SANDIA REPORT

SAND 2005-5463
Unlimited Release
Printed September 2005

Final Report for the Mobile Node Authentication LDRD Project

John T. Michalski and Andrew J. Lanzone

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Sandia National Laboratories

SAND 2005-5463
Unlimited Release
Printed September 2005

Final Report for the Mobile Node Authentication LDRD Project

John T. Michalski
Networked Systems Survivability & Assurance Department

Andrew J. Lanzone
Cryptography and Information Systems Surety Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-0785

Abstract

In hostile ad hoc wireless communication environments, such as battlefield networks, end-node authentication is critical. In a wired infrastructure, this authentication service is typically facilitated by a centrally-located “authentication certificate generator” such as a Certificate Authority (CA) server. This centralized approach is ill-suited to meet the needs of mobile ad hoc networks, such as those required by military systems, because of the unpredictable connectivity and dynamic routing. There is a need for a secure and robust approach to mobile node authentication.

Current mechanisms either assign a pre-shared key (shared by all participating parties) or require that each node retain a collection of individual keys that are used to communicate with other individual nodes. Both of these approaches have scalability issues and allow a single compromised node to jeopardize the entire mobile node community.

In this report, we propose replacing the centralized CA with a distributed CA whose responsibilities are shared between a set of select network nodes. To that end, we develop a protocol that relies on threshold cryptography to perform the fundamental CA duties in a distributed fashion. The protocol is meticulously defined and is implemented it in a series of detailed models. Using these models, mobile wireless scenarios were created on a communication simulator to test the protocol in an operational environment and to gather statistics on its scalability and performance.

Intentionally Left Blank

Contents

1 Introduction 7

2 Centralized Certificate Authority 8
- 2.1 Basic CA Responsibilities 8
- 2.2 Centralized CA Threat Model 9
  - 2.2.1 Denial of Service 9
  - 2.2.2 Signing an Invalid Certificate 11

3 Distributed Wireless CA 12
- 3.1 Cryptography of a Distributed Wireless CA 12
- 3.2 Distributed Wireless CA Functionality 13
- 3.3 Benefits of a Distributed Wireless CA 14

4 Certificate Request Protocol 15
- 4.1 Basic Certificate Request 15
- 4.2 Certificate Request with Domain Extension 16
- 4.3 Sparse CA Scenario (SCAS mode) 17
- 4.4 Protocol Benefits 18

5 Model Development 19
- 5.1 Node Model Overview 19
- 5.2 Node-level Logic 20
  - 5.2.1 NL1 Logic Conditions 20
  - 5.2.2 NL2 Logic Conditions 22

6 Node Model Implementation 23
- 7.1 Signature Request Process Block 24
- 7.2 Wireless LAN MAC Interface Process Block 27
- 7.3 Wireless LAN MAC Process Block 28
- 7.4 Signature Request Acknowledge Process Block 29
  - 7.4.1 Signature Request (sig_req) Logic Module 30
  - 7.4.2 Signature Response (sig_resp) Logic Module 34
  - 7.4.2 Other Logic Modules 35
- 7.5 Mobility Processing Block 36

8 Protocol Simulation 36
- 8.1 Basic Certificate Request Simulation 36
- 8.2 Certificate Request with Domain Extension Simulation 39

9 Conclusion 46

References 47

Figures

Figure 1. Initial certificate request 16
Figure 2. Additional certificate request with domain extension 17

Figure 3. Sparse CA scenario (SCAS mode) ... 18
Figure 4. NL1 node logic diagram ... 21
Figure 5. NL2 node logic diagram ... 22
Figure 6. Ad hoc wireless node model ... 24
Figure 7. Signature request packet header ... 25
Figure 8. Signature request (`sig_request`) process model ... 26
Figure 9. Wireless LAN MAC interface (`wlan_mac_intf`) process model ... 28
Figure 10. Wireless LAN MAC (`wireless_lan_mac`) process model ... 29
Figure 11. Mobility (`mobile_route`) process model ... 36
Figure 12. Initial certificate request transmission ... 37
Figure 13. A node receives the certificate request transmission ... 38
Figure 14. A second node receives the certificate request transmission ... 38
Figure 15. A third node receives the certificate request transmission ... 39
Figure 16. Partial certificate reception rate ... 39
Figure 17. Initial certificate request transmission ... 40
Figure 18. A node receives the request and re-broadcasts it ... 41
Figure 19. A second node receives the request and re-broadcasts it ... 41
Figure 20. A third node receives the request and re-broadcasts it ... 42
Figure 21. A fourth node receives the request and re-broadcasts it ... 42
Figure 22. A fifth node receives the request and re-broadcasts it ... 43
Figure 23. A sixth node receives the request and re-broadcasts it ... 43
Figure 24. A final node receives the request and re-broadcasts it ... 44
Figure 25. Partial certificate reception rates (1 km × 1 km) ... 44
Figure 26. Partial certificate reception rates (2.5 km × 2.5 km) ... 45
Figure 27. Partial certificate reception rates (5 km × 5 km) ... 45

1 Introduction

When sending high-consequence messages, it is important for the sending and receiving parties to authenticate each other so that they can be assured each is communicating with a trusted entity. The need for end-node authentication is especially important in mobile ad hoc wireless networks in hostile environments, such as military operational scenarios, where distribution and protection of information is paramount.

In a wired infrastructure, this authentication service is typically facilitated by a centrally located "authentication certificate generator" such as a Certificate Authority (CA) server. This centralized server provides cryptographic certificates for communicating nodes so they can authenticate each other. This centralized model is also used in small wireless networks that remain connected to a wired network. An IEEE 802.11 infrastructure mode network is one such network.

Unfortunately, the centralized authentication model is ill-suited to meet the needs of mobile ad hoc networks, such as those required by military systems. Ad hoc networks are characterized by an extremely dynamic network topology. As the nodes move, network links change rapidly and paths through the network must continually be adjusted. The speed at which the network topology changes makes the centralized approach to authentication difficult to maintain. This problem is exacerbated in military environments where a network node not only moves but also may be destroyed or disabled at any time. In order for ad hoc mobile networks to have the requisite authentication mechanisms, a resilient distributed alternative to the centralized CA approach is needed.

Presently there are no adequate solutions that provide secure, scalable means of authentication. Current mechanisms either assign a pre-shared key (shared by all participating parties) or require that each node retain a collection of individual keys that are used to communicate with other individual nodes. Both of these approaches have scalability issues and allow a single compromised node to jeopardize the entire mobile node community.

In this report, we propose replacing the centralized CA with a distributed CA whose responsibilities are shared between a set of select network nodes. Using threshold cryptography, the set of CA nodes can perform the duties of a centralized CA by working together in a distributed fashion, thereby reducing the constraints on the network topology. Section 2 describes the duties of a typical CA, and section 3 discusses how a distributed CA can fulfill those duties. The distributed CA protocol is described in detail in section 4. Sections 5 and 6 discuss modeling the nodes in a distributed CA environment, and section 8 details the simulations of our distributed CA protocol using those models. Finally, section 9 contains our conclusions.

2 Centralized Certificate Authority

A common solution to authentication and key management issues in populous environments is to use a *public key infrastructure* (PKI). In a PKI, every user possesses a public key/private key pair. The public keys of all users are publicly known and bound to those users, and the private keys are kept secret. In order to authenticate himself/herself, a user must demonstrate knowledge of the private key that corresponds with his publicly-known public key. If such knowledge is demonstrated, one can be assured that the user is who he claims to be (assuming the binding between users and public keys is trustworthy and private keys are kept secret).

The role of the CA is to provide a trustworthy binding between users and their public keys. Users in the network trust the CA to correctly issue certificates to users that essentially vouch for the authenticity of a user and their key. An electronic certificate contains, among other things, the user's name and their public key. The certificate is digitally signed by the CA so that anyone who trusts the CA can check the signature on a certificate to verify its authenticity. The Certificate Authority (CA) creates the signature using its unique private key, SK_{CA}. That way, anyone can use the well-known public key of the CA, PK_{CA}, to verify a certificate and be assured of the association between a user and their public key. A node can prove its identify by presenting a valid certificate, which establishes its' identity and public key, and demonstrating knowledge of the corresponding private key.

2.1 Basic CA Responsibilities

The CA performs several necessary administrative duties in a PKI. Generally, the CA signs certificates, stores copies of the certificates, and revokes invalid or expired certificates. These three roles are described in detail below.

1. Sign certificates:

The CA's primary duty is to sign new or updated certificates that associate a user with his public key. The user submits his public key along with other system-specific information to the CA. Upon receiving this information, the CA must verify that the user is authorized to receive a certificate and is indeed who he claims to be. This user verification step often involves querying a Registration Authority (RA) and/or the use of an out-of-band channel. Once a user has been authorized, the CA uses its private key, SK_{CA}, to sign the certificate and returns it to the user.

2. Cache certificates:

A CA is often required to cache existing certificates for two reasons. First, the CA may need to issue copies of the stored certificates upon request. In some systems, it may be more efficient to get a user's certificate from the CA than the user himself. And second, the CA caches existing certificates so that it can ensure uniqueness among certificates. When issuing certificates, the CA must be careful to issue no more than one certificate for a given user name. If more than one certificate is issued to the same user name, it will be impossible to distinguish between the two, allowing for a wide range of impersonation attacks. Ensuring uniqueness is a task that is often delegated to the RA.

3. Maintain and Publish Certificate Revocation List (CRL)

Just as a CA is tasked with issuing certificates, it is also required to revoke them. When a user leaves or is expelled from the system, his certificate must be immediately invalidated to prevent him from subsequently engaging in any unauthorized communication. To annul the certificates of invalid users, the CA maintains and publishes a Certificate Revocation List (CRL) that catalogs the certificates which should no longer be considered valid. When a user receives a certificate, he should first verify that the certificate is not on the CRL before engaging in the initialization handshake.

While the number of duties that a CA must perform is small, they are complex. For the purposes of simplicity, we only focus on the primary duty of CA, that of issuing and signing certificates. While caching and revoking certificates are important tasks for a CA, they are beyond the scope of this paper. The need for caching can be avoided by mandating that users supply their own certificates and having an RA that strictly monitors admittance to the network. For discussions of how revocation lists may be implemented in a distributed environment, see [3], [4], [14], [16], and [22].

2.2 Centralized CA Threat Model

The first step in developing a security approach in a network environment is to define an appropriate threat model. The following is a description of the adversarial threats considered when designing our network security architecture. We consider attacks against a wireless ad hoc network with a PKI that relies on a single, centralized CA to service the network.

The goal of the adversary is to disrupt or disable the CA from fulfilling its requisite duties or to subvert its authority. Thus, an adversary may have two goals for disrupting the CA services.

1. The adversary wishes to prevent the CA from signing certificates.
2. The adversary wishes to get a valid signature for an otherwise unauthorized certificate.

2.2.1 Denial of Service

It may be advantageous for an adversary to prevent the CA from performing some or all of its duties. If the adversary successfully denies service to the CA, all of the nodes in the network will no longer be able to obtain new certificates, renew old certificates, or maintain an accurate CRL.

If we assume that the adversary wishes to prevent the CA from signing certificates, there are a number of avenues of attack that he may use. Since the CA is a lone entity, the attacker can focus all of his efforts on a single point, making his job easier.

- **Disable the CA –** To achieve a denial of service (DoS), an adversary may attack the CA itself in an attempt to disable it or cause it to malfunction. An adversary can physically disable the CA by simply cutting the power to the device, causing it to go down for maintenance (scheduled or unscheduled), or by physically destroying the machine. Once the machine is physically unusable, it will obviously be unable to perform its duties.

The attacker may also be able to cause the CA machine to malfunction. If he is able to infect the CA with some sort of computer Trojan or virus, the adversary may be able to crash or temporarily disable the machine. Further, once the Trojan or virus is discovered, the CA could possibly suffer additional downtime while the system is restored to a trusted state.

- **Block Communications to the CA –** Another way in which an adversary may cause a DoS is to block all communications to and from the CA. In this situation, the CA would be functioning normally but would be unable to communicate with the rest of the network and, thus, unable to fulfill its responsibilities.

  A simple method for blocking communications would be to physically jam the RF spectrum used by the CA. Obviously, no network traffic can reach the CA if the wireless signal is jammed. Since the CA is a lone entity, an adversary can focus his jamming attack to a small physical area, making his task much easier.

  A more sophisticated attacker could employ a host of traffic-based DoS attacks, such as a flood attack or a SYN attack. These attacks swamp the CA with so much network traffic that it is unable to respond to legitimate traffic, effectively isolating the CA from the other nodes on the network.

  If an adversary controls an important network junction, such as a central node in the ad hoc network, he can perform what is called a black hole attack in which he only receives incoming traffic but does not propagate the traffic to the next hop. To help mask the attack, an adversary may be selective and choose only to block messages destined for the CA. The effectiveness of this attack depends on the topographical importance of the node that the adversary is able to compromise.

- **Corrupt Data Sent to the CA –** A third and even more sophisticated attack would be to corrupt (but not block) the data that is sent between the CA and the network nodes while the data is in transit. If an attacker can flip random bits in the communications between the CA and network nodes, he can damage the certificates and CRLs distributed in such a way that they are unusable. Since the signals are traveling wirelessly, weak RF jamming may be sufficient to damage the signals enough to render them unusable.

  It may be possible for an adversary to intercept and modify transmissions before they reach the CA. Even if the modifications are meaningless, the CA will either discard the signature request or sign an incorrect certificate and return a certificate that does not match the request the client originally sent.

  Likewise, the adversary can also try to intercept or disrupt transmissions that are sent out by the CA before they reach their destination node. If successful, the adversary can ensure that the transmission received by the node is useless.

2.2.2 Signing an Invalid Certificate

A second goal of an adversary may be to obtain a valid CA signature on an invalid certificate. With the ability to get a CA signature for a certificate of his choice, the adversary will be able to perform a wide range of devastating attacks on the network, such as impersonation and man-in-the-middle attacks.

- **Trick the CA into Signing Bogus Certificates –** One obvious way for an adversary to get a certificate signed is to just ask the CA. Assuming the CA has some means of detecting invalid signature requests (e.g. duplicate usernames), the adversary must “trick” the CA into signing a certificate request that would otherwise be rejected.

  If the attacker is able to intercept and modify messages while in transit, he may be able capture traffic of a legitimate certificate request. As the certificate request handshake is passed back and forth between the CA and the node, the attacker could try to modify the packets and replace relevant information with his own. Assuming the attacker correctly modified the handshake, the CA will believe the adversary to be some other legitimate node. In such a situation, the attacker may be able masquerade as another user and submit his own certificate requests.

  Another, more realistic attack involves recently announced attacks on hash functions. An adversary can create two certificates, one valid and one invalid, that hash to the same value. The adversary could then submit the valid certificate to the CA to obtain a signature. However, since the two certificates have the same hash value, the CA’s signature on the valid certificate will also be a correct signature for the invalid certificate. An attacker can then take the signature from the valid certificate and attach it to the invalid certificate, thereby crafting a signed, bogus certificate.

- **Find the Signing Key SK_{CA} –** A second way for the adversary to get valid CA signatures is to generate the signatures himself. In order to create valid signatures, the adversary must somehow obtain the CA’s closely-guarded private key, SK_{CA}. Compromising the CA’s key is a difficult goal for the attacker as the key is sure to be heavily protected. However, if he is able to discover the key, the adversary can forge as many valid signatures as he pleases.

  A naive attacker may try to obtain the CA’s private key by simply trying to guess it or search for it. Unfortunately for such an adversary, the probability of randomly (or methodically) finding the CA’s key pair is very small. In fact, this attack’s probability of success is so small that the it is essentially infeasible and is only mentioned for the sake of completeness.

