.

Ideally, in the WCL algorithm, the position readings from the stationary nodes would be obtained simultaneously.

4 Implementation and Results

4.1 Security

The security protocol that we designed in this project was implemented as a layered system. The layers used in our implementation, and the names of the packages where they are located are given below:

1. Security Components (`com.sun.spot.security / com.sun.spotx.crypto`)
2. Network Components (`redteam.communication`)
3. Secure Channel Components (`redteam.communication.secure`)

The Secure Channel Components layer heavily depends on the Security Components layer and the Network Components layer. Let's examine the structure and the implementation of the classes contained in these layers one by one.

4.1.1 Security Components

The first (and the most basic) layer, Security Components, contains all the cryptographic component classes which are needed in addition to the classes given in the SunSPOT API. These classes are distributed in two main packages according to the SunSPOT API's design philosophy. The first package (`com.sun.spot.security`) contains the general cryptography classes such as hashing, message authentication, and key generation. The second package (`com.sun.spotx.crypto`) comprises the classes related to encryption/decryption and key exchange. The classes in this layer are independent of any other package except the classes in the SunSPOT API. The actual names of the Java classes implemented are given below:

1. `Certificate.java`
2. `CertificateData.java`
3. `CertificateCredentials.java`
4. `CertificateEngine.java`
5. `HMAC.java`

Certificate.java: This is the representation of a certificate according to the Public Key Infrastructure model that we have implemented in this security protocol. This class has object composition relationships with the two component classes, `CertificateCredentials.java` and `CertificateData.java`. The two most important methods implemented in this class are the `serialize` and `deserialize` methods. The `serialize` method returns the byte array representation for a given certificate object, whereas the `deserialize` method returns the certificate object from a given serialized byte array. The class diagram of `Certificate.java` is shown in Figure 14.

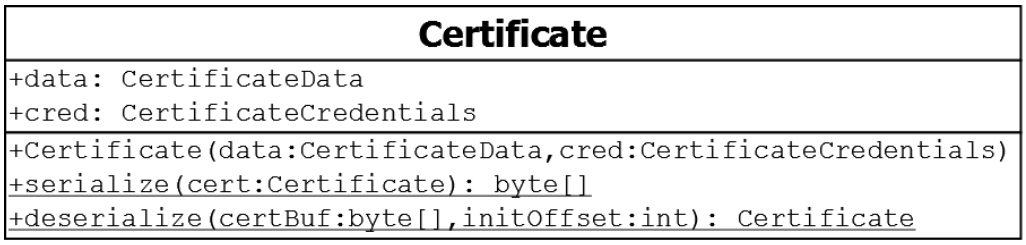


Figure 14: Certificate.java Class Diagram

CertificateData.java: `CertificateData`, constructed by the owner of a PKI Certificate, is the first half of the certificate class. This class contains the user-supplied data of a PKI certificate. In the current implementation, the data contained in this class are SPOT Type, EC Public Key, and SPOT Address. Any other fields can be added to this class without much effort. `CertificateData.java` also implements the `serialize` and `deserialize` methods which are called by the `Certificate.java` class. The most important method implemented in this class is `getDataForSignature`, which returns a byte array representation of the user supplied data that is ultimately signed by the certificate authority. The class diagram of `CertificateData.java` is shown in Figure 15.

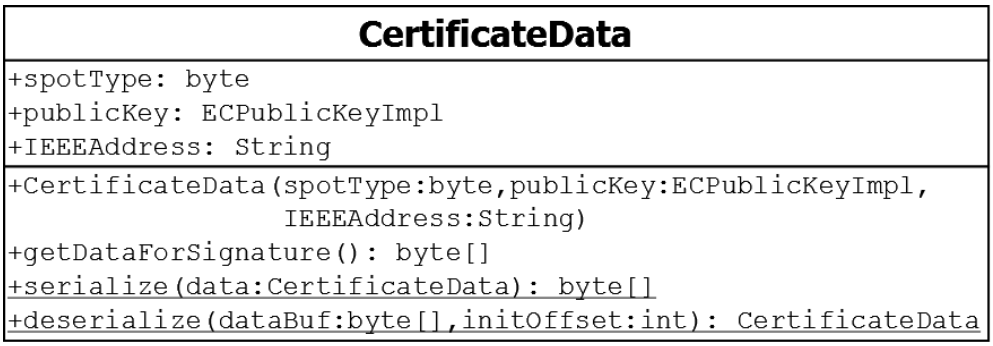


Figure 15: CertificateData.java Class Diagram

CertificateCredentials.java: `CertificateCredentials` is the counterpart of the `CertificateData.java` class for the certificate authority. The data contained in this class

are the name of the certificate authority, and the signature of the data supplied by the user via an instance of the `CertificateData.java` class. This class also implements the `serialize` and `deserialize` methods so that they can be called within the `Certificate.java` class. Class diagram of this class is given in Figure 16.

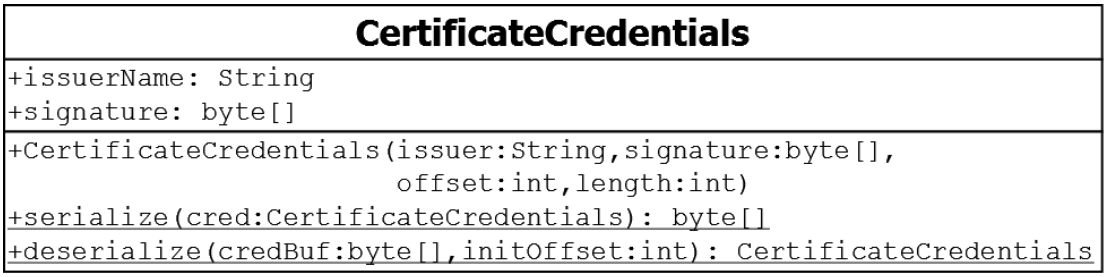


Figure 16: CertificateCredentials.java Class Diagram

CertificateEngine.java: As the name implies, `CertificateEngine` manages the creation and verification of PKI Certificates in our implementation. As Figure 17 depicts, this class contains only four methods.

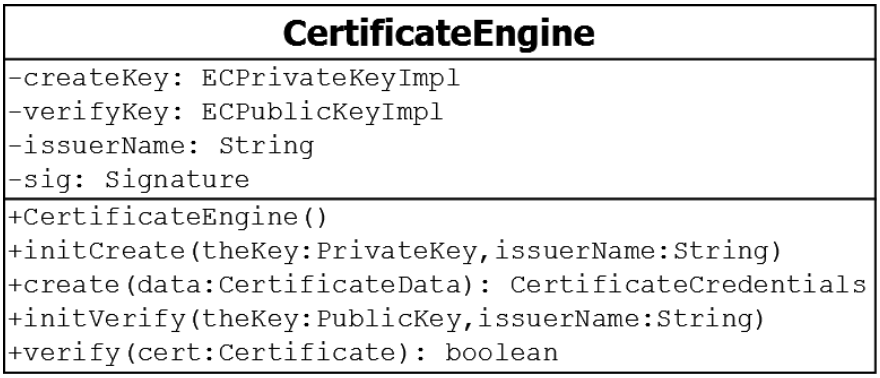


Figure 17: CertificateEngine.java Class Diagram

The sequence of methods that should be called for certificate creation is `initCreate` followed by `create`, whereas the sequence of methods for certificate verification is `initVerify` followed by `verify`.

