

Secure Group Communication over Ad-Hoc Networks
Master Thesis Report

Lars-Arne Mattsson

12 November, 2000

Abstract

One of the most interesting services in a wireless ad-hoc network is group communication. To be able to do this securely some cryptographic system is needed to stop non authorized nodes eavesdropping on the discussions within the group. This should not be a concern for applications in group communication. Instead a standard API should be available that facilitates the use of a service that takes care of this problem.

This thesis provides background information and present the state of the technologies related to security within the ad-hoc group environment.

Furthermore the thesis describes a way to solve the problem and how this solution can be used in real world scenarios.

Finally, it presents a detailed specification of an API for secure ad-hoc group communication.

Preface

I have written this master thesis as part of my studies at the Computer Science Program at Uppsala University.

The thesis is written as part of the MARCH (Mobile Aware Server Architecture) project.¹

My advisor is Yuri Ismailov at Ericsson mobile systems and network research and my examiner is professor Per Gunningberg, Department of Computer Systems (DoCS), Uppsala University.

I'd like to thank both Yuri and Per for their great support. I would also like to thank Tomas Larsson, Henrik Eriksson, Christian Gehrmann, Anders Björklund, Per Johansson, and all others that I had contact with discussing my thesis. Their input and support has given me a lot.

Finally I would like to thank my girlfriend Ingela Anderton for her support during the six months I struggled with this work.

¹This is a joint project between Uppsala University, Ericsson Research and University of New South Wales

Contents

1	Introduction	1
1.1	Background	1
1.2	Problem Description	1
1.3	Layout of The Report	1
1.4	Scenarios	2
1.4.1	General Characteristics	2
1.4.2	Meeting Environment Scenarios	2
1.4.3	Information Gathering Scenario	2
1.4.4	Search & Rescue Scenario	3
2	State-of-the-Art	4
2.1	Ad-hoc On-Demand Distance Vector Routing	4
2.1.1	Unicast	4
2.1.2	Multicast	5
2.2	Addressing	6
2.2.1	Random Selection of IP address	6
2.2.2	Naming	6
2.3	Security	7
2.3.1	PGP	7
2.4	Group Handling in Ad Hoc Networks	10
2.4.1	Trust	10
2.4.2	Setting Up Trust	10
2.4.3	Announcement of Groups	13
2.4.4	Key Exchange	13
3	Solution	14
3.1	Assumptions	14
3.2	Node Setup	14
3.3	Trust Setup	14
3.3.1	Building the Initial Trust Group	14
3.3.2	Adding New Nodes and Joining Trust Groups	15
3.4	Group Setup	15
3.4.1	Public, Private and Protected Groups	15
3.4.2	Group Master Privileges	16
3.5	The Nodes	16
3.6	Messages	16
3.6.1	Broadcast	17
3.6.2	Multicast	17
3.6.3	Unicast	17
3.7	Group Operations	17
3.7.1	Add Node	17
3.7.2	Remove Node	17

3.7.3	Handling of Group Partitioning	18
3.7.4	Joining Two Groups	18
3.7.5	Destroy Groups	18
3.8	Node Operations	18
3.9	Scenario Handling	19
3.9.1	Group Meeting	19
3.9.2	Information Gathering	19
3.9.3	Search & Rescue	20
3.10	Communication	20
3.10.1	Message Transportation	20
4	Conclusions & Future Work	22
4.1	Future Work	22
4.1.1	Implementations	22
4.1.2	Effects on Physical Devices	23
4.1.3	Security	23
4.1.4	The Protocol	23
A	Functional Requirement Specification	26
A.1	Application Controlled Functions	26
A.1.1	Key Exchange	26
A.1.2	Address Handling	26
A.1.3	Create Group	27
A.1.4	Group Add	27
A.1.5	Group Remove	28
A.1.6	Send Message	28
A.1.7	Join Groups	28
A.1.8	Become New Master	29
A.1.9	Send Group Join Request	29
A.1.10	Send Group Leave Request	29
A.1.11	List Group Members	30
A.1.12	Group Destroy	30
A.2	Event Driven Functions	30
A.2.1	handle Availabilty Change	30
A.2.2	Recieve Message	31
A.2.3	Handle Grouplist Request	31
A.2.4	Handle Master Change	32
A.2.5	Handle Groupcomposition Change	32
B	Implementation Specification	33
B.1	Datastructures	33
B.1.1	GroupInfo	33
B.1.2	NodeID	33
B.1.3	Message	34
B.2	Communication	35
B.2.1	Multi- and Broadcast	35
B.2.2	createConnection	35
B.2.3	senddata	36
B.2.4	recievedata	36
B.2.5	listener	37
B.3	Addressing	38
B.3.1	allocateIPAddress	38
B.3.2	storeNodeName	38
B.4	Group Handling	39