  Since the chance of the attacker finding the CA’s key pair on his own is remote, a more realistic approach would be to steal the key pair directly from the CA. While the likelihood that an attacker can extract the CA’s key pair directly from the CA is also small, the attacker has an advantage in that there is a lone target on which to focus his efforts. A motivated attacker can concentrate his efforts on compromising the CA and gaining unauthorized access. For example, an the attacker may try to upload a virus or Trojan to the CA that secretly watches for the key pair to be used. If an attacker is able to do so, he may be able to obtain the CA's key pair.

In the centralized model, the CA provides certification services for the entire network. For mobile wireless networks, the scalability challenges are immediately apparent. Many of the attacks associated with a centralized CA stem from the fact that the CA is a single point of failure in the network. An attacker is able to concentrate all of his resources on a single, fixed entity. Consequently, some attacks, especially denial of service, become surprisingly simple to mount. Similarly, attacks targeting the signing key can be focused on a single entity. A lone CA is a single point of failure in the network and an obvious target for an attacker.

The intermittent connections and dynamic topologies intrinsic to wireless environments further exacerbate the availability problems of a lone CA. Because node topologies in a wireless ad hoc network can change so rapidly, a path to the lone CA may not always be available. When nodes are left without a connection to the CA, they have no means of obtaining the CA's services and are effectively suffering a denial of service. Allowing normal network events, such as node topology changes, to inflict denial of service conditions on the network is unacceptable. Thus, a more robust and fault-tolerant method is needed to fulfill the role of the CA in wireless ad hoc networks.

3 Distributed Wireless CA

To provide better certification services to a mobile ad hoc network, we define a distributed wireless CA. This distributed wireless CA uses threshold cryptography to share the CA's certification duties with many nodes in the network. Instead of relying on a lone CA to sign certificates, a subset of nodes in the network can work together to perform the same task, making the network more robust and fault tolerant.

3.1 Cryptography of a Distributed Wireless CA

In a distributed CA, the CA's private signing key, SK_{CA}, is no longer located in a single location but is distributed in part to multiple nodes in the network. Each CA node possesses a small piece of the private signing key called a *secret share*. Any subset of k nodes can combine their secret shares to collectively produce a valid signature, but any collaboration of less than k nodes will be unable to generate a valid signature. An important advantage of threshold signature schemes is that the entire private signing key SK_{CA} is never seen by nor revealed to any entity in the network (except at system initialization when the secret shares are created). Once the secret shares have been created, the private signing key is thereafter kept secret.

Threshold cryptography was introduced in 1979 by Shamir [19]. While Shamir's scheme was simple and insecure, numerous implementations of and improvements to threshold cryptosystems have since been made. Particularly, improvements have been made to the robustness and efficiency of threshold secret sharing. For instance, there are improvements

[2][15] that alleviate the need for a trusted party to set up and distribute the secret shares at system initialization. Another set of improvements, called proactive secret sharing (PSS) schemes [8][9][12][13][17], periodically update the secret shares to limit the time in which a mobile adversary can stage an attack. PSS schemes force the adversary to discover *k* secret shares in the limited time period between proactive secret share updates. Verifiable secret sharing (VSS) [7][10][18][21] allows nodes actively to detect bad or corrupt secret shares when gathering input from *k* nodes. Thus, VSS provides nodes with the capability of detecting and possibly reacting to malicious nodes which are deliberately submitting false partial signatures in an attempt to sabotage the group signatures.

Threshold cryptography is used in our distributed CA. When a node needs to have a certificate signed by the CA, the node broadcasts a certificate request to all nearby CA nodes. When a CA node receives such a request, it validates the request and authenticates the node using information presumably obtained via the RA. If the node's right to a certificate is established, the CA node uses its secret share to compute a partial certificate. The partial certificate is then sent back to the requesting node. When the requesting node receives a partial certificate from a CA node, it can use VSS to verify the correctness of the partial certificate. Once the requesting node has at least *k* valid partial certificates, it can combine them and form a valid certificate with a correct signature.

In this scenario, the requesting node was able to get a new certificate signed by the CA signing key SK_{CA} in a distributed fashion. The entire signing key SK_{CA} was never visible to any of the players in the exchange. There are a several different threshold signature schemes [5][6][10][11][20] that can be used in such a system, depending on the system's needs and environment.

3.2 Distributed Wireless CA Functionality

In order for a distributed wireless CA to be a reasonable substitute for the standard centralized CA, it must be able adequately to perform the basic duties of a CA. Specifically, a distributed CA must perform the three previously identified administrative tasks of a CA. In this section, we recount the duties of a CA and discuss how they can be fulfilled by our distributed CA.

1. Sign certificates:

A CA's primary duty is to sign new or updated certificates that associate a user with his public key. The procedure for obtaining a certificate from a distributed environment was outlined above. Assuming a certificate request message is heard by at least *k* CA nodes, the request will be fulfilled. However, therein lies the difficulty: ensuring with high probability that the certificate request reaches enough CA nodes. This problem is discussed later and becomes the focus of our modeling and simulation efforts.

2. Cache certificates:

As noted earlier, the duty of caching certificates is not addressed by our protocol. Caching certificates is generally not a critical task for a CA and, in fact, is often not its duty at all but that of the RA. As such, we feel this task is of secondary importance and can be safely omitted from this discussion.

3. Maintain and Publish Certificate Revocation List (CRL)

Actively maintaining and distributing a CRL in a distributed environment is a more difficult problem. Because of the lack of centralization, determining which nodes should be revoked and maintaining an authoritative list of those nodes is not as straightforward as it was for a centralized CA. Nevertheless, CRL distribution in an environment with a distributed CA can still be achieved. Numerous solutions have been suggested [3][4][14][16][22], but their descriptions are beyond the scope of this paper. The reader is directed to those references if further detail is required.

If it assumed that a certificate request is received by at least *k* CA nodes, then that certificate request will be satisfied. The validity of that assumption will be examined later in section 8. However, when the assumption holds, the distributed CA will more than adequately fulfill the primary duty of a CA, that of signing certificates.

3.3 Benefits of a Distributed Wireless CA

In a standard PKI, the CA is usually a single machine or a small set of redundant machines that logically function as one. The CA is the sole custodian of the CA's private key, SK_{CA}, and is the only entity capable of signing certificates. In such an architecture, all certificate signature requests go to the lone CA for approval. Obviously, as the system grows larger, the burden of signing certificates grows. Since the CA must be able to handle the traffic and signing overhead for certificate requests for the entire system, it can become a bottleneck.

A distributed CA removes the bottleneck from the system, making the DoS attacks discussed in section 2.2.1 much more difficult to mount. Recall that DoS attacks against a centralized CA preyed on its uniqueness, allowing an attacker to focus all his resources on a single target. By distributing the CA duties amongst a host of nodes, DoS attacks must now disable numerous targets instead of just one. In a distributed environment, the CA is no longer a single point of failure for the network.

If there are enough CA nodes, a distributed CA is also more available than a single CA under normal ad hoc network conditions. Since the centralized CA is a single entity, there is a small number of paths to reach it at any given time. As the network topology changes, it is likely that the network will sometimes be fragmented into isolated pieces. When this fragmentation occurs, the segregated portions will be unable to communicate with the CA. In a distributed CA, even if fragmentation occurs, CA duties can still be accessed. As long as a node is in a fragment with at least *k* CA nodes, that node can access the CA services.

A distributed CA is also equipped to protect the secret signing key, SK_{CA} in a unique way. In a centralized CA, the private key is held by the lone CA. In order to obtain SK_{CA}, an adversary must compromise the CA. Since the CA is a lone entity, it can be heavily fortified against compromise or penetration. Nevertheless, it is still a single point of failure on which an attacker may focus all of his efforts. On the other hand, in a distributed CA, shares of the private signing key are dispersed among the various signing nodes. While it may be more difficult to secure multiple nodes, the distributed CA is strengthened by the fact that no single node possesses the entire signing key. In order to obtain SK_{CA}, an adversary must concurrently compromise k different nodes. Furthermore, in a system which updates the secret shares using PSS, the adversary has only a small amount of time to compromise the k nodes. So, even if the distributed signing nodes were not as individually fortified as a single centralized CA, the need to compromise k nodes bolsters the system's overall security.

4 Certificate Request Protocol

Based on the threshold certificate signature scheme discussed above, we now formally define a protocol for requesting certificates from a distributed CA. The protocol defines the procedures for requesting a certificate and for CA nodes that need to issue partial signatures.

When a node needs to have its certificate signed by the CA, it initiates a request for a signature to the distributed CA by sending out a request to a group address that collectively represents the authority of the CA. The particular group address is targeted at only those nodes that have the capability to sign a partial certificate. For purposes of discussion, these CA nodes will be said to belong to Node Level 2 (NL2). The requesting node that initiates the signature request (and all other nodes) will belong to Node Level 1 (NL1).

4.1 Basic Certificate Request

The first signature request is broadcast by an initiating NL1 node to a group address that targets all NL2 nodes within the local broadcast area. The request is sent to a group address so that a single message can be sent to all NL2 nodes within range, making certificate requests more efficient. This *groupcast* approach also alleviates the need for maintaining an accurate list of addressable receivers, a task which can be quite inefficient when addressable receivers are out of range of the requester. [1]

The basic certificate request has a type field in the transmission frame which identifies it as a "basic request." The basic identifier indicates that the message should not be propagated to other, more distant nodes who are outside of the NL1 node's transmission range. The request also contains a sequence number that uniquely identifies it from subsequent transmissions.

An NL2 node that receives a basic certificate request uses its secret share to compute a partial certificate and transmits the partial certificate back to the requesting NL1 node. Figure 1 illustrates this sequence of events.

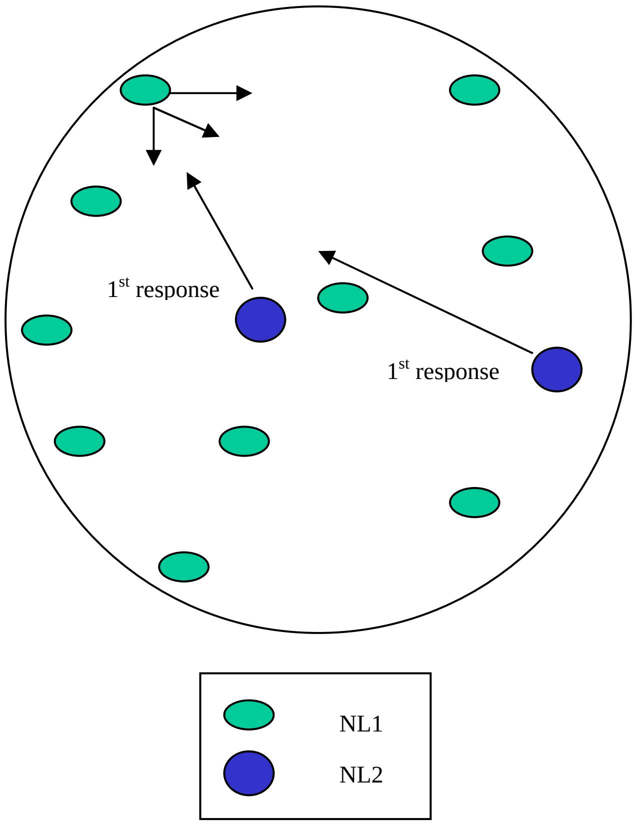


Figure 1. Initial certificate request

The NL1 node that initiated the signature request listens for responses for a specific timeout period. After the timeout period has expired, it tests the validity of each partial certificate it received using a VSS scheme and compares the number of valid responses with the needed threshold value *k*. If the threshold has been met, then the node can combine the signed partial certificates to create a valid, signed certificate, which can then be used for authentication. [1]

4.2 Certificate Request with Domain Extension

If, after the first certificate request, the number of valid partial certificates is less than the threshold, a subsequent transmission is sent out. The subsequent transmission is sent using the same group address as the first transmission. It uses the same sequence number as the first transmitted frame, which allows the first group of nodes to determine that this is a repeat request. The type value of this frame is different so that it can be identified as a "forwarding request." The type value, when received by the original responders, directs these responders to forward out this groupcast to other NL2 nodes that are located beyond the range of the original request. The NL2 nodes forward the request to more distant NL2 nodes who, upon receipt of a forwarded request, compute the signed partial certificate and transmit the result back to the NL2 nodes that forwarded them (these return transmissions are unicast). The intermediate NL2 nodes then forward the results back to the originating NL1 node.

When the NL1 node receives this second wave of signed partial certificates (within an appropriate time-out period) it compares the number of valid additional signatures with the threshold value. If the value is met, or exceeded, it combines the partial certificates and computes the needed certificate. If the threshold value is still not met, the node then aborts its attempt. Figure 2 depicts this activity.

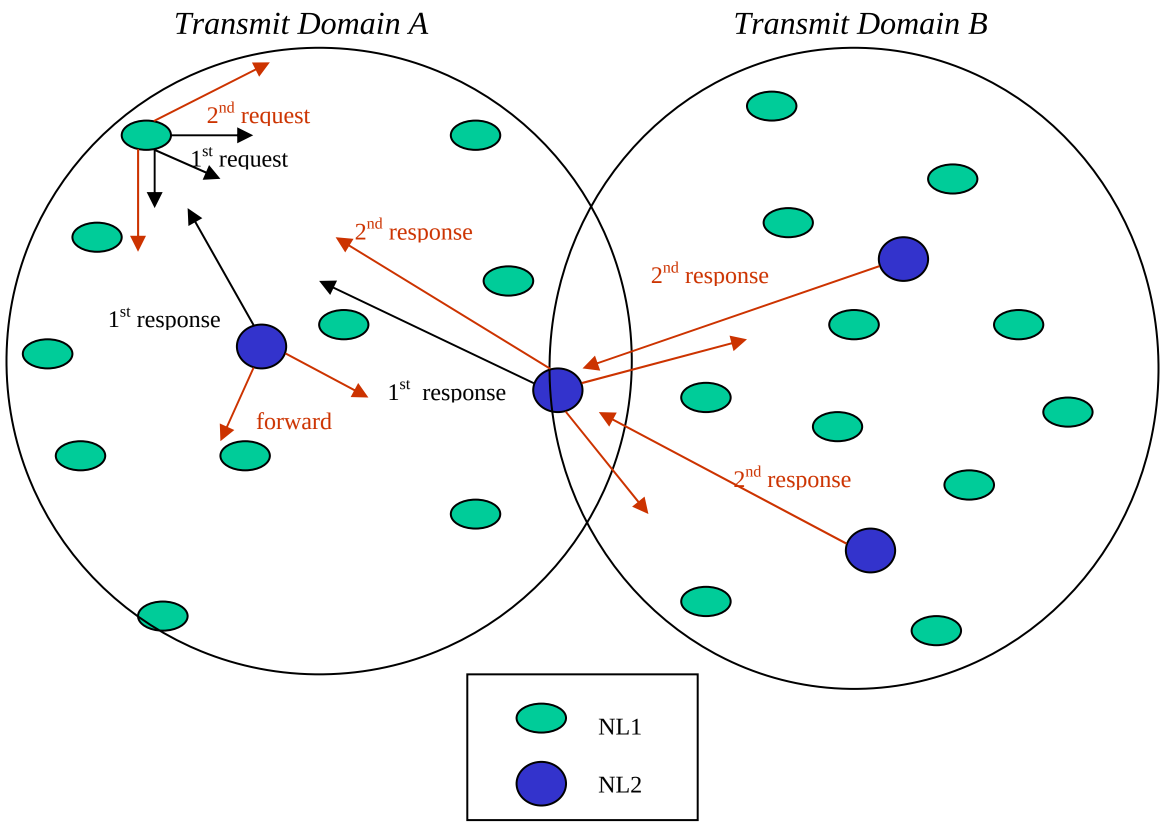


Figure 2. Additional certificate request with domain extension

4.3 Sparse CA Scenario (SCAS mode)

It may be necessary for the mobile ad hoc CA protocol to operate in a more demanding environment. Depending on the operational environment and scenario, it may be appropriate to adjust the signature resolution protocol to accommodate sparsely populated distributed CA services that still require a moderate threshold value *k*. The need for this operational mode could be attributed to excessive amounts of environmental clutter (e.g. hills, mountains, trees, buildings, or weather) that effectively reduce the overall transmission capacity of each participating node. Another potential reason for this mode could simply be a lack of participating CA nodes due to expansive territorial range or adversarial destruction and/or compromise.