HMAC.java: This is our implementation of the Hash-Based Message Authentication Code (HMAC) [1]. The internal hashing function used in HMAC is the SHA-1 function which is already implemented in the SunSPOT API. The class diagram of HMAC method (for a given key K and a message m) is show in Figure 18.

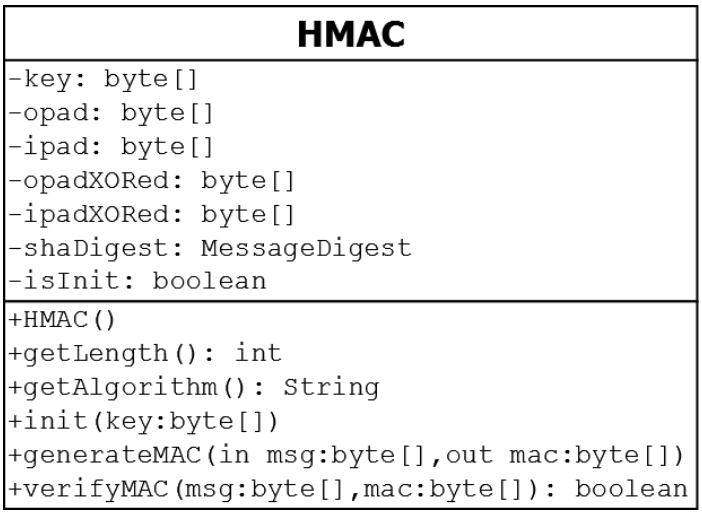


Figure 18: HMAC.java Class Diagram

4.1.2 Network Components

The second layer, Network Components, contains the required classes for basic network communication. The main purpose of this layer is to provide a generic, fast, and reliable mechanism to send and receive radiograms. As a requirement for the event-based programming paradigm used in our implementation, most of the classes in this layer are designed to run on separate threads. One should note, however, that data sent in this layer is in the clear with no security whatsoever. The protocol which ensures secure communication of data is implemented in the next layer which is Secure Channel Components. The classes implemented and the interfaces specified in the Network Components layer are given below.

- Classes:
  1. `PacketTransmitter.java`
  2. `PacketReceiver.java`
  3. `ServerCommunicator.java`
  4. `ServerBroadcaster.java`
- Interfaces:
  1. `IPacketHandler.java`
  2. `IAlerter.java`

PacketTransmitter.java: This class implements the functionality required to send data packets. More specifically, data packets are sent through instances of `Radiogram.Java` via

the connections opened with `RadiogramConnection.java`. The partial class diagram of this class is shown in Figure 19.

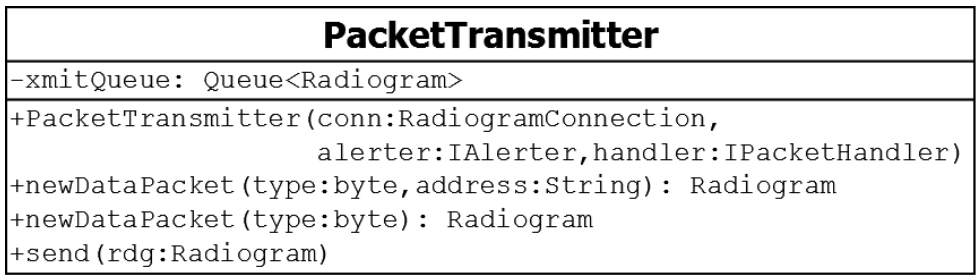


Figure 19: PacketTransmitter.java Partial Class Diagram

The `PacketTransmitter` must be instantiated with a valid `RadiogramConnection` instance and implementations of the two interfaces, `IAlerter` and `IPacketHandler`. `RadiogramConnection` is used as the medium through which the radiograms are sent. The `IAlerter` implementation is used to alert sending of data through the `PacketTransmitter` and the `IPacketHandler` implementation is used to inform the sender of any failure that may occur when sending radiograms.

Data is sent though the `PacketTransmitter` in the following order.

1. The sending entity obtains a new radiogram from the `PacketTransmitter` by calling the `newDataPacket(args...)` method.
2. The sending entity then fills the radiogram with the data to be sent by using the appropriate methods specified in the `Radiogram.java` class.
3. Next, the sending entity passes the filled radiogram object to the `PacketTransmitter` by calling the `send(args...)` method.
4. The `PacketTransmitter` sends the given data asynchronously using the internal threading and queuing mechanism.

Our network layer requires that every radiogram sent must have a corresponding type. However, the association of semantics to these types is left to the users of the network layer. When a sending entity attempts to send a packet by calling the `newDataPacket` method, the `PacketTransmitter` obtains the packet type from the entity and places that in the first byte of the returned radiogram. This is how this protocol ensures that every data packet sent has an associated type. Since this type is a byte value in the current implementation, the sending entity has 256 possibilities for data packet types.

Internally, the `PacketTransmitter` maintains a queue which is used to store the outgoing radiograms. When it receives filled radiograms from entities, it places them in this

queue in the same order that they are received. A separate thread is run inside the `PacketTransmitter` to check the outgoing queue for any radiograms. If there are any, the internal thread sends them through the `RadiogramConnection` which was supplied by the sending entity when the `PacketTransmitter` was instantiated. This mechanism ensures that the data packets are sent in a First In First Out (FIFO) fashion.

PacketReceiver.java: This implements the functionality required to receive data packets. Similar to `PacketTransmitter.java`, the data packets are received through instances of `Radiogram.Java` via the connections opened with `RadiogramConnection.java`. The class diagram of `PacketReceiver` is shown in Figure 20.

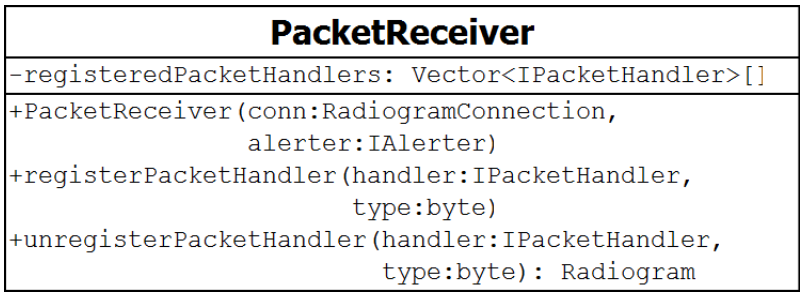


Figure 20: PacketReceiver.java Class Diagram

Similar to the `PacketTransmitter`, the `PacketReceiver` must be instantiated with a valid `RadiogramConnection` instance and an implementation of the interface `IAlerter`. After that, the receiving entity must register with the newly instantiated `PacketReceiver` to handle incoming radiograms. This is performed by using the `registerPacketHandler` method of the `PacketReceiver` which requires an implementation of the `IPacketHandler` interface and the type of the anticipated radiograms as the arguments. The receiving entity could either implement the `IPacketHandler` interface on its own or use an instance of another class which has implemented that interface. `PacketReceiver` internally maintains a vector of 256 vectors of `IPacketHandlers`, and saves a reference to the supplied `IPacketHandler` instance inside this vector indexed by the anticipated packet type.