B.4.1	createGroup	39
B.4.2	addGroupMember	39
B.4.3	removeGroupMembers	40
B.4.4	sendGroupKey	41
B.4.5	handleNewGroupKey	42
B.4.6	handleNewNodes	42
B.4.7	joinGroup	43
B.4.8	leaveGroup	43
B.4.9	composeMessage	44
B.4.10	handleJoinRequest	45
B.4.11	handleGroupLeaveRequest	46
B.4.12	handleGroupDestruction	46
B.4.13	handleGroupAvailabilityChange	47
B.4.14	handleReestablishedConectctions	47
B.4.15	handleGroupListRequest	48
B.4.16	handleGroupMembersListRequest	49
B.4.17	handleGroupPartition	50
B.4.18	handleNewMaster	50
B.4.19	transferMaster	51
B.4.20	handleMasterTransfer	52
B.4.21	joinGroups	53
B.4.22	handleJoinGroups	53
B.5	Key Handling	54
B.5.1	sendKey	54
B.5.2	receiveKey	55
B.5.3	removeKeys	55
B.6	Message Handling	56
B.6.1	sendMessage	56
B.6.2	recieveMessage	57
B.7	AODV	57
B.7.1	AODVavailabilityChange	57
B.7.2	AODVrequestRoute	58
B.7.3	AODVcreateGroup	58
B.7.4	AODVjoinGroup	59
B.7.5	AODVleaveGroup	59
B.8	Application Functions	59
B.8.1	appAcceptGroupDestroy	60
B.8.2	appAllowNodeToJoin	60
B.8.3	appBecomeNewMaster	60
B.8.4	appCheckLeaveAcceptance	61
B.8.5	appGroupChanged	61
B.8.6	appGroupDestroyed	61
B.8.7	appGroupListRequested	62
B.8.8	appHasTrustSetup	62
B.8.9	appIsMaster	62
B.8.10	appJoinGroups	63
B.8.11	appJoinThisGroup	63

Chapter 1

Introduction

1.1 Background

Today techniques more or less suitable for wireless ad-hoc networks[4] are starting to emerge in many forms. Good examples of this is Bluetooth¹, and IEEE802.11² implementations like WaveLAN³ and BreezeNet⁴. One of the most interesting services for ad-hoc networks are group communication. To be able to enter a meeting room and in a few minutes be part of a group session where everyone can communicate freely and efficiently.

Today there are lots of separate attempts to solve problems in the ad-hoc area (some problematic areas are security, limitation of power consumption and ease of use). Many of the problems are still not completely solved and those areas with working solutions has been produced very recently.

1.2 Problem Description

One of the down sides of ad-hoc networks is the problem that anyone can listen to what is being sent if they are able to intercept the traffic (which is quite easy due to the use of radio). This could be prevented by using cryptography to stop unauthorized parties from understanding what they hear if they eavesdrop.

