ANALYSIS OF A KEY MANAGEMENT SCHEME FOR WIRELESS MESH NETWORKS

BY

RAVISHANKAR SATHYAM

B.S., University of Illinois at Urbana-Champaign, 2007

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Advisers:

Professor Klara Nahrstedt
Professor William Sanders

ABSTRACT

Wireless mesh networks (WMNs) have emerged as a viable option for many applications. These include home, corporate (including power grid) and automotive monitoring/control, and low cost Internet provisioning. However, security is an important aspect of WMNs, because of applications such as corporate monitoring/control and Internet provisioning. In order to provide security properties such as data confidentiality and integrity, a solid key management scheme (that also satisfies other properties of WMNs such as support for ad hoc networks) is required. This thesis describes the design and implementation of the SMOCK key management scheme, as well as its integration with SMesh, a readily available wireless mesh network software. There are two modes of integration proposed, and experiments were conducted to determine various metrics important to SMOCK. The metrics are parameter generation times, key generation times, round trip times, and encryption times. These experiments determined that the network model (namely the average message size, average hop count of a message, number of nodes in the network, the maximum allowed resilience in the network and the transmit power of the mesh nodes) essentially determines the scenario of integration used with the mesh network.

To my parents, and others who supported my decision

to undertake graduate school

I would like to acknowledge Prof. Klara Nahrstedt for providing the opportunity to work with SMOCK, as well as helping me with the main concepts of this thesis. Also, I want to acknowledge Prof. William Sanders for also supervising my thesis and providing an environment where I can learn from others about security in the power grid. Also, thanks to Nilo Rivera and Claudiu Danilov for helping me with installation and understanding of the SMesh and Spines software. Thanks to Wenbo He and Ying Huang for helping me with the theory behind SMOCK. Thanks to Thadpong Pongthawornkamol and Hoang Nguyen for helping me with the understanding behind ad hoc wireless networks. Finally, I would like to acknowledge funding from the National Science Foundation (the grant number is A5565 NSF CNS 05-24695) which made this thesis possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AKD	area key distributor
AMI	advanced metering infrastructure
AS	authentication server
DKD	domain key distributor
KDC	key distribution center
MANET	mobile ad hoc network
NIC	network interface card
NLOS	non-line-of-sight
P2P	peer to peer
PKC	public key cryptography
RADIUS	remote authentication dial-in user service
SMOCK	Scalable Method of Cryptographic Key Management
TGS	ticket granting server
TTP	trusted third party
VAP	virtual access points
WMN	wireless mesh network

CHAPTER 1

INTRODUCTION

1.1 General Scenario

In recent times, wireless networks have witnessed rapid growth. Telecommunication companies worldwide have been improving the state of wireless networks such that not only can more traffic be carried over these networks than ever before, but the quality of service has also improved. One of the key wireless technologies is wireless mesh network (WMN). Wireless mesh networks comprise of mesh nodes and clients where mesh nodes act both as hosts to the client and also as routers, forwarding packets on behalf of other nodes that are not within direct wireless transmission range of their intended destination [1]. Ad hoc WMNs are dynamically self-organized and self-configured: the mesh nodes in the network automatically establish and maintain mesh connectivity among themselves. Due to this, "mesh networks have the advantage of low up-front cost, easy network maintenance, robustness and reliable service coverage" [1]. Clients of WMNs can vary from desktops, laptops, PDAs, pocket-PCs, phones, and any other mobile device equipped with a wireless NIC. These clients can connect directly to a mesh router. However, clients that are not equipped with NICs can connect to the mesh router via a nonwireless interface such as Ethernet (IEEE 802.3). Also, WMNs can be integrated with various existing wireless networks such as cellular, wireless-fidelity (Wi-Fi), worldwide interoperability for microwave access, wireless sensor and WiMedia networks due to the gateway/bridge functionalities in mesh routers.

In such applications that use a WMN (especially in networks which are bridged or form a gateway to the Internet), security is an important issue. In, fact security is an important issue for ad hoc networks in general. In applications like battlefield and first-responder networks, adversaries can easily affect the network if a security mechanism is not employed [2]. WMNs in particular can be easily compromised due to the "vulnerability of channels and nodes in the shared wireless medium, absence of infrastructure and dynamic change of network topology" [1], [3], [4]. There are many types of attacks specific to a WMN, which can be applicable to various levels in the network stack. Types of attacks in the routing layer can range from advertising routing updates for the wireless routing protocols to erroneous packet forwarding. In the MAC layer, the properties of the IEEE 802.11 protocol (such as back-off) may be misused in a way such that the network is always congested due to some malicious nodes [1], [5]. Furthermore, attackers may also misuse cryptographic primitives [6].

In order to prevent some of these attacks, we need to meet security requirements such as data integrity and confidentiality. Such requirements can be met by a solid key management scheme. Such a key management scheme is needed that can work with preferably low space/time requirements (space is at a premium since many mesh networks feature devices such as cellular phones, sensors, etc., which do not have a large amount of space. Such a key management scheme will help provide confidentiality and data integrity in the mesh network in the face of such attacks, while minimizing memory and processor utilization.

We desire a key management scheme which is scalable to a large number of nodes in the mesh network. Specifically, the storage and communication overhead for the network must be less than *O(n)* [7]. The desired features of the key management scheme for mesh networks are listed:

- *Minimal Overhead* - Communication and memory overhead should be minimal. This ensures that the scheme is compatible with the variety of memory resources that clients of mesh networks possess.

- *Availability* - Nodes and links in a mesh network are prone to failure. However, the scheme should be robust to a fraction of such failures.

- *Resilience* - Nodes in a mesh network (as well as clients) are vulnerable to various attacks. The key management scheme should provide security even when a fraction of the nodes are compromised.

One such scheme is known as SMOCK (Scalable Method of Cryptographic Key Management). The basic idea of SMOCK is inspired by the use of needing multiple keys to open the door to a vault [7]. SMOCK is a combinatorial public key management scheme where nodes use multiple public/private keys to encrypt and decrypt messages, respectively. Each key has a unique ID, and since nodes possess a unique combination of private keys, the concatenation of the private key IDs gives each node its unique SMOCK ID. If a node A needs to encrypt a message to another node B, it sends a request for B's SMOCK ID to infer the set of private keys owned by B [7]. A can then encrypt the message using the public key set that corresponds to the private keys owned by B. SMOCK has been shown to be efficient in simulation results and the goal of this thesis would be to analyze its performance (in terms of speed and memory utilization) on a real mesh network. That is the main contribution of this thesis, as highlighted in the next section.

1.2 Thesis Contribution

As stated before, security is an important issue in wireless mesh networks. WMN's need to meet security requirements such as data integrity and confidentiality, which are met by a solid key management scheme. This thesis first provides a background on key management schemes and an overview on mesh networks. More importantly, the thesis provides a description of the implementation of this scheme, SMOCK, as well as two scenarios of integration with an existing mesh network implementation, SMesh. Furthermore, a performance analysis of the scheme under various data and key sizes as well as various SMOCK parameters, and also various delays between the mesh nodes, is provided.

1.3 Thesis Organization

In the next chapter, an overview on wireless mesh networks is provided. Chapter 3 focuses on several key management schemes, and examines their differences, strengths, and weaknesses. Chapter 4 provides a detailed description of the SMOCK design, as well as the implementation. Chapter 5 describes the architecture of the mesh network used in the experiments, and where SMOCK lies in the big picture. Chapter 6 describes the steps taken to configure each mesh node/client. Chapter 7 describes the experiments that provide an analysis of SMOCK. In addition it showcases the experimental results and provides insight into these results. Finally, Chapter 8 provides the conclusion.

CHAPTER 2

BACKGROUND ON WIRELESS MESH NETWORKS

According to [1], mesh networks have the following characteristics:

- *Multihop coverage*: WMNs help in extending the coverage area of today's wireless standards without reducing the channel capacity. Also, they help in providing NLOS (non-line-of-sight) connectivity. These features are provided in the mesh network due to multihopping capability [8].

- *Ad hoc networking support*: WMNs are self-forming, and provide for self-reconfiguration, self-healing, and self-formation of the mesh network for optimized bandwidth-delay product. This feature also provides for easy deployment and configuration, good fault tolerance properties, high connectivity as well as "multi-point-to-multi-point communications" [1].

- *Limited mobility*: While mesh nodes have little to no mobility, clients can be mobile. However, it is the responsibility of the mesh nodes to perform handoff of clients.

- *Compatibility with existing wireless standards*: WMNs that are built on IEEE 802.11 standards must support all types of conventional Wi-Fi clients. Furthermore, such networks should use the bridging capabilities to interact with nodes that operate on different wireless standards, such as Wi-Max, ZigBee, etc. [1].

- *Multiple types of network access*: Both backbone and P2P communications are supported by WMNs. This helps in creating inexpensive ISP services, as well as integration with other wireless networks [1], [9].

Wireless mesh networks can be organized in three separate ways [1]. They are as follows:

- *Infrastructure/backbone WMNs*: This particular architecture is shown in Figure 2.1 and includes mesh routers that form an infrastructure for clients to connect to them. The mesh routers form a self-configuring, self-healing network among themselves [1]. Mesh routers can use the gateway functionality to connect to the Internet. This network can then serve as a backbone to various clients. This "infrastructure meshing" approach, which provides a backbone for clients, also allows for the integration of WMNs with existing wireless networks through the bridging functionality of the mesh routers. The infrastructure/backbone setup is the most common of the three setups [1] and is used in scenarios such as community and neighborhood networks. While Figure 2.1 shows a mesh router acting as a bridge to a WiFi network, the mesh routers in this setup can act as a bridge to any network (such as wireless sensor network, WiMax, etc.) by establishing a connection with the access point of that network.

- *Client WMNs*: Client WMNs provide a P2P network among the clients [1]. Here, the client nodes themselves constitute the network where they perform the routing and configuration among themselves, while providing application-level services to clients. As compared to an infrastructure WMN, nodes in a client WMN have more requirements since they need to perform routing and self-configuration in the mesh network apart from providing application-level services. An example of a client WMN is shown in Figure

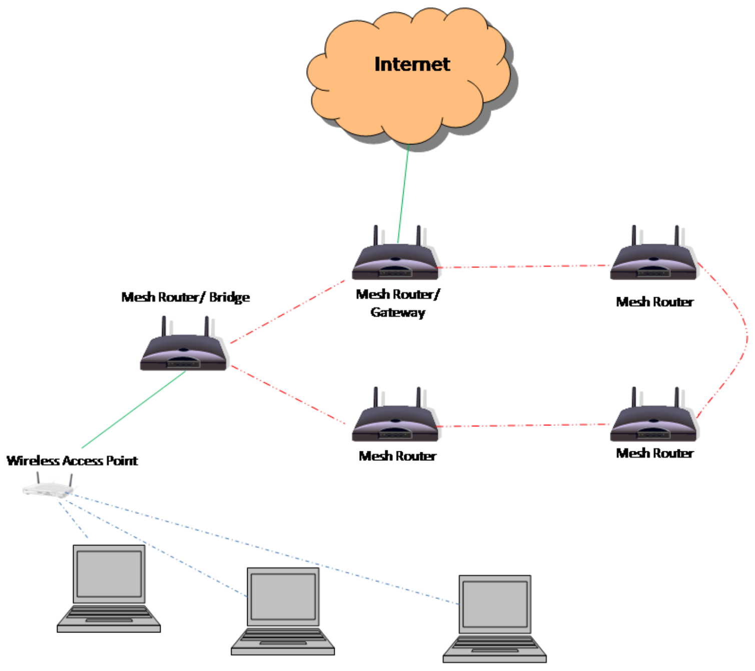

Figure 2.1: Infrastructure/backbone WMN

2.2. It should also be noted that, as compared to the backbone WMN, the client WMN typically does not provide bridging/gateway capabilities, so all the nodes in the mesh network follow the same wireless standard.

- *Hybrid WMNs*: Hybrid WMNs comprise mesh routers which purely perform routing and self-configuration of mesh networks (and provide gateway/bridge functionalities to clients), as well as client nodes which provide application level services to end users while performing routing and self-configuration services to the mesh network. An example of the hybrid network is shown in Figure 2.3.

Given the characteristics and different types of WMNs possible, it is worth noting that they have several applications, some of which include [1]:

- Community and neighborhood networks
- Enterprise networks

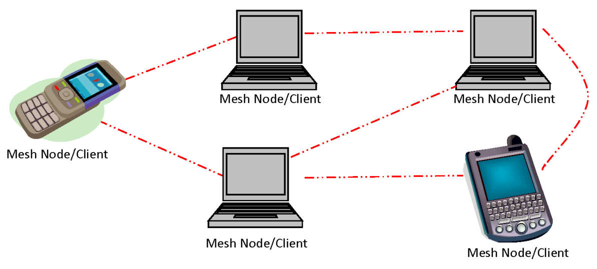

Figure 2.2: Client WMN

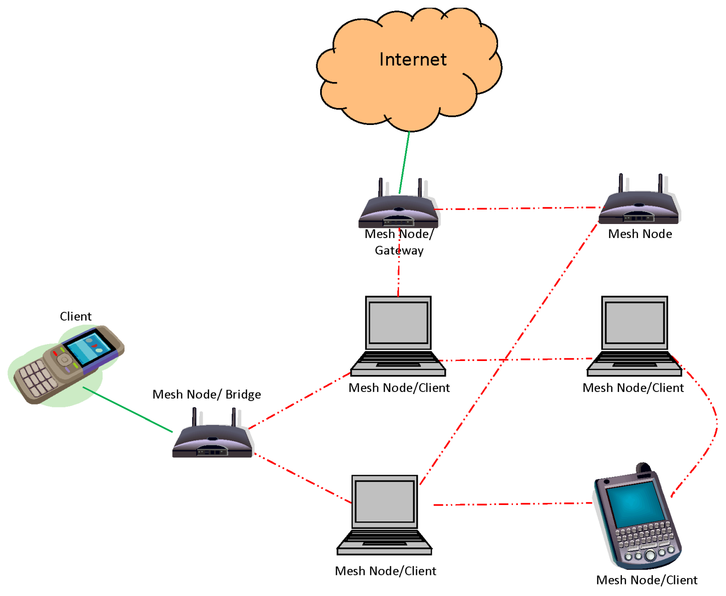

Figure 2.3: Hybrid WMN

- Home networks
- Wide area networks
- Building automation

An example of a mesh network application is its usage in the power grid. One important application is the usage of mesh networks is in the power transmission phase, as part of the Advanced Metering Infrastructure (AMI) [10]. A mesh network can be setup around individual homes in order to collect metering information and send/receive metering data to and from the AMI system, as shown in Figure 2.4. This figure shows an AMI network, where homes have "digitized

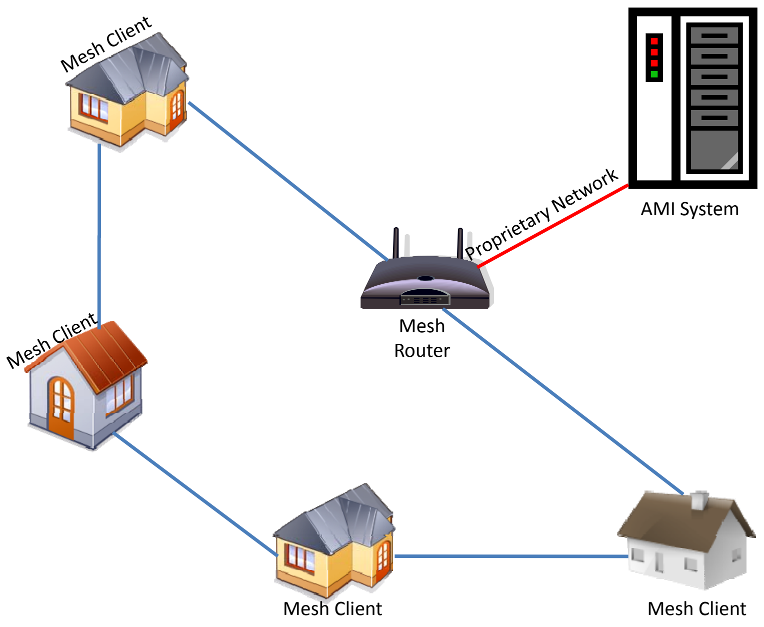

Figure 2.4: Example of a WMN for an Advanced Metering Infrastructure

meters," i.e., advanced meters that contain a wireless component. The meters at these homes are part of a mesh network. Meter data is transferred between the AMI system and the meters via a mesh router, which monitors the meters at the homes and provides a mesh network on which the meters are the clients.

The router provides bridging functionality by bridging the proprietary network (of which the AMI system and the router are a part) and the wireless network (of which the meters at the homes and the router are a part). Thus, mesh networks are useful in a variety of applications, an example of which includes the power grid.

CHAPTER 3

RELATED WORK

Key management schemes are needed to meet security requirements such as data integrity and confidentiality. They can be broadly classified into two types:

- Symmetric key management schemes
- Asymmetric key management schemes

3.1 Symmetric Key Management Schemes

Symmetric key management schemes rely on distribution of symmetric keys (which are generated using symmetric key algorithms such as DES, AES, Blowfish, etc.). There are several such symmetric key management schemes, most of which use a TTP (trusted third party) / centralized server to distribute keys (in these schemes, the secret symmetric keys are predistributed among nodes before deployment). For example, the Kerberos scheme uses a TTP, also known as a key distribution center (KDC). Kerberos is based on the idea proposed by Needham and Schroeder [11], where each client shares a single key with the KDC. When nodes need to communicate with each other, they obtain a new session key, which is encrypted with the preinitialized key from the KDC [2].

The KDC in Kerberos maintains a record of secret keys, and distributes a unique secret key to each node in the network (whether a client or a server) which is known only to the node itself and the KDC. This KDC also generates session keys to facilitate communication between two entities [12]. However, the downside is the fact that nodes have to store a separate session key for every entity in the network, and so the total storage space required is O(n^2) (where n is the number of nodes in the network). Newer protocols have been proposed which attempt to increase the availability of the KDC (for example, via replication) [2]. One such protocol is proposed by Striki and Baras in [13]. Here, data keys are generated using a domain key distributor (DKD). These are used by the session for encrypting data. Each domain is then divided into disjoint areas, and within each area, an area key distributor (AKD) helps distribute the keys to members in each area. However, the performance of such schemes is not satisfactory in terms of efficiency and scalability for ad hoc networks [14]. Moreover, the number of keys stored is again O(n^2) with respect to the number of nodes in the network.

Another scheme is the symmetric key management scheme for sensor networks by Eschenauer and Gligor [15]. In this scheme, a large pool of keys (approximately 2^{17} - 2^{20}) is generated, and a subset of keys is randomly drawn and placed into the sensor ring of each node by a TTP prior to deployment. Deployed nodes then try to either find a common symmetric key, or establish a path through other nodes with which they share symmetric keys. However, such a scheme possesses significant memory issues, since, based on experiments, a key ring size of 200 is needed per node, when the overall pool size is 100 000 keys. Under this constraint, the total space used up by such a scheme is O(n^2). Furthermore, in order to increase the probability of finding a common key between shared nodes, the key ring size of each node must increase, thus increasing memory utilization. This is because if the nodes had smaller key ring sizes, they would then have higher probability of not finding common keys, and then work on establishing a path by finding shared keys with other nodes, thus adding additional space/time cost. Also, we note that since only one shared key is needed for communication between two nodes, it hampers the resiliency of nodes against node-capture. This can be improved by increasing the number of shared keys required between two nodes for communication. This is implemented in the next scheme.

This scheme, which uses a TTP, was proposed by Chan et al., in [16]. The first scheme is called the q-composite scheme, which is quite similar to the scheme proposed in [15]. However, instead of two nodes having just a single common key to communicate with each other, nodes should have a set of common keys in order to communicate with each other. While this drastically increases the resiliency against node capture, it automatically implies that nodes need to have a larger key ring size per node, thus maintaining the total space being O(n^2). The second scheme introduced by the authors is known as the multipath reinforcement scheme. In this scheme, an existing path between two nodes A and B is reinforced by replacing the common key with a random value. This is done by A sending different random values along all the disjoint paths to B, and the final random value being the XOR of all the sent random values. Hence, an attacker then has to probe all the disjoint paths to get the random value between A and B. While this provides additional security between the two nodes, it works better with the traditional scheme presented in [15] rather than the q-composite scheme, thus perhaps defeating the purpose of the q-composite scheme. The third scheme presented in this paper is the random pairwise scheme. While this scheme provides node-to-node authentication, it does not provide a solution to the memory constraint issue.

The problem with random key predistribution schemes is that any two immediate neighbors are connected by a secure link with a certain probability p, and so there is always a probability that the network may not be fully connected (in the aforementioned scheme, this probability increases as q increases) [7]. Furthermore, there is a communication overhead associated with such schemes during the key setup phase. With this in mind, Camtepe et al. proposed a combinatorial distribution of symmetric keys. Given m as a design parameter, the management scheme supports $m^2 + m + 1$ nodes, and the key pool size is $m^2 + m + 1$ [17]. Nodes carry $m+1$ keys and each pair of nodes share exactly one key. If a node is captured, then the probability of a link in the network being compromised is 1/m. However, this scheme does not apply to an arbitrary number of nodes.

Hence, we notice that while symmetric key distribution schemes have the advantage of computational and energy efficiency, they often lag behind in communication complexity. Moreover, they can leave a large memory footprint. A combinatorial approach to symmetric keys has been proposed, but the scheme does not apply to an arbitrary number of nodes. Thus, we look into the alternate, asymmetric key management schemes.

3.2 Asymmetric Key Management Schemes

As the definition implies, asymmetric key management schemes rely on the distribution of asymmetric, or public keys (which are generated using asymmetric key algorithms such as Diffie-Hellman, ElGamal, Elliptic Curve, RSA, etc.). Initially targeted for the Internet ([7], [18]), there have been many schemes which have utilized public key cryptography (PKC). One such scheme is proposed by Zhou and Haas in [19]. In this scheme, there is one public group key K/k and n number of servers (where K is the public component and k is the private component). To tolerate t compromised servers, an $(n,\ t + 1)$ threshold cryptography scheme is used. The public component of the key K is known by all nodes in the network, while the private key k is split into several shares s_1, s_2, $s_3 ... s_n$, one for each server. In order to sign a certificate, servers generate partial signatures for the certificate and submit them to a coordinator (which is a randomly chosen server), which calculates the signature (corresponding to k) from the partial signatures.

In order to improve on the availability of this approach, Kong et al. proposed a distributed scheme where nodes in the network carry a share of the private key [20]. As before, there is one public group key (denoted as SK,PK, where SK is the system private key and PK is the system public key). Using threshold cryptography, SK is shared among all the network entities (each of which has its own certified public key). When a valid certificate is needed, a local group of K secret share holders is created on the fly [20]. Each share holder s_i provides a partial certificate signed by a value derived directly from its secret share P_{s_i}. Once K such partial certificates are collected, they are combined to form the complete certificate (that is signed by SK). While such a service increases the availability of authentication, it also increases the communication overhead for authentication [7].

Another public key-scheme is proposed by Malan and Smith [21] for sensor networks. In this scheme, Elliptic Curve Cryptography (ECC) is used in sensor networks, where it is shown that the computational requirements needed are low. However, for large networks, such a scheme may leave a large memory footprint. Montenegro and Castelluccia propose a scheme that binds node IDs to public keys [22]. In this scheme, the public key is hashed, and the hashed value is then used as a component of the IP address of the node. This removes the need to create certificates that bind the node IDs to their public keys [7].

Capkun et. al. propose a self-organized public key management scheme [23] which allows individual nodes to create, distribute, and maintain their own public/private keys. Once nodes create their public/private keys, they then issue public key certificates on behalf of other nodes. For example, a node u may issue a certificate that binds another node v to the public key K_v using its signature. Neighboring nodes periodically share certificates that they issue and hold, and using this mechanism, certificate repositories are built, enabling nodes to build an incomplete view of the certificate graph. This incomplete view enables nodes to create a certificate chain, which is then used for encryption and authentication purposes. When two nodes need to authenticate each other's public keys, they merge their local repositories and try to find an appropriate certificate chain that makes the verification possible. However, an incomplete graph can sometimes make verifications impossible (if a certificate chain does not exist between two distant nodes, then it is impossible to authenticate their public keys). Also, for a large-scale network, the certificate repositories of each node and the periodic broadcast of certificates by each node add memory and communication overhead respectively.

While the aforementioned schemes focus on providing individual public/private keys to nodes in the network, a lot of research has focused on establishing a shared secret key among nodes in a network. Since the nodes in the network agree on a common group key, such schemes are known as key-agreement protocols. In fact, the Diffie-Hellman protocol (which is used to establish a common key between two nodes) forms the basis for many key-agreement schemes. The scheme proposed by Ateniese et al. extends the Diffie-Hellman key-agreement protocol (which is for two parties) to that of an n-party key agreement protocol. In their scheme, they use the properties of safe primes in generating a public key from a randomly generated secret, and they use this to create the shared key. This contributory scheme (since all parties equally contribute to the key and guarantee its freshness [24]) offers perfect forward secrecy.

A similar scheme is proposed by Becker and Willie in [25], where the authors design a key-agreement protocol (again based on the Diffie-Hellman protocol) that minimizes the number of message rounds needed to establish a key between multiple parties. However, the issue of perfect forward secrecy is not discussed in the scheme. Another non-TTP scheme is proposed by Sherman and McGrew in [26]. This scheme constitutes the construction of a one-way function tree, whose leaves make up the group members (between whom the group key is shared). The internal nodes contain either leaves or other internal nodes as children (each internal node has two children). All nodes contain a node key and a blinded node key. Each internal nodal key is created by applying a one-way function (along with a "mixing function") on its child nodes [26]. The key of the root is the group key of all the nodes in the network.

Another similar key-agreement scheme is proposed by Kim et al. in [27]. This scheme constitutes construction of a key tree, whose leaves consist of member nodes, and the internal nodes whose children consist of leaves and other internal nodes. Each internal node possesses a key which is computed recursively (as well as a hidden blinded key), and the group key is simply the key associated with the root node. The authors consider several tree structures which are efficient with respect to the number of group operations such as member add/delete and group merge/partition [2]. While such schemes are efficient with regard to communication complexity, the derivation of the key at each internal node is a computationally intensive process.

While such schemes are distributed and do not need a centralized server/TTP, they have several disadvantages when it comes to implementation over an ad hoc mesh network [2]. They are as follows:

- In most of the key agreement schemes, several nodes need to be within direct contact of each other. However, in an ad hoc network, this is not feasible.
- Ad hoc networks consist of mobile nodes which may move rapidly, thus changing the topology. Before the key-agreement protocol has been carried out between any two clients, the clients may already have changed location

and no longer be in contact. So, the time taken to successfully establish a group key in such schemes is a big hindrance when it comes to ad hoc networks in general [2].

- Nodes in the network may be susceptible to attackers. If a node is captured, then the group key is exposed, thus compromising all links. Hence such schemes are not resilient.

In summary, key management schemes come in two basic flavors. Many symmetric key management schemes rely on random predistribution. This may result not only in a large memory overhead, but also in a probability that the graph is not fully connected, and hence pairs of nodes may not share any common keys. Another limitation is the communication overhead during the key setup phase [7]. Most of the proposed asymmetric key management schemes are decentralized (the centralized key management scheme proposed in [19] only provides message authentication as a service); the key distribution schemes suffer from extra memory/communication overhead, while the key-agreement schemes are not resilient. Furthermore, they are not applicable to ad hoc wireless mesh networks whose clients can be highly mobile. With this in mind, in the next chapter we cover SMOCK, a centralized asymmetric key management scheme that focuses on small memory footprint (at the cost of communication overhead) and increased resilience against Sybil attack.

CHAPTER 4

SMOCK DESIGN AND IMPLEMENTATION

We need a key management scheme that can provide confidentiality and authenticity for a mesh network. It should also be flexible for two scenarios, namely integration with the underlying Spines software in each mesh node, or integration with each mobile client in a mesh network. In both scenarios, the mesh network under consideration is an ad hoc wireless network. Our network model assumes that the mesh nodes are static, and the mesh topology does not change. However, the clients are mobile, and they are seamlessly transferred from one mesh node to other. Any client is connected to a single mesh node, and all packets generated by a client are sent to its mesh node, which then forwards it appropriately. Packets intended for a particular client are first sent to its mesh node, which then forwards it to the client. We assume that one of the clients in the mesh network is static, and plays the role of a centralized server which creates and distributes SMOCK keys.

These mesh nodes are organized in a ring topology, and so the scheme must be applicable to a multihop network. From a security point of view, we assume that any attacker from the outside of the mesh network has finite memory and computational power. Furthermore, we assume that attackers can get hold of the private keys of any node (by physically attacking the device). We also assume that the attacker can perform the following attacks:

- Eavesdropping: An attacker (or any unauthorized recipient) can intercept any message.

- Spoofing: An attacker may fake the message or a node's ID.
- Denial of service (DoS): A malicious mesh node, client, or an attacker from outside of the network can inject a significant amount of data traffic into the network to clog the network [7].
- Sybil attack: The attacker fakes keys which she can circulate within the network.

For each scenario we make certain different assumptions about the computational and security model, which we highlight below.

- **Integration of SMOCK with clients of mesh network**

  In this scenario, we integrate SMOCK with the clients of the mesh network. These clients may be mobile, and they send packets to each other via the mesh network. In this scenario, we make no assumptions either about the number of mesh nodes or about any computational/memory constraints for these nodes. We also assume that the number of clients as well as the size of the mesh network may change dynamically. However, we assume that while the mobile clients may vary in amount of memory they possess, they all have reasonable computational power in order to encrypt/decrypt data multiple times using public/private keys. From the security point of view, we assume that both clients and mesh nodes may be malicious.