When a new radiogram arrives at the `PacketReceiver`, it obtains the type of the received radiogram by reading the radiogram's first byte. Next, it checks if there are any `IPacketHandlers` registered to receive that type of radiograms, and if there are any, informs them individually.

ServerCommunicator.java: This implements a general purpose server communicator, which can send to and receive from any one spot at a time on a given fixed port by using the `PacketTransmitter` and `PacketReceiver` as primitive network components. This class is an example of how one could make use of the PacketTransmitter and PacketReceiver to create a high-level communication class.

ServerBroadcaster.java: This is the implementation of a general purpose server broadcaster, which can send to all spots at once on a given fixed port by using a `PacketTransmitter` instantiated with a broadcast-mode `RadiogramConnection`. However, this implementation of the server broadcaster does not have the functionality to receive radiograms. In general, the `ServerBroadcaster` is used to multicast radiograms and the `ServerCommunicator` is used to unicast and receive radiograms.

IPacketHandler.java: This interface specifies the functionality expected from a class prepared to receive radiograms through a `PacketReceiver` and also to send radiograms through a `PacketTransmitter`. This interface specifies two methods `handlePacket` and `handleSendingFailure`. The first method is called by the `PacketReceiver` to inform the receipt of a radiogram and the second method is used by the `PacketTransmitter` to inform any failure that may occur when sending radiograms.

IAlerter.java: This interface specifies the functionality expected from a class prepared to inform the sending and receiving of data through network connections. For example, if one wants to blink a LED every time a data packet is received or sent, he/she should implement this interface within a class and pass an instance of that class to the `PacketReceiver` or `PacketTransmitter`. The functions specified in this interface are `alertReceived` and `alertSent`, which are self-explanatory.

4.1.3 Secure Channel Components

The last layer, Secure Channel Components, contains the classes of the secure communication protocol designed in this project. As mentioned previously, this secure communication protocol is implemented by combining the networking capabilities of the Network Components layer and the security features of the Security Components layer. The main classes and interface of this protocol are as follows.

- Classes:
    1. `SecureChannelMetadata.java`
    2. `SecureChannelEngine.java`
    3. `SecureChannel.java`
- Interfaces:
    1. `ISecureChannelHandler.java`
    2. `ISecureDataHandler.java`

SecureChannelMetadata.java: This is a wrapper class for data about a secure channel. Although this class is not directly used by a user of the secure communication protocol, it plays a vital role in establishing secure channels between entities. Figure 21 shows a partial class diagram of `SecureChannelMetadata`.

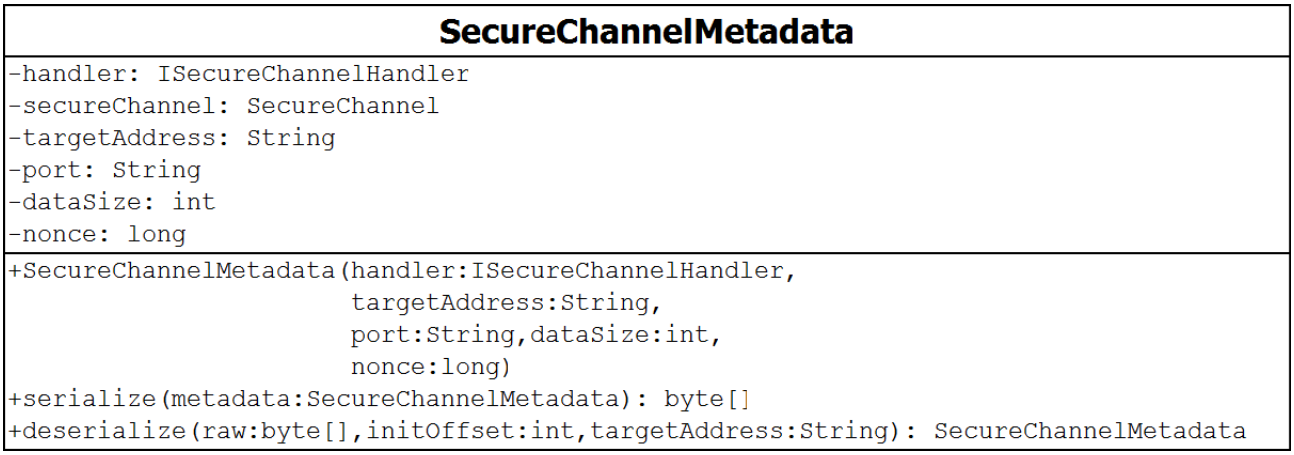


Figure 21: SecureChannelMetadata.java Partial Class Diagram

SecureChannelEngine.java: This class provides the functionality required to establish secure channels between entities by implementing the handshake process of our secure communication protocol. The partial class diagram of this class is shown in Figure 22.

After initializing a `SecureChannelEngine` with proper arguments, an entity can request secure channels from the `SecureChannelEngine` by using the `requestSecureChannel` method. When requesting secure channels, the requesting entity should provide an instance of the `ISecureChannelHandler`, the address of the target entity, the port number to be used for communication, and the maximum size of the radiograms that has to be sent through that secure channel.

Internally, the `SecureChannelEngine` maintains two vectors: `requestedChannels` and `activeChannels`. When a new secure channel is requested, the `SecureChannelEngine` creates an initial `SecureChannelMetadata` instance for that request and saves that instance in the `requestedChannels` vector. Next, the `SecureChannelEngine` starts the handshake process as mentioned in the theoretical concepts section (section 3.1.1). Finally, if the handshake process is successful, the engine:

1. Creates a new instance of the `SecureChannel` class
2. Passes that newly created `SecureChannel` instance to the process (which requested the secure channel) by using the methods specified in the `ISecureChannelHandler.java` interface