To compensate for these kinds of restraints, a different type field value was added to the CA request frame that allows the initiating NL1 node to request a signed partial certificate from multiple transmission domains on its first request cycle. This mode allows the request frame to be propagated through a predetermined number of different transmission domains on the first request without wasting time waiting for an initial request from a single domain as in the default configuration. As before, this transmission frame will be associated with a unique sequence number to allow for the detection of transmission loops. The NL2 nodes will act upon this request immediately. The original NL1 node will wait a pre-determined amount of time for the needed amount of responses before timing out and aborting the CA request. Figure 3 illustrates this SCAS approach.

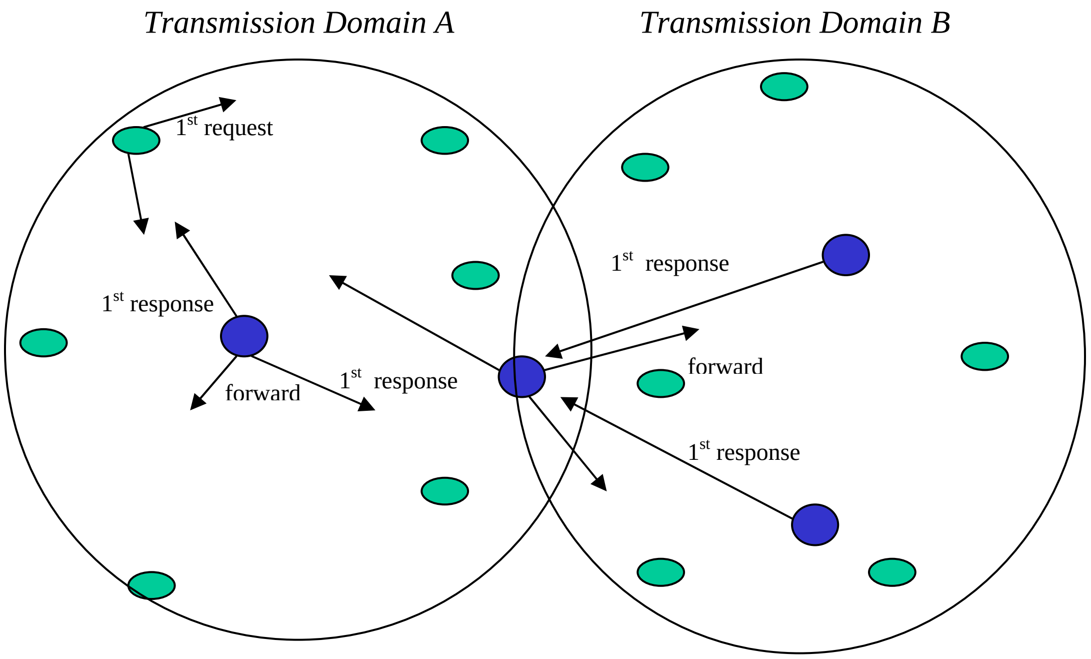


Figure 3. Sparse CA scenario (SCAS mode)

4.4 Protocol Benefits

This approach to distributed certificate generation for mobile ad-hoc nodes provides an improvement over comparable mechanisms in the following ways:

- **Groupcast certificate request –** The groupcast method of information request is more scalable and efficient than broadcast forms of the same request. A broadcast request requires that each node that receives the broadcast process it even when the receiving node does not have the capability of providing part of the answer. With the groupcast method, only nodes that are associated with the group address are asked to process and potentially forward the request. This allows only nodes that can provide a part of the answer to be queried. [1]

- **Selectable outreach –** The number of hops that the initial request is propagated is selectable as described:

    - *Default Mode* – In this mode the initial reach of the request can be limited to a user-defined broadcast domain. This allows for a very rapid response to an initial request.
    - *SCAS Mode* – In this mode the reach capability can be extended to encompass multiple broadcast domains over a widely dispersed geographical area. [1]

- **Redundant and stale request detection –** The design of the protocol provides a simple means for a receiving node to differentiate between new certificate requests and redundant requests which can to be discarded. The protocol also dictates that requests expire after a pre-selected amount of time, preventing intermediate NL2 nodes from using valuable memory resources to store stale requests. [1]

- **Adjustable *k*-threshold value** – When a threshold signature scheme is used for signature generation, the number of partial certificates that must be obtained to create a valid certificate can be adjusted on a system-by-system basis to fit varying needs. [1]

- **Transmission loop detection –** The presence of broadcast loops in the transmission medium can be easily detected, and all redundant traffic can be discarded. [1]

5 Model Development

In order to understand and test our distributed CA protocol, we implemented it in a simulated environment. To do so, a detailed model of the system was developed. The first step was to create models of the individual network nodes that take part in the protocol. In this section we describe the logical makeup of our NL1 and NL2 node models. Subsequent sections discuss the models' implementation and our system simulation results.

5.1 Node Model Overview

To model our protocol and its proscribed interactions between nodes, two different mobile objects had to be created: an NL1 node model and an NL2 node model. The NL1 node model had to provide the following minimum capabilities:

1. At random intervals, send out a request for a signed partial certificate.
2. Wait until it receives a threshold number *k* responses from NL2 devices.
3. Record the number of partial certificate responses. After a specific time interval, if the total number of partial certificate responses is at least *k*, then set the request counter to "successful;" otherwise, set the request counter to "failed."
4. Simulate combining the partial certificates by doing operations that represent an equivalent computational delay.
5. Keep track of the total number of certificate requests.

Similarly, the NL2 node model had to provide the following minimum capabilities:

1. Listen for certificate requests from NL1 objects.

2. Simulate the generation of a partial certificate by doing operations that represent an equivalent computational delay.
3. Send the partial certificate back to the requester.
4. Keep track of the total number of certificate requests.
5. Forward the certificate request to other NL2 devices[*].

These are the bare minimum capabilities that the two types of models must possess. To perform more detailed simulations, there are several possible adjustments or improvements that we explored. We list them here for the sake of completeness.

- To facilitate the request and response of the interaction between these two different node types, it may be appropriate to create a new type of broadcast frame (within the MAC layer) or, more simply, a new "type field" value within the IP header. This will be needed to construct the logic state that will provide the processing.
- The development of stat handles may be needed to help track some of the more important statistics of interest.
- A model that consists of a broadcast-only scenario was developed first. This allowed us to identify the minimum ratio of NL1 nodes to NL2 nodes vs. the *k* threshold value.
- The first models created were not concerned with HLA interactions to guide their movements. This level of detail can come later if desired.
- Initially, random mobile pattern generation was used to dictate the path of each NL1 and NL2 node. The random paths can be replaced if desired.
- The model can incorporate a wireless routing protocol that allows for multi-hop capabilities to help improve the chances of reaching the requisite number of NL2. This scenario can be compared and contrasted with the broadcast only scenario. To facilitate this multi-hop scenario, a new protocol for a multi-hop wireless threshold signature scheme will need to be developed.
- HLA interfaces can be added to allow for mobile pattern generation to be driven remotely. These interfaces allow for other users and their simulators to create different types of useful scenarios.

5.2 *Node-level Logic*

Since the wireless nodes are mobile and in constant motion, the number of NL2 nodes that receive the original transmission from the requesting NL1 node varies over time. In our simulations, the transmission range value for the NL1 and NL2 nodes is set to 300 meters. Because of this range limitation, the underlying protocol was designed with features that allow the limited transmission range to be extended. The following subsections show some of the conditions and associated logic sequences that are supported by the protocol.

5.2.1 NL1 Logic Conditions

After an NL1 node sends out a groupcast request for a partial certificate, the request is received by all nodes within the transmission range who are part of the groupcast address as specified in

[*] This capability is added for multi-hop simulations

the header of the transmitted frame. If other nodes are within range but are not part of the requesting address group, they filter out and ignore the request. The following section describes the logic sequence of the transmitting NL1 node.

Condition: An NL1 node generates a signature request frame:

```
--Generate packet
--Generate sequence number
--Set type field = signature certificate request
--Set origin field = local NL1 address
--Set forward bit
                    If forward bit = 0
--Set Range byte = 0
                    If forward bit = 1
--Set Range byte >= 1
--Cache sequence number
--Set interrupt for cache time out (interrupt time value will vary depending on range byte value)
--Set destination address = destination group address
--Set source address = local NL1 address
```

NODE LEVEL 1 (NL1) LOGIC CONDITION
Condition: Receive a returning unicast signature response frame from a previously groupcast forwarding NL2 node.

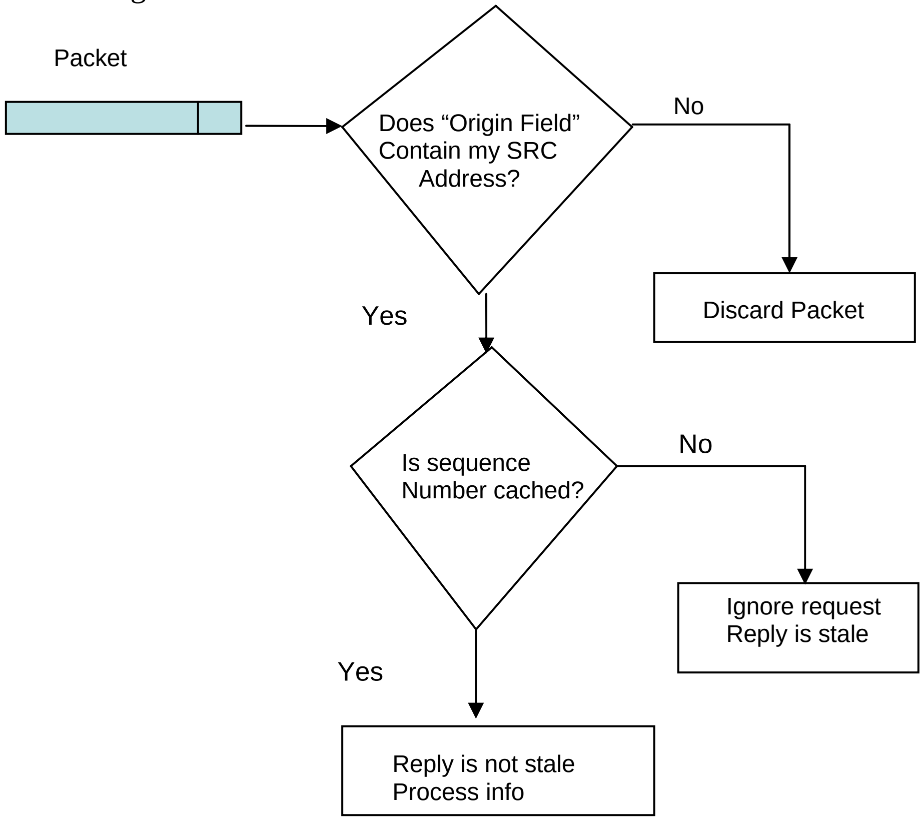

Packet
Does "Origin Field" Contain my SRC Address?
No
Discard Packet
Yes
Is sequence Number cached?
No
Ignore request Reply is stale
Yes
Reply is not stale Process info

Figure 4. NL1 node logic diagram

5.2.2 NL2 Logic Conditions

Condition: Receive a non-forwarding groupcast or unicast signature request frame from a NL1 node.

--Type field = signature request
--Forward bit = 0   (since forward bit is 0, no need to check range byte)
--Store sequence number
--Store source address

--Generate partial certificate

CREATE RETURN FRAME

--Input stored sequence number
--Set type field = signature response
--Install partial certificate
--Set destination address = stored source address
--Unicast frame back to requesting NL1 node

Condition: Receive a forwarding groupcast signature request frame from a NL1 node or a forwarding NL2 node (All forwarding signature request frames are groupcast).

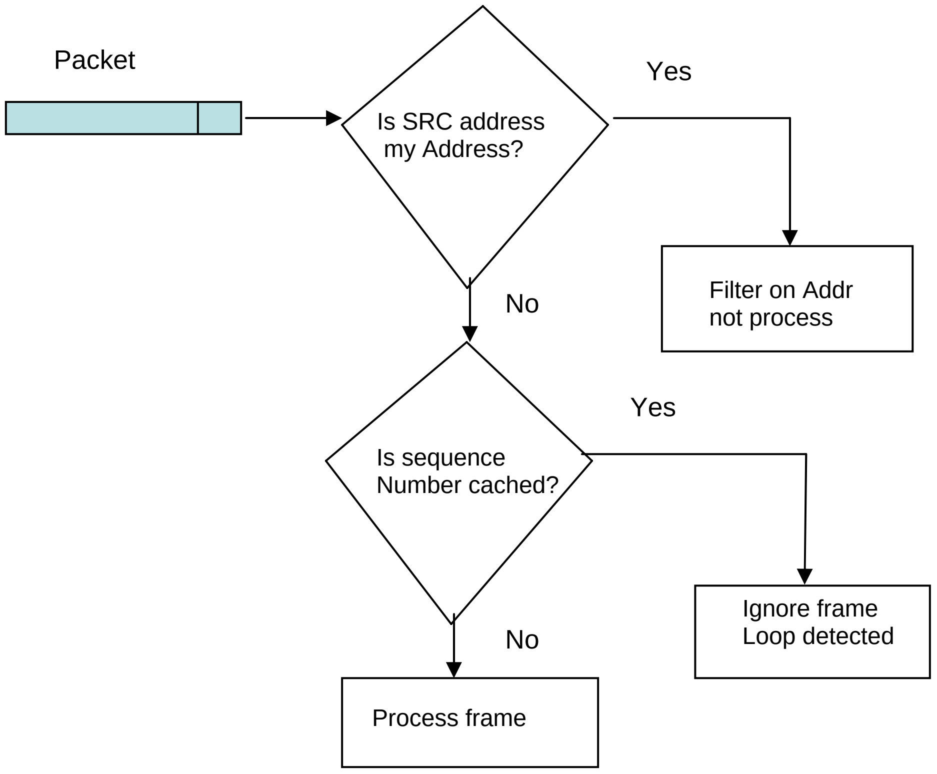

Packet
Is SRC address my Address?
Yes
No
Filter on Addr not process
Is sequence Number cached?
Yes
No
Ignore frame Loop detected
Process frame

Figure 5. NL2 node logic diagram

--Cache frame (must include source address of originating or forwarding node and "origin" field)
--Set cache interrupt to time-out frame
--Copy cached frame
--Decrement range field
--Groupcast frame out (this continues the forwarding process)

--Generate partial certificate
--Create return frame (insert source address from previous NL1 or forwarding NL2 node)
--Set type field = signature response
--Set forward bit = 0
--Unicast frame back to sender (The sender may be another forwarding NL2 node or the originating NL1 node)

Condition: Receive a forwarding multicast signature request frame from a forwarding NL2 node. Since range byte = 0, Unicast message back to forwarding node but no more groupcast forwarding

--Type field = signature request
--Forward bit = 1
--Range byte = 0

Determine if sequence number is
Already cached ----------------------→ YES   ignore request, loop detection

--------→ NO
--Cache received frame
--Set cache interrupt to time out frame
--Generate partial certificate
--Create Frame and pack fields

--Return sequence number
--Set type field = signature response
--Set forward bit = 0
--Unicast frame back to sender (The sender is a previously forwarding NL2 node)

6 Node Model Implementation

The general wireless node model, used for both NL1 and NL2 node models, is comprised of seven underlying processing blocks: a wireless LAN receiver and transmitter module (**`wlan_port_rx0`**, **`wlan_port_tx0`**), a wireless media access control module (**`wireless_lan_mac`**), a wireless MAC interface module (**`wlan_mac_intf`**), a mobile route module (**mobile_route**), a signature request module (**sig_request**), and a signature request acknowledge module (**sig_request_ack**). Figure 6 shows these interconnected modules.