- **Integration of SMOCK with mesh nodes**

  In this scenario, we integrate SMOCK with the mesh nodes. In this scenario, we make no assumptions either about the number of mobile clients or about their computational/memory constraints. However, we assume that the number of mesh nodes is relatively small, since each mesh node stores the SMOCK ID (apart from IP address, signal strength, and other data) of all other mesh nodes. We also assume that the number of clients as well as the size of the mesh network may change dynamically. However, we assume that while the mesh nodes may have an arbitrary amount of memory, they all have reasonable computational power in order to encrypt/decrypt data multiple times using public/private keys. From the security point of view, we make no assumptions about the maliciousness of clients. However, we assume that mesh nodes may not be malicious. This is because a packet sent over multiple hops is decrypted and then encrypted at each hop in the mesh network (as explained in the next chapter). Thus mesh nodes have access to the plaintext packets sent by the client. Thus, while SMOCK in this scenario protects against attackers from outside the mesh network (or against malicious clients), it does not account for malicious mesh nodes. Finally, we also assume that the link between a client and mesh node is protected using a shared symmetric key.

The basic idea of SMOCK is to have multiple keys in order to unlock the door to a vault [7]. In this scheme, multiple keys are used to encrypt a message, and the designated recipient uses multiple keys for decryption. Public-key cryptography is used for SMOCK since this provides minimal memory overhead. Nodes contain all public keys and a unique combination of private keys. An example of the SMOCK key management scheme is shown in Figure 4.1. In this figure, the total number of nodes in the network N is 10 (Node 1, Node 2,..., Node 10), while the key pool K contains $< priv_A, pub_A >$, $< priv_B, pub_B >$, $< priv_C, pub_C >$, $< priv_D, pub_D >$ and $< priv_E, pub_E >$, and key pool size $|K|$ is 5. The number of private keys assigned to each node $|K_i|$ is 2. Hence, each node i has a unique combination of two private keys. For example, Node 8 contains two private keys, namely $< priv_C, priv_D >$. Each node possesses a SMOCK ID, which is unambiguously associated with its set of private keys. For example, Node 1's SMOCK ID would

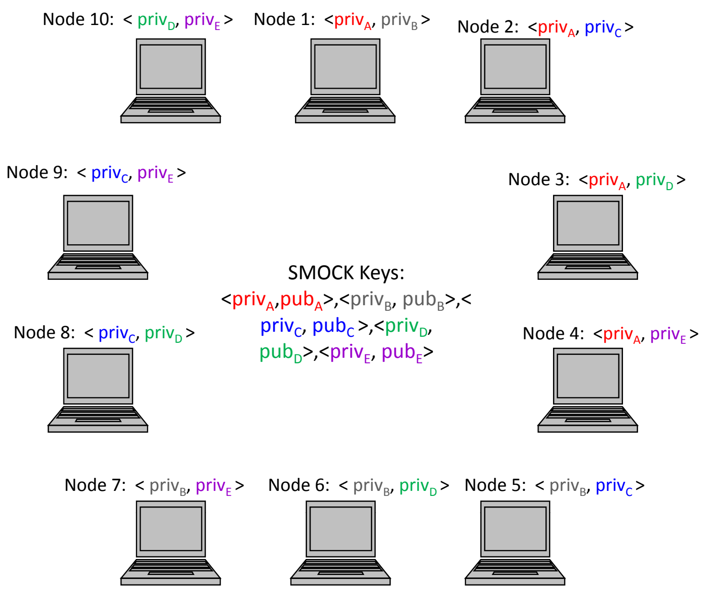

Figure 4.1: SMOCK key setup

be "12" (the concatenation of "1" and "2," which are the key ID's corresponding to $priv_A$ and $priv_B$, respectively). The size of the subsets of private keys at each node is the same. For example, we assume that a system of N nodes contains a key pool K of public-private key pairs. This key pool is generated by the SMOCK key allocation mechanism (which is explained in Section 3.2), which runs on a centralized server. This mechanism also allocates SMOCK public/private key pairs to nodes. Let $< K_A^{priv}, K_A^{pub} >$ and $< K_B^{priv}, K_B^{pub} >$ denote the private and public key sets held by nodes A and B, respectively. Then, the following properties hold:

$$|K_A^{priv}| = |K_B^{priv}| \tag{4.1}$$

$$|K_A^{pub}| = |K_B^{pub}| \tag{4.2}$$

Table 4.1: SMOCK variables

K	Key pool of the system
a	The total number of public keys in the system
b	The number of private keys held by each node
K_i^{priv}	Set of private keys held by node i
K_i^{pub}	Set of public keys held by node i
M	Memory size for key storage
$k_c(x)$	Expected number of disclosed keys when x nodes are captured
$k_v(x)$	Maximum number of disclosed keys when x nodes are captured
$V_x(a,b)$	Vulnerability metric when x nodes are captured
$C(a,b)$	Combinatorial of a and b
N	Total number of nodes in the network
V	A set of nodes in the ad hoc wireless network

$$K_A^{priv} \neq K_B^{priv} \forall B \in N \tag{4.3}$$

Furthermore, multiple copies of the same private key can be held by different users. The key allocation mechanism of SMOCK needs to ensure that these properties are always followed. Given properties 4.1, 4.2, and 4.3, SMOCK aims to achieve objectives which are discussed in the next section. The symbols used in the following chapters are explained in Table 4.1.

4.1 Objectives of SMOCK

- **Memory Efficiency** - The SMOCK key allocation scheme has to generate a key pool K to achieve the following constraint [7]:

$$\min(|K| + \max_{\forall i \in V} b_i) \textit{ such that } K_i \nsubseteq K_j, K_i \nsupseteq K_j \ \ \forall i \neq j \tag{4.4}$$

  where $b_i = |K_i| = |K_i^{priv}|$ is the number of private keys stored at Node i. Here, $(|K| + \max_{\forall i \in V} b_i)$ is the maximum amount of memory stored among all nodes, and this quantity needs to be minimized.

- **Computational Complexity** - Each node needs to minimize the number of public keys needed to encrypt any outgoing message, as well as a small number of private keys to decrypt the outgoing message [7]. This results in the following constraint:

$$\min(\max_{\forall i \in V}(b_i)) \textit{ such that } K_i \not\subseteq K_j, K_i \not\supseteq K_j \quad \forall i \neq j \tag{4.5}$$

  Here, $\max_{\forall i \in V}(b_i)$ is the maximum number of private keys stored at a node (amongst all nodes), and this quantity should be minimized.

- **Resilience Requirement** - The SMOCK key allocation scheme needs to determine the proper ratio between the number of private and public keys in order to ensure that the percentage of links compromised (when nodes are captured) is kept below a certain threshold. Reference [7] defines a vulnerability metric $V_x(a, b)$ as the percentage of communications compromised when x out of N nodes are captured. In other words,

$$V_x(a, b) = \frac{C(k_c(x), b)}{C(a, b)} \leq \wp \tag{4.6}$$

  where $\wp$ is the threshold on the number of compromised communications when x nodes are randomly captured by an attacker from outside the network. The value of $k_c(x)$ is determined as follows. First, an assumption is made that the server randomly chooses private keys for each node (with no bias) [7]. For the first node captured, b keys are revealed and so $k_c(1) = b$. If x nodes are captured, then the number of new keys disclosed k_{new} is

$$k_{new} = b * \frac{a - k_c(i-1)}{a} \tag{4.7}$$

So, when the i'th node is captured, the total of $k_c(i)$ keys are disclosed, where

$$\begin{aligned} k_c(i) &= k_c(i-1) + bk_new \\ &= b * \frac{a - k_c(i-1)}{a} \end{aligned} \tag{4.8}$$

This implies that

$$\begin{aligned} k_c(i) = \frac{a-b}{a} k_c(i-1) + b \\ k_c(1) = b \end{aligned} \tag{4.9}$$

Let $y_i = k_c(i) - a$. Then

$$\begin{aligned} y_i = \frac{a-b}{a} y_{i-1} \\ y_1 = b - a \end{aligned} \tag{4.10}$$

So,

$$y_i = (\frac{a-b}{a})^{i-1} y_1 \tag{4.11}$$

This implies that

$$k_c(i) = a - (a-b) * (\frac{a-b}{a})^{i-1} \tag{4.12}$$

So, we know that if x nodes are captured, then the expected number of keys disclosed $k_c(x)$ is $a - (a-b) * (\frac{a-b}{a})^{i-1}$. The capture of x nodes results in the compromise of $C(k_c(x), b)$ key sets on average, and $C(k_v(x), b)$ key sets in the worst case [7]. Hence, the vulnerability metric $V_x(a, b)$ is $\frac{C(\lfloor k_c(x) \rfloor, b)}{C(a,b)}$

in the average case and $\frac{C(k_v(x),b)}{C(a,b)}$ in the worst case.

The SMOCK key allocation scheme should thus achieve the aforementioned objectives, while adhering to the properties mentioned before. The next section describes this scheme.

4.2 SMOCK Key Allocation Scheme

The SMOCK key allocation scheme first determines the values of a and b in order to meet the objectives highlighted in the previous section. Then mutually exclusive key sets are distributed (on a secure channel) to each node.

4.2.1 Determination of a and b

The memory efficiency objective highlighted earlier requires $\frac{a}{N}$ to be small (for memory efficiency). However, this conflicts with the resilience requirement, which requires $\frac{a}{b}$ to be large. So, [7] provides the algorithm shown below which generates the value of a and b while providing a tradeoff between the memory and resilience requirements.

```
1: Initialize $l = 2$
2: **while** $C(l, \lfloor \frac{l}{2} \rfloor) < N$ **do**
3:     $l = l + 1$
4:     $a = l$, $b = \lfloor \frac{l}{2} \rfloor$
5: **end while**
6: **while** $C(a, b - 1) > N$ **do**
7:     $b = b - 1$
8: **end while**
9: **while** $C(a + 1, b - 1) > N$ **do**
```

```
10:     $a = a + 1, b = b - 1$
11: **end while**
12: **while** $V_x(a, b) \leq \wp$ **do**
13:     **if** $C(a + 1, b - 1) > N$ **then**
14:         $a = a + 1, b = b - 1$
15:     **else**
16:         $a = a + 1$
17:     **end if**
18: **end while**
19: $|K| = a$ and $|K_i| = b$
```

This algorithm first calculates the minimum amount of memory needed to store public keys in order to support secure communication among N nodes. Then, the algorithm aims to satisfy Equation (4.4) in Step 2. Step 6 then optimizes Equation (4.5) while keeping Equation (4.4) intact. Step 9 ensures that the key allocation scheme satisfies Equation (4.6) [7]. When a and b do not satisfy the resilience requirement, then either a is increased, or a is increased and b is decreased simultaneously.

However, often the maximum amount of memory needed to store the public keys is limited by M. Reference [7] also provides an algorithm which fully utilizes the memory provided to optimize Equations (4.5) and (4.6). This algorithm is stated as follows:

```
1: Let $a = \lceil \frac{2M}{3} \rceil, b = \lfloor \frac{M}{3} \rfloor$
2: **while** $C(a + 1, b - 1) > N$ **do**
3:     $a = a + 1, b = b - 1$
4: **end while**
5: $|K| = a$ and $|K_i| = b$
```

In this way, a and b parameters are determined by a centralized server. The server then generates a number of keys and assigns a unique ID to each key. Finally, it distributes mutually exclusive key sets to nodes.

4.2.2 Key distribution to nodes

After key generation, the centralized server then waits for new nodes in the network. Given a and b, the maximum number of nodes that can be supported (assigned unique SMOCK key subsets) is $C(a, b)$. However, if additional nodes join the network, the centralized server then generates additional keys ($\acute{a}$ more keys) and assigns a unique subset of b keys to each additional new node (each subset must assign at least one out of $\acute{a}$ keys). Each new node broadcasts the value of $\acute{a}$ as well as the $\acute{a}$ public keys to all the other nodes in the network. It should be noted that with $\acute{a}$ new pairs, the network can accommodate $\sum_{i=0}^{\acute{a}} C(a+i, b-1)$ additional nodes [7].

As new nodes enter the network, they communicate with the server in order to receive their SMOCK keys. The server designates a unique subset of keys (each subset being of size b) and assigns this subset to the new node. It should be noted that any given key can be assigned to at most $\frac{b}{a}C(a, b)$ nodes (this ensures that $C(a, b)$ nodes can be successfully assigned unique subsets of SMOCK keys). In addition, the server also generates a unique SMOCK ID for each node (this ID is unique as long as the key sets are mutually exclusive). The SMOCK ID is a composite key index, since it is comprised of the indexes of the individual keys. In other words, this ID is simply an ordered concatenation of the IDs of the private keys that the node possesses. For example, let $keyID_1{}^j$,$keyID_2{}^j$,...,$keyID_b{}^j$ be the key IDs of b private keys held by node v_j, such that

$$keyID_1{}^j < keyID_2{}^j < ... < keyID_b{}^j \tag{4.13}$$

The SMOCK ID of v_j would then be $keyID_1{}^j keyID_2{}^j ... keyID_b{}^j$. For example, if $b = 3$ and a node v_j had keys with IDs 2, 5, and 3, respectively, then v_j's SMOCK ID would be "235." According to [7], the ID of each key takes $\lceil \log a \rceil$ bits, and thus the SMOCK ID would take $b\lceil \log a \rceil$ bits. Another example of the key generation and allocation scheme is shown in Figure 4.2. In this case, $b = 2$, and a node is assigned the private keys $priv_A$ and $priv_E$ which have ID's "1" and "5," respectively, thus giving it the ID "15." Nodes then can use the SMOCK keys to provide either confidentiality or integrity. The next subsection provides the usage of SMOCK keys in individual nodes.

4.3 SMOCK Usage

Individual nodes can use their SMOCK keys to either prevent eavesdropping and protect privacy or provide integrity [7]. Assume that a node v_i wants to send M_{ij} to node v_j. Then, v_i follows the following algorithm:

```
1: Request $v_j$ for its SMOCK ID.
2: After receiving $v_j$'s SMOCK ID, encrypt with the corresponding keys.
3: Send encrypted message to $v_j$.
```

The encryption algorithm is based on whether the nodes prefer data confidentiality or authenticity. In order to provide data confidentiality, v_j uses the following equation for encryption:

$$E_{ij} = Enc(...(Enc(Enc(M_{ij}, K_{j1}^{pub}), K_{j2}^{pub}), ...K_{jb}^{pub}) \tag{4.14}$$

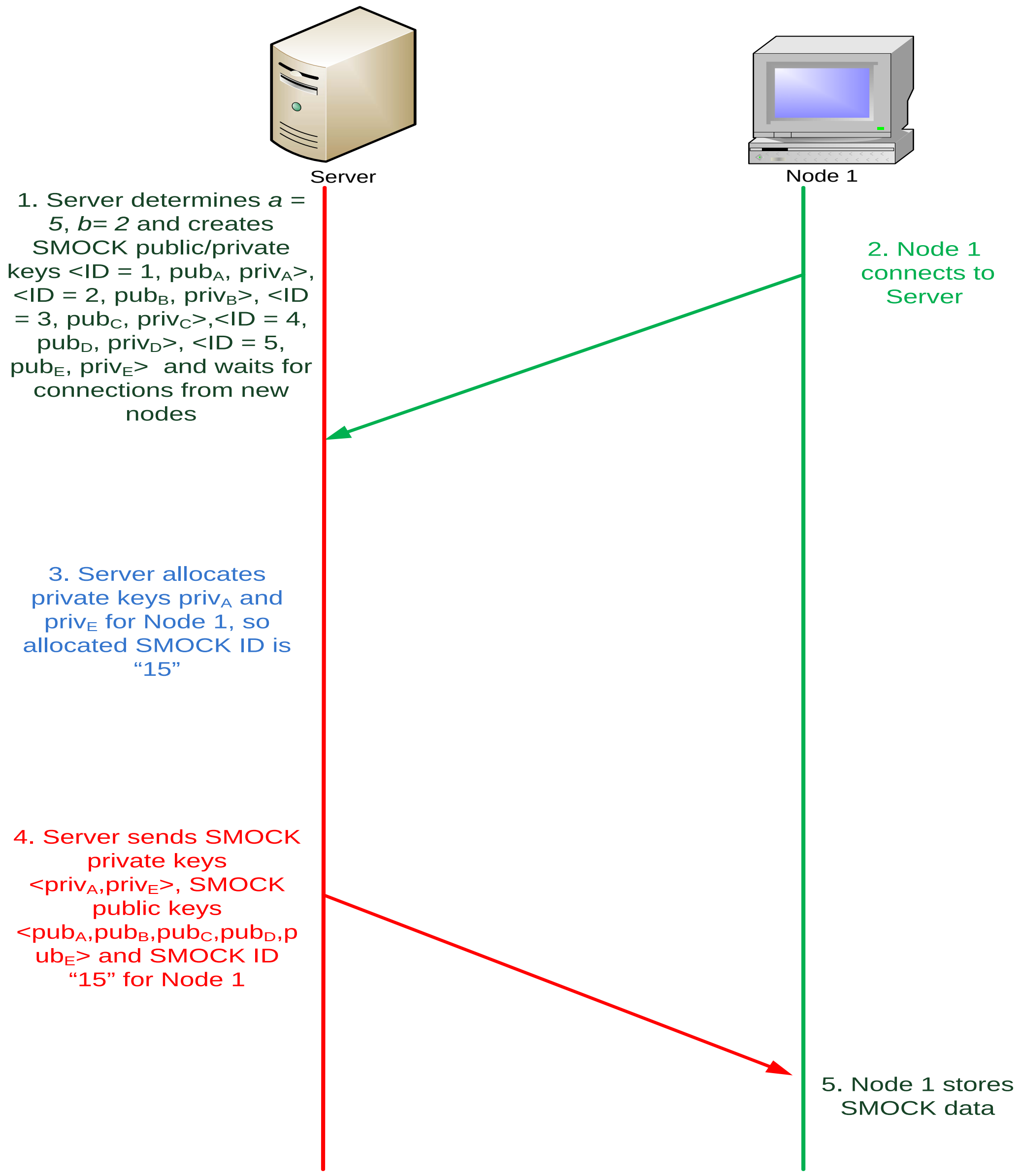
Server
Node 1
1. Server determines *a* = *5*, *b*= *2* and creates SMOCK public/private keys <ID = 1, pub_A, $priv_A$>, <ID = 2, pub_B, $priv_B$>, <ID = 3, pub_C, $priv_C$>,<ID = 4, pub_D, $priv_D$>, <ID = 5, pub_E, $priv_E$> and waits for connections from new nodes
2. Node 1 connects to Server
3. Server allocates private keys $priv_A$ and $priv_E$ for Node 1, so allocated SMOCK ID is "15"
4. Server sends SMOCK private keys <$priv_A$,$priv_E$>, SMOCK public keys <pub_A,pub_B,pub_C,pub_D,pub_E> and SMOCK ID "15" for Node 1
5. Node 1 stores SMOCK data

Figure 4.2: SMOCK offline key deployment phase

For decryption, v_j uses the following equation:

$$M_{ij} = Dec(...(Dec(Dec(E_{ij}, K_{jb}^{priv}), K_{j(b-1)}^{priv}), ...K_{j1}^{priv}) \tag{4.15}$$

On the other hand, in order to provide integrity, v_i encrypts using the following equation:

$$E_{ij} = Enc(...(Enc(Enc(M_{ij}, K_{i1}^{priv}), K_{i2}^{priv}), ...K_{ib}^{priv}) \tag{4.16}$$

Then, v_j uses the following equation for decryption:

$$M_{ij} = Dec(...(Dec(Dec(E_{ij}, K_{ib}^{pub}), K_{i(b-1)}^{pub}), ...K_{i1}^{pub}) \tag{4.17}$$

An example of this is shown in Figure 4.3, where two nodes (Node 1 and Node 2) interact with the "Server" entity (from Figure 4.2) and receive their two SMOCK private keys, 5 SMOCK public keys and SMOCK ID. Here, Node 1 receives $priv_A$ and $priv_E$ and hence its SMOCK ID is "15." On the other hand, Node 2 receives $priv_B$ and $priv_C$, respectively, and hence its SMOCK ID is "23." So when Node 1 wishes to encrypt a message M_{12} to Node 2, it acquires Node 2's SMOCK ID and then encrypts with the appropriate public keys. In this case, the encryption equation is

$$E_{12} = Enc(Enc(M_{12}, pub_B)pub_C) \tag{4.18}$$

So, the message is first encrypted with pub_B and then with pub_C. The message is decrypted in reverse order at Node 2. The decryption equation in this case is

$$M_{12} = Dec(Dec(E_{12}, priv_C)priv_B) \tag{4.19}$$

So, the message is first decrypted using $priv_C$ and then using $priv_B$.

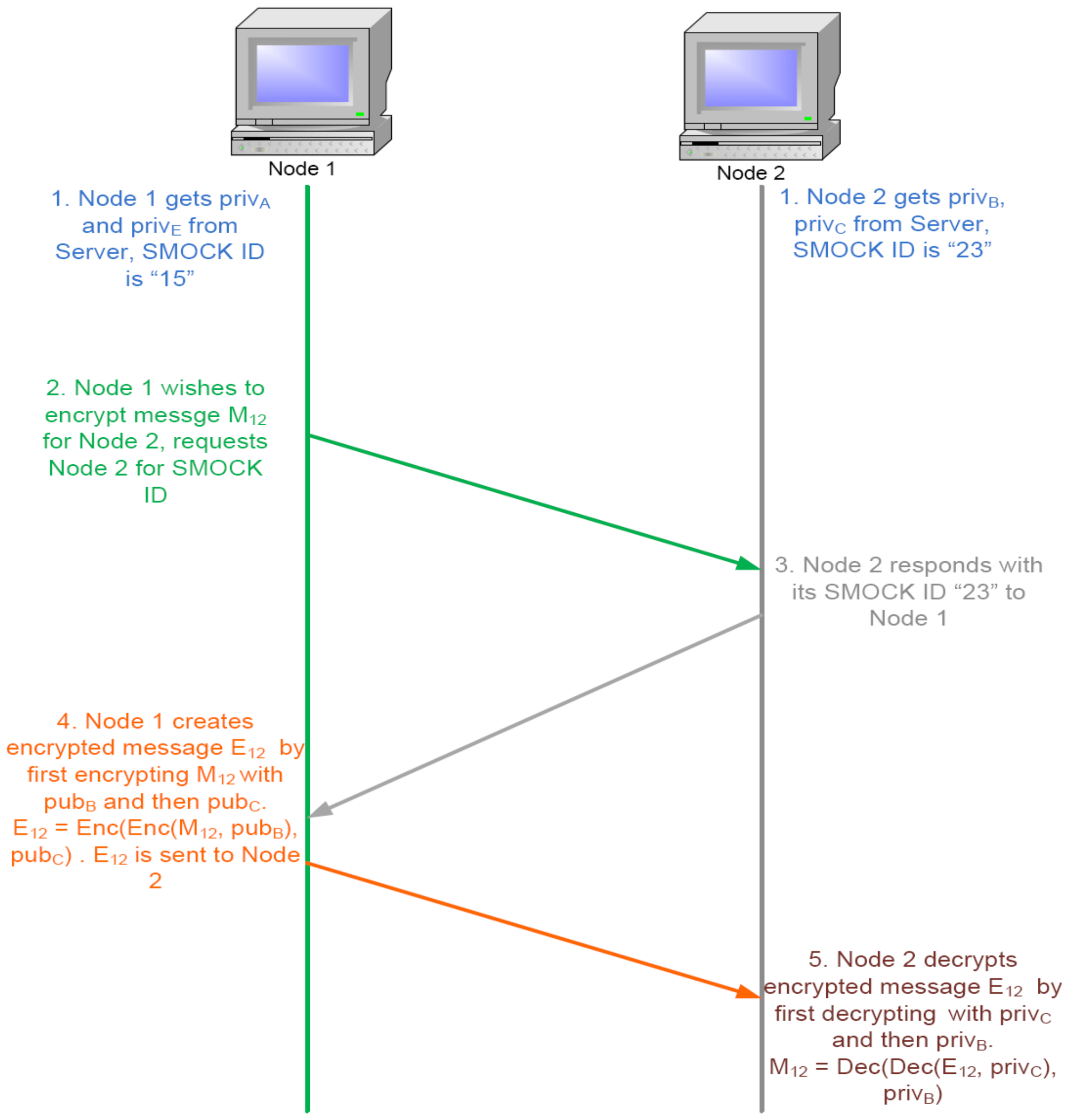
Node 1
Node 2
1. Node 1 gets $priv_A$ and $priv_E$ from Server, SMOCK ID is “15”
1. Node 2 gets $priv_B$, $priv_C$ from Server, SMOCK ID is “23”
2. Node 1 wishes to encrypt messge M_{12} for Node 2, requests Node 2 for SMOCK ID
3. Node 2 responds with its SMOCK ID “23” to Node 1
4. Node 1 creates encrypted message E_{12} by first encrypting M_{12} with pub_B and then pub_C. $E_{12} = Enc(Enc(M_{12}, pub_B), pub_C)$. E_{12} is sent to Node 2
5. Node 2 decrypts encrypted message E_{12} by first decrypting with $priv_C$ and then $priv_B$. $M_{12} = Dec(Dec(E_{12}, priv_C), priv_B)$

Figure 4.3: SMOCK communication between nodes

In this manner, data is encrypted and decrypted using the SMOCK keys. This scheme needed to be successfully implemented so as to be experimented upon. However, this implementation needs to be robust, flexible, and simple. The implementation details of this SMOCK scheme are highlighted in the next section.

4.4 SMOCK Implementation

As stated before, SMOCK is to be implemented for experimental purposes. The SMOCK implementation mainly relies on creating the *SMOCKhelpers* library as well as the freely available *OpenSSL* library ([28]), out of which two main modules are built. The *OpenSSL* library is used for key generation/storage and RSA encryption/decryption purposes while the *SMOCKhelpers* library implements the SMOCK key generation and key allocation as well as SMOCK encryption/decryption algorithms mentioned in previous sections. The two modules that comprise the implementation are:

- ***SMOCK Server***

  The *SMOCK Server* module runs on the centralized server, and is responsible for generating the SMOCK public/private key pairs based on input parameters. The main program in this module is *RSAKeygen*, which is run with certain inputs, namely the desired key size of each SMOCK key, the maximum number of nodes in the network, the resilience threshold $\wp$, and memory constraints, if any. For example, if we wish to have SMOCK keys for a network containing a maximum of 1000 nodes, with a resilience threshold of 0.7 and the key size of 1024 bits, then we use the following command:

```
./RSAkeygen -m 1000 -s 1024 -p 0.7
```

After getting the resilience threshold and the maximum number of nodes in the network and any posssible memory constraints, *RSAKeygen* mainly relies on the *generate_smock_keys()* function from the *SMOCKhelpers* library, which uses the aforementioned algorithms (from Section 3.2) in order to compute the a and b parameters. After the parameters are determined, the keys are generated using the *RSA_generate_key()* function. Each generated key is also assigned a numerical ID. Furthermore, the server also generates its own public/private keys with which it signs the assigned SMOCK IDs. This is to bind each node to a SMOCK ID, so that if a node is assigned a particular SMOCK ID, then another node cannot assume the same SMOCK ID (since it cannot produce the SMOCK ID signed by the server). After the keys are generated, they are stored in a link list of structs which stores the key along with its key ID. Then, a communication channel is created (with a call to *server_create_control _channel()*) which handles control messages from any client node. The *SMOCK Client* module (described below) on new client nodes first establishes a secure SSL socket (using the OpenSSL library). It then sends a *New Client Connection* message to the server, upon which the server creates a new thread to handle this request. The thread handler, *new_conn_thread()*, picks a random subset of SMOCK private keys to be assigned to this node (after ensuring that each SMOCK key is not overused and the subset of keys is unique) and sends them to the new client, along with the SMOCK public keys. It also generates a SMOCK ID by concatenating the binary equivalents of the key IDs of the assigned private keys. The server signs this SMOCK ID with its private key and sends this along with its public key and the assigned SMOCK keys to the new client. This interaction between the *SMOCK Server* module on the server and the *SMOCK Client* module on the new client is shown in Figure 4.4, which highlights each step in the SMOCK bootstrapping phase.

- ***SMOCK Client***

  The *SMOCK Client* module runs on every client, and comprises of a set of API which are to be utilized for communication with the server as well as for encryption/decryption of data for any application running on the client. The *get_smock_keys()* function is used in order to get the SMOCK keys (and SMOCK ID) from the *SMOCK Server*. It establishes a secure socket with *SMOCK Server* and then sends a *Client New Connection* message to *SMOCK Server's* control channel and waits for a response. As stated before, the *SMOCK Server* allocates a set of SMOCK private keys (and a SMOCK ID), and sends them to the *SMOCK Client*, along with all the SMOCK public keys. The *SMOCK Client* also opens a control channel in order to handle any *SMOCK ID Request* messages.