There are security issues involved in wireless ad-hoc networks that aren't the same as in wired local networks. I think mainly of man-in-the-middle attacks[16] which are a lot easier to administer when using radio. This kind of attack is very troublesome when exchanging keys to build trust between different nodes that are going to communicate.

To make it easy to develop applications that wants secure group communication I will design an API that gives this service. The API should handle as much as possible of the problems that the ad-hoc environment gives the applications, with as little involvement as possible from the application. The goal is a transparent system, but with control over what happens (any limitations in control should be in the applications, not in the API).

1.3 Layout of The Report

The rest of this report is divided into the following chapters. In chapter 2 I present the state-of-the-art techniques in group communication over ad-hoc networks. In chapter 3 I present my solution to the problem described above and in chapter 4 I draw conclusions and present possible

¹Read more about Bluetooth at <http://www.bluetooth.com/>

²See <http://www.ieee.org/> for more info about 802.11.

³See <http://www.lucent.com/wirelessnet/products/solutions/wavelan.html> for more info.

⁴See <http://www.breezecom.com/Products/brznprd.htm> for more info.

future work. Appendix A contains a functional requirement specification for the solution and appendix B contains a implementation specification of the API that I have designed.

1.4 Scenarios

Here I will present three scenarios which all should be handled by the system.

1.4.1 General Characteristics

A basic need for security is obvious (it's the whole point of the system). All scenarios demand a basic level of secrecy and authentication.

To be able to find malicious nodes in the group, non repudiation is important. The reason for this is that if it isn't possible to identify who sent a message, a malicious node could send incorrect data to the group trying to fool the other nodes. Of course, this can happen even if the originator of a message can be identified. The difference is that if malicious data are found, the sender can easily be identified if all messages are signed in a way they can not be repudiated.

For each scenario below I will identify the characteristics of the scenario in three different important areas. These are *security demands*; how important is security in this scenario, *partitioning*; how likely is a partition of the network and finally what kind of *problems* is evident in this scenario.

1.4.2 Meeting Environment Scenarios

Assume a group of people, possibly without any previous connections, who sits down at a meeting table and wants to communicate using a secure and private group. The application used will be a shared white board or something with similar characteristics. The characteristics of this scenario also matches very well a computer game scenario. The main difference is the need for secrecy in the public part of the game. It is not so bad if a game gets compromised, compared to an important business meeting where industrial espionage can cost a company enormous amounts of money.

A common situation matching this scenario would probably be like this. People from two companies meet. One person is selected as a master and starts a group and the others join this group. During the meeting, some people may join and others may leave. This should not be any problem and it should not cause any disturbances except that the members should see that nodes come and go.

Security Demands Depends mainly on what kind of information that will be transmitted over the network. In general secrecy is most interesting but non repudiation could be important later on to be able to solve disputes over who said what.

Partitioning The risk of partitions are very low, but if they occur, the group will probably be severely crippled if the missing nodes can't be ignored (which is unlikely).

Problems there are no obvious problems in this scenario that doesn't hold for all scenarios.

1.4.3 Information Gathering Scenario

Assume a doctor on his rounds that stops at a patient's bed and ask for the vital stats from all the monitoring equipment together with the journal. Together with the doctor, a group of people from the staff are present and they should also be able to view the material.

Security Demands Security is very crucial here. Only a limited group of people should be able to receive data from the devices. This gives administrative problems because each device must know who to trust to be able to give authorized nodes access.

Partitioning The risk of partitions are low, unless the radio system that is used is very limited.⁵ To tighten the security even more, partitions should be counted as dangerous, and should probably not be handled automatically (the master must accept rejoins of partitions, at least if the partition has lasted more then a certain time).

Problems Since security is so important, the administration of users of the service will be complicated. The devices must get access information from some administrative node that is trusted.

Physical security is a serious problem, nodes can easily be stolen and this must not mean that the system security is broken.

A factor that makes it even harder to handle the security is that the nodes move around a lot, jumping from different rooms with short intervals.