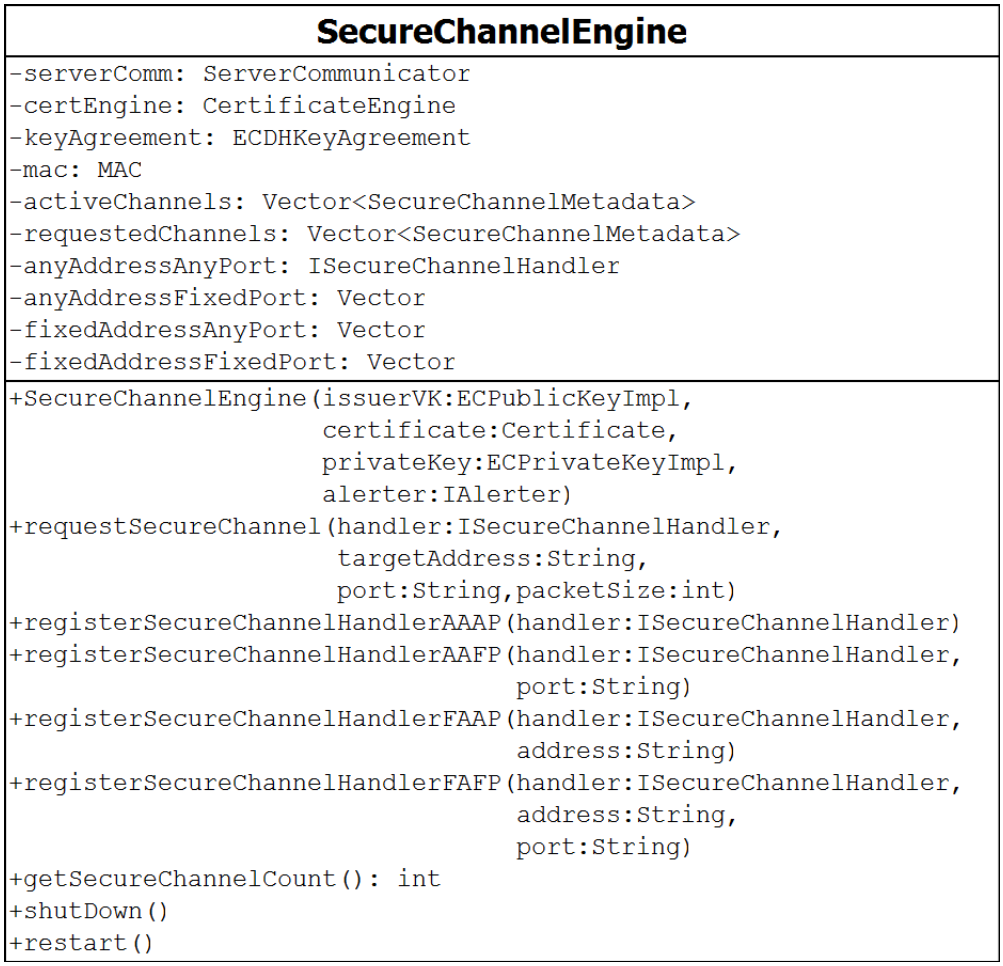


Figure 22: SecureChannelEngine.java Partial Class Diagram

3. Updates the corresponding `SecureChannelMetadata` instance, which is located in the `requestedChannels` vector, with a reference to the newly created `SecureChannel` instance

4. Moves this corresponding metadata object from the `requestedChannels` vector to the `activeChannels` vector

Requesting secure channels is only half the picture. In order for a secure channel to be established, the targeted entity must have registered with its own instance of the `SecureChannelEngine` to accept secure channel requests. As shown in Figure 22, our implementation of the `SecureChannelEngine` provides four methods for entities to register with the engine so that they are able to accept incoming secure channel requests. Let's examine these methods one at a time in the order of increasing specificity.

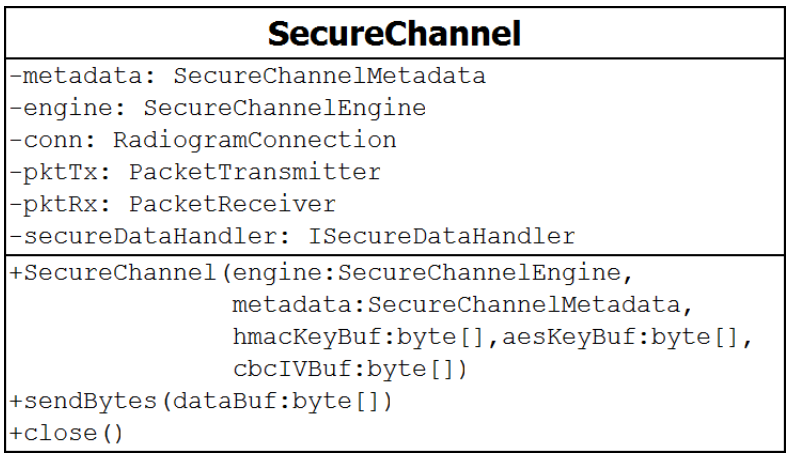


Figure 23: SecureChannel.java Partial Class Diagram

- `registerSecureChannelHandlerAAAP(args...)`, which is the most flexible of all, allows an entity to inform the engine that it's ready to accept secure channel requests from *any address on any port* (AAAP).
- `registerSecureChannelHandlerAAFP(args...)` is used by entities willing to accept secure channel requests from *any address on a fixed port* (AAFP). When using this method, the registering entity should provide the fixed port as one of the arguments to the method.
- `registerSecureChannelHandlerFAAP(args...)` is the fixed address counterpart to the AAFP method mentioned above. This method is more specific since it is used by entities willing to accept secure channel requests from *a fixed address on any port* (FAAP).
- `registerSecureChannelHandlerFAFP(args...)` is the most specific of all. This method is used by entities prepared to accept secure channels from *a fixed address on a fixed port* (FAFP).

The engine internally keeps track of all the active secure channels that it has created. This allows the engine to close all those active channels when the engine is restarted (by a call to the `restart` method) or shutdown (by a call to the `shutdown` method).

SecureChannel.java: This class provides the infrastructure required to send/receive data through a secure communication channel between two entities. The partial class diagram of `SecureChannel.java` is shown in Figure 23.

Sending data is realized through the method `sendBytes` which accepts the data to be sent in the form of a byte array. As mentioned in the theoretical concepts section (section 3.1.2),

the data sent through `sendBytes` is encrypted with AES block cipher in CBC mode. The HMACs of the encrypted data is also attached with the outgoing data packets in order to enforce authenticity of the encrypted data.

When new radiograms with encrypted data arrive at an instance of `SecureChannel`, it first verifies the HMAC of the encrypted data. If the HMAC verification is successful, the secure channel instance decrypts the data and passes that as a byte array to the instance of the class which is prepared to accept data from this secure channel instance. This delegation of received data is achieved via the methods specified in the `ISecureDataHandler` interface, which must be implemented by any class prepared to accept data through `SecureChannel` instances.

Once an entity has finished communicating through a secure channel, it should close that connection by calling the `close` method specified in `SecureChannel.java`. When at least one of the two participating entities closes its secure channel, the `close` method in our protocol implementation ensures that the secure channel is closed from both ends.

4.2 Location

4.2.1 Triangulation

While attempting to implement the triangulation algorithm, further device research revealed that the SunSPOT radios are not suitable for getting accurate distance measurements using RSSI or Distance Bounding, which are necessary for triangulation. This is due to the nature of radio waves, and in particular the radial nature and frequency of the waves transmitted by the SunSPOTs. Radio waves are prone to interference by their surroundings, and as such, signal transmission can be absorbed, deflected, and diffracted by objects encountered during their propagation in space, the amount of each depending on the frequency of the waves and other factors. Particularly troublesome can be the additive and subtractive effects of having the radio waves reach their destination after being partially absorbed or reflected, causing a signal, for instance to wander off when reflected by obstacles, splitting the original signal into multiple routes, and then finally arrive at its destination out of phase with the originally transmitted version. Other factors that affect radio waves include atmospheric conditions present during propagation and the discrepancies between the signals radiating from the SunSPOT during transmission [9]. These factors led us to the conclusion that the SunSPOT by itself would not be adequate for triangulation.

Further research exposed us to the existence of an attachable module built specifically for localization by Texas Instruments. Chipcon CC2431 is a low power, high performance microcontroller which is IEEE 802.15.4 compliant, with a built-in Location Engine and an in-system programmable flash [14]. With some modification, the Chipcon CC2431 microcontroller can be used in conjunction with the SPOTs to significantly enhance the precision of distance estimation.