  The *SMOCK Client* is also responsible for encryption/decryption of data. For example, if an application on Client 1 wishes to to send encrypted data to an application on another client, Client 2, then it uses the SMOCK API. Any application wishing to encrypt the data may use the *smock_encrypt()* function. A sample call to the function is as follows:

  ```
  smock_encrypt(msg, msg_size, dest, enc, enc_size)
  ```

  Here, *msg* is the plaintext message to be encrypted, *msg_size* is the size of the message, *dest* is the IP address of the destination, *enc* is the buffer which is to hold the encrypted data, and *enc_size* is the size of this buffer. The *smock_encrypt* function loads the SMOCK public keys and first sends

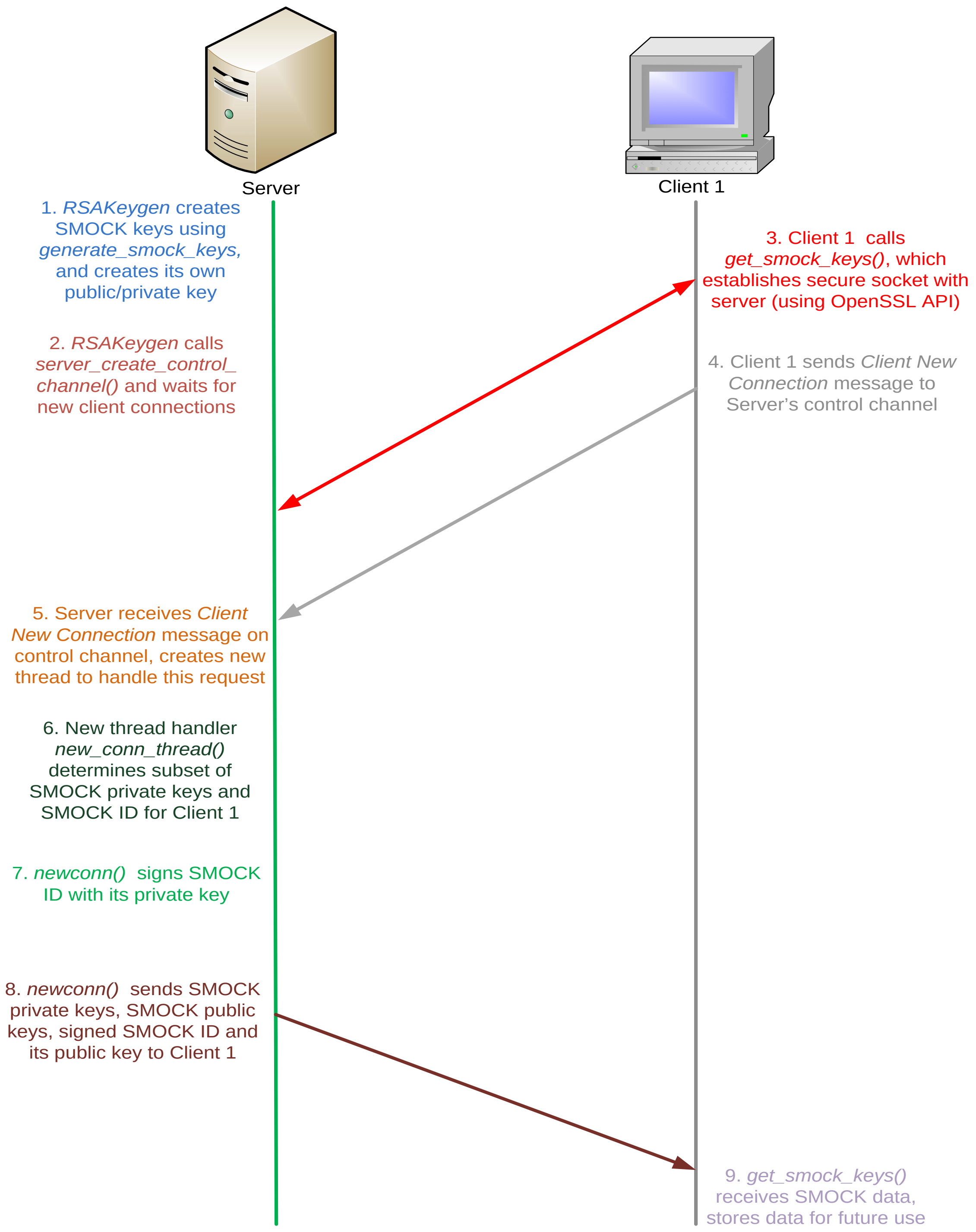
Server
Client 1
1. *RSAKeygen* creates SMOCK keys using *generate_smock_keys*, and creates its own public/private key
2. *RSAKeygen* calls *server_create_control_channel()* and waits for new client connections
3. Client 1 calls *get_smock_keys()*, which establishes secure socket with server (using OpenSSL API)
4. Client 1 sends *Client New Connection* message to Server's control channel
5. Server receives *Client New Connection* message on control channel, creates new thread to handle this request
6. New thread handler *new_conn_thread()* determines subset of SMOCK private keys and SMOCK ID for Client 1
7. *newconn()* signs SMOCK ID with its private key
8. *newconn()* sends SMOCK private keys, SMOCK public keys, signed SMOCK ID and its public key to Client 1
9. *get_smock_keys()* receives SMOCK data, stores data for future use

Figure 4.4: Step-by-step description of SMOCK bootstrapping phase

a *SMOCK ID Request* message on Client 2's control channel. Client 2 then responds with its SMOCK ID signed by the server's private key. The *smock_encrypt* function then verifies Client 2's SMOCK ID (by decrypting it using the server's public key), parses the SMOCK ID, and determines the public keys with which it encrypts the data. The actual encryption is done using the *RSA_public_encrypt()* function. As stated before, encryption is done starting with the keys in ascending order of their key IDs. It should be noted that if the message size is larger than the key size, then the message is split into several chunks (each chunk being equal to the key size) and each chunk is encrypted separately using SMOCK keys. For example, if each client has three SMOCK private keys (i.e., the b parameter is 3), the key size is 128 bytes and the message is 512 bytes, then *smock_encrypt()* breaks the message into four chunks, and each chunk is then encrypted three times using the appropriate SMOCK public keys (using the *RSA_public_encrypt()* function). This is because the RSA scheme does not allow for encryption of a buffer of data larger than the key size. Once encrypted, the message is then sent to Client 2. Client 2, upon receiving an encrypted message, invokes the *smock_decrypt* function. A sample call to the function is as follows:

```
smock_decrypt(msg, msg_size, dec, dec_size)
```

Here, *msg* is the encrypted message to be decrypt, *msg_size* is the size of the encrypted message, *dec* is the buffer which to hold the decrypted data, and *dec_size* is the size of this buffer. The *smock_decrypt* function loads the SMOCK private keys and then determines if the message size is larger than the key size. If this is the case, then the message is split into several chunks, and each chunk is decrypted seperately. The actual decryption is done using the *RSA_private_decrypt()* function, and decryption for each chunk is done starting with the keys in descending order of their key IDs. The decrypted message is then sent up to the application. This encryption/decryption scheme using SMOCK keys is shown in Figure 4.5.

Such a scheme can be applied to encryption/integrity services of nodes in a mesh network, or the clients in the network. In this thesis, we consider both scenarios while we evaluate the performance of SMOCK. In the next chapter, we describe the architecture of the underlying mesh network, as well as the integration of SMOCK with this underlying mesh network setup.

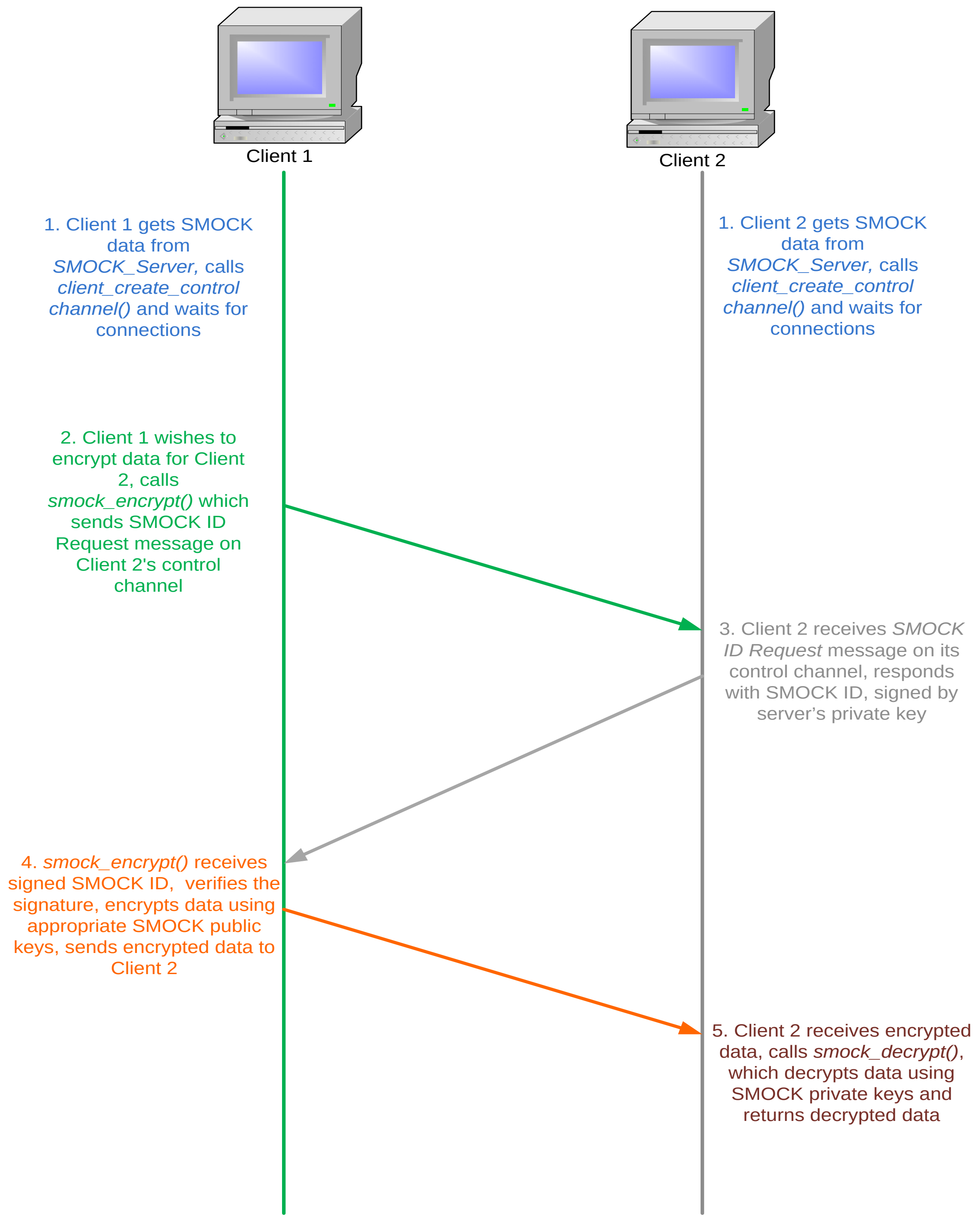
Client 1
Client 2
1. Client 1 gets SMOCK data from *SMOCK_Server,* calls *client_create_control channel()* and waits for connections
1. Client 2 gets SMOCK data from *SMOCK_Server,* calls *client_create_control channel()* and waits for connections
2. Client 1 wishes to encrypt data for Client 2, calls *smock_encrypt()* which sends SMOCK ID Request message on Client 2's control channel
3. Client 2 receives *SMOCK ID Request* message on its control channel, responds with SMOCK ID, signed by server's private key
4. *smock_encrypt()* receives signed SMOCK ID, verifies the signature, encrypts data using appropriate SMOCK public keys, sends encrypted data to Client 2
5. Client 2 receives encrypted data, calls *smock_decrypt()*, which decrypts data using SMOCK private keys and returns decrypted data

Figure 4.5: Step-by-step description of encryption/decryption of data using SMOCK keys

CHAPTER 5

SMESH - UNDERLYING MESH NETWORK ARCHITECTURE

In the previous chapter, the SMOCK key management scheme was described. In order to analyze its performance on the mesh network, a mesh network setup was needed. For these experiments, the SMesh mesh network software [29] is used. SMesh is a transparent wireless mesh network that provides fast, seamless handoff and supports real-time application data such as VoIP traffic. The clients for SMesh are unmodified devices which support 802.11 and run on Linux, Windows XP, Mac OS X, Palm OS, and Windows Mobile Pocket PC. SMesh allows for these clients to roam freely within the area covered by the wireless mesh nodes without interruption in the services provided by the network. So, the goal of SMesh is to appear as a singular access point for these clients. The mesh network of SMesh comprises mesh nodes which service clients and forward packets to other mesh nodes. These nodes also discover and periodically monitor their neighbors, as well as automatically adjusting their routing tables in the case of topology changes, which occur when the wireless connectivity between the mesh nodes changes, or when nodes crash and recover, or when nodes are added/removed from the network. The next section describes the architecture of SMesh.

5.1 SMesh Architecture

The two main components of the SMesh architecture are its communication infrastructure and its interface with mobile clients [29].

5.1.1 SMesh Communication Infrastructure

The mesh nodes in SMesh need to create and maintain a stable ad hoc wireless network. Moreover, they need to handle client requests. In order to facilitate this, they need to communicate with each other by forwarding packets over multiple hops. The communication infrastructure that enables this in SMesh is the the Spines messaging system [30]. Spines provides unicast, multicast, and any-cast communications between wireless mesh nodes. Spines runs a software daemon on each node, to which client applications can connect using API which is very similar to the Unix socket interface [30]. For example, the *spines_socket()* call creates and returns a TCP/IP connection to the daemon. Any client application can then use this socket to bind, listen, connect, send, and receive data using Spines library calls.

The architecture of the Spines daemon (shown in [30]) is shown in Figure 5.1. The daemon communicates with clients using a session layer. The daemon instantiates a unique Spines *Session* for each client connection. The *Overlay Link* component consists of three different components. The *Unreliable Data Link* sends and receives data without regard for reliability. The *Reliable Data Link* sends and receives data while providing reliability through congestion control and packet ordering. The *Control Link* is used to send and receive control data between daemons. The *Overlay Node* component shown in Figure 5.1 is responsible for forwarding data packets to its clients and other nodes, and also for maintaining connections to its neighbors. The *Hello Protocol* module is responsible for creating, monitoring, and destroying overlay links between the neighbor nodes. Spines daemons periodically send information about its links to its neighbors through the reliable data link. The *Link State Protocol* module is responsible for providing information about existing overlay links, out of which the Routing module chooses

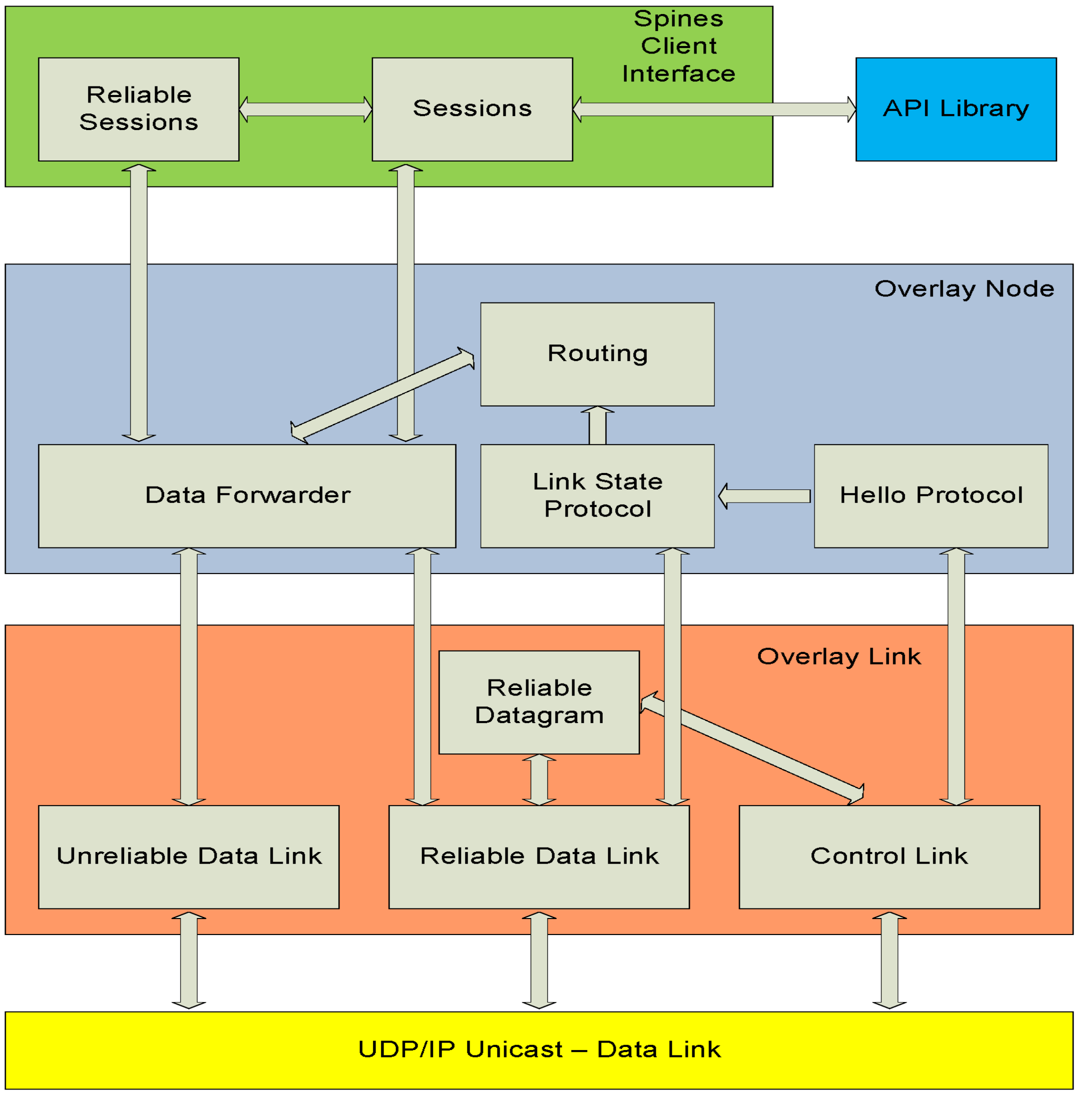

Figure 5.1: Spines daemon architecture

the neighbor providing the shortest path to a destination [30].

Spines allows for multicast and any-cast functionality in a multihop wireless environment [30]. In Spines, multicast groups are treated as class D IP address and any-cast groups are defined as a class E IP address. When mesh nodes join/leave a group, the local Spines daemon informs all other nodes via flooding. Group membership is maintained in Spines in tuples of the form (*mesh_node_address, group_address)* [30]. Based on group membership, Spines builds multicast trees throughout the network. These trees are built by optimizing on a metric such as number of hops. The utilization of Spines groups by SMesh is explained in the next subsection, which describes the SMesh interface with the mobile clients.

5.1.2 SMesh Mobile Clients Interface

SMesh serves as a singular access point for all mobile clients. This is done by first providing connectivity via a *DHCP Server* module running on each SMesh node. The role of the *DHCP Server* is to provide a unique IP address to each mobile client. The IP address is computed by applying a hash function on the client's MAC address, mapped to a class A private IP address of the form 10.A.B.C [29]. The *DHCP Server* module forces every packet to be routed through SMesh by setting the client's default gateway to be a generic global gateway and providing a network mask of 255.255.255.254. This forces every packet to be routed through this gateway. Further details of the SMesh *DHCP server* interactions are provided in [29]. The SMesh *DHCP Server* module, along with the rest of the components in the *SMesh Mobile Clients Interface*, is shown in Figure 5.2 (taken from [29]). As stated above, mesh nodes serve as the default gateway for the mesh clients. All clients are associated with a unique multicast group of mesh nodes known as the *Client Data Group*, which is the group that receives data originating from a

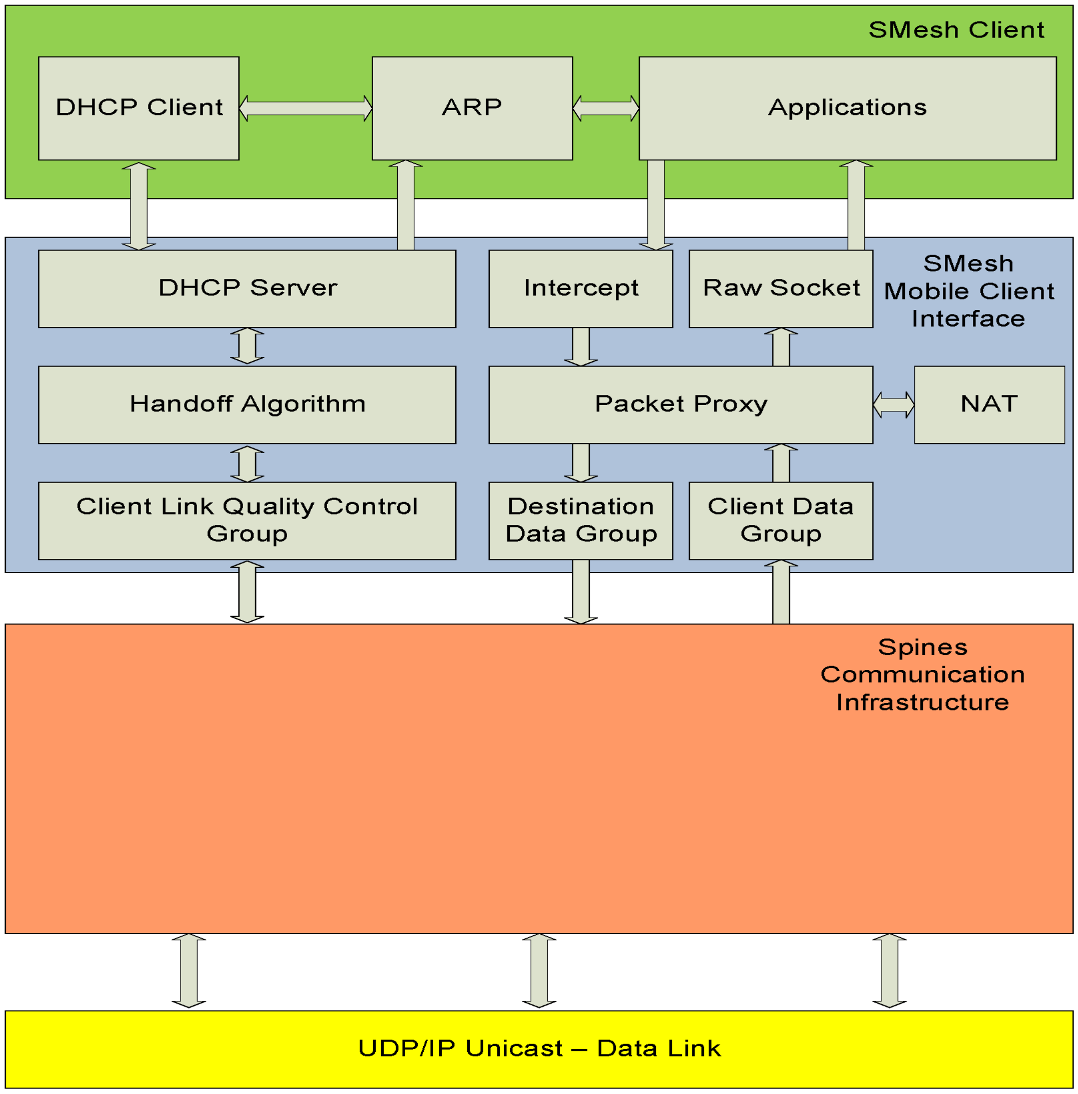

Figure 5.2: SMesh Mobile Clients Interface

particular client. One or more mesh nodes lying in the vicinity of a client will be a part of that client's *Client Data Group* [29]. The SMesh interface on each mesh node contains a *Packet Proxy* module, which uses the *Packet Interceptor* module to grab packets from clients. If the destination of a packet is the Internet, the packet is then sent to the closest Internet gateway by forwarding it to the any-cast group (via the NAT). However, if the destination of the packet is another mobile client, then the packet is forwarded to mesh nodes which belong to the *Destination Data Group* (in other words, the destination client's *Client Data Group*). This packet forwarding is done by Spines using a multicast tree, so that if the mobile client moved and a different mesh node joined its *Client Data Group*, then the packets are also forwarded to this mesh node. Each mesh node in the destination client's *Client Data Group*, upon receiving the packet, then forwards it to the mobile client using the *Raw Socket* module.

Apart from the aforementioned *Client Data Group*, each client also has another multicast group known as the *Client Control Group*. This group is used to coordinate handoffs to other mesh nodes in the client's vicinity based on the link quality. The link quality of each client is periodically (every 2 s) evaluated using the number of DHCP requests received according to the following decay function [29]:

$$M_{new} = M_{old} * D_f + Current * (1 - D_f), 0 < D_f < 1 \tag{5.1}$$

Here, M is the link quality measure, D_f is a decay factor (which decides our dependence on the older M measurement), and $Current$ is a constant if the mesh node received DHCP request in the previous 2-s interval and zero otherwise. For any given client, each SMesh node in the vicinity notes the link quality to the client and sends this to that client's *Client Control Group*. The *Handoff Algorithm* module on each SMesh node determines to join/leave *Client Data Groups* (or hand them to another SMesh node) based on the quality of the link to the client. The SMesh handoff mechanism uses gratuitous ARP requests in order to handle client handoffs. If a node believes that it has the best connectivity to a client, then it decides to serve that client, and sends a gratuitous ARP message by unicast directly to the client, changing the MAC address of its default gateway. Further details of the *Client Data Group* and *Client Control Group* membership management as well as the client management by various mesh nodes are given in [29].

While SMesh does provide fast, seamless handoff of clients between various mesh nodes, it does not provide security. In the next section, we go through two possibilities of integration of SMOCK with SMesh.

5.2 SMOCK Modules in SMesh

The SMOCK scheme, as described in Chapter 3, can be used to provide security for mesh networks. In this section, we explore two possibilities of integration with the SMesh framework. The first scenario involves integration of the *SMOCK Client* module with the mobile client, while the second scenario involves integration of the module with the underlying Spines daemon.

5.2.1 Application layer security - SMOCK integration with SMesh mobile clients

Before sending data packets over to the SMesh *Mobile Interface*, clients can first send them through the *SMOCK Client* module, which first sends a *SMOCK ID Request* message to the *SMOCK Client* module of the destination client, and then based on the received SMOCK ID, encrypts data, thus providing for confidentiality at the client end. At the receiver end, the module waits for any incoming data/*SMOCK ID Request* messages. The receiver sends its ID to the sender as a response to the *SMOCK ID Request* message. The *SMOCK Client* module then receives the encrypted message from the source and sends it to the application. This scenario is described in Figure 5.3. The advantage of such a scenario is that SMesh nodes need not worry about providing encryption, thus enhancing the speed of packet routing. This is useful for applications which do not need/want

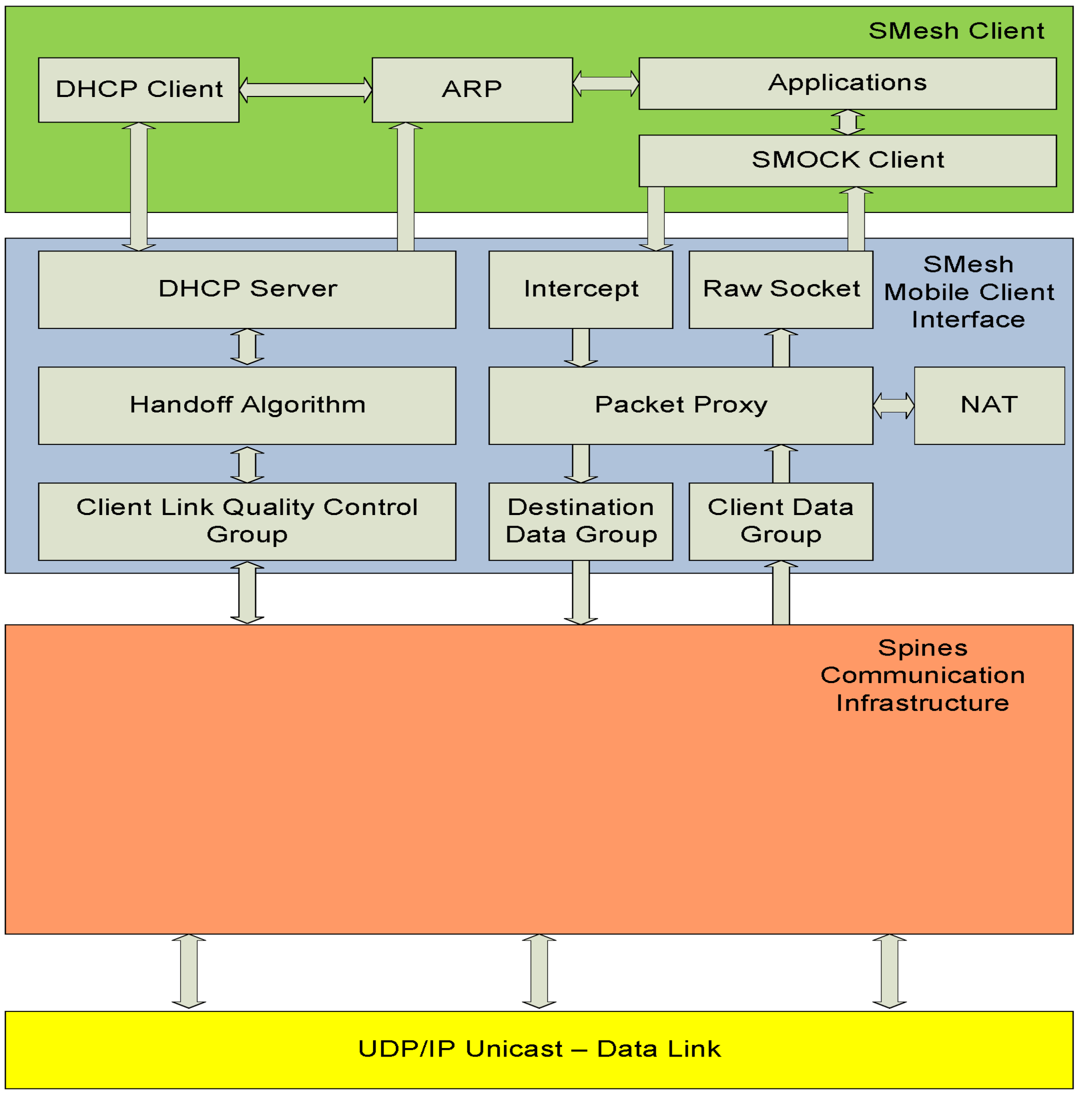

Figure 5.3: SMOCK integration with SMesh mobile clients

encryption services. Any application that wants encryption services can then pass the data through the *SMOCK Client* module, which then encrypts/decrypts the data. Furthermore, applications need not make any assumptions about the intent of clients and mesh nodes. That is, this scheme works even when clients and mesh nodes are malicious. However, this leaves applications to bear the brunt of security, which may not be desirable for certain situations. Application developers have to integrate the *SMOCK Client* module with their product, which may incur costs in terms of both time and money. Furthermore, the module needs to make a *SMOCK ID Request* message for every time a message needs to be encrypted, and needs to wait for the response from the destination client. This can add to the end-to-end delay of sending encrypting data, especially if the round-trip time between the messages is large. Another disadvantage of this scenario is that multicast between various mobile clients over the mesh network becomes challenging. This is because every node in a multicast tree, which aims to distribute multiple copies of the same packet to multiple destinations, must first contact each destination separately for its SMOCK ID, encrypt each copy of the packet separately, and then distribute the different, encrypted packets. This adds additional time overhead for every single multicast packet, thus making multicast infeasible. In this thesis, this scenario will also be referred to as Scenario 1.