1.4.4 Search & Rescue Scenario

Assume a group of people that are on a search and rescue missions in a disaster area. They want to know where everyone are located right now and want to know which geographical areas has been searched. This can be done using GPS equipment and with a thin client the runs a map application with this information presented.

Security Demands Security demands are moderate. The most important security issue is that no one should be able to insert false data into the system to fool the searchers to skip some areas. Of course it is also important that sensitive information are not spread to the wrong people.⁶ There is no need to try to keep the group secret.

Partitioning The risk of partitions are very high. Applications running in this scenario must be very good at updating nodes that reappear after a partitioning.

Problems When doing search and rescue, the nodes may be spread far apart, the number of nodes within range will be low (possibly only one if a node is in the end of the line). The transmission through the center node of a line will be very high. It's therefor very important that the master is in the center so all traffic to the master will go the shortest route possible for all nodes.

Because we use thin clients and probably during quite a long time, the battery consumption will be a big problem. The transmissions must be kept as low as possible. If nodes stop sending to limit the power consumption, nodes might be lost (even though they are within transmission range).

⁵In a hospital, this is unfortunatley quite likely due to sensitive electronic equipment that can't handle powerfull radio sources nearby.

⁶Examples of sensitive information are where dead or injured people has been found. This should not be leaked to thieves or reporters.

Chapter 2

State-of-the-Art

There are quite a lot of different techniques needed to make this system work. Here I will describe the techniques I use in the solution together with some motivations why I select this particular technique.

2.1 Ad-hoc On-Demand Distance Vector Routing

Ad-hoc On-Demand Distance Vector Routing (AODV)[6] is a routing protocol developed specifically for use in an Ad-hoc environment. The protocol is currently in draft version 4[11] at IETF and a lot of efforts are put into the further development of this protocol for example at Ericsson[13].

AODV works on top of the IP-layer and supports unicast, multicast and broadcast¹ traffic. This is important in my work because the group communication in the system should preferably be built on multicast and not broadcast (even though broadcast is needed for some operations).

AODV needs IP-addresses to work, so all nodes need an IP-address before it can get packets routed to it.

The functionality given by AODV matches the needs in group communication very well, which is the major reason I choose this algorithm. There are quite a few routing protocols that doesn't support multicast and are for that reason not interesting (for instance these [2, 9]). The protocol described in [18] does support multicast, but there hasn't been much work done on this protocol until just recently when a new version of the specification was released.² Another multicast routing protocol is [10] that uses a mesh-based instead of treebased. This protocol is a bit more developed than [18], but it doesn't support unicast directly which AODV does very well. Another reason to select AODV is that it is used a lot in research here at Ericsson, which is an advantage³.

2.1.1 Unicast

Here follows the tools available for the routing of unicast traffic.

Route Request

A route request is sent out when a node (A) needs a route to another node (B). This either happens if A never spoken with B before, or if A has lost the route to B. A route can be lost for several reasons. Movement of the nodes is one, timeouts in the routing tables is another.⁴

¹Only on the local network using the address 255.255.255.255.

²When it was much to late to change the used protocol, even if it would be motivated to choose that protocol instead out of functionality considerations. If it is motivated, I leave for further study.

³There are more people around to ask questions...

⁴Routes are only kept alive for a limited time to avoid overload in the routing tables.

Route Reply

A route reply is always sent back when a working route has been found via a route request. The reply is either sent by the requested node or by a node having a current route to it.

Route Error

Route errors are used to indicate that a node can not be reached any more. This message is sent back to the source by a node that realizes that the node listed as the next hop in the route is unavailable. A node can also forward a route error message if it gets it from a node that it maintains a active route to. The message is then forwarded to all nodes that is part of that route.

2.1.2 Multicast

Multicast in AODV is tree based and it is designed in the following way.

After deciding which multicast IP-address to use, one node initiates the group. This node will also become the group master. The group master is responsible for sending out group hello messages to the whole network, informing all the nodes about the existence and availability of the group. It also works as a base node in the current multicast tree.