Figure 24: Chipcon CC2431

The Location Engine in CC2431 is used to estimate the position of nodes in a wireless network (Figure 25). Reference nodes (our observer SPOTs) exist within known coordinates. Blind nodes (our roamer SPOTs) are other nodes which are often mobile, whose coordinates need to be estimated. The Location Engine receives RSSI values from the reference nodes, which will decrease with increasing distance. The main feature of the CC2431 Location Engine is that the location is performed in each blind node, and each localization calculation can use up to 16 reference nodes. As opposed to the SPOTs built-in values which ranges from -60 dBm to 60 dBm, and can be retrieved from the *getRSSI()* function, the Location Engine's RSSI values ranges from -40 dBm to -90 dBm, where -40 dBm is the highest value and corresponds to the signal strength on a distance of one meter.

Localization using CC2431's Location Engine consists of three phases [13]:

1. Broadcast phase: the blind node sends out broadcast messages.
2. Data Collecting phase: the blind node gathers data from each reference node it uses.
3. Position Calculating phase: the blind node calculates its position and transmits it to the Packet Sniffer, which is usually connected to a PC.

The CC2431 itself comes with the AES encryption already implemented for secure data transfers. Therefore, the security part of our project would be implemented only on the SPOTs, and not on the CC2431 microcontrollers - only the localization algorithm would be implemented in the microcontrollers, as illustrated in Figure 26.

Unfortunately, due to time constraints, shipping problems, and the steep learning curve for programming the CC2431 modules, we were not able to implement our localization algorithm to include the CC2431 Location Engine.

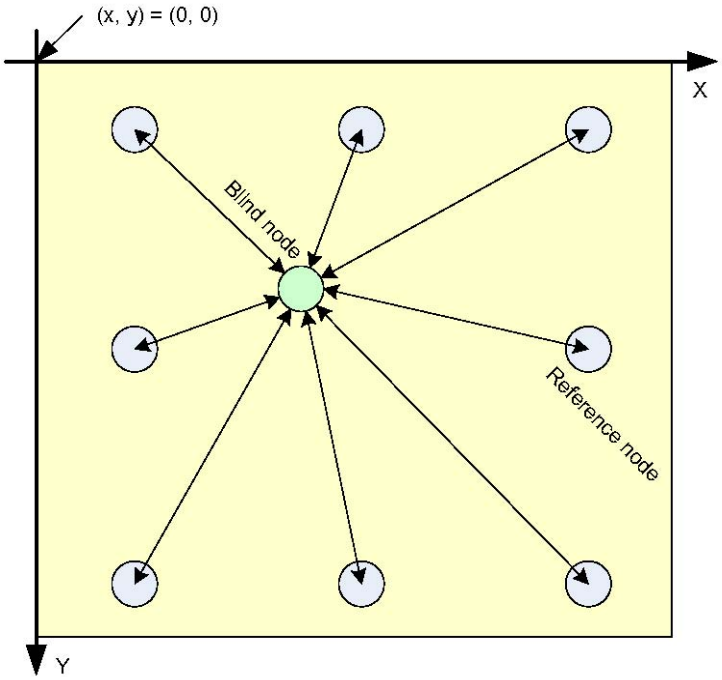


Figure 25: Location Estimation with CC2431

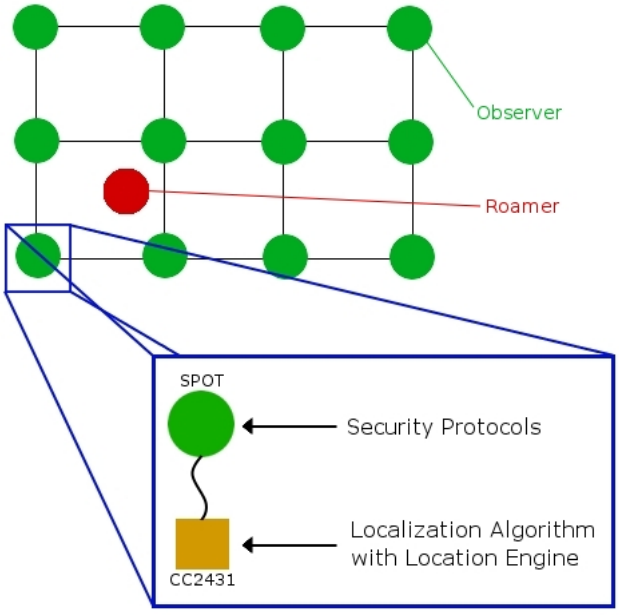


Figure 26: Our Sensor Network with CC2431 modules

4.2.2 Distance-Bounding

In our team's scenario, getting an accurate measure of distance is even more crucial for emergency situations. The distance-bounding protocol was one of the options of the security protocol that was going to be implemented, but in the end the team opted to go with other security protocols as described in the security implementation section. The distance-bounding protocol is basically a combination between the two main parts of our project: security and triangulation. After we found out about the limitations of the SunSPOT devices, we decided on a new scenario: implementing the security and the triangulation components separately, as illustrated in Figure 26. With this new scenario, the team felt that implementing the distance-bounding protocol would pose some redundancies.

4.2.3 Radio Signal Strength Indication (RSSI)

We attempted to roughly estimate distance between two SunSPOTs using RSSI values. At first, we converted the RSSI value to a percentage for easy reading but it turned out that this gave inconsistent values. When two SunSPOTs were placed right next to each other, the RSSI percentage ranged between 60-80%. As they were spread further apart the RSSI percentage decreased, which was expected, but the uncertain dispersion of the radio signal strength as discussed earlier made the percentage readings fluctuate, leading to readings that did not decrease linearly with increased distance.

Trial 1: Distance of about 1 foot apart gave an RSSI percentage value of about 60%. When two SPOTs were spread about 2 feet apart, the readings were lower, close to 40-50% but occasionally, it fluctuated back to 60%.

Trial 2: When the SunSPOTs were 1 foot apart and stationary, the RSSI percentage was consistent. Once an object was introduced between the two SPOTs however, the RSSI percentage greatly decreased. This is evidence of what was explained before: radio interference and propagation.

After these findings, the team attempted to find some estimate of distance despite the revealed inconsistency. Using the raw RSSI values, after numerous trials, a table was created of a range of RSSI values and the *approximate* distance (in feet) between two SunSPOTs. It is also worth noting that the RSSI value we used were taken at an average of 5 readings per second. Every 200ms a packet was sent, the RSSI was computed, and after 5 readings they were averaged and given on a per second basis, as shown in Figure 27.

RSSI Value	>20	$20-12$	$12-9$	$9-4$	$4-(-2)$	$(-2)-(-10)$	<-10
Distance	0.5	1	2	3	4	5	6

Figure 27: RSSI Value Range vs Distance

For the purposes of the tests, we decided to sample RSSI readings at 10 second intervals,

receiving 1 packet per second. For the first set of images (Figure 28 and Figure 29), the SunSPOTs were placed one foot apart and we noticed that over the numerous trials, the RSSI values were initially constant, but they eventually decreased and fluctuated. Thus, the estimated distance readings were more than one foot, up to five feet at one point, even though the SunSPOTs were actually one foot apart.

After a few trials, the readings of distance between two SunSPOTs were consistent and this remained true for up to 4 feet. Once the distance increased beyond 5 feet or more, the RSSI values were so inconsistent that the readings were above 20. The distance estimate output was therefore 0.5ft (6 inches) which was completely off. It was evident that using RSSI would not help us solve localization problems. Using a small, random set of collected data, the figures show a visual representation of the erratic behavior of the SunSPOT's RSSI values and the distance calculated based on preset intervals.