5.2.2 Link layer security - SMOCK integration with Spines

The second scenario considered is shown in Figure 5.4. In this scenario, the *SMOCK Client* module is integrated with the underlying Spines daemon. When a client sends a packet to the SMesh node, it is then sent over to the underlying Spines *Client Interface* via a session. As soon as it reaches the *Overlay Node Component*, the packet is then sent to the *Data Forwarder*, which determines the

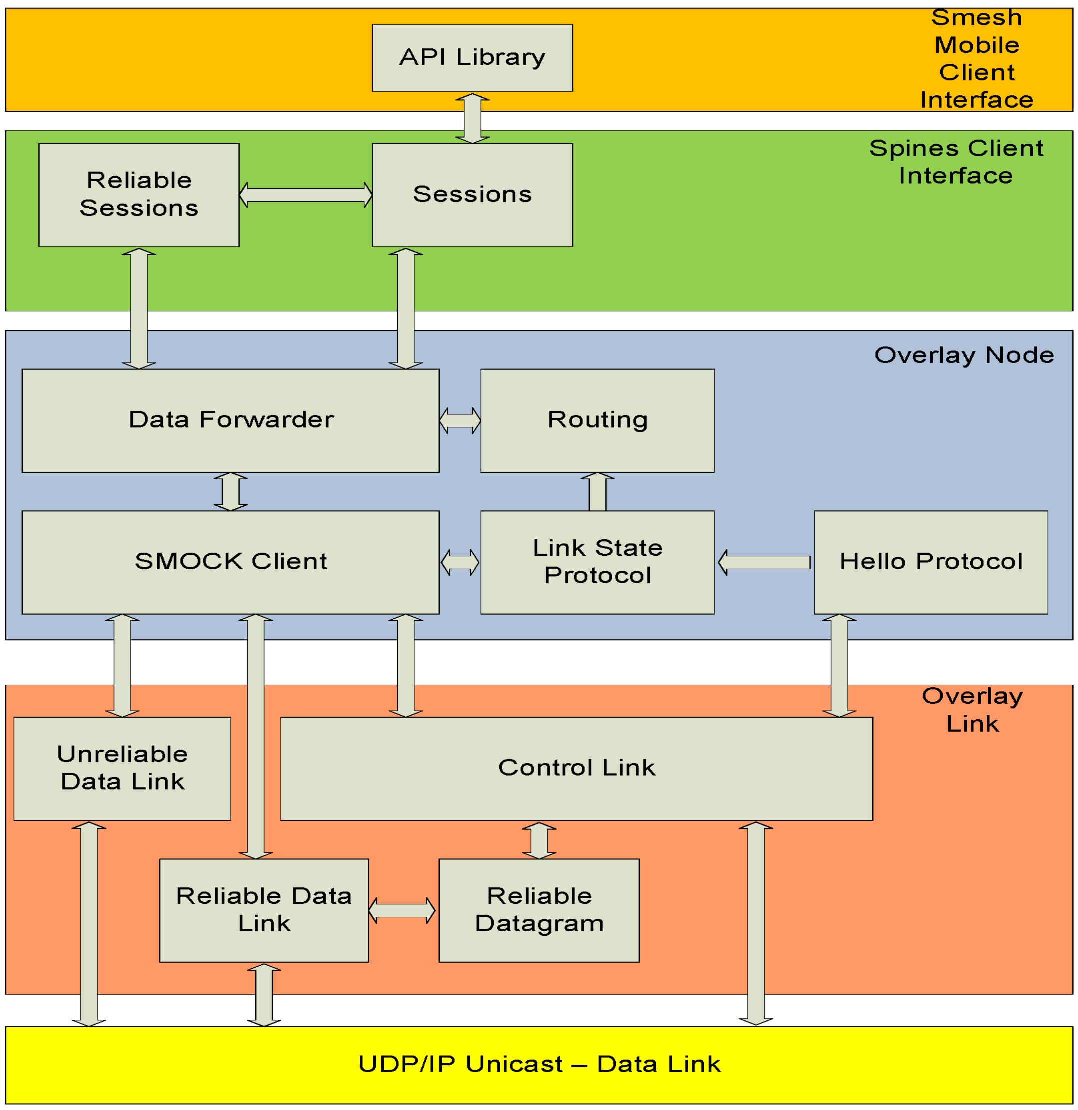

Figure 5.4: SMOCK integration with Spines

next hop for this packet and then sends it to either the *Unreliable Data Link* or the *Reliable Data Link* (depending on SMesh preferences). However, before it is sent to the Overlay Link component, the packet is then sent through the *SMOCK Client* module, which encrypts the packet based on the ID of the mesh node which serves as the next hop for this packet. Furthermore, any incoming packet from either the *Unreliable Data Link* or the *Reliable Data Link* components first goes through *SMOCK Client*, which determines if the packet came from another mesh node. If this is the case, the packet is decrypted based on the SMOCK ID of the previous hop for this packet and then sent to the *Data Forwarder*. Furthermore, all *Hello* packets carry the SMOCK ID of the originating node, so that each mesh node knows the SMOCK ID of its neighbor. Moreover, any outgoing message sent by the *Link State Protocol* is first sent through *SMOCK Client*, which encrypts the message with the public keys based on the SMOCK ID of the destination of the link information packet. At the receiver end, any packet received via the *Control Link*, if intended for the *Link State* module, is first sent through the *SMOCK Client* where it is decrypted and then sent to the *Link State* module. This is to ensure that the correct destination received the intended *Link State* packet, and ensures that any mesh node receives *Link State* packets only from other appropriate mesh nodes. Thus, the mobile clients are assured of data confidentiality on the mesh network (given an assumption on the nonmalicious behavior of mesh nodes, which is highlighted below). As a matter of fact, the only links that clients have to secure is the immediate link between themselves and the mesh node (this can be achieved by the client and the mesh node agreeing on a symmetric key).

The advantage of this approach is that multicast between clients is now feasible. This is because clients now do not have to worry about encryption on their part. Hence, multicast trees can now be created with clients being easily able to forward multiple copies of a data packet to various other clients in the tree. The fact that the clients do not need to worry about encryption on their part ensures that application developers need not worry about incorporating the *SMOCK Client* module (or any other encryption scheme) when they write applications for the mobile clients for the mesh network. However, a major disadvantage is that the assumption that all mesh nodes are nonmalicious has to be made. This is because at each mesh node, the data packet is decrypted (if received from another mesh node). Moreover, the task of decrypting and encrypting the message at each mesh node can be time-consuming, especially if the number of hops, key size, or number of private keys per node, b, is large. In this thesis, this scenario is also referred as Scenario 2.

These two scenarios have to be tested by having a performance analysis of the SMOCK key management scheme on the mesh network. In order to do this, we need to configure nodes either as mesh nodes (running SMesh and Spines) or as mobile clients (that connect to a mesh node). The next chapter describes the steps taken to configure laptop computers to function as mesh nodes/clients.

CHAPTER 6

EXPERIMENTAL SETUP

In this chapter, we cover the experimental setup required to do a performance analysis of the SMOCK key management scheme on the mesh network. This consists of three main sections, namely the networking setup, SMesh software setup, and wireless testbed setup.

6.1 Networking Setup

For experimental purposes, five laptop computers were set up to run the SMesh and Spines software and two additional laptop computers served as SMesh mobile clients, each running Linux Fedora Core 8. In order to provide wireless capability, each laptop was provided a Netgear WG511T wireless PC card. However, there was a need for chipset-level software which helps the computer detect the wireless card. For our experiments, we used the openly available *MadWifi* software, version 0.9.4 [31], since the wireless card had an Atheros-based chipset, and *MadWifi* supports such chipsets. One of the major advantages of the *MadWifi* is that it supports virtual access points (VAPs), with which it can support multiple access points on a single physical access point. On any VAP, *MadWifi* supports several modes, such as ad hoc mode, monitor mode, and auto mode.

After the *MadWifi* software is downloaded, first any and all *MadWifi* devices are shut down and current *MadWifi* modules (if any) are removed. Assuming that we are in the *MadWifi* directory, the following commands remove the current modules from the computer [31]:

```
cd scripts
./madwifi-unload.bash
./find-madwifi-modules.sh \$(uname -r)
cd ..
```

Once the old modules are removed, we go back to the *MadWifi* directory and type

```
make && make install
```

to create and install the *MadWifi* module. While the module is now loaded, it needs to be loaded into the running system. Before loading the newly created *MadWif*i module, we need to ensure that the *ath5k* module is blacklisted. This can be done by opening the **blacklist** file, which is in the directory **\etc\modprodprobe.d**, and adding *ath5k* to the end of that file. After this, if a *MadWifi* module is still currently running (as an interface) then it needs to be unloaded. This can be done by typing the following command:

```
modprobe -r ath_pci
```

Once this is done, we install the *MadWifi* module (with the ad hoc mode) using the following command:

```
modprobe ath_pci autocreate=adhoc
```

With this, we create a VAP with the ad hoc mode since the SMesh software requires each wireless device to be in ad hoc mode. By default, the name of this VAP is **wlan0**. The **modprobe** command creates a new VAP in ad hoc mode, which is by default called **wlan0**. For each of the nodes that will run SMesh, we edit the properties of this VAP by going to **\etc\sysconfig\network-scripts** and editing the **ifcfg-wlan0** file. A sample edited **ifcfg-wlan0** file is shown below.

```
#Atheros Communications, Inc. AR5212 802.11abg NIC
DEVICE=wlan0
ONBOOT=yes
BOOTPROTO=none
HWADDR=00:1e:2a:10:28:fc
TYPE=Wireless
NETMASK=255.255.255.0
IPADDR=10.0.0.5
USERCTL=no
IPV6INIT=no
PEERDNS=yes
ESSID=substation2
CHANNEL=1
MODE=Ad-Hoc
```

While most of the entries in the **ifcfg-wlan0** file are automatically filled, we need to fill out three important entries: the **ESSID** of the mesh network, which identifies the underlying network at the IP layer; the **IPADDR**, which is the unique IP address of the node; and the **NETMASK**, which determines the number of nodes in the network denoted by the **ESSID**. The new **wlan0** virtual interface, along with the actual physical interface (**wifi0**) can be seen in the output of the **ifconfig** command. A sample **ifconfig** output for one of the mesh nodes showing the VAP along with the physical interface **wifi0** is shown below.

```
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:337788 errors:0 dropped:0 overruns:0 frame:0
          TX packets:337788 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:73069970 (69.6 MiB)  TX bytes:73069970 (69.6 MiB)

wifi0     Link encap:UNSPEC  HWaddr
00-1E-2A-10-28-FC-48-08-00-00-00-00-00-00-00-00
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:890683101 errors:0 dropped:71473688 overruns:0
                                                       frame:60011889
          TX packets:11591409 errors:3012578 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:199
          RX bytes:60677174 (57.8 MiB)  TX bytes:1377486357 (1.2 GiB)
          Interrupt:11

wlan0     Link encap:Ethernet  HWaddr 00:1E:2A:10:28:FC
          inet addr:10.0.0.5  Bcast:10.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::21e:2aff:fe10:28fc/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2511750 errors:0 dropped:0 overruns:0 frame:0
          TX packets:173534 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:195849626 (186.7 MiB)  TX bytes:50062094 (47.7 MiB)
```

A sample **iwconfig** (shows all functional wireless interfaces) output is also shown below.

```
wlan0     IEEE 802.11g  ESSID:"substation2"
```

```
Nickname:"localhost.localdomain"
          Mode:Ad-Hoc  Frequency:2.412 GHz  Cell: 02:0E:9B:85:78:1F
          Bit Rate:0 kb/s   Tx-Power=18 dBm   Sensitivity=1/1
          Retry:off   RTS thr:off   Fragment thr:off
          Encryption key:off
          Power Management:off
          Link Quality=33/70  Signal level=-63 dBm  Noise level=-96 dBm
          Rx invalid nwid:6429041  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:0  Invalid misc:0   Missed beacon:0
```

In this manner, we install MadWifi on five computers and modify the networking configuration files such that they are all part of the same network (with **ESSID** "substation2"). In order for SMesh to actively monitor packets for RSSI measurement (used to gauge link quality), another VAP (in monitor mode) needed to be created. The additional VAP can be created using the following commands:

```
wlanconfig wlan create wlandev wifi0 wlanmode monitor ifconfig wlan1
up echo '802' > /proc/sys/net/wlan1/dev_type
```

The first command creates the new VAP **wlan1** (in monitor mode). This VAP is then turned on, and in the third command is set to attach prism2 type headers to each packet. In this manner, we add the MadWifi module and configure the wireless setup on each laptop designated as a mesh node.

Apart from the five laptops, two additional laptops will serve as the SMesh mobile clients. They will receive their IP address from the nearest mesh node. In a manner similar to the mesh nodes, *MadWifi* was downloaded and installed on each client. However, since the clients do not have static IP addresses (and in fact, get them from the mesh network), their **ifcfg-wlan0** file was to be edited differently. A sample **ifcfg-wlan0** file for a mesh client is shown below.

```
#Atheros Communications, Inc. AR5212 802.11abg NIC DEVICE=wlan0
ONBOOT=yes BOOTPROTO=dhcp TYPE=Wireless NETMASK=255.255.255.0
USERCTL=no IPV6INIT=no PEERDNS=yes ESSID=substation2 CHANNEL=1
MODE=Ad-Hoc
```

Here, the two main changes are that there is no **IPADDR** field (since the client does not use a static IP address), and that the boot protocol (**BOOTPROTO**) field is changed to DHCP (so that the DHCP client can then negotiate with the DHCP server in a mesh node in order to get assigned an IP address).

On each of the five laptops designated as mesh nodes, we needed to install and configure SMesh and Spines software in this manner. The installation and configuration of SMesh on each of these nodes is covered in the next section.

6.2 SMesh Software Setup

In this section, we go through the SMesh software setup. Provided that we have the executables for both SMesh and Spines, we create a directory **/jffs** and place them there. Also placed in the directory is the runSMesh script and the SMesh configuration file. In the default configuration file, we had to add some information and make some changes. These changes/additions are as follows:

```
MESHIF=wlan0
```

We set the mesh interface to be **wlan0**.

```
KERNEL=0
```

We operate in overlay mode (packets are routed through the Spines daemon), and hence set this parameter to 0.

```
NEIGHBORS=
```

We can either choose autodiscovery of neighbor with the following command:

```
NEIGHBORS="-d 225.5.5.5"
```

However, we can also set static neighbors (which is what was done for experiments) by explicitly specifying neighbors, using the following command:

```
NEIGHBORS="-a Neighbor1Addr -a Neighbor2Addr -a Neighbor3Addr ..."
```

where Neighbor1Addr, Neighbor2Addr, and Neighbor3Addr, etc., are IP addresses of the neighbors to which we want to connect statistically.

```
MONIF=wlan1
```

Here we specify our monitoring interface to be **wlan1**, which we created specifically for promiscuous monitoring of packets.

```
DEBUG=1
```

An optional command with which we can view the debug output.

Finally, we run the SMesh software by typing

```
./runSmesh
```

The **runSmesh** shell script runs the SMesh software with parameters based on the **smesh.conf** file. This also automatically starts the Spines daemon. The next task at hand was to create a testbed of nodes running the SMesh software, which will be covered in the next section.

6.3 Wireless Testbed Setup

After the SMesh software was installed on each machine, the next task was to create a testbed of nodes which run SMesh. SMesh and Spines were installed

Table 6.1: Table of computational/memory info of the mesh nodes

Node IP Address	Computational Power (MHz)	Memory Info (MB)
10.0.0.3	1800	248.33
10.0.0.4	550	249.27
10.0.0.5	1700	502.52
10.0.0.6	1800	248.71
10.0.0.7	550	376.08

on five machines, each of which was assigned a unique IP address. The five machines were given IP addresses 10.0.0.3, 10.0.0.4, 10.0.0.5, 10.0.0.6, and 10.0.0.7, respectively. For the remainder of this thesis, these nodes will be known as Nodes 3, 4, 5, 6, and 7.

The processor and memory information for each machine is given in Table 6.1. The five nodes are set in the third floor of the Siebel Center for Computer Science in a manner shown in Figure 6.1. These nodes are set in such a way

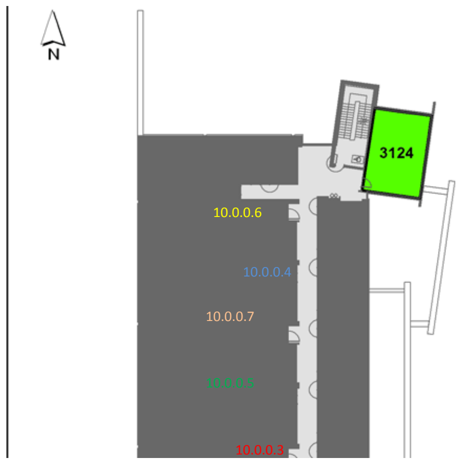

Figure 6.1: Placement of SMesh nodes

as to get the maximum amount of physical distance possible (for each hop) to minimize interference between the nodes. However, the nodes are set in a chain topology. In other words, three mesh nodes have exactly two neighbors, while two mesh nodes have exactly one neighbor. In addition to the five mesh nodes, two additional laptops were used as the mobile clients of the mesh network. Also, one of the clients also took the role of the *SMOCK Server*, which generates and allocates SMOCK keys. In this client, the *SMOCK Server* and SMesh run as different processes. The topology of the five mesh nodes, along with the two client nodes, are shown in Figure 6.2. Once all mesh nodes and clients were properly configured and deployed, experiments which cover both SMOCK scenarios could be performed. The next chapter covers the applications developed on each client, the parameters that are varied in the experiments (in both scenarios), and the metrics used to evaluate each scenario.

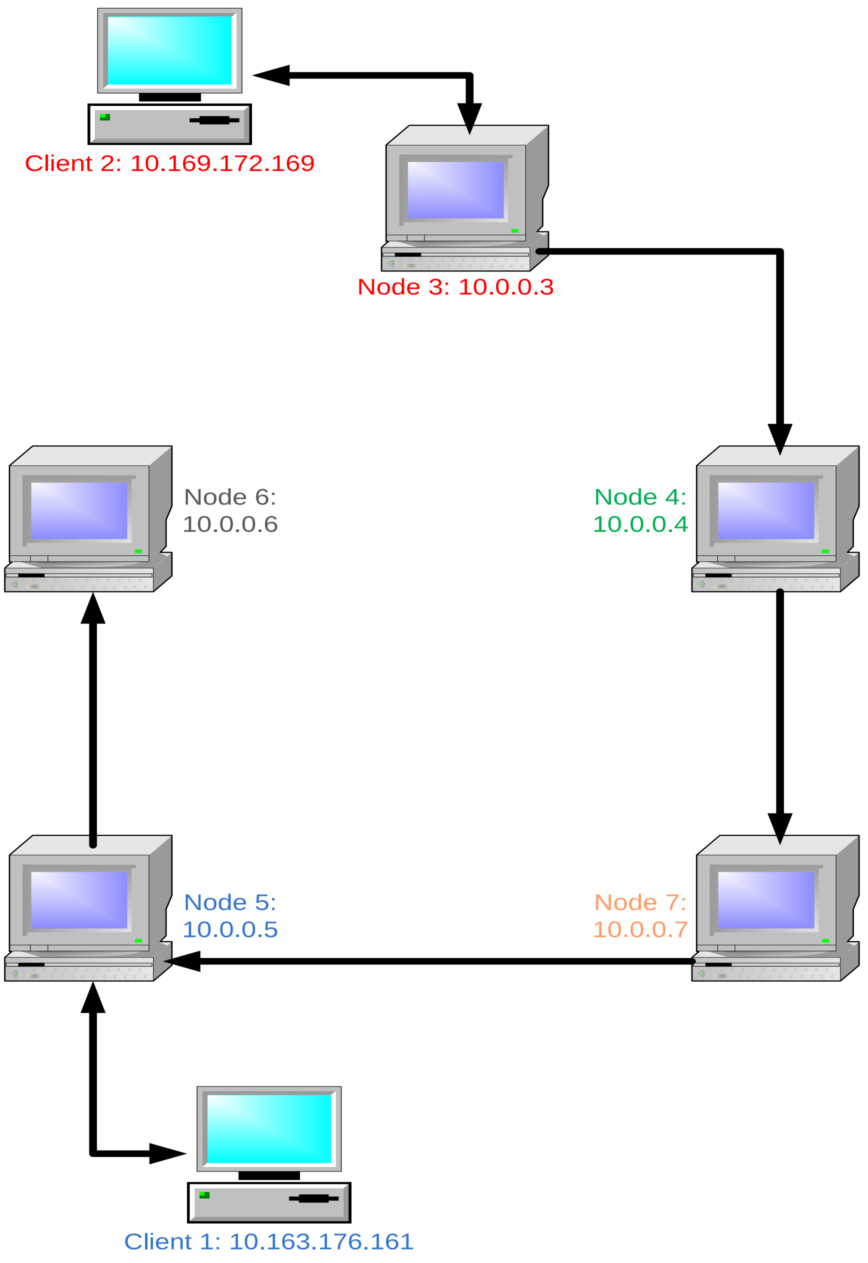

Figure 6.2: SMesh node topology (along with SMesh mobile clients)

CHAPTER 7

EXPERIMENTAL RESULTS

After the configuration and deployment of mesh nodes and clients, experiments were then to be conducted in order to do a performance analysis of the SMOCK scheme of the mesh network. In this chapter, we analyze the performance by first defining the metrics with which we evaluate the SMOCK key management scheme, the parameters which are varied for each metric, and then describe the evaluation results.

7.1 Metrics Used for Evaluation

Using the given topology, a performance analysis of the two SMOCK integration scenarios was to be done. For such a performance analysis, there are various metrics which can be considered important. The four main metrics that are considered in this chapter are the SMOCK parameter (a and b) generation time, SMOCK key generation time, SMOCK memory utilization, and round trip time (that includes SMOCK encryption/decryption). To begin with, *SMOCK Server* generates the SMOCK keys, and the time taken to generate the SMOCK parameters, as well as SMOCK keys for various key sizes, network sizes, and resilience parameters is an important issue. Furthermore, after the bootstrapping phase, the amount of memory used by the *SMOCK Client* module in both the storage of SMOCK data as well as temporary memory utilized by the encrypt/decrypt functions needs to be measured. This is because mesh networks may contain mesh nodes/clients which have high memory constraints. Furthermore, since nodes/clients of the mesh network also have varied processor power, it is necessary to monitor the encryption/decrytion time taken by the SMOCK scheme. Finally, applications (running on mesh clients) that utilize the mesh network may have real-time requirements. Due to this, it is important to monitor round trip time of sending encrypted data between clients of the mesh network. Hence, given these four metrics, we can now proceed with a performance analysis of SMOCK on the mesh network. The results of the performance analysis are shown in the next section

7.2 Experimental Results

In this section, we present the experimental results of the SMOCK performance analysis. As stated in the previous chapter, there are two nodes that act as the clients of the mesh network. Apart from the SMOCK key generation performance analysis (which was done on the *SMOCK Server* module), the rest of the experiments were conducted between these two clients (the exact topology will be shown before each different experiment). These clients will be known as Client 1 and Client 2. Client 1 is assigned an IP address (from a SMesh node) of 10.163.176.161, while Client 2 is assigned an IP address of 10.169.172.169.

In the case of Scenario 1 SMOCK integration, where the *SMOCK Client* module is integrated directly with the SMesh clients (and thus applications on SMesh clients can use the SMOCK encrypt/decrypt functionality), both clients, upon getting their IP address from a mesh node, contact the *SMOCK Server* module and get their SMOCK data. This data is stored in a predetermined location, and is later utilized by the *SMOCK Client* module. For the purpose of these experiments, Client 1 runs an application, App1, which takes a destination IP address and message size D, creates a buffer (of size D) containing random data (from a predetermined seed), encrypts it using SMOCK keys, and then sends the data to the destination IP address, and waits for a reply. Client 2 runs application App2 which waits for incoming requests. When it receives the data from Client 1, it first uses its SMOCK keys to decrypt the data and then verifies that the decrypted buffer is what was expected (since Client 2 also generates the buffer containing random data). It then generates a response (which is a buffer of the same size as the incoming buffer, containing random data from a second predetermined seed), encrypts it using SMOCK keys, and then sends it back to Client 1. Upon receipt, Client 1 decrypts this data, ensures that the decrypted data is what was expected, and then computes the time taken for the round trip operation (the round trip time metric).

In the case of Scenario 2 of SMOCK integration, all SMesh nodes receive their SMOCK data from the *SMOCK Server*, and then pass their signed SMOCK IDs to each other via periodic *Hello* messages. This eliminates the need for the *SMOCK ID Request* message, as well as the need for Client 1 and Client 2 to encrypt data. Hence, while Client 1 and Client 2 in this case perform similar operations as in Scenario 1, they do not encrypt/decrypt the data (as that is done by the mesh nodes while forwarding data).

As stated in the previous section, the time taken to generate SMOCK keys in the bootstrapping phase in the *SMOCK Server* module is important. This is analyzed in the next subsection.

7.2.1 SMOCK key generation time

The SMOCK key generation time is the time taken to generate the SMOCK keys, and is dependent on four main factors: the number of nodes in the network N, the number of nodes captured x, the resilience bound on the number of communica-

Table 7.1: a and b generation time (μs), $\wp$ vs. x (which is a percentage of N), 10 nodes

$\wp \backslash x$	30%	50%	70%
0.1	41.62	46.31	40.78
0.3	41.69	48.34	40.99
0.5	45.23	49.51	41.27

tions compromised when x nodes are captured $\wp$, and the key size. The values of x, $\wp$, and N determine the size of the key pool a as well as the number of private keys per node b (as shown in Chapter 4). Hence, these four parameters are varied and we note the key generation time, as well as the time taken to determine a and b for each scenario of parameters. We note the results for three different key sizes, namely 1024-bit, 2048-bit, and 4096-bit keys.

Parameter (a and b) generation time

It should be noted that the time taken to generate was independent of key size, and purely dependent on $\wp$, x, and N. So for all key sizes, we measured the parameter generation time by first varying N (between 10, 100, and 1000 nodes). For each network size value, we varied the resilience bound $\wp$ between 0.1, 0.3, and 0.5 and varied x between 30%, 50%, and 70% of N, respectively. For each case, 10 trials were conducted and the average of each trial was noted. For 10 nodes and varied values of x and N, 10 trials of the key generation algorithm were conducted and the average time taken to generate the values of a, b is shown in Table 7.1.

Also, the actual values of a and b are shown in Table 7.2. In all the tables, the columns represent x, while the rows represent $\wp$. Table 7.1 shows that the parameter generation time for 10 nodes hovered around 40 μs. This is because for all combinations of $\wp$ and x, (a,b) was determined to be (5,2). In the next set of

Table 7.2: Value of (a, b), $\wp$ vs. x (which is a percentage of N), 10 nodes

$\wp\backslash x$	30%	50%	70%
0.1	(5,2)	(5,2)	(5,2)
0.3	(5,2)	(5,2)	(5,2)
0.5	(5,2)	(5,2)	(5,2)

Table 7.3: a and b generation time (μs), $\wp$ vs. x (which is a percentage of N), 100 nodes

$\wp\backslash x$	30%	50%	70%
0.1	47.20	46.51	46.86
0.3	46.16	47.34	46.01
0.5	47.55	46.65	47.69

experiments, the number of nodes N was increased to 100. The results are shown in Tables 7.3 and 7.4.

In the 100-node case, (a,b) was increased to (10,3), and this increase is attributed to the larger number of nodes to support. In any case, the amount of time taken to determine a and b does not change. For the final set of experiments in this category, the number of nodes N was set to 1000. The results is shown in Tables 7.5 and 7.6. The results show that for the 1000-node case, (a, b) is set to (15,4) (for the given combinations of $\wp$ and x). Furthermore, the time taken to generate these parameters is slightly higher than that for the 10-node and 100-node cases.

Table 7.4: Value of (a, b), $\wp$ vs. x (which is a percentage of N), 100 nodes

$\wp\backslash x$	30%	50%	70%
0.1	(10,3)	(10,3)	(10,3)
0.3	(10,3)	(10,3)	(10,3)
0.5	(10,3)	(10,3)	(10,3)

Table 7.5: Value of (a, b), $\wp$ vs. x (which is a percentage of N), 1000 nodes

$\wp \backslash x$	30%	50%	70%
0.1	(15,4)	(15,4)	(15,4)
0.3	(15,4)	(15,4)	(15,4)
0.5	(15,4)	(15,4)	(15,4)

Table 7.6: a and b generation time (μs), $\wp$ vs. x (which is a percentage of N), 1000 nodes

$\wp \backslash x$	30%	50%	70%
0.1	54.53	53.14	52.72
0.3	57.61	53.98	53.56
0.5	53.21	53.21	53.42

After generating a and b, the *SMOCK Server* module can then proceed to generate the SMOCK keys. The next set of experiments note the time taken to generate these keys. The time taken to generate the SMOCK keys is highly dependent on the key size, and so we look at the key generation time for three main key sizes, namely 1024-bit keys, 2048-bit keys, and 4096-bit keys.

Key Generation Time - 1024-bit keys

For this and the other two sizes, the number of nodes N was first varied (between 10, 100, and 1000 nodes), and as in the previous experiments that measured the time taken to determine a and b, the number of nodes compromised was varied between 30%, 50%, and 70% of N. For each of these variations, the resilience bound on the number of compromised communications was varied from 0.1, 0.3, and 0.5. We first note the key generation time for 10 nodes, shown in Table 7.7.