Group Hello

Used to be able to know if all nodes in the multicast group are still available. The result is a list of nodes that aren't available. This message is sent out periodically by all nodes. Normal nodes only send the message to its neighbors to indicate that it is still available. The master of the group broadcasts its hello message to the whole network. This is used to help all nodes in the network to keep their multicast tables updated. This message won't be sent if any other broadcast has been sent since the last group hello.

Join Group

Route requests are also used to join a multicast group. A node sends a route request with the groups address as destination and sets a flag in the header of the message indicating that it wants to join the group. If the node knows who the master of the group is, it inserts this information in the header and unicasts the message to the node that is one hop closer to the master.

Multicast Activation

Used to configure which route is active. The node waits a defined amount of time for replies to a route request. When the deadline has passed a MACT message is sent out to indicate which of the routes it plans to use. The MACT message is sent to the node it wants to have as next hop in the multicast tree and this will then forward the message to the next node in the found route.

MACT can also be sent out by a node that wants to leave a multicast group or by a master that wants to hand over the masters role to some other node. This is done by setting the relevant flag in the message header and then sending it to the appropriate node(s).

Note that even if a node leaves the multicast group, it may not stop forwarding messages in the multicast tree if it isn't a leaf node. As long as the node stays in the network, it has to keep working as a router for the data sent through the network.

Leave Group

When a node wants the leave a multicast group, it sends a MACT message, with the *prune* flag set. This makes the node leave the multicast group. The node may still be part of the multicast tree, if it wasn't a leaf in the tree.

Group Partition Handling

When a group is partitioned a new group master must be appointed. The first node realising that the old master is unavailable, will also become the new multicast group master if it is part of the group. This node will start a route reconstruction and when this fails (due to the partitioning) it will instead make itself the master of the group. Only one node can do this at a time, if another node wants to send data via the broken route, it will have to wait for the reconstruction before doing anything more. A node that isn't part of the group can still initiate the reconstruction, but it will have to ask a node that is part of the group to become the new master. The group node that will be chosen to become master is the node that is part of the group and closest to the node doing the reconstruction and on the route that the node has to the groups master.

The groups can start reconnecting when a node in one of the groups reconnects with a node in the other group(s). The nodes find each other by receiving group hellos for the same group as it is part of but with a different group master.

When the two partitions reconnect, the master with the lowest IP-address will be chosen as the new master for the joined groups.

2.2 Addressing

The routing algorithm needs an IP-address for each node. An easy way to handle addresses is to use DHCP (Dynamic Host Configuration Protocol)[5]. Unfortunately DHCP is a bit unpractical for this scenario. You could insert a DHCP server at all key nodes⁵ but this results in problems with getting the DHCP-servers to cooperate. A more attractive solution would be to randomly select an address in a predefined address range in the private or link local address space.

2.2.1 Random Selection of IP address

A node should select an address that has a high chance of being unique. This could be done in several ways, but I propose the following approach based on the algorithm described in [17].

An address is selected by random from the list of available addresses in the link local address space (169.254/16). This address is then looked up using an ARP⁶ request. This is done to make sure that the address is available. This also tells all nodes on the available part of the network that this address is about to be taken, and in that way tell nodes trying to get an address not to take it for themselves.

The probability that an address is taken and thereby creating duplicated addresses on the network is quite low (with almost 2^{16} possible addresses, and with few used addresses the risk is extremely low as long as a good random generator is used).

In [17] DHCP is used when available. This isn't relevant in this scenario because there will definitely not be any DHCP servers available.

To make sure no duplicated addresses show up in the system, all nodes must listen for answers to ARP question for its own address. If any other node answers, claiming that it owns the address, actions must be taken to remove the duplicates.