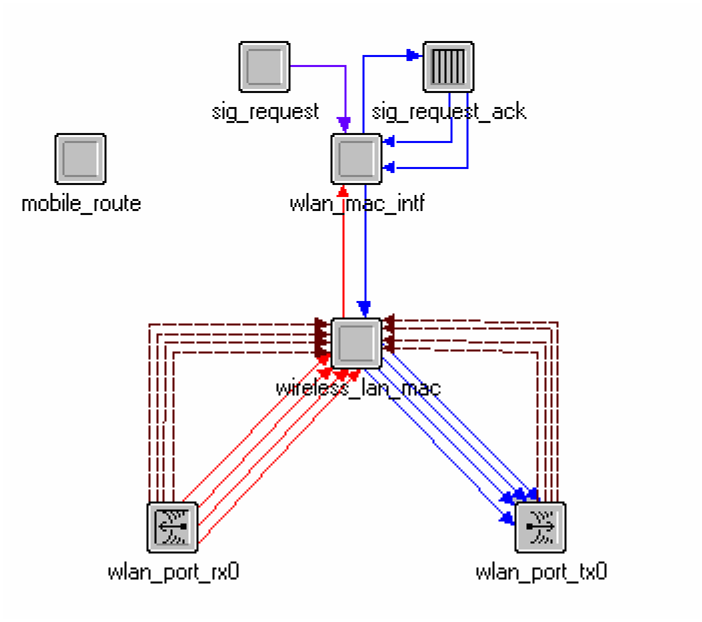


Figure 6. Ad hoc wireless node model

The wireless blocks (**wlan_port_rx0**, **wlan_port_tx0**) provide a means to emulate a wireless transceiver. The bit error rate, receiver sensitivity, and transmission power are set to provide a range value of approximately 300 meters. Access to the wireless medium is governed by the **wireless_lan_mac** module, which emulates an ad hoc wireless node. Wireless traffic that is received from the MAC layer is pushed up to the **wlan_mac_intf**. This module has a two-fold purpose: to forward incoming data up to the signature request acknowledgment process block (**sig_request_ack**), and to receive communications from the **sig_request** and **sig_request_ack** process blocks. The following sections describe each of these process blocks in detail.

7.1 Signature Request Process Block

The signature request process block, shown in Figure 6 as **sig_request**, is designed to send out certificate request packets from an NL1 node. NL1 nodes need to be capable of generating certificate requests and transmitting them out. Recall that, to provide the most efficient means of utilizing the wireless medium, the certificate requests are sent to all NL2 nodes via a groupcast transmission.

To facilitate a groupcast transmission, a new packet type had to be created. This new packet was formed by adding a new identifier to the MAC wireless header. In addition to unicast and broadcast identifiers, a groupcast identifier was created. This new identifier allows listening nodes to filter on this type of broadcast if the listening node has the capability of processing these packets. (All NL2 nodes were designed with the capability of filtering and processing groupcast packets.) Within the `sig_request` processing block, a packet is generated that represents a signature certificate request. The packet header contains fields that allow the packet to be uniquely identified and propagated throughout the wireless domain. Figure 7 shows the packet header construct.

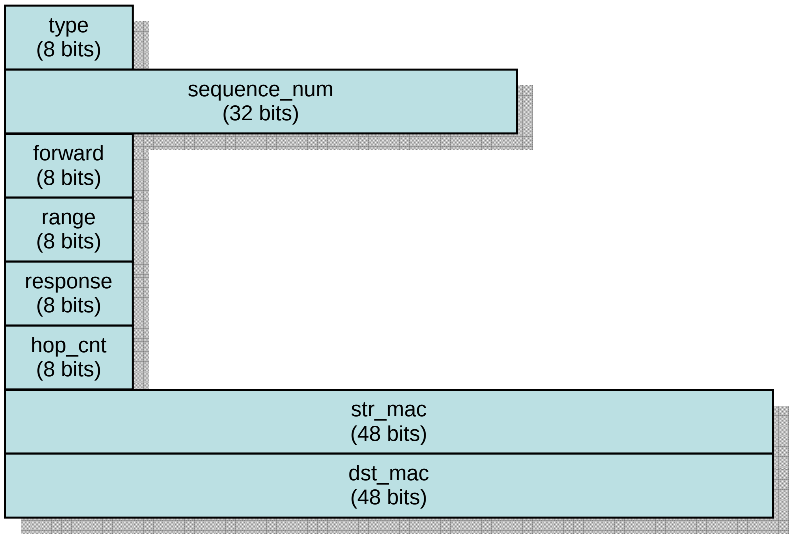
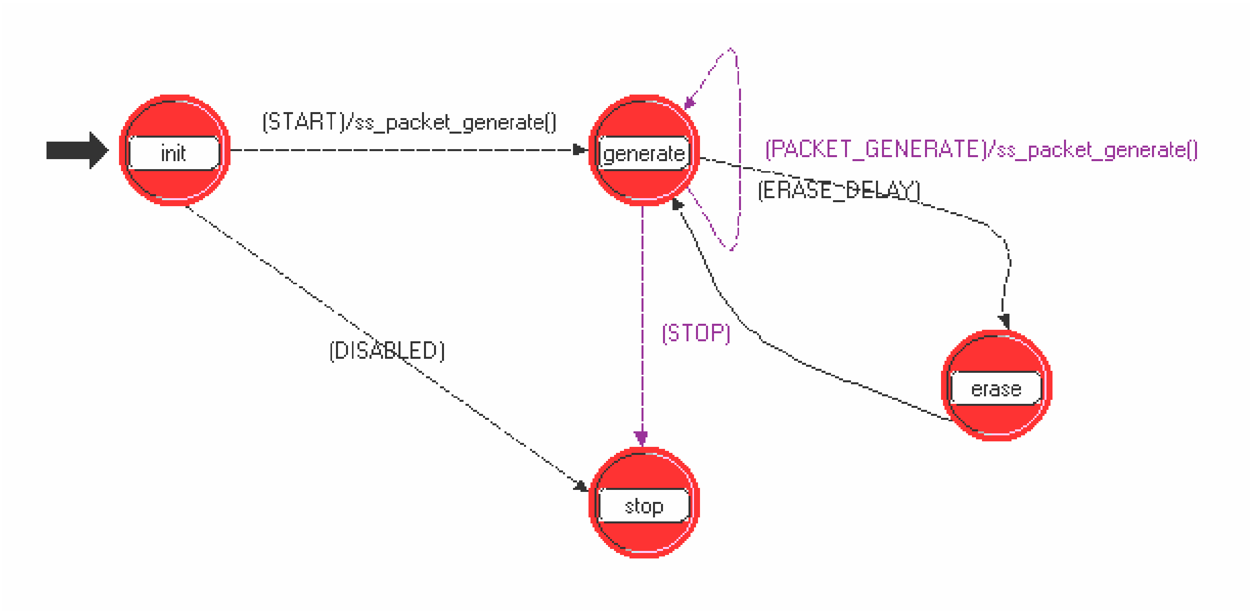


Figure 7. Signature request packet header

The "type" field identifies two different types of packets, a signature request packet and a signature response packet. The header also contains a "sequence_num" field which is used to store a sequence number that uniquely identifies each individual signature request packet, an "origin" field used to identify the source of the request, and "forward" and "range" fields that specify the propagation distance of the packet.

The `sig_request` process model, shown below in Figure 8, is comprised of four logic blocks: **init**, **stop**, **generate**, and **erase**. The **init** block is used to initialize the process model by registering itself with the simulation kernel and providing some preprocessing directives. From this starting point, the process model can move to the **stop** state directly if it is disabled by the user. Otherwise, it remains in the **init** state and waits until it is triggered to move to the **generate** state. This trigger comes in the form of a call to the function **ss_packet_gernerate()**. The **ss_packet_generate** code creates a packet structured as in Figure 7, initializes the appropriate fields with default and user-selectable attributes, and forwards the packet out. The **erase** logic state is triggered by an "**erase_delay**" logic condition, which is used to facilitate animation during the simulation.



init
generate
erase
stop
(START)/ss_packet_generate()
(PACKET_GENERATE)/ss_packet_generate()
(ERASE_DELAY)
(STOP)
(DISABLED)

Figure 8. Signature request (**`sig_request`**) process model

After the packet header is constructed, a data field value is appended and the whole packet is sent out the packet stream to the next processing block. The following is some of the code that creates the header and initializes its fields.

```
/* At this initial state, we read the values of source attributes       */
/* and schedule a set interrupt that will indicate our start time */
/* for packet generation.                                                */

/* Obtain the object id of the surrounding module.                       */
own_id = op_id_self ();
/* Get my assigned MAC address                                           */
parent_id = op_topo_parent (own_id);
//op_ima_obj_attr_get (parent_id, "Wireless LAN MAC Address", &my_address);
//printf ("My src MAC address is: %d\n", my_address);

/* Read the values of the packet generation parameters, i.e. the        */
/* attribute values of the surrounding module.                           */
op_ima_obj_attr_get (own_id, "Packet Interarrival Time", interarrival_str);
op_ima_obj_attr_get (own_id, "Packet Size",              size_str);
op_ima_obj_attr_get (own_id, "Packet Format",            format_str);
op_ima_obj_attr_get (own_id, "Start Time",               &start_time);
op_ima_obj_attr_get (own_id, "Stop Time",                &stop_time);



/* Create a packet with the specified format. */
pkptr = op_pk_create_fmt (format_str);
op_pk_total_size_set (pkptr, pksize);

/* Pack the packet fields, type, sequence num, forward, range and address */
```

```
pkptr = op_pk_create_fmt ("sig_request_pkt_1");
op_pk_nfd_set (pkptr, "type", 90);
if (op_ima_obj_attr_get (parent_id, "wireless_lan_mac.dest group address",
            &group) == OPC_COMPCODE_SUCCESS)
{
      op_pk_nfd_set (pkptr, "group_addr", group);
}
/* Seed the rand using a number */
if (num_gen == 65000)
{
      num_gen = 0;
}
num_gen = num_gen + 1;
srand (num_gen);
/* Call the rand, get the number */
result = rand();

/* Set the sequence number */
op_pk_nfd_set (pkptr, "sequence_num", result);
printf("Sending out a groupcast packet the sequence number is %d \n",result);
op_ima_obj_attr_get (parent_id, "name", &name_buffer);
printf("My requester node name is [%s] \n", name_buffer);

/* Set packet attributes */
op_ima_obj_attr_get (parent_id, "Wireless LAN MAC Address", &my_address);
op_ima_obj_attr_get (parent_id, "wireless_lan_mac.dest group address",
            &group_address);
op_pk_nfd_set (pkptr, "src_mac", my_address);
op_pk_nfd_set (pkptr, "origin_addr", my_address);

op_pk_nfd_set (pkptr, "dst_mac", group_address);
op_ima_obj_attr_get(parent_id, "forward", &forward);
op_pk_nfd_set (pkptr, "forward", forward);
op_ima_obj_attr_get(parent_id, "range", &range);
op_pk_nfd_set (pkptr, "range", range);

/* Send the packet via the stream to the lower layer. */
op_pk_send (pkptr, SSC_STRM_TO_LOW);
```

The packet sent by the **`sig_request`** block arrives at the wireless MAC interface block (**`wlan_mac_intf`**), which then assigns the packet a destination group address in the destination address field. This address value is retrieved from a node-level attribute table which is user-selectable. The packet is then sent out a packet stream to the **`wireless_lan_mac`** processing block, which finally transmits the packet out into the environment.

7.2 *Wireless LAN MAC Interface Process Block*

As seen in Figure 6, the wireless LAN MAC process block (**`wlan_mac_intf`**) takes inputs from the signature request block (**`sig_request`**), signature request acknowledgement block (**`sig_request_ack`**), and the wireless transceivers (**`wireless_lan_mac`**). It is comprised of several blocks, as shown in Figure 9. The first two blocks (**init**, **init2**) are initialization blocks, which execute preprocessor directives. The **wait** block queries the model registry for MAC-layer information, such as the assigned object ID and the packet stream numbers associated with this wireless LAN MAC layer process. The wireless LAN MAC process block also checks to determine if the current outgoing transmission will be associated with a standard broadcast or the newly defined groupcast.

The **idle** block remains in the idle state until interrupted by an incoming packet from either the **`sig_request`** process block, the **`sig_request_ack`** process block, or the **`wireless_lan_mac`** process block. The **fwd_grp_out** block processes packets originating from a call to the function **ss_packet_generate_fwd** contained within the **`sig_request_ack`** process block. This module packs the appropriate header fields of the packet which includes assigning a groupcast destination address. It is then forwarded out to the **`wireless_lan_mac`** process block.

The **app_layer_arrival** block processes packets originating from either the **`sig_request_ack`** and **`sig_request`** process blocks, or from its own **mac_layer_arrival** module. Packets originating from the higher level processes are created by a call to the **ss_packet_generate** function located in the header block of both processes. The module assigns the IP destination address to the packet which has been set by the user as an attribute associated with the node model.

The **mac_layer_arrival** module simply forwards an incoming packet from the lower **`wireless_lan_mac`** process model to the resident **appl_layer_arrival** block, which in turn forwards the packet to the higher layer **`sig_request_ack`** process block. Figure 9 shows the **`wlan_mac_intf`** process model.

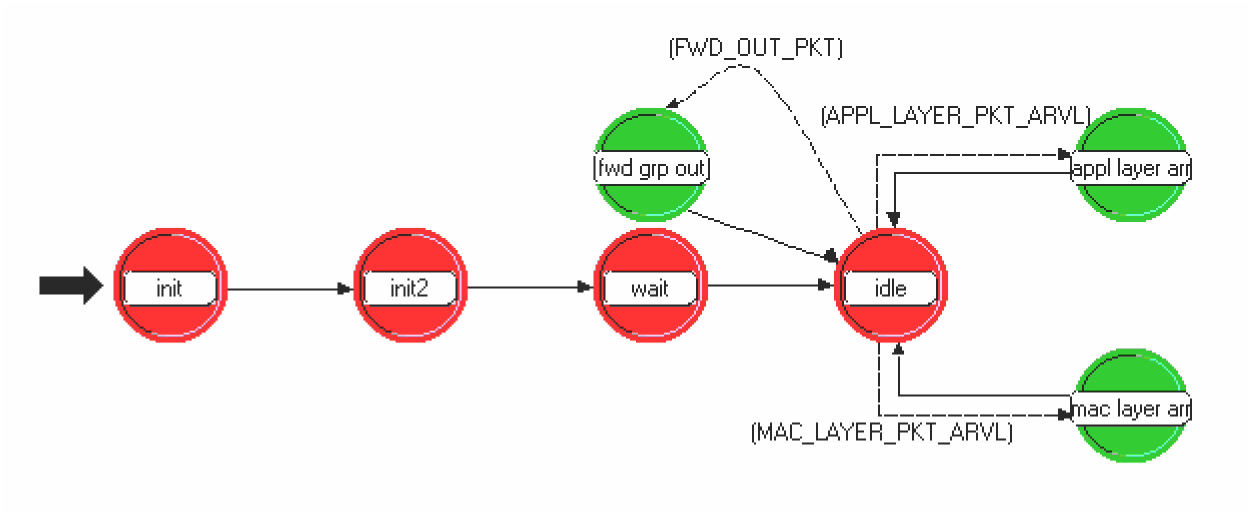


Figure 9. Wireless LAN MAC interface (**`wlan_mac_intf`**) process model

7.3 Wireless LAN MAC Process Block

From Figure 6, it is clear that the last processing stage before the physical interface in the wireless node model is the **`wireless_lan_mac`** process block. This process block is responsible for emulating the media access communication process within the wireless model. It is also responsible for orchestrating access to and reception from the wireless medium. This media access control is accomplished by buffering incoming messages from the higher layers within the communication stack, and then listening to the wireless media to determine if those messages can be sent without collision. If collisions occur, this block retransmits the wireless frames until they reach their intended destination. Finally, this stage is also responsible for discerning if frames on the wireless media are destined for itself. Figure 10 shows some of the logic associated with this process block.