After this, we looked at creating keys for a 100-node network. Tables 7.8 shows the key generation times when N is 100. From the key-generation time for 100 nodes, we note that it takes approximately twice the amount of time (order of

Table 7.7: SMOCK key generation time (s), $\wp$ vs. x (which is a percentage of N), 10 nodes, 1024-bit keys

$\wp \backslash x$	30%	50%	70%
0.1	0.899	1.115	1.032
0.3	1.934	0.945	1.259
0.5	1.080	1.386	1.290

Table 7.8: SMOCK key generation time (s), $\wp$ vs. x (which is a percentage of N), 100 nodes, 1024-bit keys

$\wp \backslash x$	30%	50%	70%
0.1	2.008	2.036	2.188
0.3	2.191	2.086	1.850
0.5	2.286	1.955	1.823

2000 ms) to generate keys as in the case for 10 nodes (where the key generation time is on the order of 1000 ms). In the next set of experiments, N was set to 1000 nodes, and the key generation time for this case is shown in Table 7.9

The results for the 1000-node case show that the key generation time is on the order of 3000 ms. In the next part, we consider the key generation time for 2048-bit keys.

Table 7.9: SMOCK key generation time (s), $\wp$ vs. x (which is a percentage of N), 1000 nodes, 1024-bit keys

$\wp \backslash x$	30%	50%	70%
0.1	2.272	2.955	3.164
0.3	2.655	2.940	3.035
0.5	3.534	2.831	3.473

Table 7.10: SMOCK key generation time (s), $\wp$ vs. x (which is a percentage of N), 10 nodes, 2048-bit keys

$\wp \backslash x$	30%	50%	70%
0.1	6.139	6.281	6.000
0.3	6.414	6.126	5.930
0.5	6.348	5.974	6.513

Table 7.11: SMOCK key generation time (s), $\wp$ vs. x (which is a percentage of N), 100 nodes, 2048-bit keys

$\wp \backslash x$	30%	50%	70%
0.1	13.104	9.091	11.866
0.3	9.604	10.594	13.800
0.5	13.045	11.743	13.349

Key Generation Time - 2048-bit keys

As in the previous part, we consider the key generation time for three values of N, namely 10, 100, and 1000 nodes. Also $\wp$ changes between 0.1, 0.3, and 0.5, while x changes between 30, 50, and 70%. The result for the key generation time when N has 10 nodes is shown in Table 7.10.

We see that the key generation time is much higher than that of 1024-bit keys. As a matter of fact, the results in Table 7.10 show that the key generation times for the 2048-bit key and $N = 10$ are around six times as high as the key generation times for 1024-bit key for the same N. The next set of experiments had $N = 100$. The results for this case are shown in Table 7.11.

For $N = 100$ we note a greater variance in the key generation time as compared to previous cases. Furthermore, the average key generation times for the 2048-bit case is around six times that of that 1024-bit case when $N = 100$. The next set of experiments had $N = 1000$, and the results are shown in Table 7.12.

As in the previous two cases, the key generation times for the 2048-bit, 1000-

Table 7.12: SMOCK key generation time (s), $\wp$ vs. x (which is a percentage of N), 1000 nodes, 2048-bit keys

$\wp\backslash x$	30%	50%	70%
0.1	19.072	23.032	21.608
0.3	21.592	23.259	19.824
0.5	19.548	19.111	21.761

Table 7.13: SMOCK key generation time (min:s), $\wp$ vs. x (which is a percentage of N), 10 nodes, 4096-bit keys

$\wp\backslash x$	30%	50%	70%
0.1	1:19.228	1:21.00	1:14.188
0.3	1:1.113	1:16.275	1:18.818
0.5	1:1.623	1:17.316	1:6.232

node network takes approximately six times the key generation times for the 1024-bit, 1000-node network. The next set of experiments measure the key-generation time for 4096-bit keys.

Key Generation Time - 4096-bit keys

As in the previous two parts, we consider the key generation time for three values of N, namely 10, 100, and 1000 nodes. Also $\wp$ changes between 0.1, 0.3 and 0.5, while x changes between 30, 50 and 70 %. The key generation time when N has 10 nodes is shown in Table 7.13.

The key generation time is much higher than that of 2048-bit keys. The results in Table 7.13 show that the key generation times for the 4096-bit keys and $N = 10$ is around 13 times as high as the key generation times for 1024-bit key for the same N. The next set of experiments had $N = 100$. The results for this case are shown in Table 7.14.

Similar to when $N = 10$, the key generation time is much higher than that of

Table 7.14: SMOCK key generation time (min:s), $\wp$ vs. x (which is a percentage of N), 100 nodes, 4096-bit keys

$\wp \backslash x$	30%	50%	70%
0.1	2:44.908	2:40.591	2:35.186
0.3	2:19.604	2:10.594	2:24.380
0.5	2:15.452	2:54.743	2:35.534

Table 7.15: SMOCK key generation time (min:s), $\wp$ vs. x (which is a percentage of N), 1000 nodes, 4096-bit keys

$\wp \backslash x$	30%	50%	70%
0.1	3:6.278	3:17.965	3:31.382
0.3	4:48.375	3:33.866	3:38.231
0.5	4:16.847	3:38.569	3:54.348

2048-bit keys. The results in Table 7.14 show that the key generation times for the 4096-bit key and $N = 100$ are around 12 times as high as the key generation times for the 2048-bit key for the same N. The final set of experiments had $N = 1000$, and the results are shown in Table 7.15.

As in the previous two cases, the key generation times for the 4096-bit, 1000-node network are much higher than those of the 2048-bit, 1000-node network. As a matter of fact, the key generation times are approximately 10 times those of the 2048-bit, 1000-node network.

So far, we had measured the time taken by the *SMOCK Server* module to generate keys. In the next subsection, we explore the memory usage of SMOCK.

7.2.2 SMOCK memory usage

In this subsection, we observe the memory usage of the SMOCK scheme for various key sizes and number of nodes N. Since the a and b parameters do not change greatly over various values of $\wp$ and x, it was decided to fix the value of $\wp$ as 0.1

and x as 30% of N. With that in mind, we took into consideration three key sizes (namely, 1024-bit keys, 2048-bit keys, and 4096-bit keys) as before and altered N. In this experiment, we arranged a topology shown in Figure 6.2 on the mesh network and used the Scenario 1 integration, where SMesh client applications use the *SMOCK Client* to encrypt/decrypt data. SMOCK memory usage consists of two main components, namely the SMOCK data storage (which is the major factor), and additional temporary memory required by the encryption/decryption algorithms.

For each key size, we vary N between 10, 100, and 1000, which corresponds to a b value between 2, 3, and 4. In each experiment, we sent data ranging from $2^0 - 2^9$ bytes. Each experimental result was the average of ten trials, and in each trial the memory usage was computed by averaging the memory utilized by the encryption/decryption algorithms in both clients, as well as averaging key storage from both clients. So, Figure 7.1 shows the memory usage for 1024-bit SMOCK keys, with respect to the log of the data encrypted. For example, 5 on the X-axis implies that the data to be encrypted was 2^5 bytes.

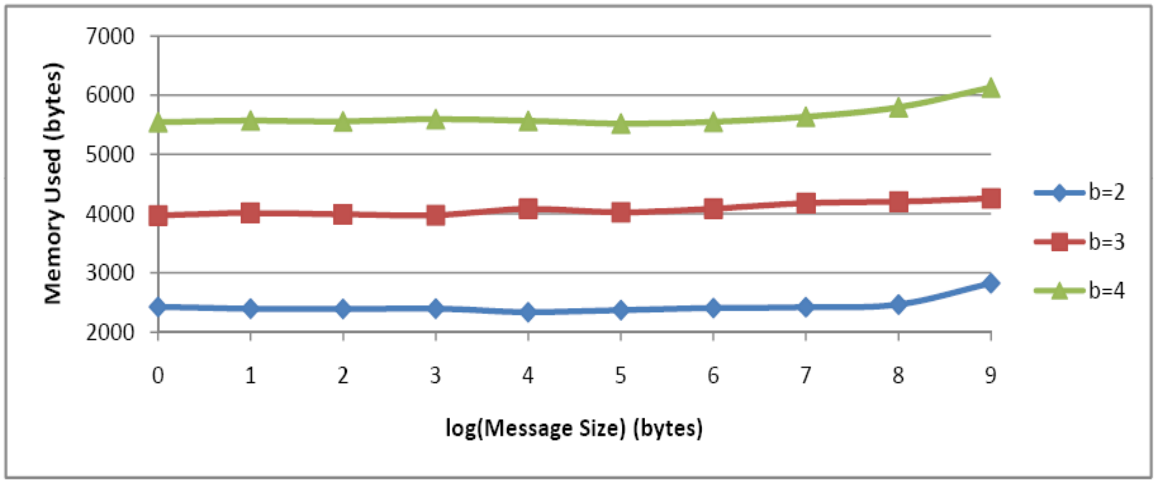

Figure 7.1: Memory usage vs. log(message size) - 1024-bit keys

As can be seen from Figure 7.1, the memory usage ranges from 2500 to 6000 bytes. The next set of experiments focuses on the usage of 2048-bit keys (with b varying as usual). The results from these experiments are shown in Figure 7.2. The memory usage with 2048-bit keys ranges from approximately 4000 to 10 000 bytes. The final set of experiments uses 4096-bit keys, and the results of these experiments are shown in Figure 7.3.

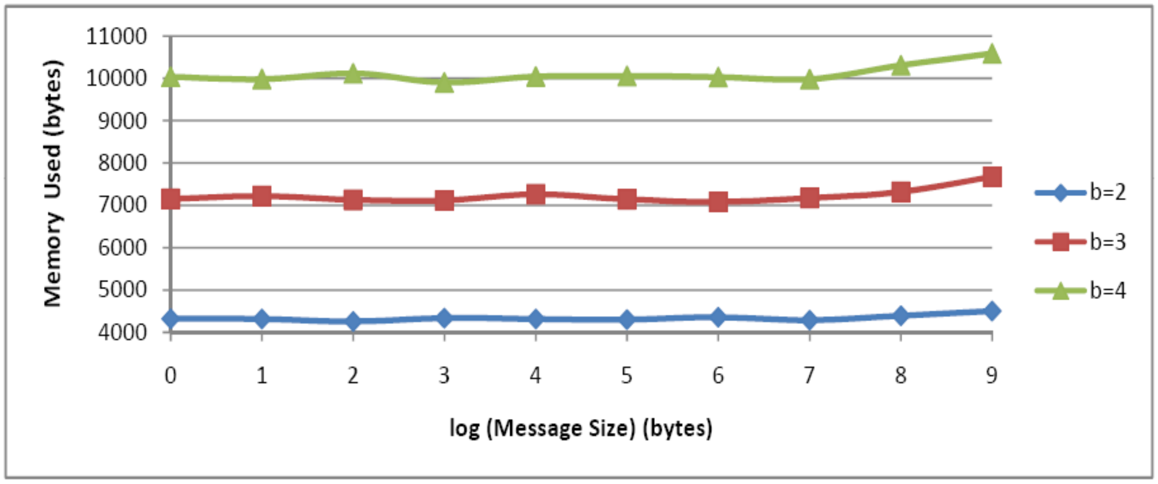

Figure 7.2: Memory usage vs. log(message size) - 2048-bit keys

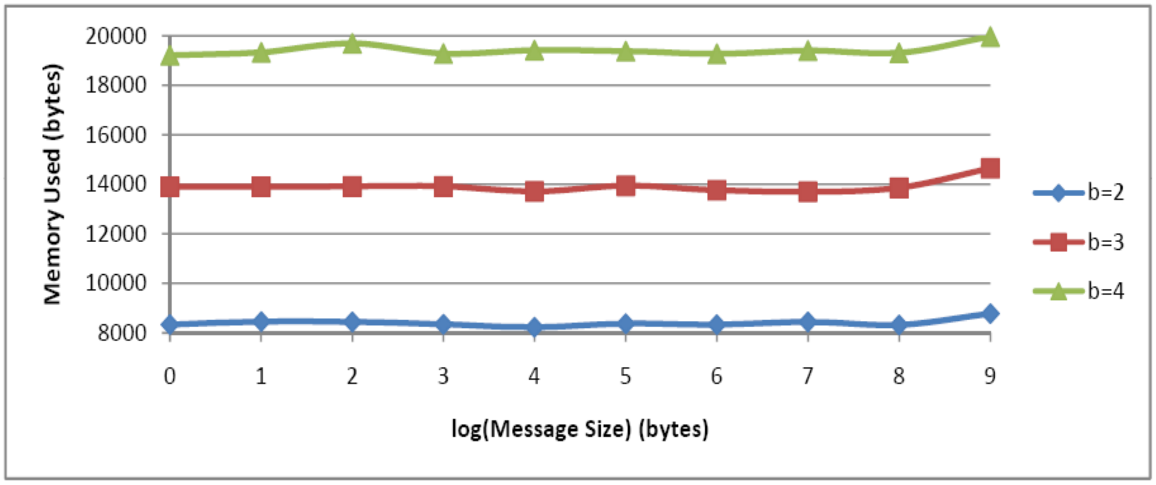

Figure 7.3: Memory usage vs. log(message size) - 4096-bit keys

We see that with 4096-bit keys, the memory usage ranges from approximately 8000 to 19 000 nodes. So, the difference between memory usage for 4096-bit and

2048-bit keys is much higher than the difference between 2048-bit and 1024-bit keys. This fact is highlighted in Figure 7.4, which keeps b as a constant and notes the memory uses for the different key sizes.

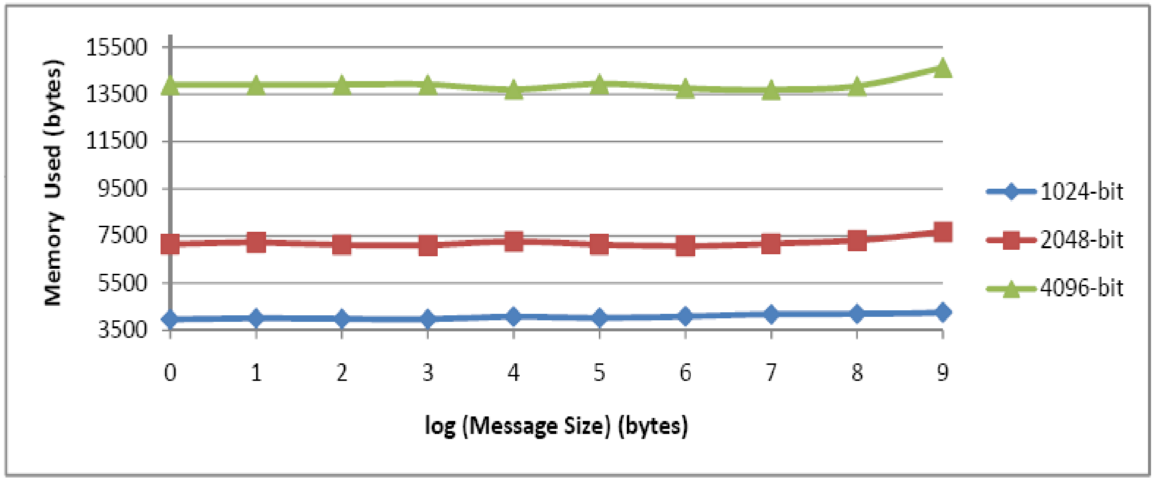

Figure 7.4: Memory usage vs. log(message size) - b = 3

In this figure, we see the huge gap between the 4096-bit and 2048-bit keys as compared to 2048-bit and 1024-bit keys. This shows that SMOCK may be impractical with anything larger than 2048-bit keys. However, as the results for 1024-bit and 2048-bit keys show, SMOCK is very memory efficient with less than 10-kB storage in all cases. This is equivalent to storing only 10 certificates from other nodes (since a certificate is typically 1 kB). However, as shown, SMOCK can support 1000 nodes with approximately 10 kB of memory. Hence, SMOCK is more memory efficient than a certificate-based key management scheme.

In the next subsection, we focus on the third metric for evaluation, the encryption/decryption time.

7.2.3 SMOCK encryption/decryption time

In this part, we observe the time taken by the the SMOCK scheme to encrypt/decrypt data (of various sizes) for various key sizes and number of nodes N. As in the memory usage evaluation, it was decided to fix the value of $\wp$ as 0.1 and x as 30% of N. With that in mind, we took into consideration three key sizes (namely, 1024-bit keys, 2048-bit keys, and 4096-bit keys) as before and altered N. Similar to the memory usage evaluation, we arranged a topology shown in Figure 6.2 on the mesh network and used the Scenario 1 integration, where SMesh client applications use the *SMOCK Client* to encrypt/decrypt data.

For each key size, we vary N between 10, 100, and 1000, which corresponds to a b value between 2, 3, and 4. In each experiment, we sent data ranging from $2^0 - 2^9$ bytes. Each experimental result was the average of 10 trials, and in each trial the encryption and decryption time was computed by averaging the encryption and decryption times in both clients (since in that scenario, each client encrypts and decrypts once). So, Figures 7.5 and 7.6 show the encryption/decryption times for 1024-bit SMOCK keys, with respect to the log of the data encrypted.

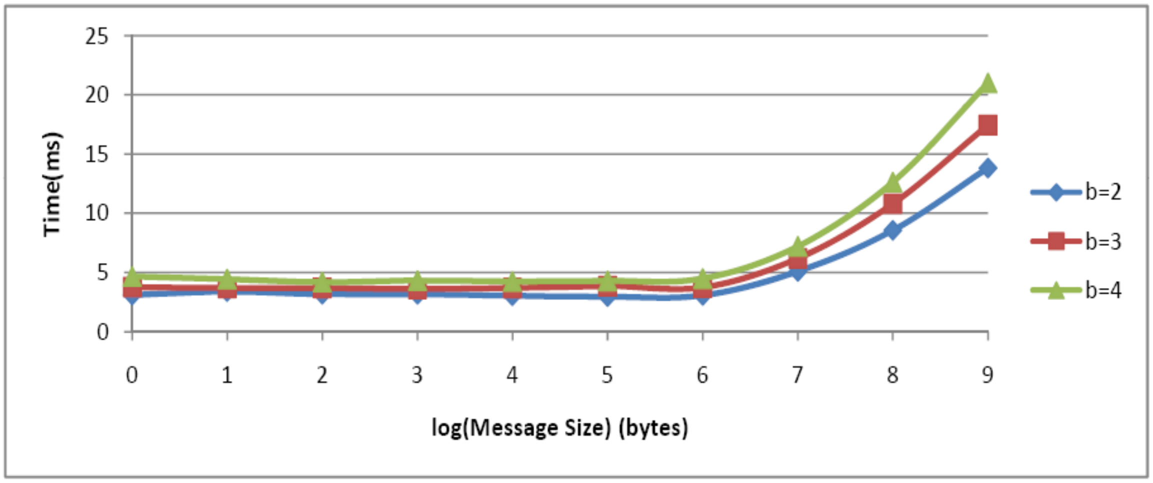

Figure 7.5: Encryption time vs. log(message size) - 1024-bit keys

As can be seen from Figures 7.5, the encryption time for this key size ranges from 4 ms to 20 ms. It is interesting to note that the encryption times stay similar while the message size remains less than the key size, but increases exponentially with respect to the message size, when the message size begins to approach the

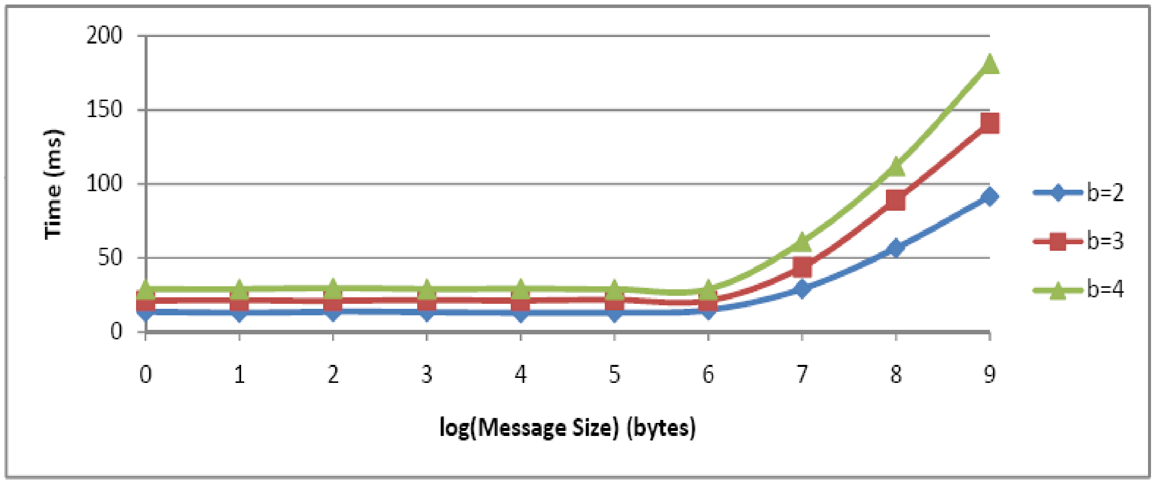

Figure 7.6: Decryption time vs. log(message size) - 1024-bit keys

key size. This is because RSA keys can only encrypt data if the message size is smaller than the key size. So, when presented with data whose size is greater than the key size, we break the data into smaller chunks (each chunk being the key size minus a certain constant) and encrypt each chunk separately. So for example if $b = 3$, the SMOCK key size is 1024 bits, and the message size is 64 bytes, then the SMOCK scheme, in order to encrypt and decrypt data, performs three encryption operations and three decryption operations (namely three calls to *smock_encrypt()* and *smock_decrypt()*), thus performing six operations in total.

However, if the message size is 256 bytes, then the message is broken down into three chunks (of sizes 100, 100, and 56), and each chunk is encrypted/ decrypted separately. In this case, the SMOCK scheme performs nine encryption and nine decryption operations, thus performing 18 operations in total. So clearly we see that increasing the data size beyond the key size results in exponential increase in time with respect to the data size. This is thus reflected in Figure 7.5 as well as 7.6, which shows the decryption times (for 1024-bit keys) ranging from approximately 20 to 170 ms (with the latter figure also showing the exponential increase in decryption time when the message size is greater than or equal to 128 bytes). The next set of experiments focuses on the usage of 2048-bit keys (with b varying as usual). The results from these experiments are shown in Figures 7.7 and 7.8.

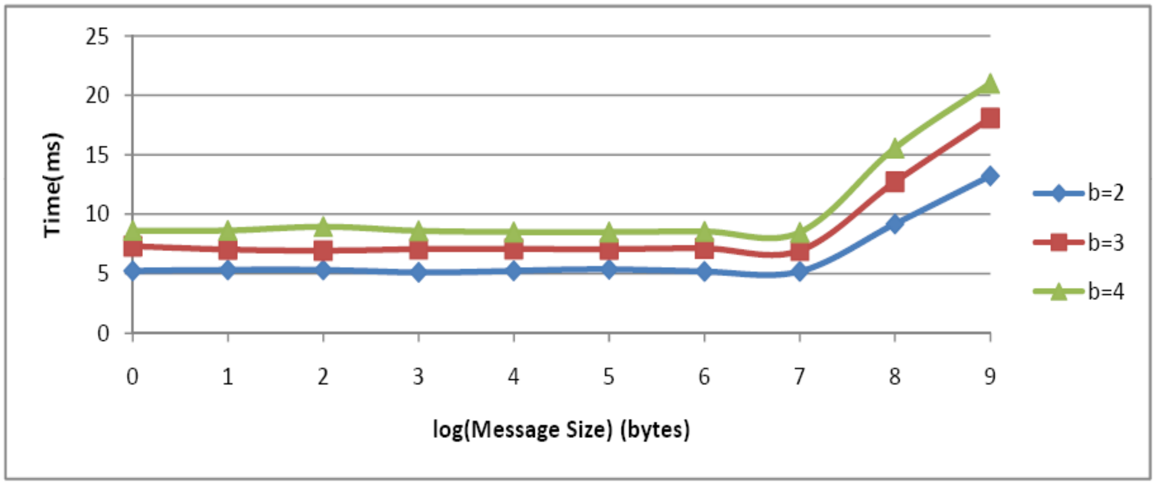

Figure 7.7: Encryption time vs. log(message size) - 2048-bit keys

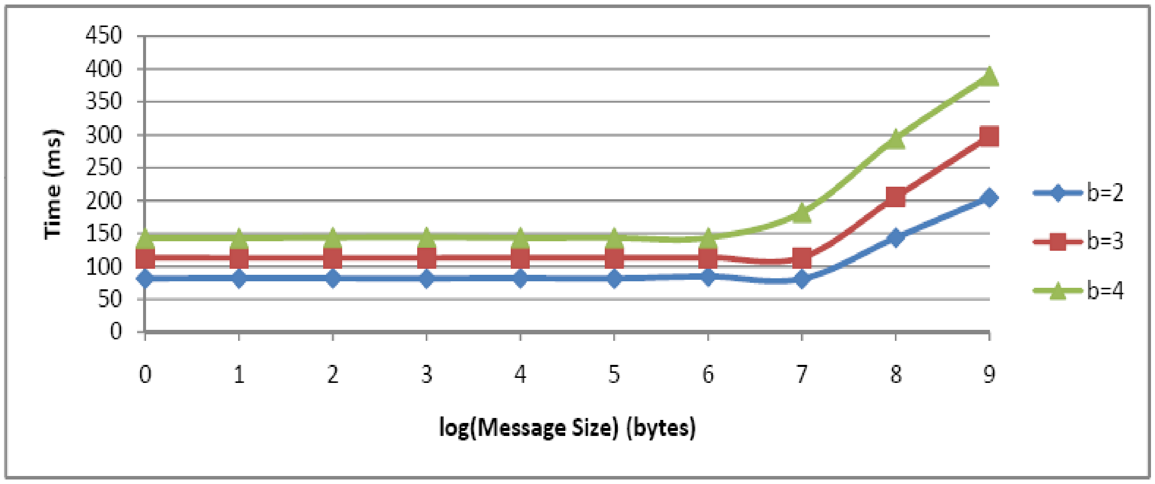

Figure 7.8: Decryption time vs. log(message size) - 2048-bit keys

Figure 7.7 shows the encryption times (for 2048-bit keys) ranging from 5 to 20 ms, which is not too different than that for the 1024-bit keys. However, Figure 7.8 shows the decryption times ranging from approximately 80 to 380 ms, which is a starker difference with its 1024-bit counterpart. As in the 1024-bit key case, we note the exponential increase in encryption/decryption times when the message size begins to approach the key size (which is 256 bytes in this case).

The next set of experiments focuses on the usage of 4096-bit keys (with b varying as usual). The results from these experiments are shown in Figures 7.9 and 7.10.

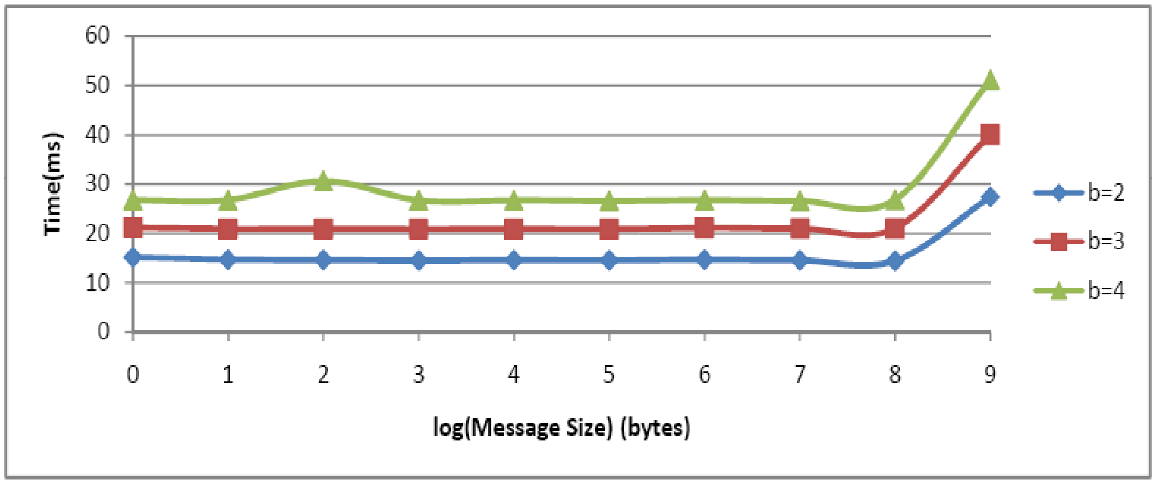

Figure 7.9: Encryption time vs. log(message size) - 4096-bit keys

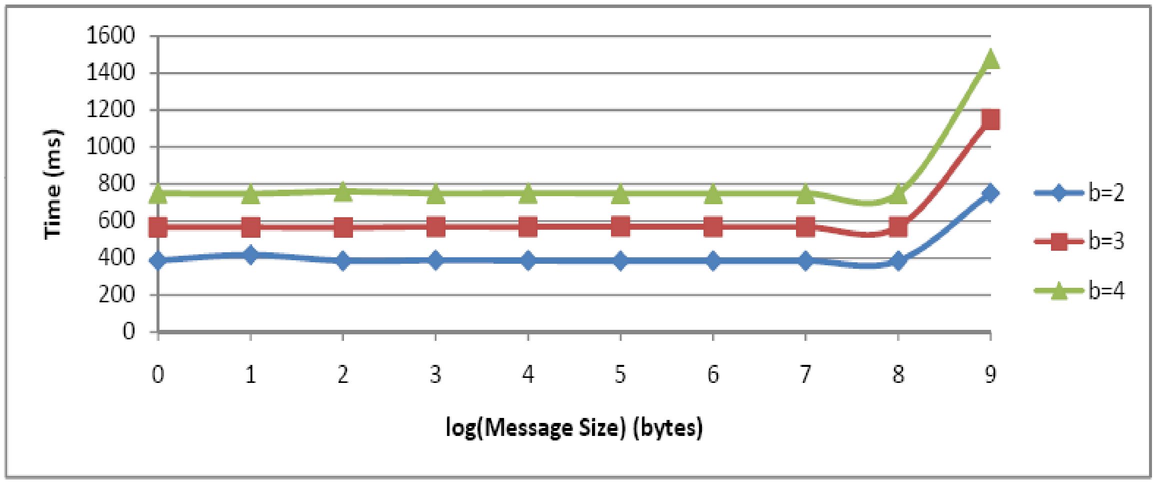

Figure 7.10: Decryption time vs. log(message size) - 4096-bit keys

Figure 7.9 shows the encryption times (for 4096-bit keys) ranging from 20 to 50 ms, which is much larger than that for the 2048-bit keys. Moreover, Figure 7.10 shows the decryption times ranging from approximately 400 to 14200 ms, which is also much different from its 2048-bit counterpart. These differences are highlighted in Figures 7.11 and 7.12. Here we kept b as a constant ($b = 3$) and observed the change in encryption/decryption times with respect to the key sizes. As with the memory usage case, there was a big difference in the encryption/decryption times between the 4096-bit key case and the 2048-bit key case.