2.2.2 Naming

The normal method for mapping names of nodes to addresses is to use a DNS⁷ service. In this system DNS is a bit of an overkill. Of course you could use it, but it would be a bit complicated to make sure it works correctly because there are no real server node in this scenario. Of course, a master node could work as a DNS-server, but it would make the system less dynamic. The master would still need a high level of caching of name to address mappings to lower the traffic to, and dependence of the master. If you still want to use DNS, a dynamic version of DNS is needed. The

⁵More exactly, all nodes that is powerful enough to support it

⁶Address Resolution Protocol.

⁷Domain Name System

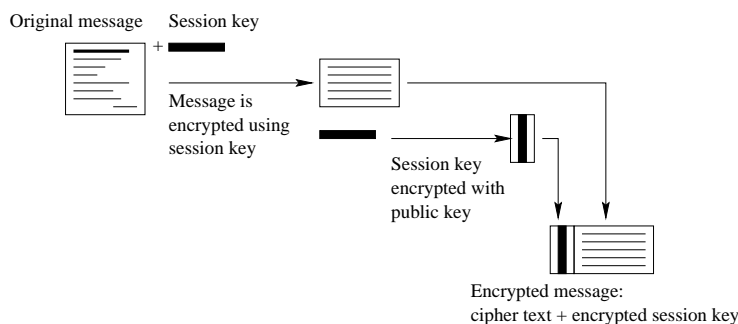


Figure 2.1: How PGP encryption works

reason is that since DNS normally is static, all information must be entered by an administrator. With dynamic DNS new information can be added and the records in the DNS database can be changed without direct interference from the administrator of the server. In [15] one version of how dynamic DNS can be implemented is suggested.

In this system with a limited amount of people, a naming scheme could easily use the name of the users together with the organization the user belongs to. For nodes not belonging to a user (usually some kind of device), some service lookup protocol would be better to use then naming. One protocol that could be used for this is SLP [8].

A system using the naming schema described above could work like this. Each node belongs to a organization and a user (or a category⁸). Each node then has a name that is unique to the user or category. In this way, each node has a unique name that is easy to understand and administrate. Each node is then responsible for detecting duplicates in the naming and also answer questions about what address this name corresponds to.

2.3 Security

In this system some means of achieving secrecy, non-repudability and integrity is needed. One well known technology is Pretty Good Privacy (PGP). It is using public key cryptography and achieves a high level of security with reasonable efficiency. An argument to choose this technique is that it is well tested and implementations are available in several forms.

2.3.1 PGP

Pretty Good Privacy [3] guarantees the secrecy of the data sent using PGP, keeping the efficiency needed for a commercial solution. Another feature of PGP is that it makes it possible to create digital signatures of messages, giving non-repudiation and integrity. The third major feature is that there are good support for certificates that can be used when deciding if a public key can be trusted. The following section about PGP is a summary of the information found in [12].

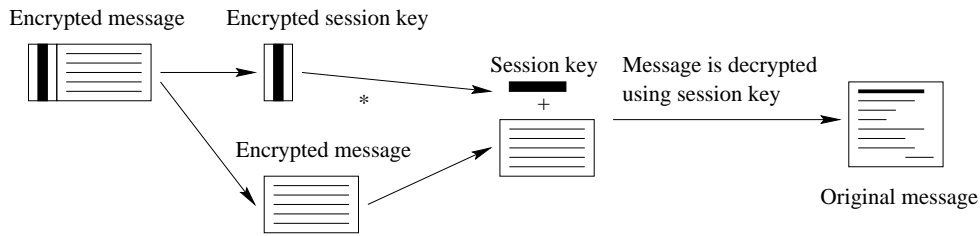
Basic Design of PGP

PGP uses both symmetric session keys and asymmetric public/private keys. Encryption using asymmetric keys are very expensive compared to using symmetric keys (a factor 1000/1). That is why both methods are combined in PGP.

In figure 2.1 and 2.2 the steps used in the encryption and decryption are described. In words, the algorithm works like this.

Encode

⁸One category could be EKG-machines.