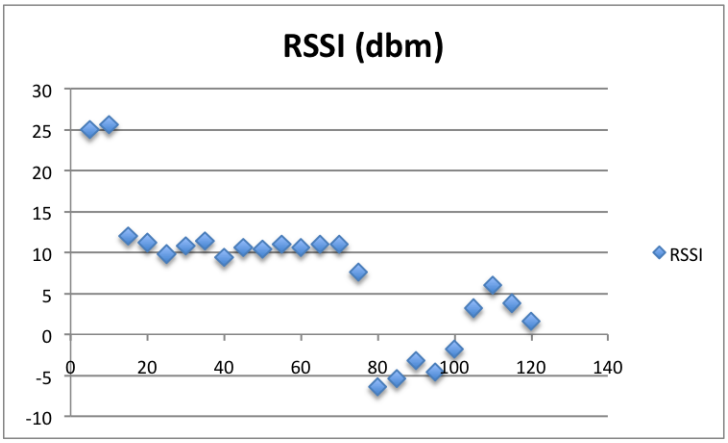


Figure 28: RSSI vs Time (one foot apart)

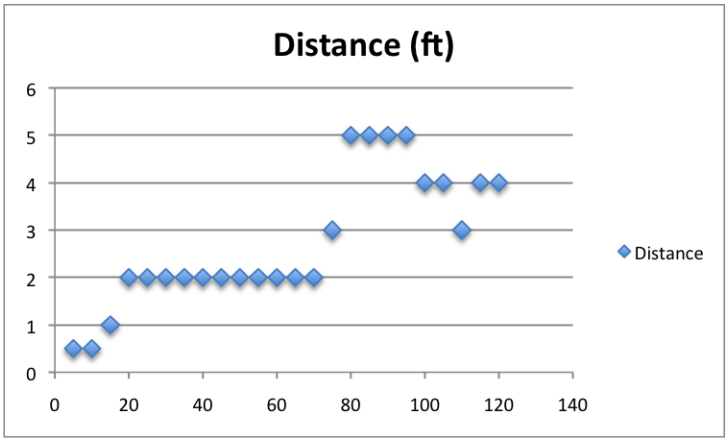


Figure 29: Distance vs Time (one foot apart)

Another trial tested the SunSPOTs at a larger distance. At four feet apart, the readings indicated that the SunSPOTs were 6 feet apart (Figure 30 and Figure 31).

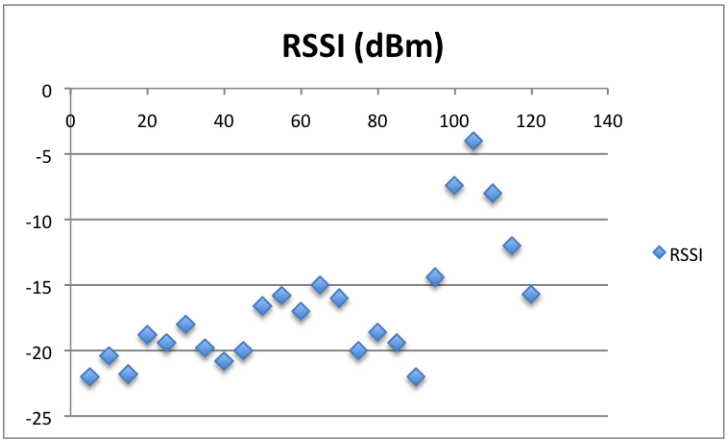


Figure 30: RSSI vs Time (four feet apart)

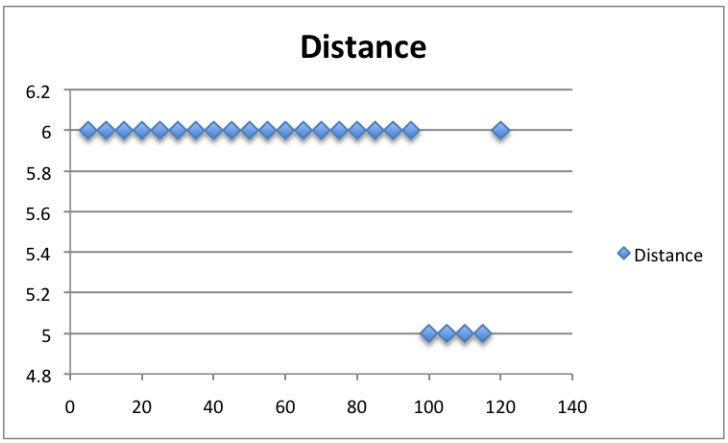


Figure 31: Distance vs Time (four feet apart)

5 Application

As a way to demonstrate some of the results of the research done by the group and to test some theories, we implemented a Java application, using the SunSPOT SDK supplied by Sun Microsystems. The application framework for the SunSPOTs comes in two parts:

SPOT or Device-Based These kinds of applications are designed to be run on the SunSPOT devices. The applications take advantage of the on-board Sqawk Java virtual machine, and use a small subset of the standard Sun Java VM. The devices are designed to be able to run multiple applications, called midlets. Interactivity is achieved using the two switches on the front of the SPOTs. A switch can be pushed to make a choice, or selection. To facilitate ease of use, and efficient application switching, an application was designed to be used as a loader for the numerous midlets running on the device.

Host-Based A host-based application is designed to be run from a computer. Also written in Java, it requires a **base-station**, a SunSPOT that comes without sensors. It is used to communicate with the regular SunSPOTs wirelessly. Due to the fact that a host application runs on a computer, it can be a full fledged Java application, without the space and resource constraints of a device-based application. The host application typically has a graphical user interface to interact with and send commands to the wireless SunSPOTs.

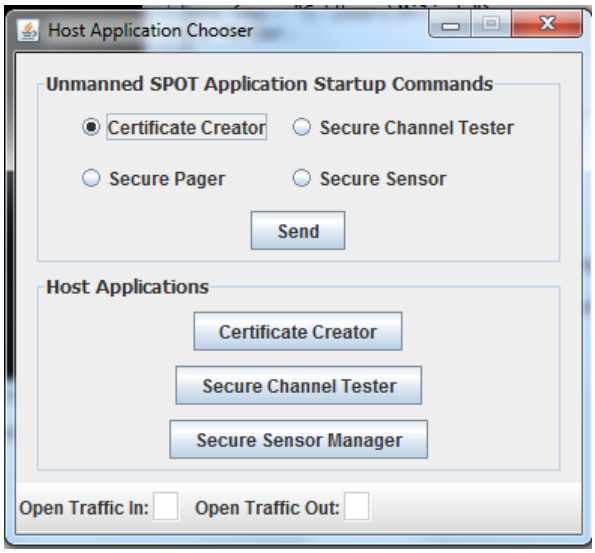


Figure 32: Application Home Screen

The applications created for this project include the "Certificate Creator", an implementation of a certification authority, used to issue and manage certificates to the mobile SunSPOTs that participate in the closed network. Another application is the "Secure Sensor Manager", the main graphical user interface of the application, which implements a subset of the functionality originally envisioned for the full application.

5.1 Certificate Creator

The window for Certificate Creator is divided into two sections. The top of the window displays the name and the verification key of the certificate creator, the host application.