However, while there is a negligible difference between the encryption times of the 4096-bit and 2048-bit SMOCK keys, there is a significant difference (of around 90 ms) between the 2048-bit and 1024-bit keys. As in the previous cases, there was an exponential increase in encryption/decryption times when the message size begins to approach the key size (which is 512 bytes in this case).

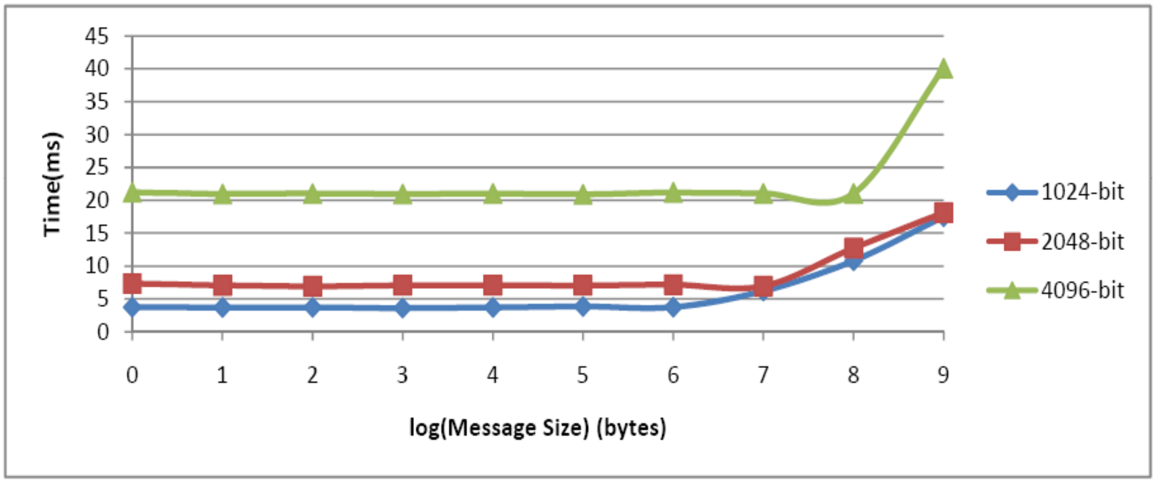

Figure 7.11: Encryption time vs. log(message size) - b = 3

Since the encryption/decryption times using 4096-bit SMOCK keys are much higher than those using 1024-bit and 2048-bit keys, it is impractical to consider using 4096-bit keys in the SMOCK scheme. So, only 1024-bit and 2048-bit SMOCK keys were considered in further experiments. So far, we had only considered the processor and memory utilization by the SMOCK modules. However, in order to provide a complete overview of SMOCK performance analysis, we need to con-

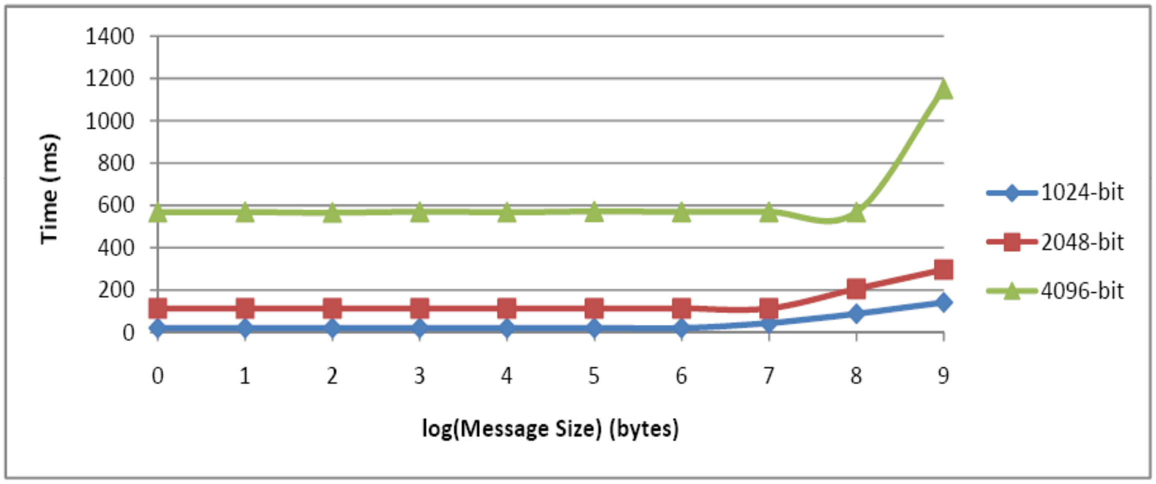

Figure 7.12: Decryption time vs. log(message size) - b = 3

sider the effects of SMOCK on the round trip time between nodes in the mesh network. This is reviewed in the next part.

7.2.4 SMesh round trip time

In this part, we consider the effects of SMOCK on the round trip time between nodes in the mesh network. The nodes in the mesh network are organized in a topology shown in Figure 6.2, and can have various ping times between each other based on the transmit power. The transmit power for each node is the amount of power dedicated for data transmission, and is measured in units of dBmW (decibels per milliWatt). If given a power value W (in watts), then it can be converted to a dBmW value P, using the following equation:

$$P = \frac{30 + 10 * log(W)}{1000} \tag{7.1}$$

By default, we assume that clients maintain the default transmit power of 19 dBmW (also the maximum possible transmit value). For all the mesh nodes, the transmit power can be modified by typing the following command:

```
iwconfig XXX txpower YYY
```

where the name of the VAP (whose transmit power we wish to modify) is substituted for XXX and the transmit power value is substituted for YYY. For the purposes of this performance analysis, we consider two values for transmit power, namely the minimum possible value of 1 dBmW and the default (and maximum possible) value of 19 dBmW. For all cases, we consider the round trip time over three cases when it comes to hop count. In other words, we consider the following cases:

- **Three hops** In this case, we consider the mesh node topology we physically place the Client 2 machine (which hosts both *SMOCK Server* and App2) close to the Node 3, and the Client 1 machine to Node 4. In this case, Node 3 becomes the only SMesh node in Client 2's client data group (and so provides Client 2 its IP address and routes all data packets to and from Client 2), while Node 4 becomes the only node in Client 1's client data group.

- **Four hops** In this case, we place Client 2 next to Node 3 and Client 1 next to Node 7.

- **Six hops** In this case, we place Client 2 next to Node 3 and Client 1 next to Node 6.

As stated before, we consider two scenarios of encryption, namely Scenario 1, which refers to end-to-end encryption of data and Scenario 2, which refers to a hop-by-hop encryption of data. We first consider Scenario 1. Here, data is encrypted by the SMOCK Client modules which reside on Client 1 and Client 2, respectively.

Scenario 1 (end-to-end encryption) performance

In this case, we first set the transmit power of each mesh node to 1 dBmW and measure the average ping time between each node and its neighbor over 10 trials. The node topology along with the mean/variance of RTT values between the nodes is shown in Figure 7.13.

We consider the performance of SMOCK (over various hops) for two key sizes, and three b values for each key size. Figure 7.14 shows the round trip time for each b value using 1024-bit SMOCK keys.

The results show an enormous difference between the RTT values for three, four, and six hops. For the three-hop case, the RTT values vary from 650 ms to 1000 ms. It should be noted that for each b value, if the data size is greater than or equal to the key size, the RTT time increases exponentially with respect to the data size. This behavior is explained before (caused due to multiple encryptions/decryptions, thus increasing the round trip time). For the four-hop case, the RTT values vary from 920 ms to 1230 ms. For the six-hop case, the RTT values vary from 1210 ms to 1580 ms. As expected, there is an increase in both RTT values, as well as the variance between the RTT values for different hop counts. The increase in RTT values is due to the fact that routing overhead is added at each hop in the mesh network. At each hop, the packet is sent all the way up to the SMesh application (through the Spines application), which determines the next hop to send the packet to, and then sends the packet to the next hop. The increase in variance of RTT values occurs due to the fact that the RTT values between neighboring nodes have a certain amount of variance due to packet retransmissions (the low transmit power results in lost packets). So, the variances sum up as the number of hops increases, thus leading to high variance in RTT values for the six-hop case.

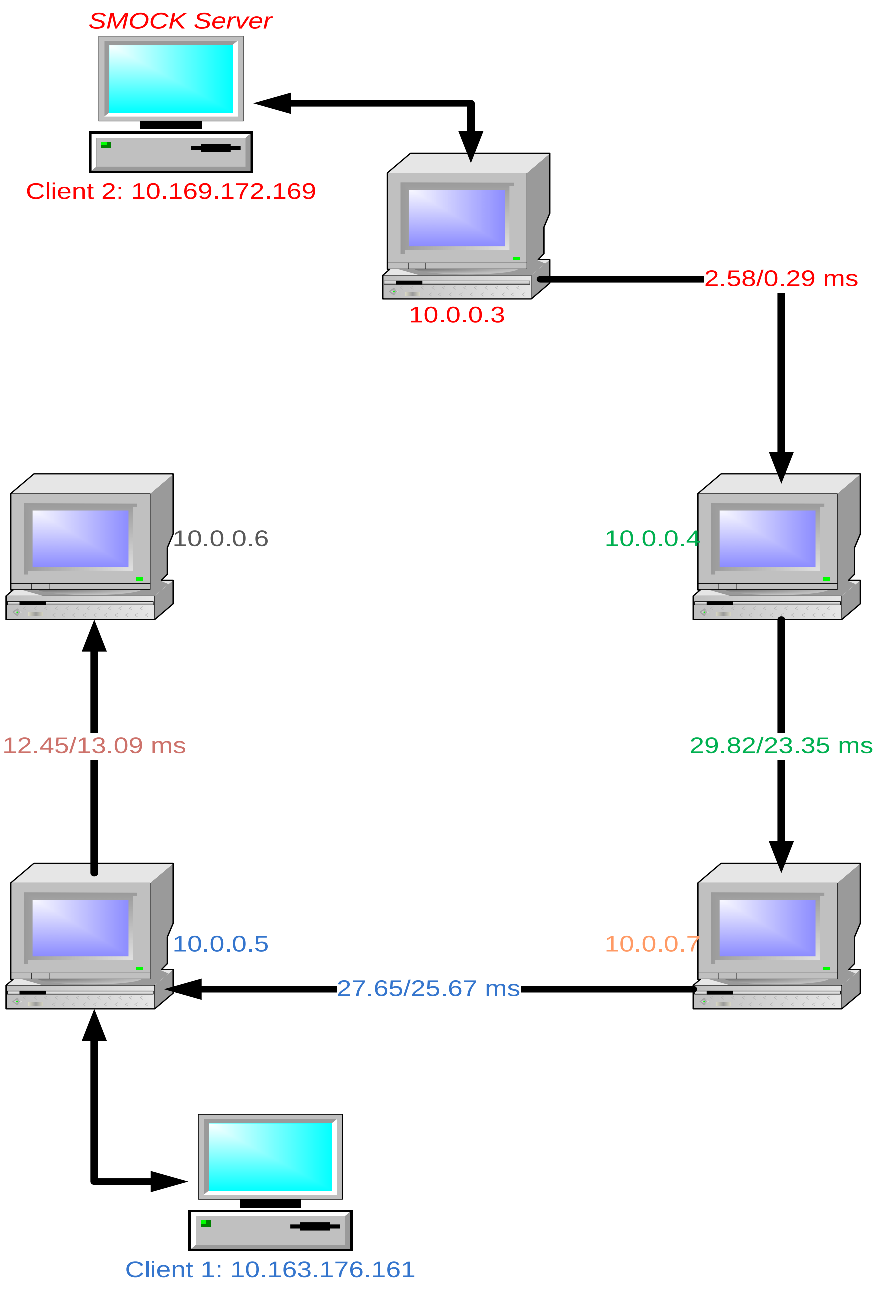

Figure 7.13: Topology of mesh nodes (with mean/variance of round trip times) - 1-dBmW transmit power

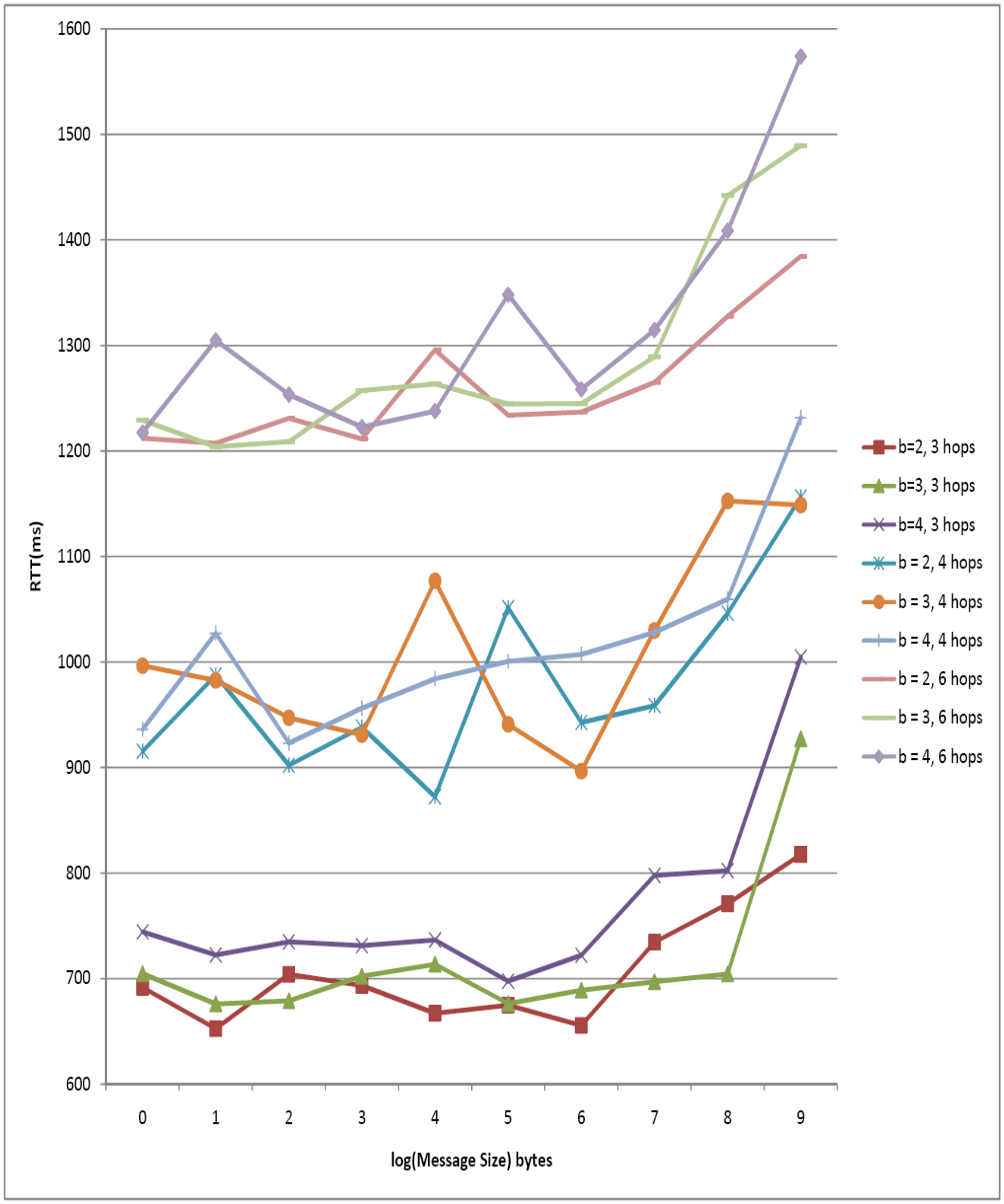

Figure 7.14: Average round trip time vs. log(message size) - Scenario 1 (end-to-end encryption), 1024-bit keys, 1-dBmW transmit power

In our next set of experiments, we changed the key sizes to 2048 bits and observed the round trip times. The results for the three-hop, four-hop and six-hop cases (for all b values) are shown in Figure 7.15. As in the 1024-bit key case, there is an enormous difference between the RTT times for three, four, and six hops. As noted before, the RTT value increases exponentially with respect to the data size when the data size is greater than or equal to the key size. In this case, the cutoff mark is 8 (since 2048 bits corresponds to 2^8 bytes). For the three-hop case, the RTT values range from 730 ms to 1390 ms. For the four-hop case, the RTT values range from 1130 ms to 1630 ms. For the six-hop case, RTT values vary from 1260 ms to 1880 ms. From Figure 7.15, we note that the variance of RTT times using 2048-bit keys is much higher than that of RTT times using 1024-bit keys. For example, the RTT values for the network which have 2048-bit SMOCK keys and $b = 4$ have higher variance than the RTT values for the network which uses 1024-bit keys and $b = 4$. This is because the processing/routing overhead between neighboring nodes increases due to the increased data size (since in RSA, the encrypted data has the same size as the key).

We next considered the performance under a different transmit value, namely 19 dBmW (which is the default and maximum power value). Similar to the previous scenario, we consider the performance of SMOCK (over various hops) for two key sizes, and three b values for each key size. The RTT values between the mesh nodes are shown in Figure 7.16. Figure 7.17 shows the round trip time for each b value using 1024-bit SMOCK keys. For three-hops, the RTT values range from 350 ms to 800 ms. For the four hop case, the values range from 360 ms to 830 ms. For the six-hop case, the values range from approximately 370 ms to 900 ms. The results show that there is little difference between the round trip times for all b values and all hops. This is because, compared to the 1 dBmW case, there is little overhead due to packet transmissions/reordering (since the transmit power is

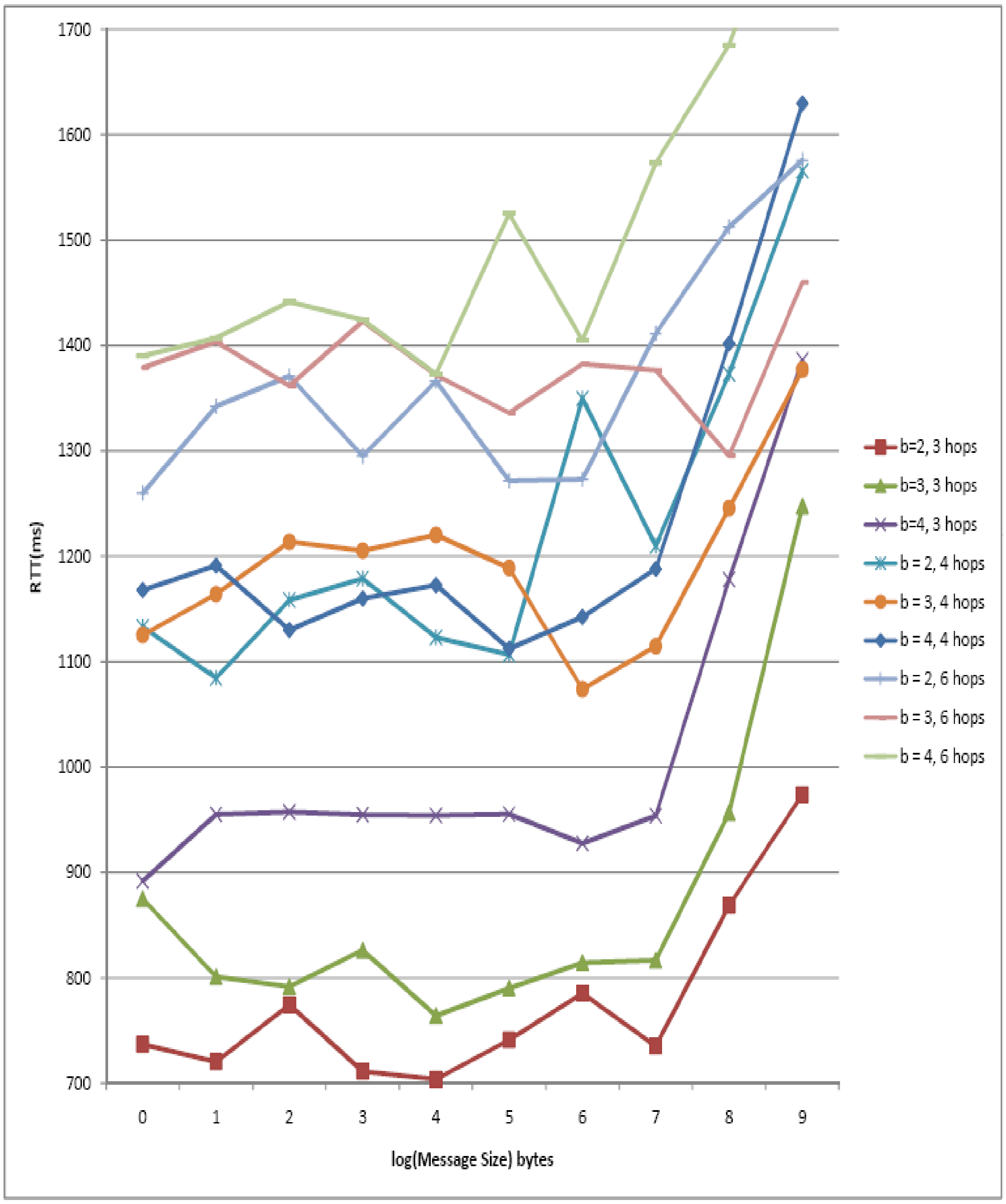

Figure 7.15: Average round trip time vs. log(message size) - Scenario 1 (end-to-end encryption), 2048-bit keys, 1-dBmW transmit power

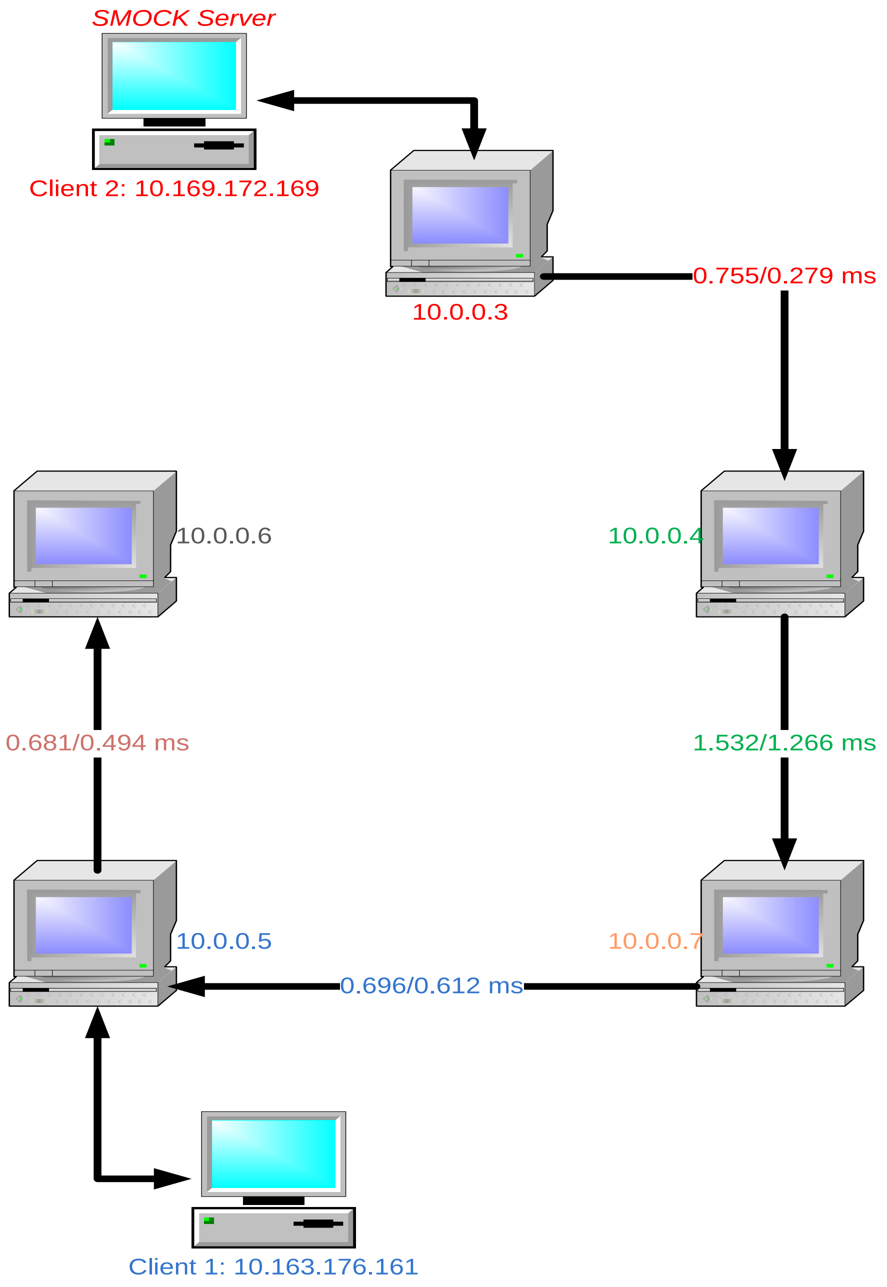

Figure 7.16: Topology of mesh nodes (with mean/variance of round trip times) - 19-dBmW transmit power

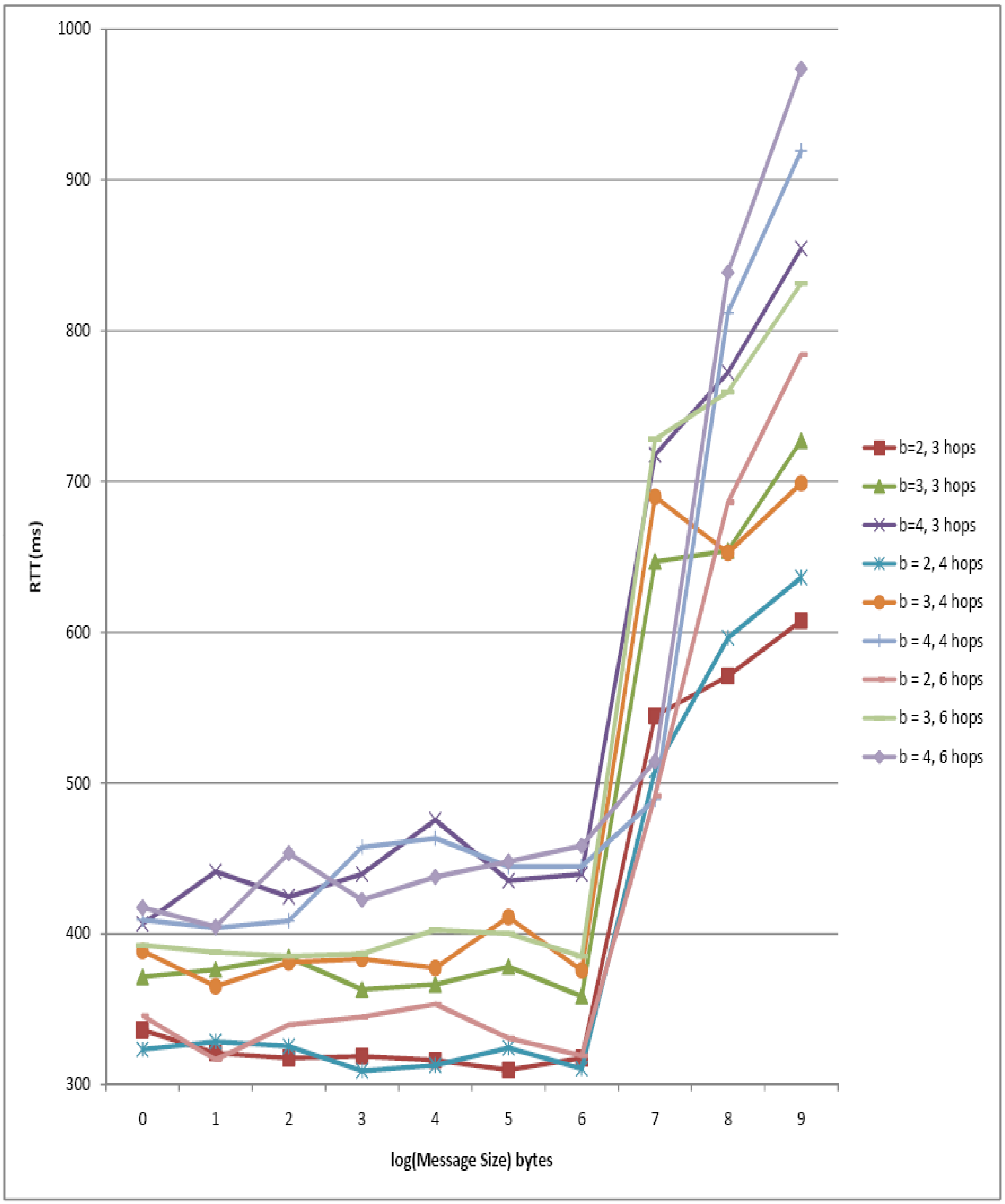

Figure 7.17: Average round trip time vs. log(message size) - Scenario 1 (end-to-end encryption), 1024-bit keys, 19-dBmW transmit power

maximum, thus leading to minimal packet transmission errors). Because of this, the only factor contributing to the round trip time is the overhead due to two factors, namely the delay between the nodes (which is now entirely dependent on the physical barriers between the nodes) and the encryption/decryption using the SMOCK keys. However, since the round trip times are similar for all b values, the overhead due to the number of encryptions/decryptions is much less significant as compared to the overhead due to the delay.