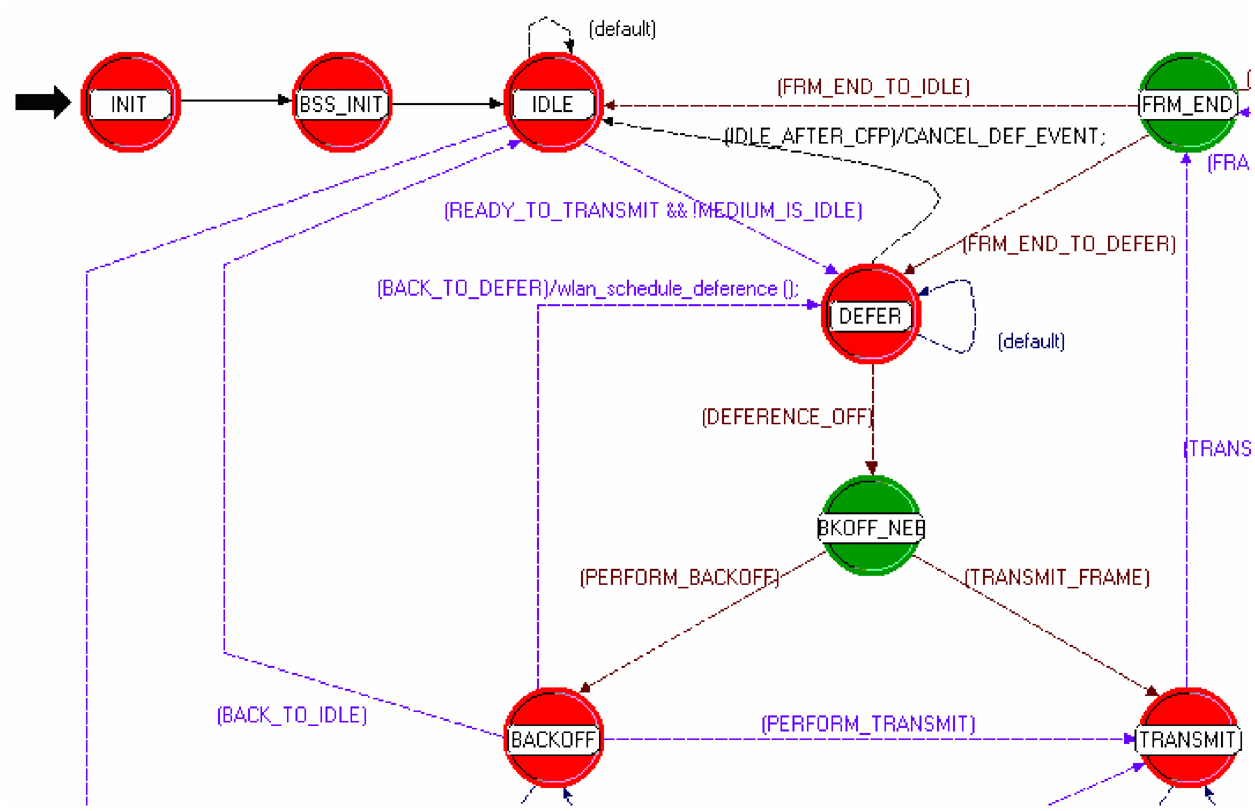


Figure 10. Wireless LAN MAC (`wireless_lan_mac`) process model

7.4 Signature Request Acknowledge Process Block

The process block that provides most of the logic associated with the certificate request protocol is the signature request acknowledge (`sig_request_ack`) process block. Within this process block, signature request packets are processed, returned to sender via unicast, and forwarded out via groupcast to other nodes in more distant domains.

The **init** block of the `sig_request_ack` process initializes this module by obtaining the object identification number, registering handles for gathering statistics, and registering macros that display the transmission and reception of wireless frames. After this process block is initialized it moves to the **idle** block. The **idle** block waits for stream interrupts and, when one occurs, determines to which logic module the packet needs to be forwarded. There are five logic modules to which the **idle** block can forward packets. For instance, signature request packets are forwarded to the signature request (**sig_req**) module. Below is a description of the **sig_req** and **sig_resp** logic modules.

7.4.1 Signature Request (sig_req) Logic Module

The signature request state block is responsible for processing incoming signature request packets. One of its first checks is to determine if the incoming request has its forwarding bit set. If the bit is not set, then the packet is locally processed and returned to sender. The following source code shows this processing.

```
if (op_ima_obj_attr_get (parent_id, "name", node_name) ==
            OPC_COMPCODE_SUCCESS)
{
      printf ("%s has received a signature request frame (%f) \n", node_name,
                  op_sim_time());
      op_pk_nfd_access (pkptr, "src_mac", &src_mac);
      printf ("The source MAC address of this frame is: %d \n", src_mac);
      op_pk_nfd_access (pkptr, "origin_addr", &origin);
      printf ("The origin MAC address of this frame is: %d \n", origin);
}
/* Check forward byte to determine if requester wants to groupcast      */
/* this packet to another domain                                        */

op_pk_nfd_access (pkptr, "forward", &forward);

if (forward == 0) // No forwarding, stay within domain
{
      cp_pkptr = op_pk_copy(pkptr);

      /* Grab sequence number so it can be pushed into returning packet */
      op_pk_nfd_access (pkptr, "sequence_num", &sequence);

      /* Retrieve and store src_addr field so it can be used to         */
      /* pack returning packet                                          */
      op_pk_nfd_access (pkptr, "src_mac", &return_src_addr);

      /* Call delay interrupt, send certificate back to requester       */
      op_intrpt_schedule_self(op_sim_time() + DELAY_TIME, PROCESS_DELAY);
}
```

The last line in the code above schedules an interrupt after an appropriate processing delay. The processing delay is representative of the time taken to generate a partial certificate. After the processing delay, the module transitions from the **idle** state to the **delay** state. The **delay** state creates a response containing the partial certificate and unicasts it back to the requester.

If the signature request (**sig_req**) module receives a request whose forwarding bit is set, it forwards out the request in hopes of reaching other nodes that are located beyond the transmission range of the requester. The following code performs this activity.

```
/* You are here because the forward byte was 1, check range byte to     */
/* determine the depth of this request                                  */
```

```
if (forward == 1)
{
      cp_pkptr = op_pk_copy(pkptr);
      op_pk_nfd_access(cp_pkptr, "forward", &forward);
      op_pk_nfd_access(cp_pkptr, "range", &range);
      printf("The range value is: %d \n", range);
      op_pk_nfd_access(cp_pkptr, "dst_mac", &dst_mac);
      printf("The destination address is: %d \n", dst_mac);
      op_pk_nfd_access(cp_pkptr, "src_mac", &src_mac);
      printf("The source address is: %d \n", src_mac);

      printf("the forward byte was 1  \n");
      count = count + 1;
      //printf ("Total of signature request frames received, fwd = 1 is: %d
                  \n\n", count);
      //op_pk_nfd_access (cp_pkptr, "sequence_num", &sequence);

      /* Check queue, is this the first packet? */
      /* If so push packet into queue */
      if ((testnum_pkts = op_subq_stat(0, OPC_QSTAT_PKSIZE)) == 0)
      {
            printf("The number I print should be 0 [%d] \n", testnum_pkts);

            /* Push packet into queue */
            op_subq_pk_insert (0, cp_pkptr, OPC_QPOS_HEAD);

            /* Get the size of the queue */
            num_pkts = op_subq_stat (0, OPC_QSTAT_PKSIZE);
            printf("The number of packets in the queue is [%d] \n",
                        num_pkts);
            pkptr_1 = op_subq_pk_access(subq_num, OPC_QPOS_HEAD);

            op_pk_nfd_access(pkptr_1, "sequence_num", &sequence_compare);
            printf("Accessing the sequence # stored = : %d \n",
                        sequence_compare);
```

Note that this code includes a packet cache which stores a copy of the request packet. The packet cache is necessary so that the node can act as an intermediary and forward partial certificate responses from distant NL2 nodes back to the original NL1 node.

The distance in which a certificate request is forwarded is determined by the range byte. If the range byte is greater than zero, then the node decrements the range byte and forwards the packet back out. When the range byte is zero, the depth has been reached and the packet is no longer forwarded. The following code shows this process.

```
      /* Is range value 0? */
      if (range == 0)
      {
            /* Grab sequence num so it can be inserted into returning   */
            /* unicast packet                                       */
            op_pk_nfd_access(cp_pkptr, "sequence_num", &sequence);

            /* Call delay interrupt send certificate back to requester  */
```

```
                op_intrpt_schedule_self(op_sim_time() + DELAY_TIME,
                PROCESS_DELAY);
        }

        if(range >= 1)
        {
                printf("range value was 1 or greater \n");

                /* Grab sequence num so it can be inserted into returning   */
                /* unicast packet                                            */
                op_pk_nfd_access(cp_pkptr, "sequence_num", &sequence);

                /* Call delay interrupt, send certificate back to requester */
                printf("Sending a unicast packet back to requester \n");
                op_intrpt_schedule_self(op_sim_time() + DELAY_TIME,
                PROCESS_DELAY);

                /* Grab sequence num so it can be inserted into forwarding  */
                /* packet                                                    */
                op_pk_nfd_access(cp_pkptr, "sequence_num", &sequence_2);
                /* Decrement range byte and send to forwarding routine      */
                range = range - 1;
                printf("Forwarding out a groupcast packet to another domain_1
                        \n\n");
                ss_packet_generate_fwd();
        }
}
```

The aforementioned packet cache is also necessary for loop detection. Because nodes are often forwarding packets to the same group address, it is likely that multiple copies of the same request will be received. Without logic to filter out identical requests, the forwarding nodes will answer the same request multiple times. To overcome this potential inefficiency, a loop detection mechanism was deployed to identify and ignore duplicate requests. The following code provides this technique by comparing all incoming requests with cached requests. The comparison is based on the sequence number that uniquely identifies each request. The following code shows this process.

```
else
{
        /* There is at least 1 packet in the queue, Perform loop detection */

        op_pk_nfd_access(cp_pkptr,"sequence_num", &sequence_3);
        printf("The sequence value for received packet is [%d] \n",
                sequence_3);

        num_pkts = op_subq_stat(subq_num, OPC_QSTAT_PKSIZE);
        printf (" The number of packets in the test queue is [%d] fwd =1 pkt >
                1 \n", num_pkts);

        /* Loop through each packet in subqueue */
        for(i = 0; i < num_pkts; i++)
        {
                /* Grab packet from queue */
                pkptr_2 = op_subq_pk_access(subq_num, i);
                // printf("subq_num = [%d], i = [%d] \n", subq_num, i);
                /* Grab packets'sequence field */
                op_pk_nfd_access(pkptr_2, "sequence_num", &sequence_compare);

                /* Check for loop, compare field values */
                //printf("The sequence# = [%d] the sequence compare = [%d] \n",
                            sequence_3, sequence_compare);
                if (sequence_3 == sequence_compare)
                {
                printf("loop was detected!!! packet ignored \n\n");
                //set loop detector counter
                loop_cntr = loop_cntr + 1;
                }
                printf("I just jumped out of the loop detector, \n\n");
        }


        if ( loop_cntr == 0 ) // Loop was not detected
        {
                /* Grab range byte */
                printf("loop was not detected \n");
                op_pk_nfd_access(cp_pkptr, "range", &range);

                /* Push packet into queue */
                op_subq_pk_insert (0, cp_pkptr, OPC_QPOS_HEAD);

                /* Set cache timer */
                //op_intrpt_schedule_self(op_sim_time() + PACKET_REMOVE,
                            CACHE_DELAY);

                //printf ("The timer is set for [%f] + Packet_Delay \n",
                            op_sim_time());
                /* Get size of queue */
                num_pkts = op_subq_stat (subq_num, OPC_QSTAT_PKSIZE);
                printf("The number of packets in the queue is [%d] fwd =1 & seq
                            != seq \n", num_pkts);
```

At this juncture the loop detection mechanism has been employed. If a loop was detected, then the packet is ignored and no additional processing is needed. If, on the other hand, there is no loop, then the processing continues. The following code processes the request and checks the range byte to determine if this request needs to be propagated further.

```
        /* Is range byte equal to zero? */
        if(range == 0)
        {
              /* Do not groupcast this packet out to other domains, */
              /* just call delay and send packet back to requester   */

              /* Retrieve sequence # so it can be inserted into      */
              /* returning packet                                    */
              op_pk_nfd_access (cp_pkptr, "sequence_num", &sequence);

              /* Call delay interupt, send certificate back to       */
              /* requester                                           */
```

```
                        op_intrpt_schedule_self(op_sim_time() + DELAY_TIME,
                        PROCESS_DELAY);
                }


                if(range != 0)
                {
                        printf("range value is :%d \n\n", range);
                        range = range -1;
                        /* Grab sequence num so it can be inserted into       */
                        /* forwarding groupcast packet                        */
                        op_pk_nfd_access(cp_pkptr, "sequence_num", &sequence_2);

                        /* Forward out packet to another domain               */
                        ss_packet_generate_fwd();

                        /* Also process local received packet for partial     */
                        /* certificate generation and send back to upstream   */
                        /* requester                                          */

                        op_intrpt_schedule_self(op_sim_time() + DELAY_TIME,
                        PROCESS_DELAY);
                        //                                                }

                }
                if (loop_cntr >= 1)
                {
                        //printf("loop was detected!!!, Packet discarded \n\n");
                }
        }
    }
}
else
{
        //printf ("I'm a requester node I will not respond \n \n");
}
```

7.4.2 Signature Response (sig_resp) Logic Module

After a signature request is received and processed, a signature response is transmitted. When a node received a signature response, it must either process the response or forward it to the node which requested it. The node checks each response to see if the response is destined for itself or another node then acts accordingly. The following code performs the processing of the signature response messages.

```
/* Obtain the incoming packet.                                            */
//pkptr = op_pk_get (op_intrpt_strm ());
/* This has been moved to the IDLE block                                  */

/* This is the signature response logic module. It has one of two        */
/* functions:                                                             */
/* Function 1, Process end of the line certificate packets that          */
/*   originated from this node as signature requests or                  */
/* Function 2, Relay a partial certificate response back to the          */
/*   originating node. This entails retrieving a cached packet with a    */
/*   matching sequence number and inserting the proper destination      */
/*   address of the previous (upstream node)                            */

/* Is this packet addressed to me?                                      */
op_pk_nfd_access (pkptr, "dst_mac", &address);
op_ima_obj_attr_get (parent_id, "Wireless LAN MAC Address", &address_1);

if (address == address_1)
{

      if (op_ima_obj_attr_get (parent_id, "name", node_name_1) ==
                  OPC_COMPCODE_SUCCESS)
      {
            printf ("%s has received a signature response frame %f \n",
                        node_name_1, op_sim_time());
            //printf ("Node [%d] has received a signature response frame \n",
                        parent_id);
            op_pk_nfd_access (pkptr, "sequence_num", &sequence);
            printf ("The sequence number is : %d \n", sequence);
            op_pk_nfd_access (pkptr, "src_mac", &src_addr_1);
            printf ("The source address of this packet is : %d \n",
                        src_addr_1);
            op_pk_nfd_access (pkptr, "dst_mac", &dest_mac_1);
            printf ("The destination address of this packet is : %d \n",
                        dest_mac_1);
            op_pk_nfd_access(pkptr, "origin_addr", &origin);
            printf("The origin address of this packet is: %d \n", origin);

            //num_pkts = op_subq_stat(0, OPC_QSTAT_PKSIZE);
            //printf ("the number of packets in queue is %d\n",num_pkts);
            if (address == origin)
            {
                  printf ("Packet has arrived Home, the sequence # is:[%d]
                              \n", sequence);
                  signature_response = signature_response + 1;
                  printf ("The total number of signature responses is [%d]
                              \n\n", signature_response);
            }
            else
            {
                  op_pk_nfd_access (pkptr, "origin_addr", &src_mac);
                  printf ("This response packet is not mine, unicast to
                              origin \n\n");
                  ss_packet_generate_1();
            }
```

7.4.2 Other Logic Modules

The signature request acknowledge (**`sig_request_ack`**) process block has several other logic modules, **delay**, **cache**, and **erase**, that support the two primary state processing blocks, **sig_req** and **sig_resp**. Their descriptions are omitted.