* The recipients private key is used to decrypt the session key.

Figure 2.2: How PGP decryption works

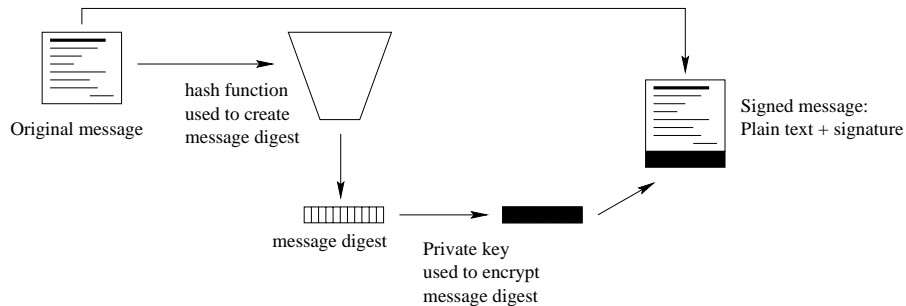


Figure 2.3: Secure digital signatures

1. A session key is generated of the decided size (maybe 1024 bits).
2. The plain text message is encrypted using the session key.
3. The session key is encrypted using the public key of the receiver and the private key of the user.
4. The encrypted key and the encrypted message are joined into one package and then sent to the receiver.

Decode

1. The message is split into two parts, the encrypted session key and the encrypted message.
2. The session key is decrypted using the private key and the sender's public key.
3. The message is decrypted using the session key into the original plain text message.

Digital Signatures

PGP gives the ability to create digital signatures using public/private keys.

Digital signatures use the fact that if a message is encrypted using a user's private key, it can be decrypted using the public key. If you trust the public key, you know that the message originated from the alleged source.

As said above, encrypting the whole message using asymmetric keys is very expensive. To avoid this, a cryptographically strong hash function is used to create a fixed size digest of the message that then is encrypted and sent together with the whole message. When decrypted, the hash is then used to check if the message received is the same as the message that was sent. Take a look at figure 2.3 for an illustration of how a message is signed.

Certificates

To be sure that a public key belongs to the same source as it says it does, *digital certificates* (*certs*) can be used. A cert is message containing signed data from a node that (if you trust it) guarantees that the public key from one source actually belongs to it and not to someone pretending to be that source.

A digital certificate contains three things:

- A public key.
- Certificate information (identity information about the owner of the key).
- One or more digital signatures.

The certificates guarantee that the public key and the identity certificate information are joined. It is up to the user to trust if the certificate is genuine in its whole.

Certificates can be distributed through *Certificate servers* that is storage-only repositories or in systems called PKI (Public Key Infrastructures).

In a certificate (**cert**) server, keys can be stored and retrieved by users when needed.

In a PKI additional features are added to the functionality in a cert server. These features are usually the ability to introduce, revoke, store, retrieve, and trust certificates. The main feature of PKI is the *Certification Authority* or *CA* which is a human entity (i.e a person, company, group, department or other association). This CA is authorized by a organisation to issue certificates to its users.

The CA creates a certificate and signs it using its own private key. This makes it possible for any user that wants to verify a public key to use the CA's public key to check the validity and identity of a public key for some other user.

Trust

You trust people to validate certificates. If you can't check a certificate personally, you must trust someone else to validate it.

The one you trust to validate the certificate is the CA. This means that the CA should do the manual check of validity for each client it serves. This works ok as long as there isn't too many clients. When the number of clients grow, the work of validating the certificates has to be distributed to more people. This is done by creating *trusted introducers* that has same privileges as a CA (or as it can be called *root meta-introducer*), except that they lack the ability to create other trusted introducers.

Trust Models

There are three different ways of setting up trust.

Direct trust This is very simple, you only trust keys that has been handed to you personally.