In our next set of experiments, we again change the SMOCK key sizes to 2048 bits and observe the round trip times. The results for the three-hop, four-hop and six-hop cases (for all b values) are shown in Figure 7.18. From the results, we see that for three hops, the RTT values range from 490 ms to 1020 ms. For four hops, the RTT values range from 600 ms to 1140 ms. For six hops, the RTT values range from 630 ms to 1270 ms. The results show that, similar to the 1024-bit (and 19-dBmW transmit power) case, having a lower b value and higher hop count results in a lower RTT value than having higher b value and lower hop count. For example, the RTT values for when $b = 3$ and the hop count is three, are higher than those for when $b = 2$ and the hop count is six. This shows that the encryption overhead is more significant than the communication overhead. Also, the encryption overhead when using 2048-bit SMOCK keys is higher than when using 1024-bit SMOCK keys, due to the fact that the RSA encrypt and decrypt operations take longer time on larger key sizes (as shown in the subsection covering encryption/decryption times).

Scenario 2 (hop-by-hop encryption) performance

In this case, we first set the transmit power of each mesh node to 1 dBmW and measure the average ping time between each node and its neighbor over 10 trials. The node topology along with the mean/variance of RTT values between the

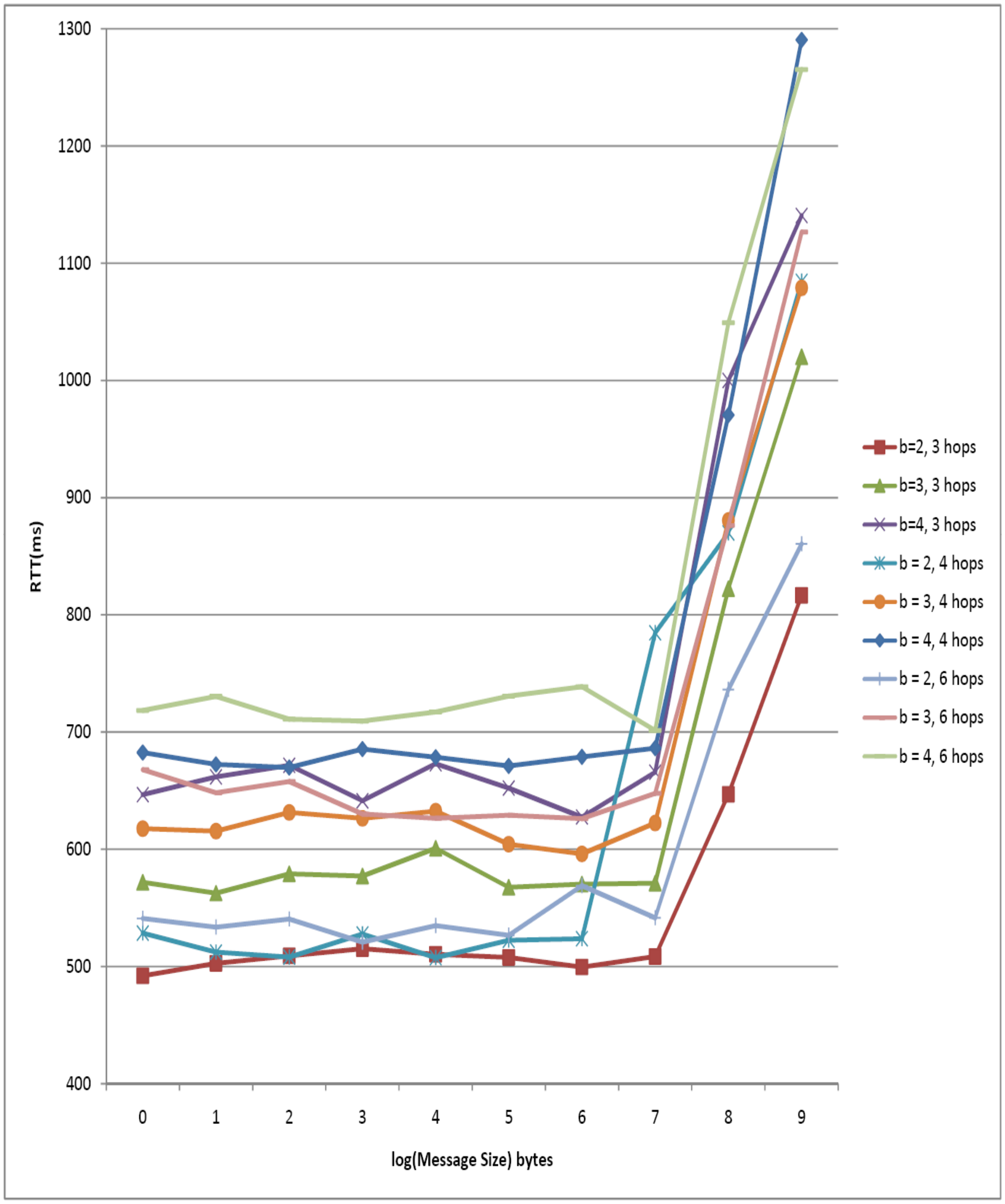

Figure 7.18: Average round trip time vs. log(message size) - Scenario 1 (end-to-end encryption), 2048-bit keys, 19-dBmW transmit power

nodes is shown in Figure 7.13. In Scenario 2, we consider the performance of SMOCK (over various hops) for two key sizes, and two b values for each key size. Figure 7.19 shows the round trip time for each b value using 1024-bit SMOCK keys. The results show a sizable difference between the RTT values for three hops versus four and six hops. For the three-hop case, the RTT values vary from 280 ms to 1300 ms. For the four-hop case, the RTT values vary from 840 ms to 2800 ms. For the six-hop case, the RTT values vary from 1390 ms to 4570 ms. A few observations can be made from this graph. First of all, there is an enormous increase in RTT values as the message size is greater than or equal to the key size. Secondly, the RTT values increase as the b value increases (keeping the number of hops constant). This is because the message is encrypted and decrypted at each hop, and thus an increase in b leads to additional overhead at each hop. So, for example, if the message size is 64 bytes (and key size is 128 bytes), the number of hops is three (which implies that the packet travels through two mesh nodes), and $b = 2$, then there are four calls to the *smock_encrypt()* function and four calls to the *smock_decrypt()* function (when we send the packet and receive a same-sized reply from the destination), thus resulting in a total of eight calls.

However, if $b = 3$, then sending and receiving the same data size over the same number of hops results in six calls to the *smock_encrypt()* function and six calls to the *smock_decrypt()* function, resulting in a total of 12 calls, thus leading to a larger RTT value. Also, if b is kept constant, then the RTT value increases as the number of hops increases. This is because each additional hop leads to an additional encrypt/decrypt call each way, thus adding to the round trip time. For example, if the message size is 64 bytes (and key size size is 128 bytes), $b = 3$ and the number of hops is three, then there are a total of 12 encryption-related calls, as shown above. However, if the number of hops is four, then sending and receiving the same data size using the same b value results in 12 calls to

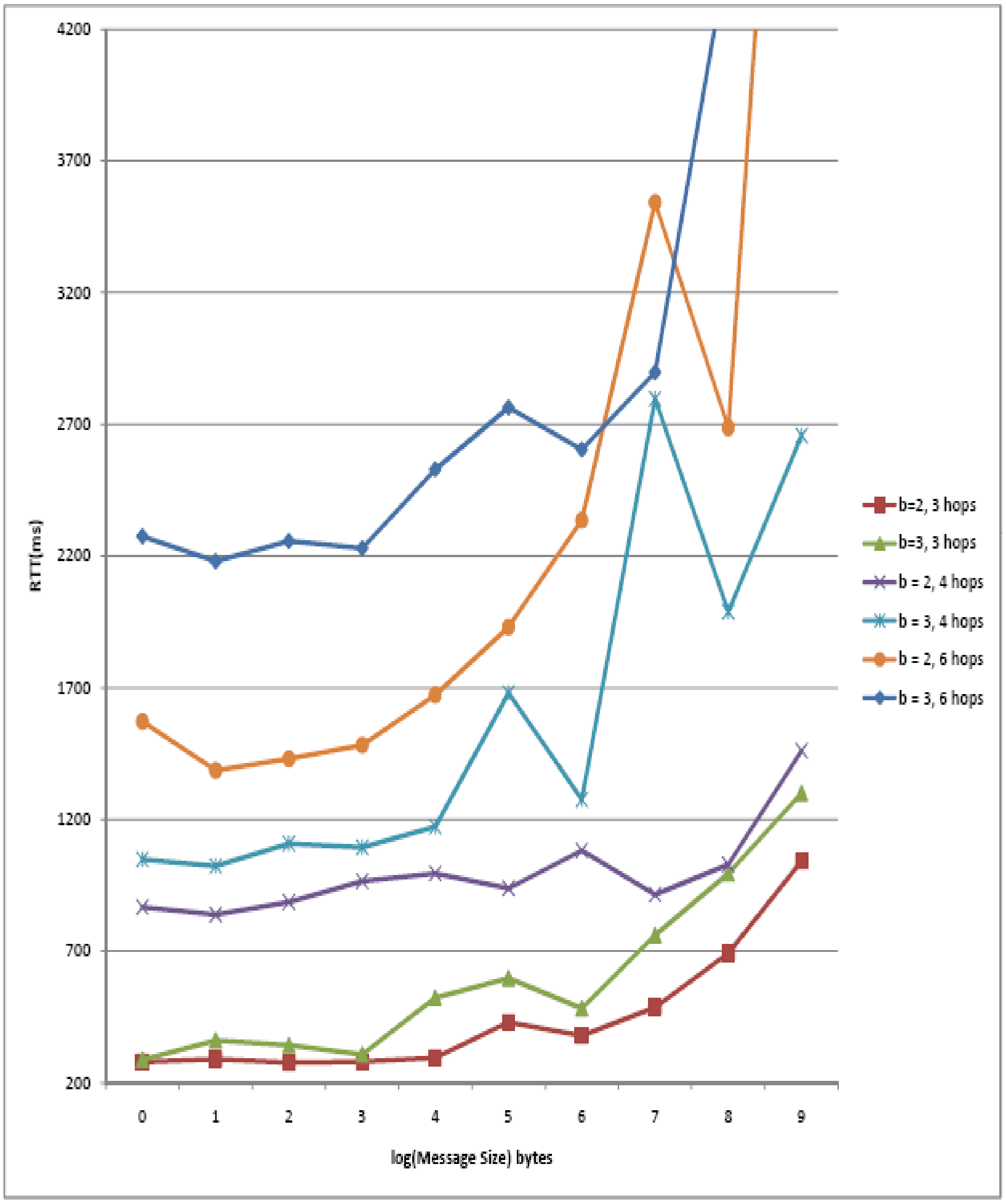

Figure 7.19: Average round trip time vs. log(message size) - Scenario 2 (hop-by-hop encryption), 1024-bit keys, 1-dBmW transmit power

smock_encrypt() and 12 calls to *smock_encrypt()*, thus resulting in a total of 24 calls. Finally, as the number of hops increases, the difference in RTT values for $b = 2$ and $b = 3$ increases. For example, the difference between the RTT values (for when $b = 2$ and $b = 3$) when the number of hops is three, is less than the difference between the values when the number of hops is six. This is unlike Scenario 1, where this difference stays constant. This increasing difference between the values (with respect to the number of hops) is due to the fact that in Scenario 2, every additional hop carries not only a communication overhead, but also an encryption overhead (unlike in Scenario 1, where every additional hop results in only a communication overhead) which increases with respect to b.

As in Scenario 1, it should be noted that for each b value, if the data size is greater than or equal to the key size, the RTT time increases exponentially with respect to the data size. Furthermore, the rate of exponential increase is directly proportional to the number of hops (if b is kept constant). This is because when the message size is larger than the key size, the number of calls to *smock_encrypt* and *smock_decrypt* increases exponentially with respect to the message size. However, the number of encryption calls is also dependent on the number of hops, since each additional hop leads to an additional exponential overhead due to encryption. So, the number of hops (that the packet has to travel) determines the rate of exponential increase.

In our next set of experiments, we change the SMOCK key size to 2048 bits and consider the round trip time under Scenario 2. The results are shown in Figure 7.20. From the figure, we note that for the three-hop case, the RTT values range from 760 ms to 1521 ms. For the four-hop case, the RTT values range from 2350 ms to 5020 ms. For the six-hop case, the RTT values range from 4140 ms to 11780 ms. In this figure, we make all the observations that we made in the previous case (namely 1024-bit keys). The round trip time is directly proportional to b, as well

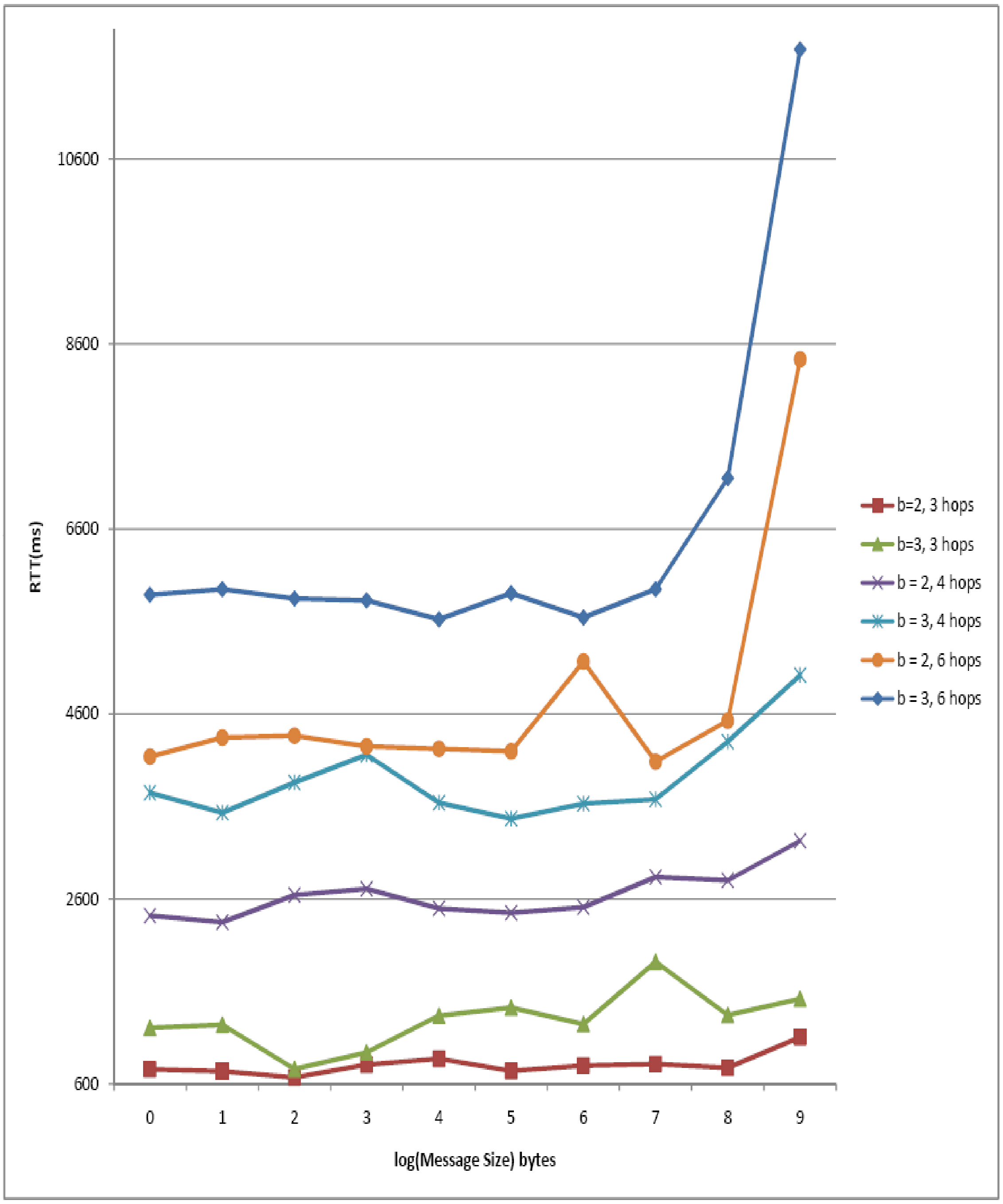

Figure 7.20: Average round trip time vs. log(message size) - Scenario 2 (hop-by-hop encryption), 2048-bit keys, 1-dBmW transmit power

as the number of hops. Also, the difference between the RTT values (for $b = 2$ and $b = 3$) is directly proportional to the number of hops. Moreover, the RTT values increase exponentially when the message size is greater than or equal to the key size. The rate of this exponential increase is determined by the number of hops in the network. Furthermore, the RTT values are directly proportional to both the b value as well as the number of hops (similar to the 1024-bit case).

In the next set of experiments, we modified that transmit power to 19 dBmW and measured the average round trip time for Scenario 2, using 1024-bit keys. The results for this are shown in Figure 7.21. From the figure, we note that for the three-hop case, the RTT values range from 180 ms to 630 ms. For the four-hop case, the RTT values range from 600 ms to 1780 ms. For the six-hop case, the RTT values range from 860 ms to 2370 ms. As before, we note that the RTT values are directly proportional to both b and the hop count. Also, the difference between the RTT values (for $b = 2$ and $b = 3$) is directly proportional to the number of hops. In this result, we note that the RTT values for the case where $b = 2$ and the hop count is six are less than those of the case where $b = 3$ and the hop count is four. This is due to the fact that with maximum transmit power, there is minimized transmission overhead at each hop, since fewer packets are sent out of order, or are lost. Due to this, we note that with maximized transmit power, the RTT values for smaller b and larger hop count may be smaller than those with larger b and smaller hop count (and this happens as b increases).

We then changed the key size to 2048-bit keys and measured the RTT values. For this case, the results are shown in Figure 7.22. For the three-hop case, the RTT values range from 460 ms to 1420 ms. For the four-hop case, the RTT values range from 1450 ms to 3620 ms. For the six-hop case, the RTT values range from 2070 ms to 4180 ms. As before, we note that the RTT values are directly proportional to both b and the hop count. Similar to previous experiments, the

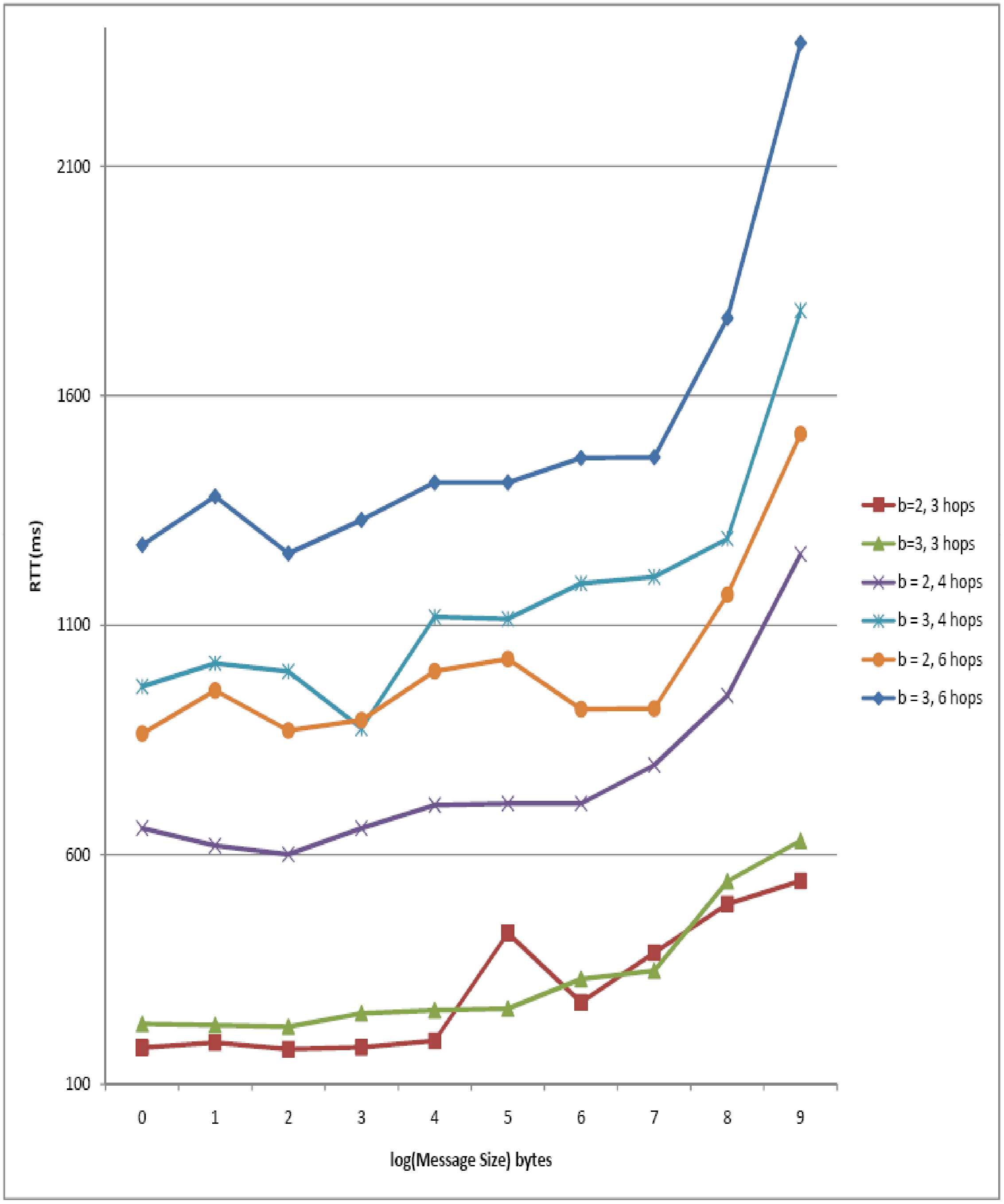

Figure 7.21: Average round trip time vs. log(message size) - Scenario 2 (hop-by-hop encryption), 1024-bit keys, 19-dBmW transmit power

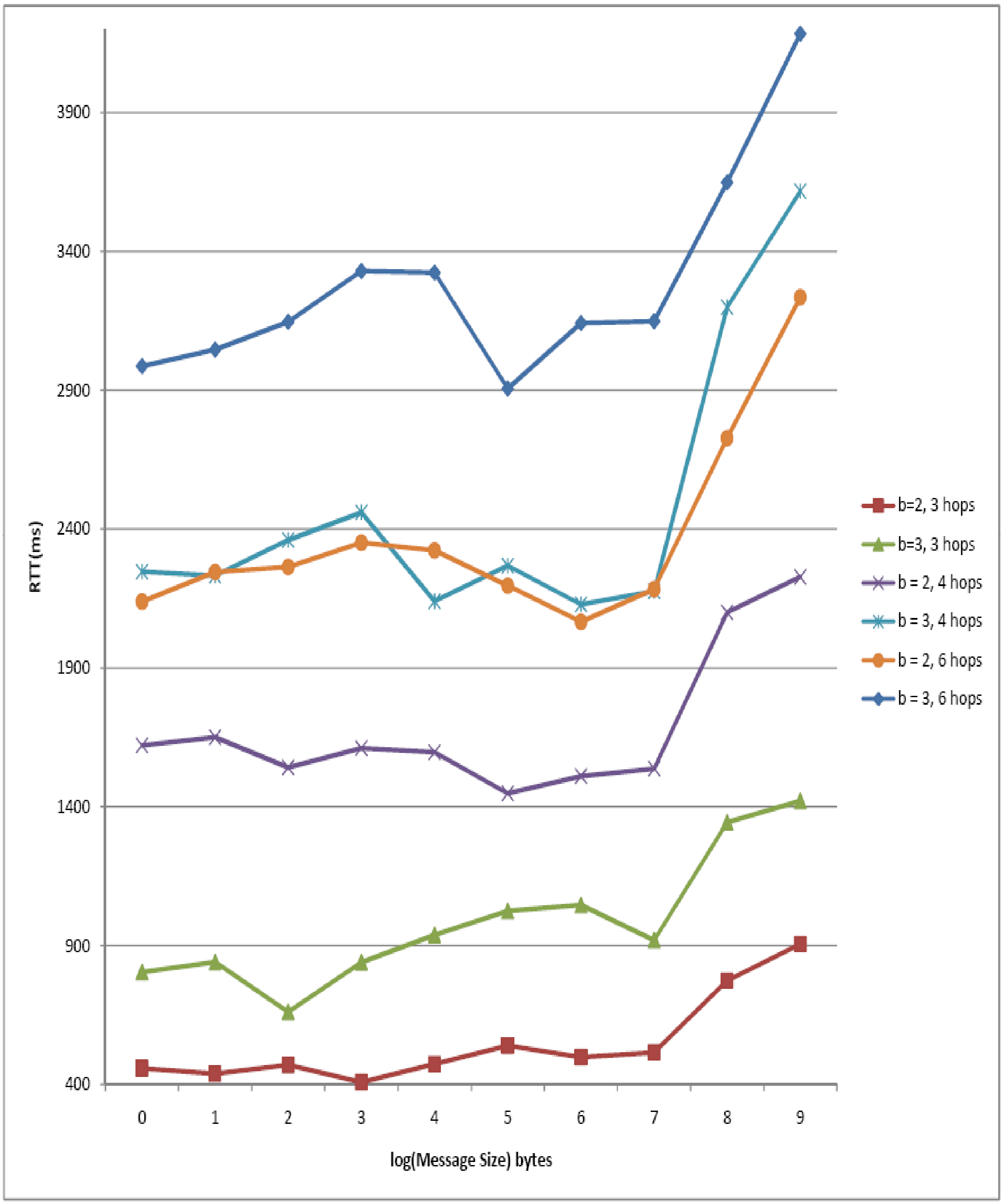

Figure 7.22: Average round trip time vs. log(message size) - Scenario 2 (hop-by-hop encryption), 2048-bit keys, 19-dBmW transmit power

difference between the RTT values (for $b = 2$ and $b = 3$) is directly proportional to both b and the hop count, and the difference between the RTT values (for $b = 2$ and $b = 3$) is directly proportional to the number of hops. As in the 1024-bit key case, the RTT values for when $b = 2$ and the hop count is six are less than those for when $b = 3$ and the hop count is four.

Scenario 1 versus Scenario 2 Performance

Here, we compare the RTT values as a result of using Scenario 1 versus Scenario 2 of SMOCK integration. In the first case, we compare the RTT values when using 1024-bit SMOCK keys and setting the transmit power to 1 dBmW. The results are shown in Figure 7.23. In the figure, we note that for the three-hop case, the RTT values for Scenario 2 are less than those for Scenario 1. This is due to the fact that the encryption overhead is the same in both scenarios, but the communication overhead is higher for Scenario 1. In this case, both scenarios make $2 * b$ calls to *smock_encrypt*() and *smock_decrypt*(), resulting in $4 * b$ calls in total (where b is the number of private keys stored in each node). However, Scenario 1 has a larger communication overhead since in Scenario 1, a *SMOCK ID Request* message needs to be sent to the destination node in order to receive its SMOCK ID. This is unlike in Scenario 2, where nodes exchange SMOCK IDs with their neighbors. However, if the message size becomes larger than the key size, the encryption overhead becomes more significant than the communication overhead, so the RTT values for Scenario 2 exceed those for Scenario 1 (since in Scenario 2, the Spines daemon breaks down the original data into smaller chunks, thus resulting in more calls to *smock_encrypt*()).

However, when the hop count is four, we notice that for $b = 2$, the RTT values are identical for both scenarios. This indicates that the increased encryption overhead for Scenario 2 (the additional hop results in at least eight additional calls

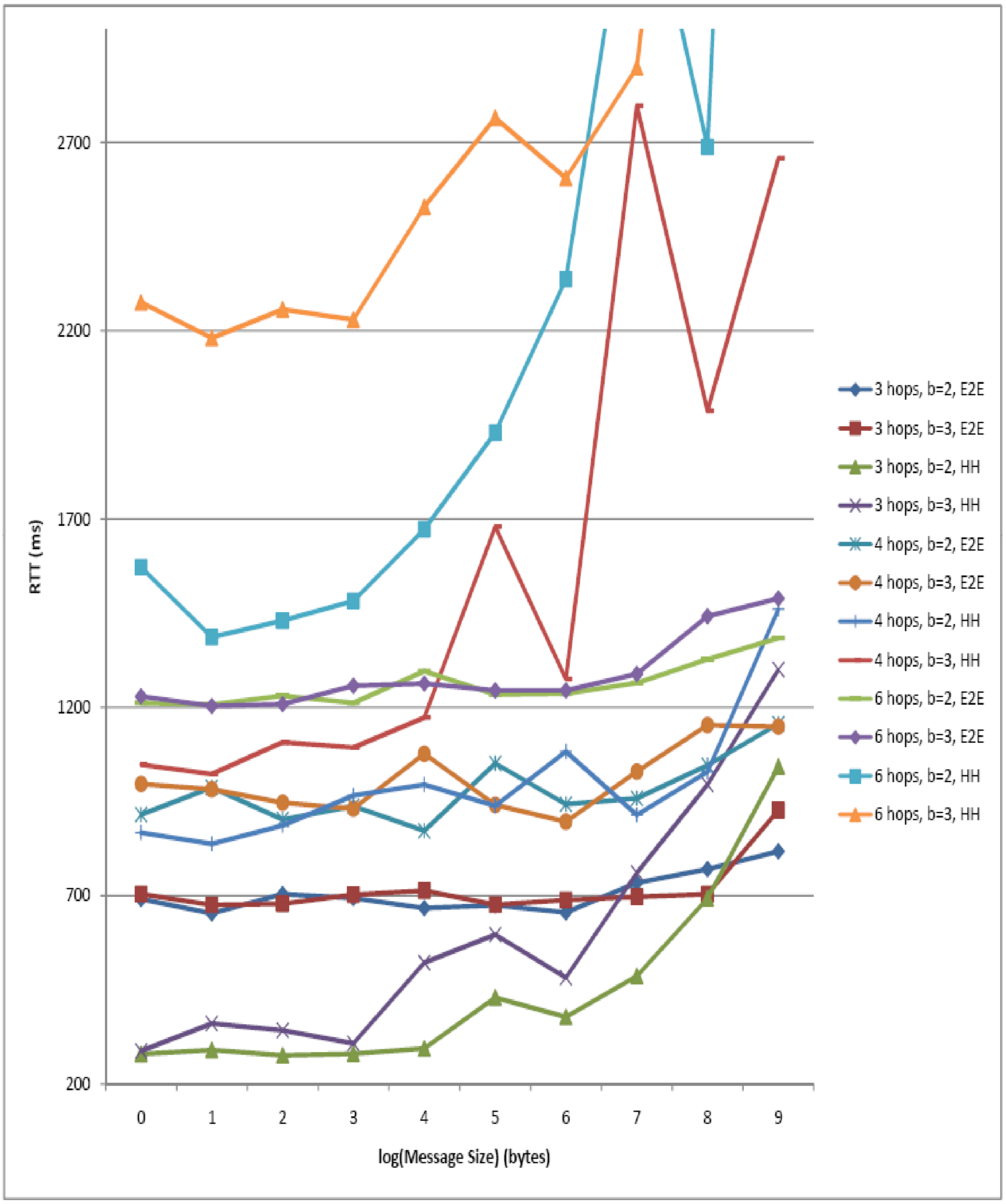

Figure 7.23: Average round trip time vs. log(message size) - Scenario 1 vs. Scenario 2, 1024-bit keys, 1-dBmW transmit power

to *smock_encrypt*() and *smock_decrypt*()) offsets the additional communication overhead in Scenario 1. However, when $b = 3$, the RTT values for a message traveling four hops is higher when in Scenario 2. This is because the encryption overhead is increasingly significant in comparison with the communication overhead, since there are now a total of at least 24 calls to *smock_encrypt*() and *smock_decrypt*() (provided the message size is less than the key size) versus 12 total calls for Scenario 1. The encryption overhead becomes even more significant when the hop count increases to six (since there are $8 * b$ calls to *smock_encrypt*() and $8 * b$ calls to *smock_decrypt*()). This results in the RTT values when using Scenario 2 being much higher over six hops than those for Scenario 1.