7.5 *Mobility Processing Block*

Each node has a mobility processing block that dictates its movement in the simulation. This process block is called **`mobile_route`**, as seen in Figure 6. The starting point of each node is generated randomly based on a uniform distribution. The maximum and minimum values are associated with the size of movement area grid. Once the starting point of the node is known, then a target position, or stopping point, is randomly selected. The node will then move on a vector linking its starting point to its target destination. After reaching its stopping point, the node pauses based on some preset pause time and calculates a new path. Additional variables allow the user to set the velocity of the nodes. Figure 11 shows the node mobility process model.

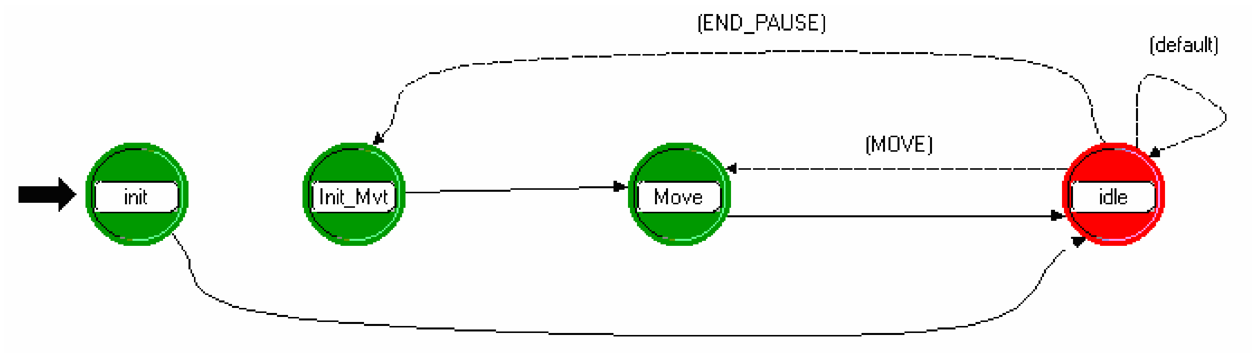


Figure 11. Mobility (**`mobile_route`**) process model

8 Protocol Simulation

In our distributed CA, if a certificate request is heard by at least *k* NL2 nodes, then those nodes can satisfy the request. That assumption is valid if the mobile network is dense enough or if *k* is small enough. However, the difficulty is ensuring that *k* nodes receive the request, especially in sparse networks or networks in which *k* is large (relative to the number of nodes). We used our models to simulate the system and further examine this problem.

The simulation model is comprised of a mobile NL1 requester node and multiple mobile NL2 response nodes. The intent of the simulation was to examine visually and statistically features associated with the protocol. Specifically, it was important to test the effectiveness of the protocol in mobile ad hoc environments. In order for the protocol to work, the initiating NL1 node needs to receive at least *k* partial certificate responses. With that goal in mind, our simulations focused on analyzing the response rate of the mobile ad hoc network to the different types of certificate requests.

8.1 *Basic Certificate Request Simulation*

The first scenario we considered was the basic certificate request scenario. In this scenario, an NL1 node broadcasts a request for a certificate, and the NL2 nodes within transmission range compute and return partial certificates. No domain propagation occurs.

This scenario was intended to gauge the efficiency gained when transmitting via groupcast as opposed via unicast. Standard unicast transmissions are limited to a one-to-one ratio of transmissions to responses. For our distributed CA protocol to work efficiently, the groupcast transmissions must be efficient. This scenario was also designed to test the feasibility of receiving enough partial certificates using only basic certificate requests. Therefore, the metric used was the ratio of transmitted certificate requests to successfully received responses.

The simulations were run with ten NL2 responder nodes and one NL1 requester node randomly positioned in one square kilometer. The nodes all selected a random starting location within the area and randomly plotted a vector course. Each node then followed its own movement vector until reaching the area boundary, paused, recalculated another vector, and continued its movement. The requester sent a certificate request once per minute.

Figures 12-15 show the simulated sequence of events. The simulation starts in Figure 12 with a certificate request broadcast to a group address. The concentric rings represent the omni-transmission pattern and range of the requester. When the outermost circle encompasses any other node, that node is highlighted in blue indicating that the transmission has been received and is being processed. This “highlighting” takes place almost simultaneously but is shown sequentially on the simulator because receiver activity is based on the actual arrival time of the transmitted signal. Because each node is at a slightly different distance from the requester node, the transmitted energy arrives at slightly different times. Figures 10-13 show a single certificate request transmission (Figure 12) that is received by three NL2 nodes (Figures 13-15.

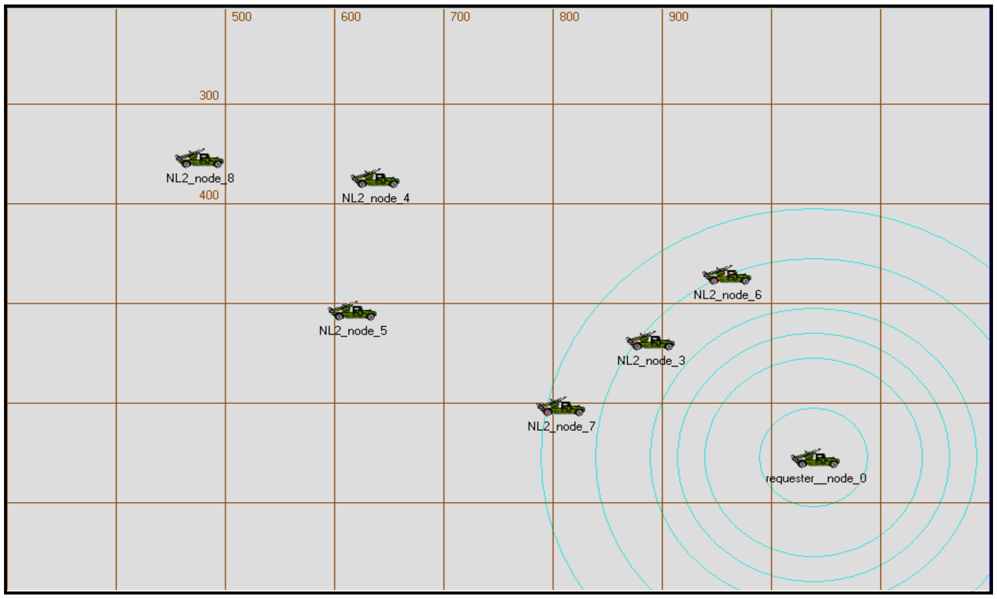


Figure 12. Initial certificate request transmission

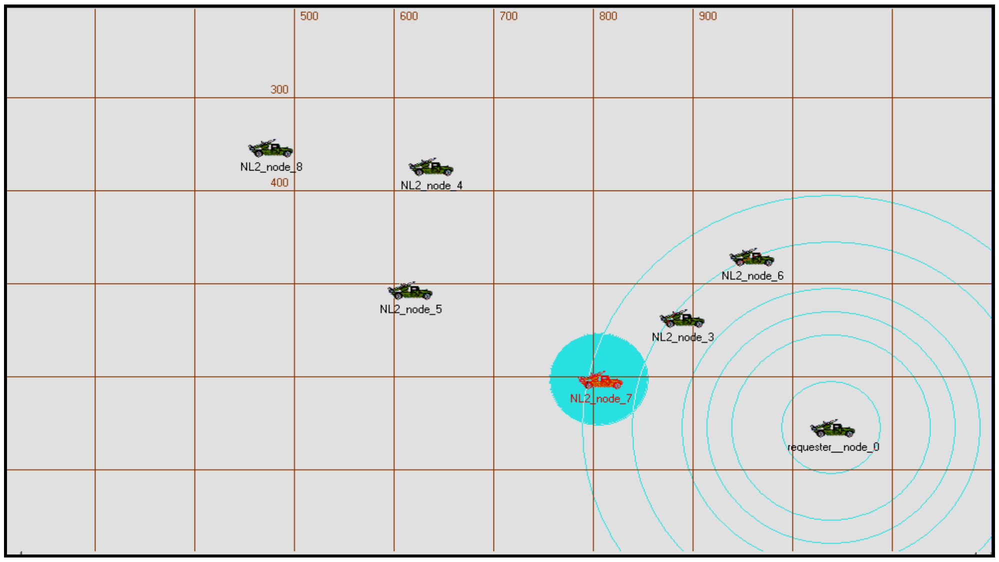


Figure 13. A node receives the certificate request transmission

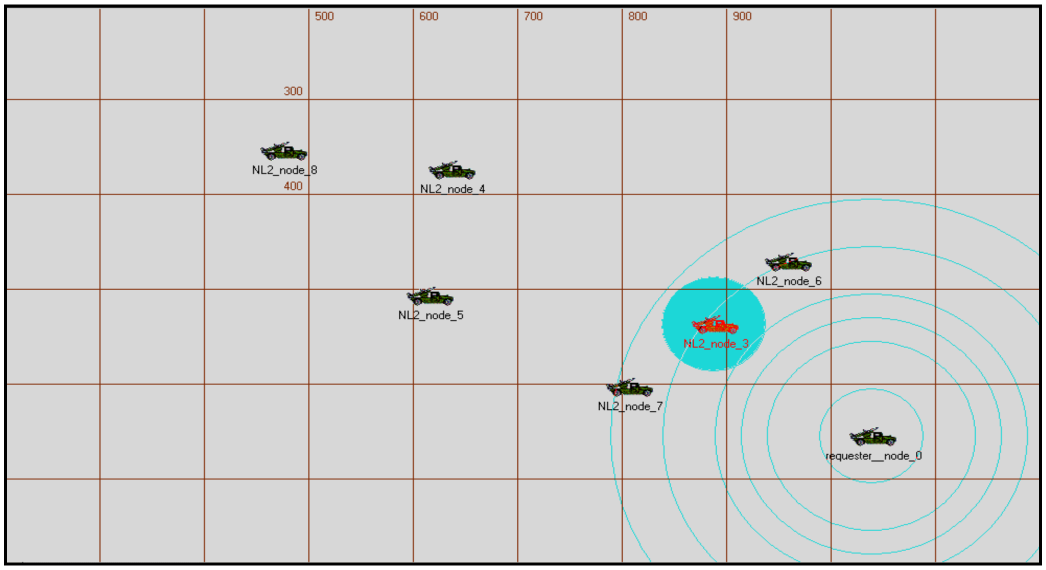


Figure 14. A second node receives the certificate request transmission

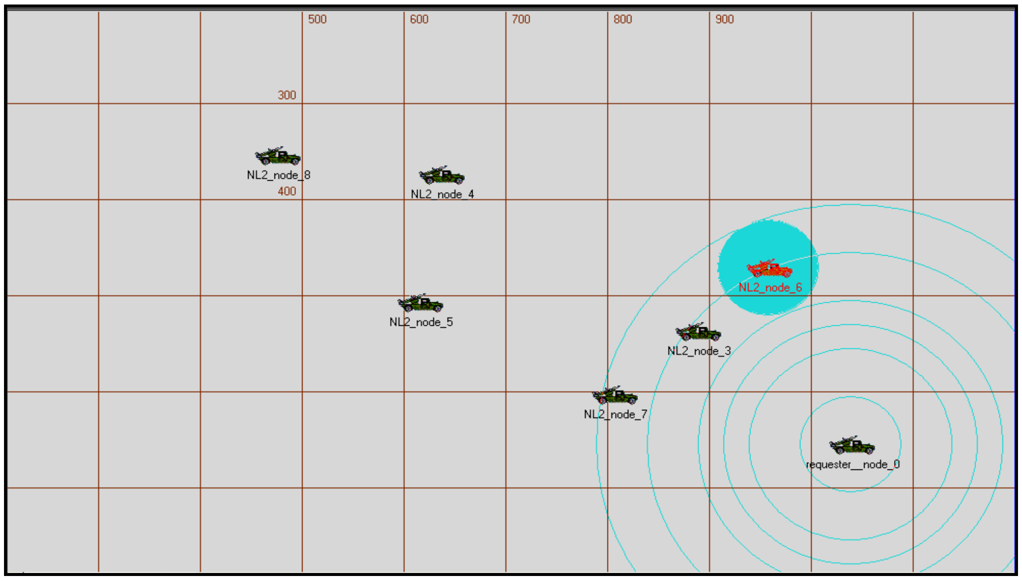


Figure 15. A third node receives the certificate request transmission

For each certificate request, the NL1 node tallied the number of responses that it received from the neighboring NL2 nodes. The number of responses was based on how many responders could be reached per each transmitted request. As seen in Figure 16, in our simulations the number of nodes that received each transmitted request varied over time based on the location of each receiving node relative to the requester. Depending on the system's threshold value *k* and the density of the nodes, this variation may or may not be acceptable. In our scenario (ten nodes per square kilometer), if *k* is greater than 20% of the NL2 nodes, the CA protocol will experience difficulties.

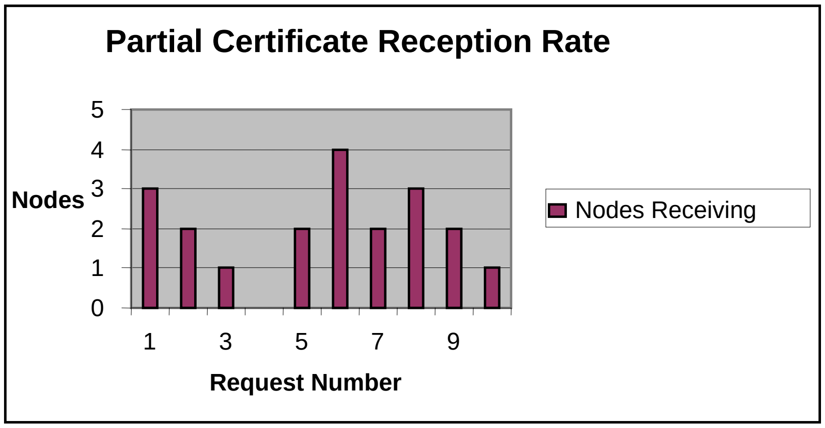


Figure 16. Partial certificate reception rate

8.2 Certificate Request with Domain Extension Simulation

The second scenario focused on the certificate requests that are propagated beyond the initial transmission domain. In these simulations, the initiating NL1 node sends out a certificate request that is marked for domain extension. When the request is received by NL2 nodes, they process the request, re-broadcast the transmission, and send a partial certificate back to the initiating NL1 node. The original NL1 node waits a pre-determined amount of time in hopes of receiving the requisite number of partial certificates before timing out and aborting the certificate request.