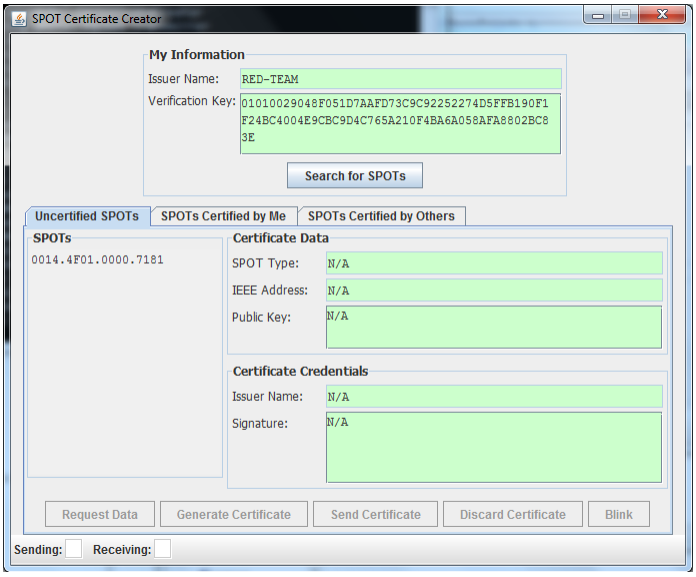


Figure 33: Certificate Creator - Uncertified SPOTs

The bottom of the window is divided into three tabs. The tabs display a list of SPOTs in the vicinity that have certificates issued by the current host application, a list of SPOTs with certificates issued by other certificate authorities and a list os SPOTs without certificates (Figure 33).

Pressing the search button sends out a broadcast signal to all the SPOTs in the vicinity. All SPOTs respond with their certificate status, indicating what type of certificates they have, if any, and who the issuer is.

To issue out a certificate to any uncertified SPOT, the SPOT is selected in the listbox and the public key is requested. On the SPOT, the user gets to select what type of SPOT it is using the buttons on the devices, a roaming SPOT or a stationary SPOT. Once selected, and the information is sent to the host, the administrator at the host can issue out a certificate to the SPOT.

5.2 Secure Sensor Manager

The Secure Sensor Manager, the main window of the application is designed to be used during a rescue session. The window has several buttons. At the beginning of the session, the administrator pings all the SPOTs to be used. The responding SPOTs are listed in the

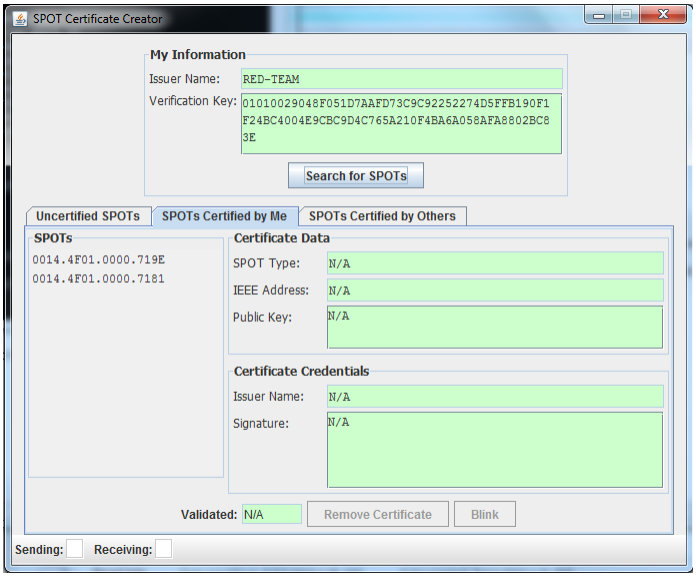


Figure 34: Certificate Creator - Certified SPOTs

grid labeled "Secure Sensors". All SPOTs will need to have secure certificates to be used in the session. The grid also displays the light and temperature readings from the different SPOTs. Before the session begins, all SPOTs connect to the base-station computer using a secure channel as described earlier.

The button "Start Sensing All" enables the display of the sensor data received from the SPOTs in the Sensor Movement window. This windows displays a graph of sensor data, particularly the accelerometer data being sent from all participating SPOTs.

5.2.1 Active Tracking

By default, all participating SPOTs are started in passive tracking mode. This mode sends data every few seconds. When a user is in distress or needs help, the user can push a button on the SPOT to indicate that it needs help, switching that SPOT into active tracking mode. In this mode, any SPOT nearby that decides to engage in a search for the person in distress gets an indicator of how close it is to the user in distress using the lights on the SPOT. The closer the user is to the SPOT in distress, the closer the searcher is, the more lights are displayed on the SPOT. Conversely, as the user moves further away from the SPOT in danger, the lights displayed are reduced.

At the end of the session, all secure channels can be closed and data transmission can be terminated by a command from the secure sensor manager.