When using 2048-bit SMOCK keys (and keeping the transmit power at 1 dBmW), the comparison between Scenario 1 and Scenario 2 is shown in Figure 7.24. When using 2048-bit SMOCK keys, the encryption overhead is higher than when using 1024-bit SMOCK keys. This is due to the higher encryption/decryption times, as shown in the section describing the encryption/decryption times using various SMOCK key sizes. Hence, the RTT values (for when $b = 2$) for Scenario 1 are similar to that for Scenario 2. However, when $b = 3$, the RTT values for Scenario 2 are much higher than that for Scenario 1. This is unlike the case when using 1024-bit SMOCK keys (where the RTT values are similar), since the encryption overhead is much higher. As expected, the RTT values for Scenario 2 are much higher than Scenario 1 for all other hop counts and b values (this is clearly seen in the figure). This makes Scenario 2 impractical for large key sizes, since larger key sizes result in larger encryption/decryption times. This in turn results in added delay at each hop, thus increasing the round trip time.

We then examine the differences between Scenario 1 and Scenario 2 when the transmit power at each mesh node is set to 19 dBmW. Figure 7.25 shows the RTT values when using 1024-bit SMOCK keys. This figure shows that when the hop

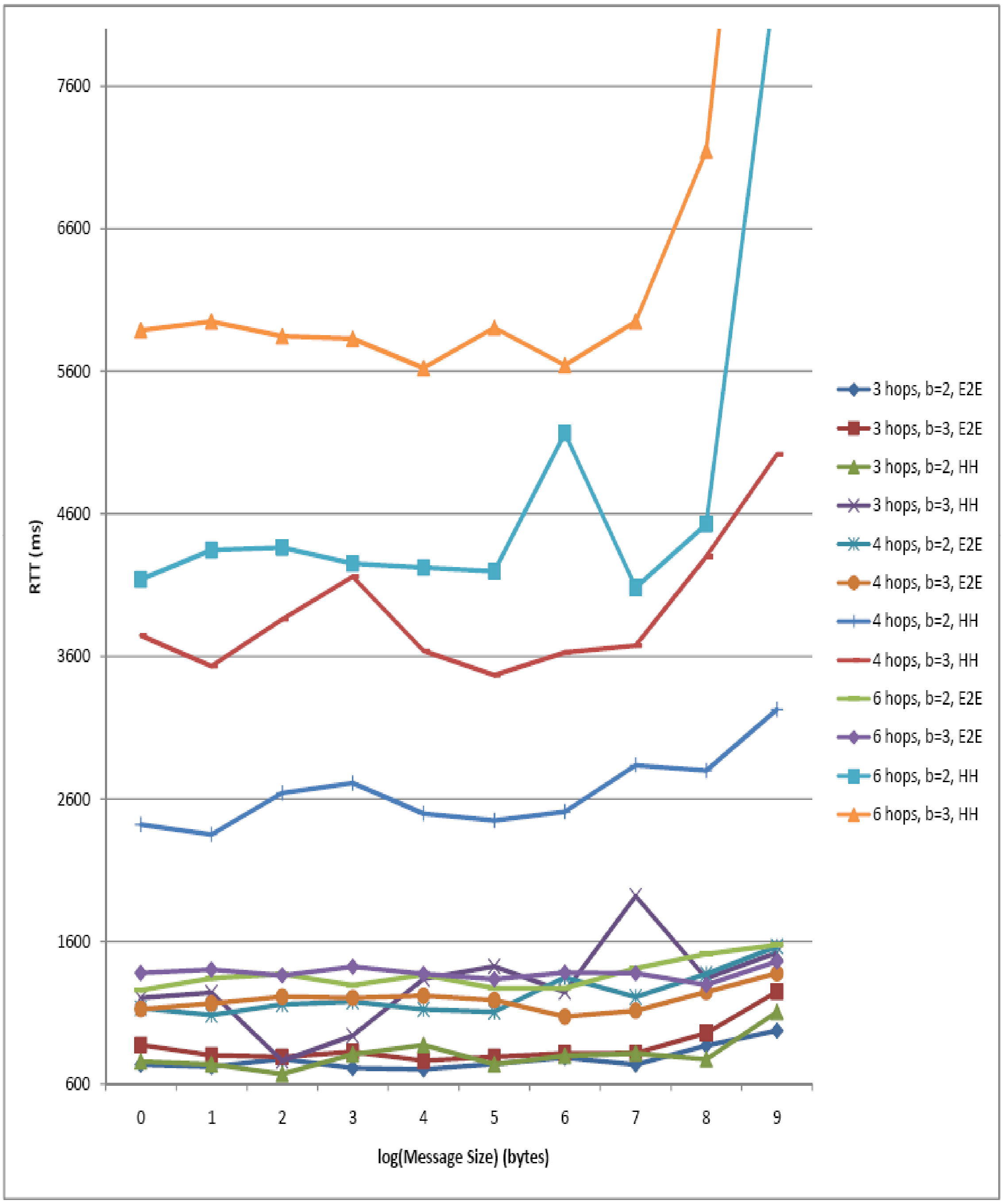

Figure 7.24: Average round trip time vs. log(message size) - Scenario 1 vs. Scenario 2, 2048-bit keys, 1-dBmW transmit power

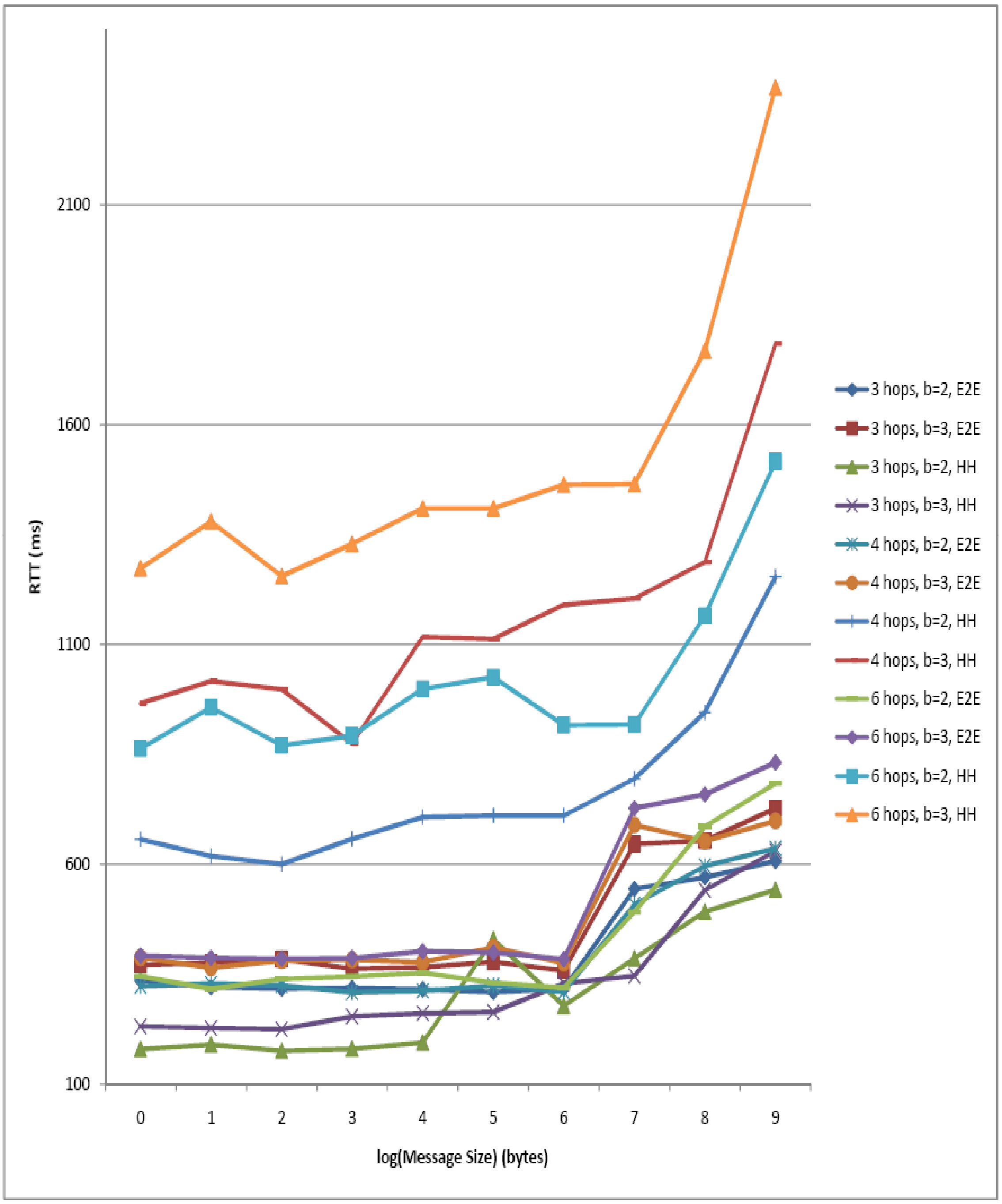

Figure 7.25: Average round trip time vs. log(message size) - Scenario 1 vs. Scenario 2, 1024-bit keys, 19-dBmW transmit power

count is three, the RTT values for Scenario 2 are much less than that of Scenario 1. This is due to the fact that as stated before, Scenario 1 has an additional communication overhead due to sending and receiving the *SMOCK ID Request* message. The RTT values of Scenario 1 are larger than that of Scenario 2 for all other hop counts and b values. This is due to the fact that maximizing the transmit power minimized the communication overhead, thus making it far less significant than the encryption overhead.

We then observed the comparisons between Scenario 1 and Scenario 2 using 2048-bit SMOCK keys, and this is shown in Figure 7.26. In this case, the encryption overhead is much higher than the communication overhead (since the key size is large and the transmit power is maximized). Thus, the RTT values for Scenario 2 are lower than those for Scenario 1 only when $b = 2$ and the hop count is three. However, in all other cases, the RTT values for Scenario 2 are much higher than those for Scenario 1. In fact, the RTT values for Scenario 2 are quite distinct in the figure. This shows that hop-by-hop encryption is not feasible for 2048-bit keys unless the average hop count for messages is less than four.

Thus, the SMOCK performance analysis comprised measuring the parameter generation time, key generation time, memory usage, and encryption/decryption times due to SMOCK keys. The parameter generation time was on the scale of microseconds, but the encryption/decryption times were on the scale of tens of milliseconds. The encryption/decryption times when using 4096-bit SMOCK keys were much higher than when using smaller sized keys. Therefore, it was considered infeasible to use key sizes larger than 2048 bits, and thus only 1024-bit and 2048-bit SMOCK keys were to be used when measuring the round trip times. When measuring round trip times, both the end-to-end (Scenario 1) and hop-by-hop (Scenario 2) encryption scenarios were used. For each scenario, we considered two different transmit powers, namely the minimal and maximal powers of 1 dBmW

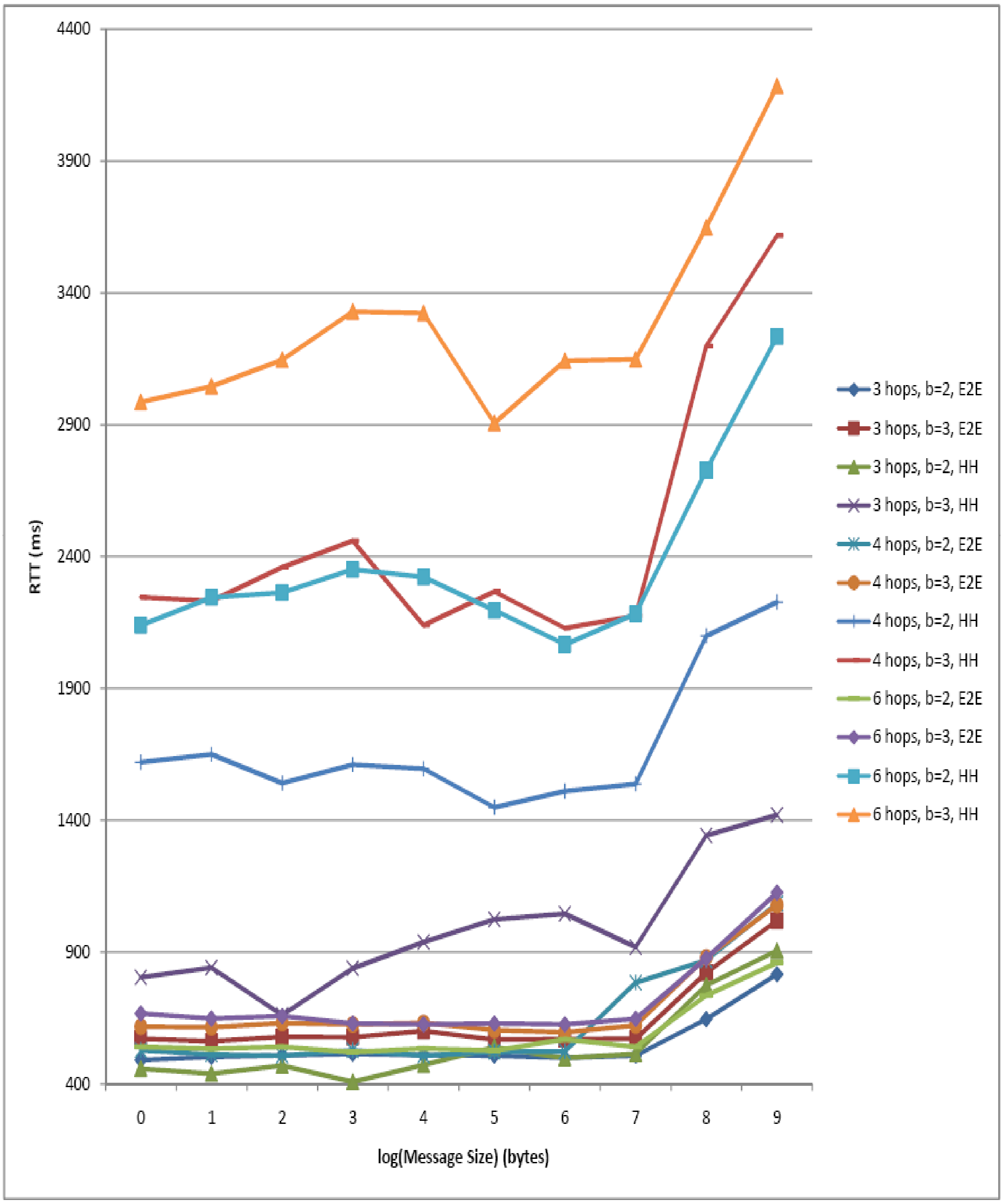

Figure 7.26: Average round trip time vs. log(message size) - Scenario 1 vs. Scenario 2, 2048-bit keys, 19-dBmW transmit power

and 19 dBmW, respectively.

First, we analyzed end-to-end encryption. For the 1-dBmW scenario and both 1024-bit and 2048-bit keys, there is an increase between RTT values for when b increases from two to four when using end-to-end encryption. However, there is a more significant increase between the RTT values for when the hop count is three, four, and six. This shows that the communication overhead (which increases for each hop) is more significant than the encryption overhead when the transmit power is 1 dBmW and we are using end-to-end encryption. Also, for both transmit powers and for all b values, key sizes, and hop counts, the round trip time increases exponentially (with respect to the message size) when the message size is greater than or equal to the key size. The transmit power was then adjusted to the maximum (19 dBmW). With this, when using 1024-bit keys, there is a very small increase in round trip time when the hop count increases from three to six. However, there is a much larger increase in RTT values when b increases from two to four. Hence, in this case, the encryption overhead is much larger than the communication overhead. The difference between the round trip times for various b values is much higher when using 2048-bit keys, since the encryption overhead increases when using larger sized keys.

When using hop-by-hop encryption and having 1-dBmW transmit power, we note that as before, the round trip times increase as both b and the hop count increase. However, as the hop count increases, the difference between the round trip times for various b values increases, due to the fact that the encryption overhead not only increases as b increases, but also increases as the hop count increases (since encryption/decryption operations are performed at each hop). This is the observation for both key sizes. When the transmit power for all nodes is set to 19 dBmW, we see the same result where the difference between the round trip times for various b values increases as the hop count increases.

When comparing the round trip times of end-to-end encryption versus hop-by-hop encrytion, we note that where there is a high delay between mesh nodes (as a result of low transmit powers, high latencies, or other factors), hop-by-hop encryption results in lower round trip times when the hop count is low (less than or equal to four hops in our case). However for high hop counts, the total encryption overhead increases significantly when using hop-by-hop encryption, thus resulting in higher round trip times. Since the encryption overhead increases as the key size increases, we note that there is little difference between end-to-end encryption and hop-by-hop encryption for the minimum hop count of three. However, as the hop count increases, the RTT values for hop-by-hop encryption are much higher than the RTT values for end-to-end encryption. When the hop count is increased to 19 dBmW and the key size is 1024 bits, the RTT values for hop-by-hop encryption are much higher than those for end-to-end encryption (except when $b = 2$ and the hop count is three). When the key size is 2048 bits, the RTT values for hop-by-hop encryption are always higher than those for end-to-end encryption (but marginally smaller when $b = 2$ and the hop count is three).

Thus, the SMOCK implementation utilized in the mesh network depends on the network model. The space/time constraints (if any) on the mesh nodes and clients, size of the network (which affects the b parameter), the average number of hops that each message travels, the average size of each message, and the average delay between the mesh nodes (which is a factor of physical distance between mesh nodes as well as the transmit powers of wireless devices) determine whether to utilize either end-to-end or hop-by-hop encryption. As stated before, the end-to-end encryption scenario stores the SMOCK keys on the mesh clients, and thus may require a certain amount of minimum memory/processor power in each client. This scenario works best when the communication overhead is low, since it needs to send *SMOCK ID Request* messages when encrypting every message. On the other hand, the hop-by-hop scenario places no requirements on mesh clients (except for minimal space/time requirements to perform symmetric-key encryption and store a key shared with the host mesh node), as they have the mesh nodes store the SMOCK keys. This scenario works best when the communication overhead is high with respect to the encryption overhead. However, if the average number of hops for each message in the network is high, then the hop-by-hop scenario is not the best option since each additional hop adds an additional encryption overhead. Hence, the network model determines the type of SMOCK encryption utilized on mesh nodes.

CHAPTER 8

CONCLUSION

This thesis provides the implementation of a key management scheme for wireless mesh networks. WMNs have various characteristics, which include multihop coverage, ad hoc networking support, limited mobility, compatibility with existing wireless standards, and multiple types of network access. Furthermore, they can be organized in three separate ways, namely infrastructure/backbone WMNs (where mesh clients provide application-level services and mesh nodes form a self-configuring, self-healing network which provides routing/gateway functionalities), client WMNs (where mesh clients provide both routing/gateway as well as application-level services), and hybrid WMNs (where there are nodes which provide only application-level or routing services, as well as nodes which provide both types of services). We then explored various key management schemes, which are roughly broken down into two types - symmetric and asymmetric key management schemes.

However, the other key management schemes did not provide all the desired qualities for a key management scheme on the mesh network, which were the following: it needed to support a network of nodes that need not be in direct contact with each other; it needed to support a network of nodes that may move rapidly (thus changing the topology); it needed to support a network that may be susceptible to attacks. In order to facilitate all this, a key management scheme (which was introduced in [7]), namely SMOCK, was introduced. SMOCK is a combinatorial key management scheme that generates a key pool and provides mutually exclusive subsets of the key pool to each node. It provides memory efficiency and tries to minimize computational complexity and maximize resiliency. We then described the SMOCK implementation, which consisted of two modules, namely *SMOCK Server* and *SMOCK Client*. *SMOCK Server* takes in parameters such as network size and creates SMOCK keys (as well as its own public key) and waits for new connections from *SMOCK Client* modules. After this, it sends SMOCK public/private keys, generates a SMOCK ID (based on the subset of SMOCK private keys), encrypts this ID with its SMOCK key, and then sends this data to *SMOCK Client*. *SMOCK Client* consists of a set of API (including the two main functions, *smock_encrypt*() and *smock_decrypt*()) which applications can use to receive keys from *SMOCK Server* and use them to encrypt/decrypt data.

We then described a software, namely SMesh, that can be used to implement a mesh network. SMesh is a software that is installed on various nodes to create a mesh network, and provides various services to clients including gateway/routing functionality. SMesh utilizes Spines, a routing software that creates a router daemon and a set of API that can be used to access the daemon to route data. This API is utilized by SMesh which provides smooth handoff to clients, thus making it a viable option for applications that provide real-time services such as VoIP. Two integration scenarios of SMOCK with the SMesh software were proposed. The first scenario (end-to-end encryption) has the clients (of the mesh network) utilizing the SMOCK API to encrypt/decrypt data and send it to other clients. The second scenario (hop-by-hop encryption) comprises integrating *SMOCK Client* with the underlying Spines daemon. As data is sent by the SMesh clients to the Spines daemon (in order to be routed to another mesh node) via the *Smesh Client Interface*, it is first encrypted using SMOCK keys. The advantages and disadvantages of both scenarios were discussed. Then, the wireless mesh network testbed setup was described. We used 7 laptops (5 mesh nodes and 2 clients), each equipped with a wireless card and installed Linux, MadWifi, and SMesh software. We then described the topology setup and described the metrics that are measured and the parameters that are altered.

We analyzed the performance of *SMOCK Server* and *SMOCK Client* under both encryption scenarios (end-to-end and hop-by-hop encryption) as well as various parameters such as transmit power, b value (the size of the mutually exclusive subset of keys assigned to each node), key size, hop count, and message size and measured various metrics such as parameter generation time, key generation time, SMOCK memory usage, SMOCK encryption/decryption time, and round trip time. The results showed that key sizes greater than 2048 bits were impractical for SMOCK, and that the decision to utilize either of the scenarios depended on the network model. The average hop count, average message size, b parameter, and transmit power determine the scenario of encryption.

Thus, for a mesh network, SMOCK is a viable option. There are two scenarios of integration of SMOCK with the mesh network. Determining the approriate mode of integration depends entirely on the network model. That being said, SMOCK offers reasonable results when it comes to round trip times of messages, while satisfying properties desired of a key management scheme for mesh networks. It is hopeful that with advances in memory/processor power/networking, the highlighted deficiencies in SMOCK may be minimized, making it viable for all applications regardless of the network model.

REFERENCES

[1] I. F. Akyildiz, X. Wang, and W. Wang, “Wireless mesh networks: A survey,” *Computer Networks*, vol. 47, no. 4, pp. 445–487, 2005.

[2] A. C.-F. Chan, “Distributed symmetric key management for mobile ad hoc networks,” *IEEE INFOCOM*, vol. 4, pp. 2414–2424, 2004.

[3] H. Yang, J. Shu, X. Meng, and S. Lu, “Scan: Self-organized network-layer security in mobile ad hoc networks,” *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 2, pp. 261–273, 2006.

[4] L. Buttyán and J.-P. Hubaux, “Report on a working session on security in wireless ad hoc networks,” *Mobile Computing and Communications Review*, vol. 7, no. 1, pp. 74–94, 2003.

[5] V. Gupta, S. Krishnamurthy, and M. Faloutsos, “Denial of service attacks at the mac layer in wireless ad hoc networks,” in *MILCOM*, 2002. [Online]. Available: citeseer.ist.psu.edu/gupta02denial.html

[6] N. Borisov, I. Goldberg, and D. Wagner, “Intercepting mobile communications: the insecurity of 802.11,” in *MOBICOM*, 2001, pp. 180–189.

[7] W. He, Y. Huang, K. Nahrstedt, and W. C. Lee, “Smock: A scable method of cryptographic key management for mission-critical networks,” University of Illinois at Urbana-Champaign, Tech. Rep. UIUCDCS-R-2006-2734.

[8] L. Krishnamurthy, S. Conner, M. Yarvis, J. Chhabra, C. Ellison, C. Brabenac, and E. Tsui, “Meeting the demands of the digital home with high-speed multi-hop wireless networks,” *Intel Technology Journal*, vol. 6, no. 4, pp. 57–68, 2002. [Online]. Available: citeseer.ist.psu.edu/krishnamurthy02meeting.html

[9] J. Jun and M. Sichitiu, “The nominal capacity of wireless mesh networks,” *IEEE Wireless Communications*, vol. 10, no. 5, pp. 8–14, 2003. [Online]. Available: citeseer.ist.psu.edu/jun03nominal.html

[10] D. Highfill, “Field asset security in a smart grid world,” June 18, 2008.

[11] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Commun. ACM*, vol. 21, no. 12, pp. 993–999, 1978.

[12] J. Kohl and B. Neuman, "The kerberos network authentication service(v5)," September 1993. [Online]. Available: http://www.ietf.org/rfc/rfc1510.txt.

[13] M. Striki and J. S. Baras, "Key distribution protocols for multicast group commununication in manets," University of Maryland at College Park, Tech. Rep. TR2003-17, 2003.

[14] J. Nam, S. Kim, H. Yang, and D. Won, "Secure group communications over combined wired/wireless networks."

[15] L. Eschenauer and V. D. Gligor, "A key-management scheme for distributed sensor networks," in *ACM Conference on Computer and Communications Security*, 2002, pp. 41–47.

[16] H. Chan, A. Perrig, and D. X. Song, "Random key predistribution schemes for sensor networks," in *IEEE Symposium on Security and Privacy*, 2003, pp. 197–213.

[17] S. A. Çamtepe and B. Yener, "Combinatorial design of key distribution mechanisms for wireless sensor networks," *IEEE/ACM Trans. Netw.*, vol. 15, no. 2, pp. 346–358, 2007.

[18] S. Kent and T. Polk, "Public-key infrastructure (x.509) (pkix) charter," May 2002. [Online]. Available: http://www.ietf.org/pkix-charter.html.

[19] L. Zhou and Z. J. Haas, "Securing ad hoc networks," *IEEE Network*, vol. 13, no. 6, pp. 24–30, 1999. [Online]. Available: citeseer.ist.psu.edu/zhou99securing.html.

[20] J. Kong, P. Zerfos, H. Luo, S. Lu, and L. Zhang, "Providing robust and ubiquitous security support for mobile ad hoc networks," in *ICNP*, 2001, pp. 251–260.

[21] D. Malan, M. Welsh, and M. Smith, "A public-key infrastructure for key distribution in tinyos based on elliptic curve cryptography," in *First IEEE International Conference on Sensor and Ad Hoc Communications and Network*, 2004, pp. 71–80.

[22] G. Montenegro and C. Castelluccia, "Statistically unique and cryptographically verifiable (sucv) identifiers and addresses," in *NDSS*, 2002, pp. 221–232.

[23] S. Capkun, L. Buttyán, and J.-P. Hubaux, "Self-organized public-key management for mobile ad hoc networks," *IEEE Trans. Mob. Comput.*, vol. 2, no. 1, pp. 52–64, 2003.

[24] G. Ateniese, M. Steiner, and G. Tsudik, "New multiparty authentication services and key agreement protocols," *IEEE Journal on Selected Areas in Communications*, vol. 18, no. 4, pp. 628–639, 2000. [Online]. Available: http://citeseer.ist.psu.edu/ateniese99new.html.

[25] K. Becker and U. Wille, "Communication complexity of group key distribution," in *ACM Conference on Computer and Communications Security*, 1998. [Online]. Available: http://citeseer.ist.psu.edu/becker98communication.html pp. 1–6.

[26] A. T. Sherman and D. A. McGrew, "Key establishment in large dynamic groups using one-way function trees," *IEEE Trans. Software Eng.*, vol. 29, no. 5, pp. 444–458, 2003.

[27] Y. Kim, A. Perrig, and G. Tsudik, "Communication-efficient group key agreement," in *SEC*, 2001, pp. 229–244.

[28] M. J. Cox, R. S. EngelSchall, S. Henson, and B. Laurie, "The openssl project." [Online]. Available: http://www.openssl.org.

[29] Y. Amir, C. Danilov, M. Hilsdale, R. Musaloiu-Elefteri, and N. Rivera, "Fast handoff for seamless wireless mesh networks," in *MobiSys*, 2006, pp. 83–95.

[30] Y. Amir and C. Danilov, "Reliable communication in overlay networks," in *DSN*, 2003, pp. 511–520.

[31] S. Leffler and G. Chesson, "Madwifi," 2005. [Online]. Available: http://madwifi.org.