A series of simulations was conducted to determine the effectiveness of using the domain extension capability of the protocol. The first simulation was executed twice, once using the basic certificate request with domain extension disabled, and a second time with domain extension enabled. As before, the positions of all participating nodes were restricted to one square kilometer, and all starting positions of the nodes and their paths during the simulation were randomly generated. The requester node (`requester_node_0`) sent a certificate request once per minute. NL2 nodes that received the certificate request processed the request and sent partial certificates back to the requesting node. In the second simulation, the NL2 nodes also re-broadcast the request.

Like that of the previous simulations, the goal was to analyze the rate of partial certificate responses. As before, the number of responses was based on how many NL2 nodes could be reached by each request. The metric used was the ratio of transmitted certificate requests to successfully received responses.

Figures 17-24 are snapshots from the communication simulator. In Figure 17, the certificate request is sent out. In Figure 18 the first NL2 node receives the certificate request, processes it, and then re-broadcasts the request. In Figure 19, a second NL2 receives the request and acts in a likewise manner. The same pattern continues in Figures 20-24.

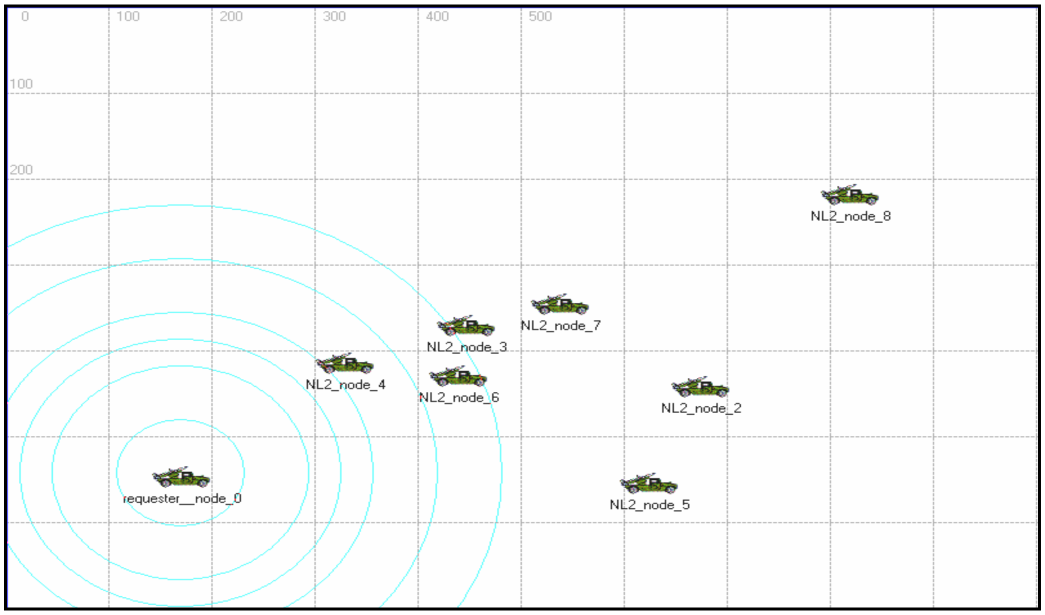


Figure 17. Initial certificate request transmission

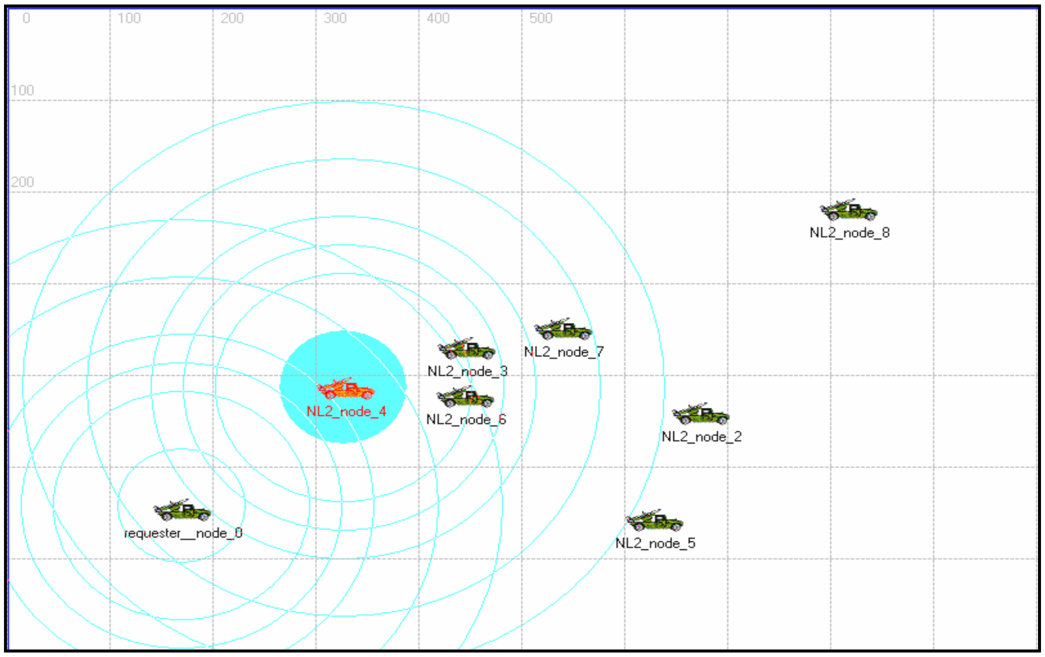


Figure 18. A node receives the request and re-broadcasts it

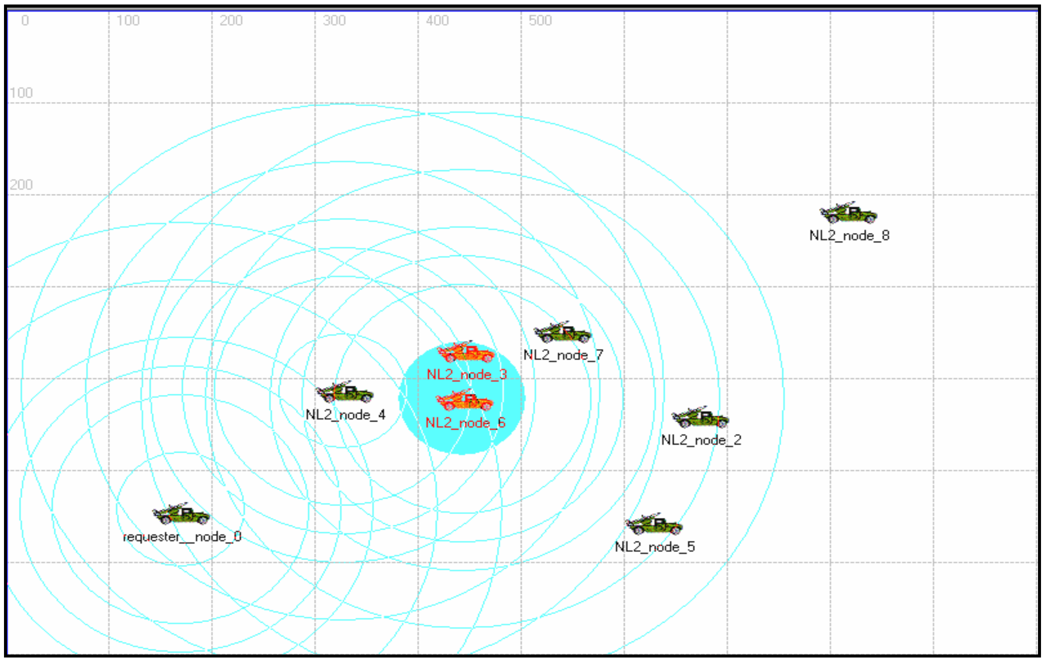


Figure 19. A second node receives the request and re-broadcasts it

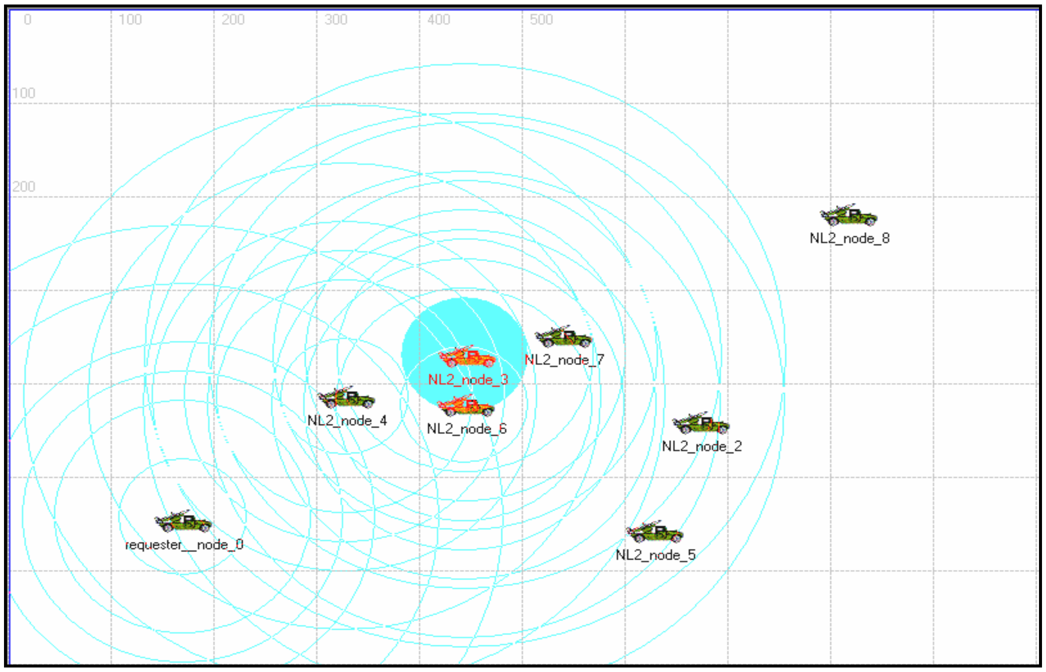


Figure 20. A third node receives the request and re-broadcasts it

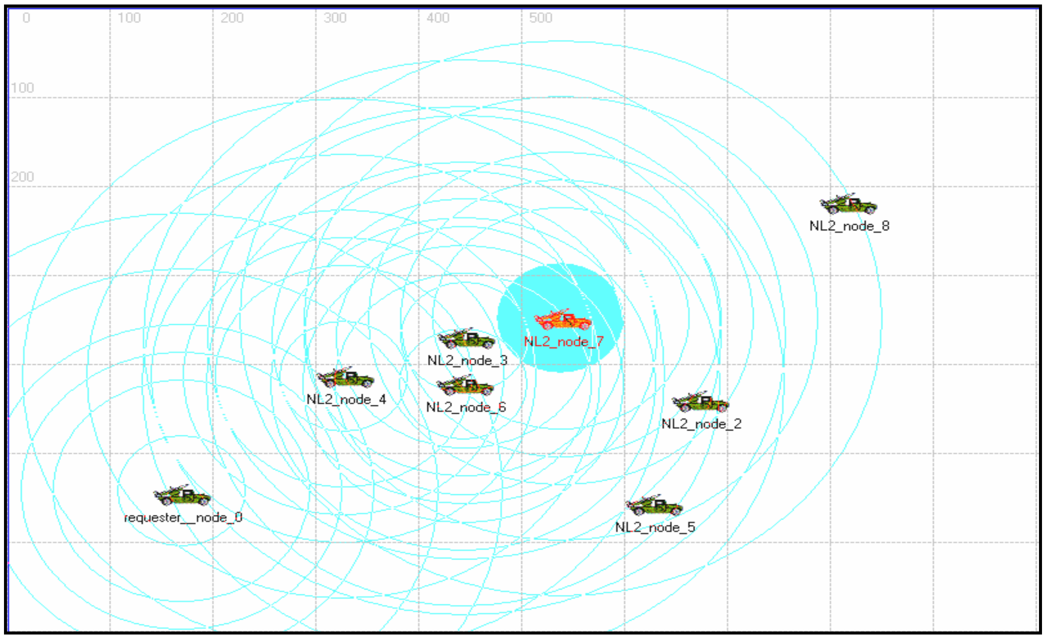


Figure 21. A fourth node receives the request and re-broadcasts it

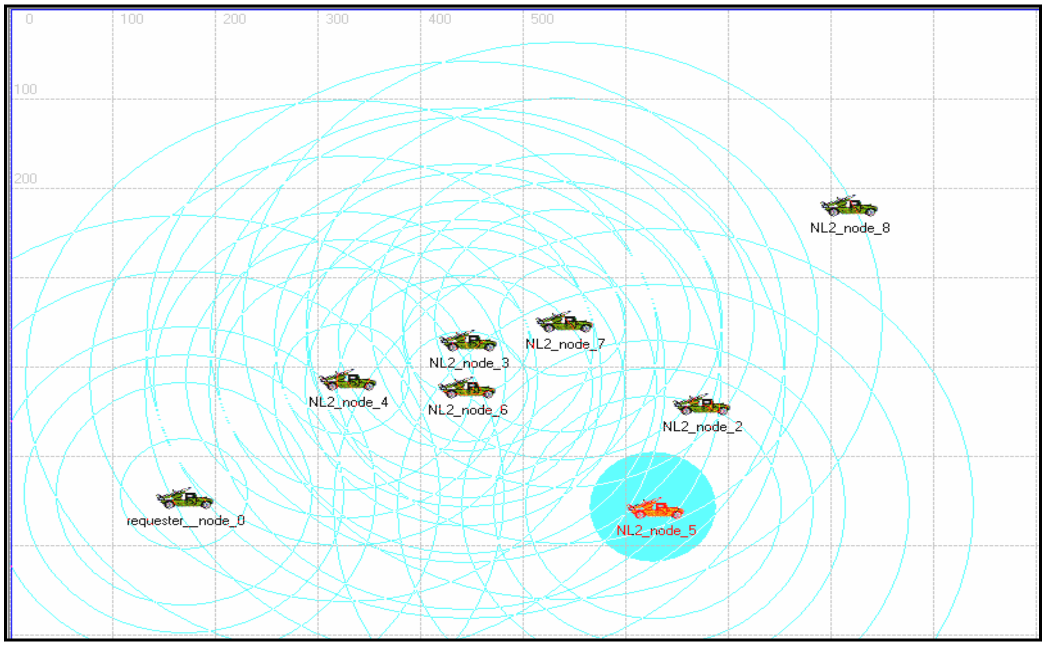


Figure 22. A fifth node receives the request and re-broadcasts it

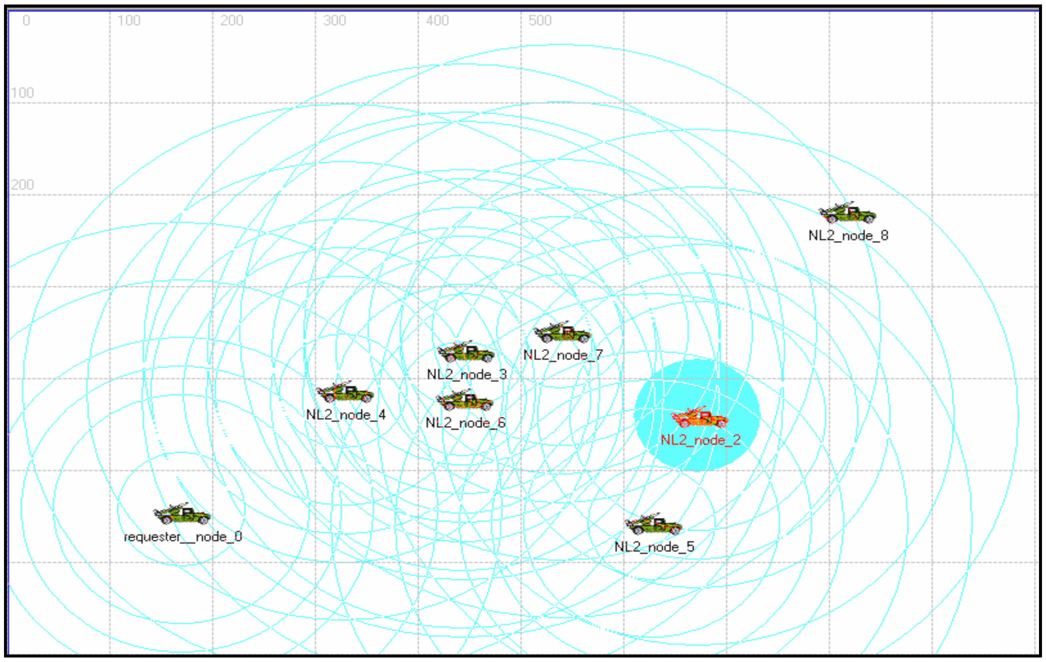


Figure 23. A sixth node receives the request and re-broadcasts it

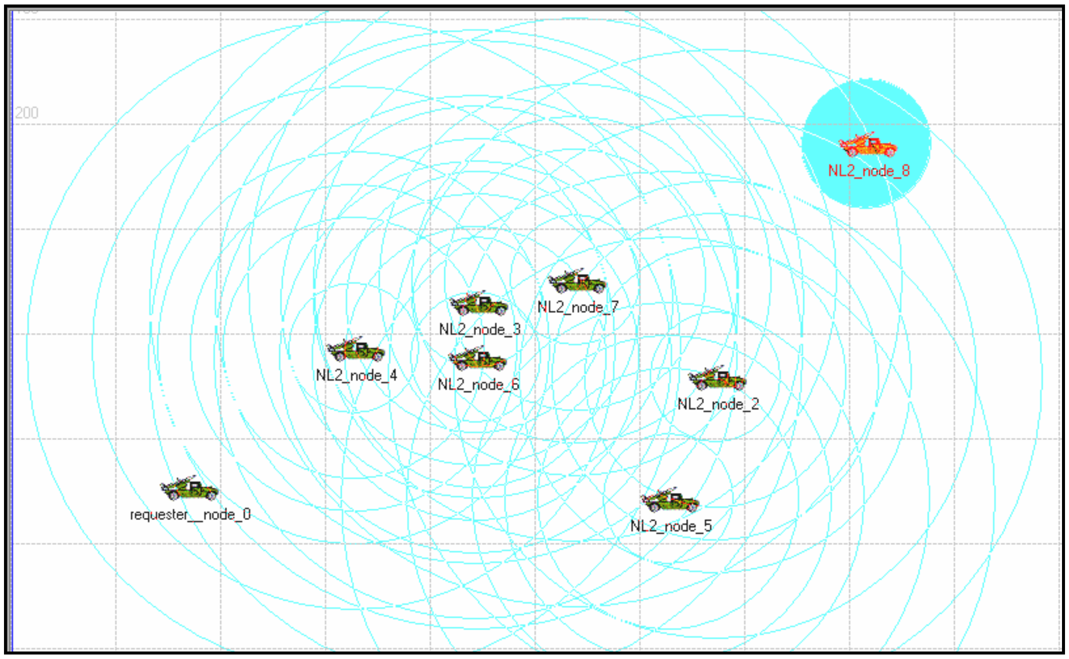


Figure 24. A final node receives the request and re-broadcasts it

The response rates of the two simulations are shown in Figure 25. The first simulation, during which domain extension was disabled (range = 0), is depicted in blue. The second simulation, which made use of the domain extension capability (range = 1), is depicted in red.

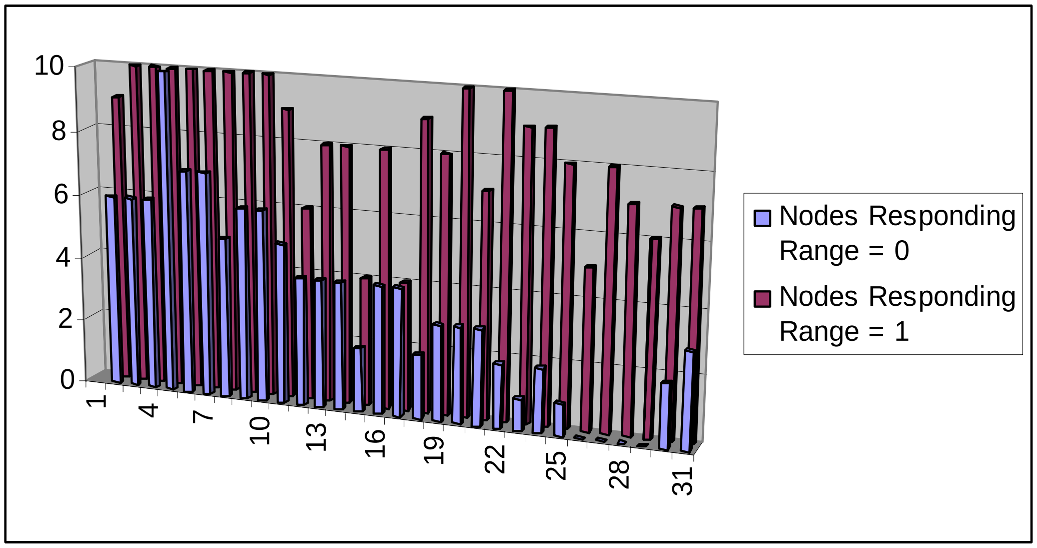


Figure 25. Partial certificate reception rates (1 km × 1 km)

According to Figure 25, the range extension bit does have an impact on the numberof responses received. The blue bars on the graph represent the number of responses received by the originating requester with transmission limited to a single transmission domain. The red bars on the graph represent the number of responses received by the originating requester with transmission extension enabled to propagate out into an additional domain. The average number of partial certificates received for the basic request without domain extension was 3.5 compared to an average of 8.2 partial certificates when using domain extensions. That is a 134% increase.

The same pair of simulations was executed in environments of 2.5 kilometers by 2.5 kilometers and 5 kilometers by 5 kilometers, and the results are show in Figure 26 and Figure 27, respectively. The results support similar conclusions. As the network grows more sparse, the need to use domain extension grows. As seen in Figure 27, in very large, sparse environments, using domain extension is essential for a distributed CA to operate properly.

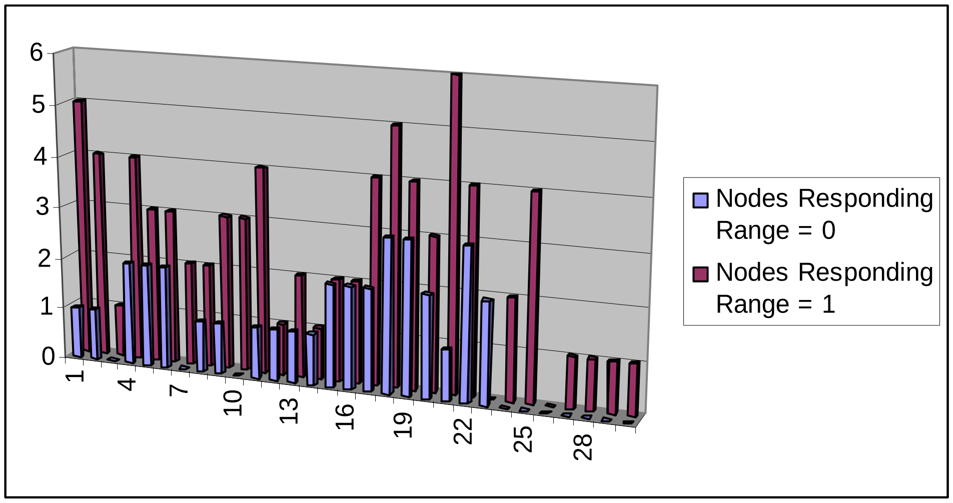


Figure 26. Partial certificate reception rates (2.5 km × 2.5 km)

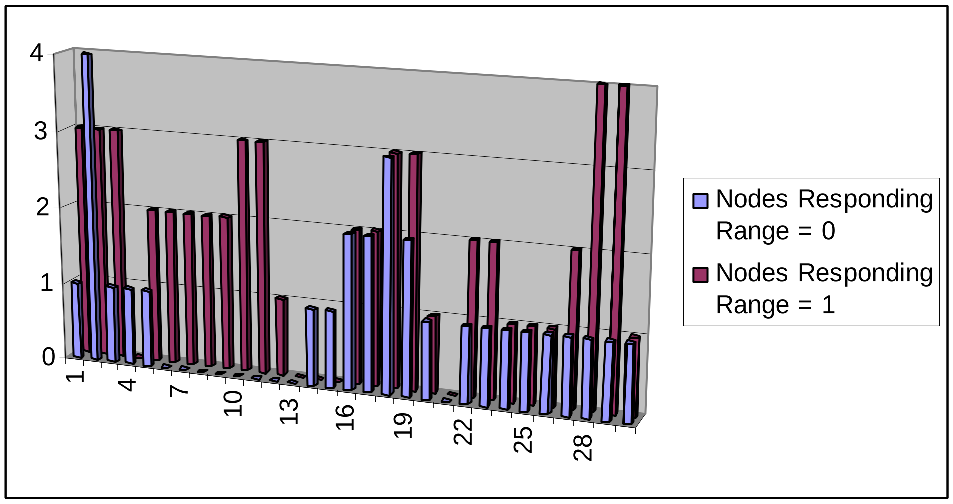


Figure 27. Partial certificate reception rates (5 km × 5 km)

9 Conclusion

In a mobile wireless environment, a distributed certificate authority offers many advantages over a centralize one. For this study, we validated this statement by creating and testing a general distributed CA protocol. The protocol specified the basic functionality of both regular and CA nodes. This distributed CA was then prototyped using detailed models that implemented the protocol. Using our model, several detailed simulations were performed. These simulations showed that our distributed wireless CA protocol with domain extension can provide CA services even in sparse environments when connectivity is sporadic, a scenario in which a centralized approach fails. The statistics we gathered and evaluated showed the performance improvements of our distributed wireless CA protocol.