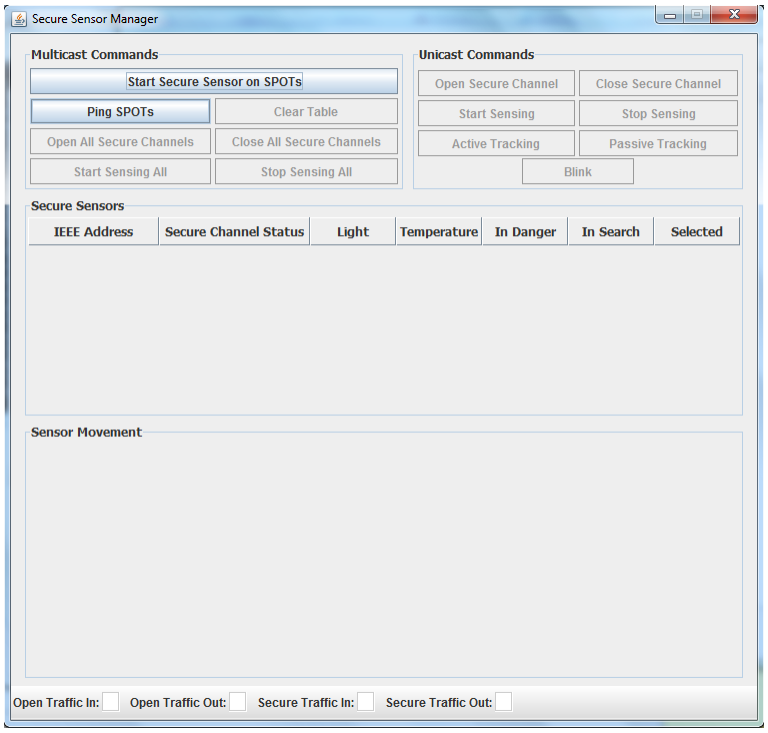


Figure 35: Secure Sensor Manager - Home Screen

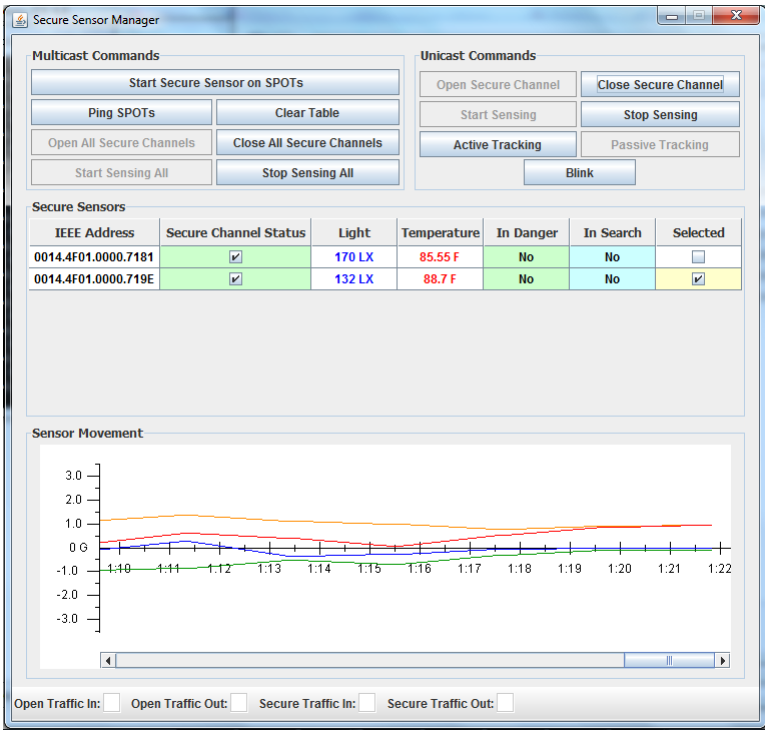


Figure 36: Secure Sensor Manager - Readings

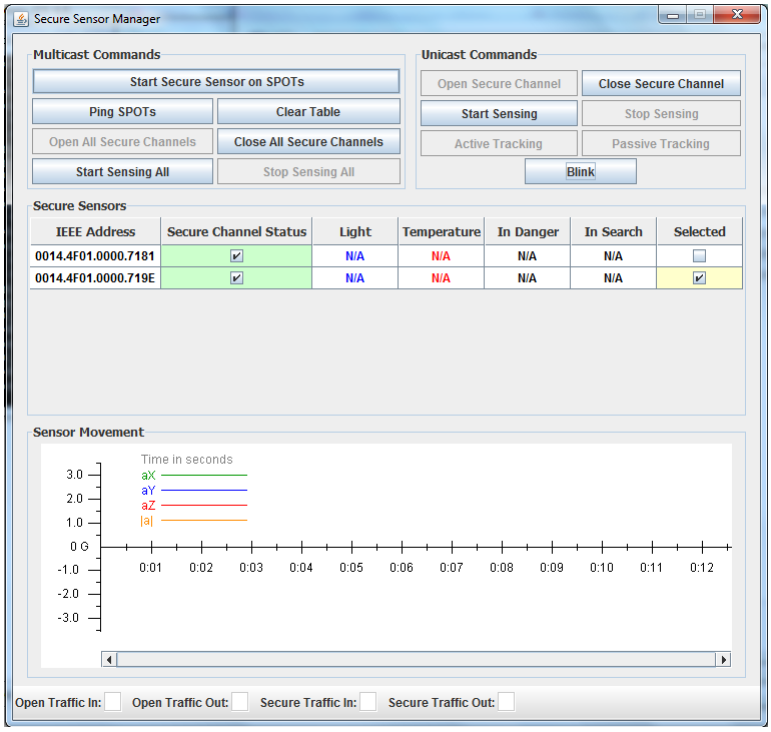


Figure 37: Secure Sensor Manager - Secure Channel

6 Conclusion

The results of the research on this project revealed that it is possible to achieve communications of real-time data in a secure, closed environment under ideal circumstances. The hybrid method of achieving secure communications using both symmetric and asymmetric protocols as implemented in this project is one possible way of achieving this goal successfully.

Our research also revealed that localization and positioning can be achieved using a variety of methods such as triangulation, distance bounding and RSSI. However, the ability to estimate distances is crucial to all of these methods. The SunSPOT radio proved to be unsuitable for this purpose. Further research is necessary to explore the possibility of improved accuracy using additional equipment added to the SunSPOTs, such as the CC2431 location chip.

7 Appendix

7.1 Vocabulary

1. SunSPOT - Sun Microsystems Small Programmable Object Technology
2. Symmetric Cryptography - Encryption methods where both sender and receiver share the same key.
3. Asymmetric Cryptography - Encryption methods where sender and receiver do not share the same key and so they must create keys on-the-fly.
4. Certificate Authority - Gives certificates to verify authenticity of parties. For this project, the base-station was the certificate authority authorizing SunSPOTs.
5. Elliptic Curve Diffie-Hellman Key Agreement - A variation of the Diffie-Hellman protocol based on the structure of elliptic curves over finite fields.
6. Lexicography - Ordered alphabetically.
7. AES - Advance Encryption Standard
8. HMAC - Hash-based Message Authentication Code
9. CBC mode - Cipher-Block Chaining
10. Triangulation - Process of determining the location of a point by measuring angles and distance to it from known, fixed points.
11. Radio Signal Strength Indicator (RSSI) - Measures the strength of signal from one node to another. The range of RSSI for our project was +60 (strong) to -60 (weak).
12. Link Quality - The quality of the received packet. Value ranges from 0 (bad) to 255 (good).
13. Accelerometer - The SunSPOTs are equipped with LIS3L02AQ Accelerometer which measures changes in speed in the X, Y, and Z-axes.
14. Distance Fraud attacks - A scenario where a dishonest prover claims to be closer/further than he actually is.
15. Nonce - A number used once.
16. Reference node - A node located on a static location.
17. Blind node - A node using CC2431 and the Location Engine to calculate its own position.

References

[1] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology—CRYPTO*, pages 1–15, 1996.

[2] J. Blumenthal, R. Grossmann, F. Golatowski, and D. Timmermann. Weighted centroid localization in Zigbee-based sensor networks. In *in IEEE International Symposium on Intelligent Signal Processing—WISP*, pages 1–6, 2007.

[3] S. Brands and D. Chaum. Distance-bounding protocols (extended abstract). In *Advances in Cryptology—EUROCRYPT*, pages 344–359, 1993.

[4] Certicom Research. SEC 1: Elliptic curve cryptography. Technical report, Standards for Efficient Cryptography (SEC), 2000.

[5] W. F. Ehrsam, C. H. W. Meyer, J. L. Smith, and W. L. Tuchman. Message verification and transmission error detection by block chaining, 1976. US Patent 4074066.

[6] R. Goldman. *Using the LIS3Lo2AQ Accelerometer*, 2007.

[7] V. R. Joan Daemen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[8] R. Meike. Location, location, location (accelerometer), November 2008. `http://blogs.sun.com/roger/entry/location_location_location_accelerometer`.

[9] R. Meike. Location, location, location (radio), August 2009. `http://blogs.sun.com/roger/entry/location_location_location_radio`.

[10] V. S. Ronald Cramer. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33:167–226, 2001.

[11] M. Saraogi. Security in wireless sensor networks. Technical report, University of Tennessee Knoxville, 2005.

[12] Sun Microsystems Inc. *Sun SPOT Library API (Javadoc) v5.0*, 2008.

[13] Texas Instruments Inc. *CC2431DK Development Kit User Manual Rev. 1.3*, 2006.

[14] Texas Instruments Inc. *System-on-Chip for 2.4 GHz ZigBee/ IEEE 802.15.4 with Location Engine*, 2009.

[15] J. Undercoffer, S. Avancha, A. Joshi, and J. Pinkston. Security for sensor networks. Technical report, University of Maryland Baltimore County, 2000.