As wireless technology continues to expand and demands on its operation increase, it will be important to continually pursue more efficient ways of providing authentication wireless node authentication. This study provided a basic proof-of-concept design, but more detailed analysis is warranted.

References

[1] W. Anderson, J. Michalski, and B. Van Leeuwen. Enhancements for distributed Certificate Authority Approaches for Mobile Wireless Ad Hoc Networks. Sand Report SAND2003-4395, December 2003.

[2] D. Boneh and M. Franklin. Efficient Generation of Shared RSA Keys. *Advances in Cryptology – CRYPTO '97*, pages 425-439. Lecture Notes in Computer Science 1294. Springer-Verlag, 1997.

[3] C. Crepeau and C. R. Davis. A Certificate Revocation Scheme for Wireless Ad Hoc Networks. *Proceedings of the 1st ACM workshop on Security of ad hoc and sensor networks*, pages 54-61. ACM, 2003.

[4] C. R. Davis. A Localized Trust Management Scheme for Ad Hoc Networks. *3rd IEEE International Conference on Networking*. IEEE, 2004.

[5] A. De Santis, Y. Desmedt, Y. Frankel, and M. Yung. How to Share a Function Securely. *26th Annual ACM Symposium on Theory of Computing*, pages 522-533. ACM, 1994.

[6] Y. Desmedt and Y. Frankel. Shared Generation of Authenticators and Signatures. *Advances in Cryptology – CRYPTO '91*, pages 457-569. Lecture Notes in Computer Science 576. Springer-Verlag, 1991.

[7] P. Feldman. A Practical Scheme for Non-interactive Verifiable Secret Sharing. *28th IEEE Symposium on Foundations of Computer Science*, pages 427-437. IEEE, 1987.

[8] Y. Frankel, P. Gemmell, P. MacKenzie, and M. Yung. Optimal Resilience Proactive Public-Key Cryptosystems. *38th IEEE Symposium on Foundations of Computer Science*, pages 384-393. IEEE, 1997.

[9] Y. Frankel, P. Gemmell, P. MacKenzie, and M. Yung. Proactive RSA. *Advances in Cryptology – CRYPTO '97*, pages 440-452. Lecture Notes in Computer Science 1294. Springer-Verlag, 1997.

[10] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust and Efficient Sharing of RSA Functions. *Advances in Cryptology – CRYPTO '96*, pages 157-172. Lecture Notes in Computer Science 1109. Springer-Verlag, 1996.

[11] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust Threshold DSS Signatures. *Advances in Cryptology – EUROCRYPT '96*, pages 354-371. Lecture Notes in Computer Science 1070. Springer-Verlag, 1996.

[12] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public-key and signature schemes. *4th Annual Conference on Computer Communications Security*, pages 100-110. ACM, 1997.

[13] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive Secret Sharing or: How to Cope with Perpetual Leakage. *Advances in Cryptology – CRYPTO '95*, pages 339-352. Lecture Notes in Computer Science 963. Springer-Verlag, 1995.

[14] C. Y. Liau, S. Bressan, and K. Tan. Efficient Certificate Revocation: A P2P Approach. ASIAN 2002 Workshop on Southeast Asian Computing Research, 2002.

[15] M. Malkin, T. Wu, and D. Boneh. Experimenting with Shared Generation of RSA keys. *Internet Society's 1999 Symposium on Network and Distributed System Security (SNDSS)*, pages 43-56. Internet Society, 1999.

[16] B. C. Popescu, B. Crispo, and A. S. Tanenbaum. A Certificate Revocation Scheme for a Large-Scale Highly Replicated Distributed System. Technical report, Vrije University, Amsterdam, 2002.

[17] T. Rabin. A Simplified Approach to Threshold and Proactive RSA. *Advances in Cryptology – CRYPTO '98*, pages 89-140. Lecture Notes in Computer Science 1462. Springer-Verlag, 1998.

[18] B. Schoenmakers. A Simple Publicly Verifiable Secret Sharing Scheme and its Application to Electronic Voting. *Advances in Cryptology – CRYPTO '99*, pages 148-164. Lecture Notes in Computer Science 1666. Springer-Verlag, 1999.

[19] A. Shamir. How to Share a Secret. *Communications of the ACM 22 (11)*, pages 612-613, 1979.

[20] V. Shoup. Practical Threshold Signatures. *Advances in Cryptology – EUROCRYPT 2000*, pages 207-220. Lecture Notes in Computer Science 1807. Springer-Verlag, 2000.

[21] M. Stadler. Publicly Verifiable Secret Sharing. *Advances in Cryptology – EUROCRYPT '96*, pages 190-199. Lecture Notes in Computer Science 1070. Springer-Verlag, 1996.

[22] P. Zheng. Tradeoffs in Certificate Revocation Schemes. *ACM SIGCOMM Computer Communication Review 33 (2),* pages 103-112, 2003.

Distribution

	MS	
1	0785	R. L. Hutchinson, 5616
5	0785	A. J. Lanzone, 5614
2	0785	J. T. Michalski, 5616
1	0785	T. S. McDonald, 5614
1	0784	R. E. Trellue, 5610
1	1161	T. K. Stalker, 5432
1	MS9018	Central Technical File, 8945-1
2	MS0899	Technical Library, 9616