Hierarchical trust A root certificate server are trusted by everyone. The trust is then distributed by using trusted introducers that has the privilege (from the root server) to hand out certificates to other nodes. This builds a hierarchical system for handing out trust.

Web of trust this model encompasses both previous models. Any node can be a trusted introducer but it is up to you if you trust the introducers opinion or not. One important factor when deciding if to trust a certificate is how many nodes trusts a certificate. Another way to increase the certainty that a certificate is genuine is to seek as short chains as possible to trusted meta-introducer.

In PGP there are also different levels of trust.

- Complete trust
- Marginal trust
- No trust

This trust level indicates how much you trust a particular node. In a web of trust, you might not trust keys completely if the trust has passed through several steps.

Certificate Revocation

Usually it isn't motivated to trust a certificate forever. That is why most certificates has time constraints on its validity.

Sometime a certificate gets invalid before the time limit has passed. A introducer of a certificate can invalidate it by sending out a revocation message. This means that the introducer doesn't trust that the public key is connected to the identity information included in the certificate.

The revocation of certificates should be transmitted to the nodes in the group in some way. Normally this is done by publishing a Certificate Revocation List (CRL) at the CA that all users check for at regular intervals. The CRL could also be sent out to all available nodes in the network to minimize the risk of losing the revocation information due to a malfunction in the network.

2.4 Group Handling in Ad Hoc Networks

On top of the multicast group, a secure group built on public keys must be created. To setup the secure group, the nodes must be able to trust each other, how this could be done is described below.

2.4.1 Trust

A secure group is built on trust between all the nodes that want to communicate. Computers can't trust each other, it must be the users who trust each other. To transfer the trust to the computer level, keys must be exchanged under the supervision of users.

To make a system like this work, the users must trust each other enough to also trust the users the other users trust. This can easily digress to mean that you trust all people on the earth.⁹

2.4.2 Setting Up Trust

In PGP you can either use a server that you trust to get public keys for other nodes. Or you can build up a web of trust by exchanging keys with nodes you trust. A system that uses a web of trust is described in [7]. This system uses a web of trust to create an environment where a group of nodes eventually trust each other and then can work as a secure group. Methods for building the trust relations and handle changes in the group are also described. Here follows a summary of the ideas of the paper.

Creating a Trust Group

When you look at figure 2.4 you see a group of nodes. The nodes contained within the dashed lines trust each other already. This trust has been created earlier by exchanging keys that are signed by the nodes.

1. To start with, all nodes in the network broadcasts a list of the nodes they trust (including their public keys, signed by this node).
2. All nodes gather the broadcasted lists, storing them.

⁹ A theory says that you can find a trust link to any person on this earth by only going through six trust relations.

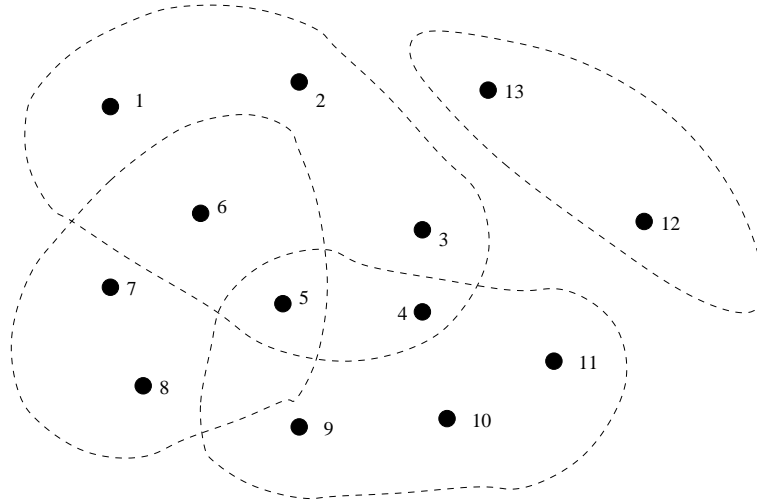


Figure 2.4: The initial trust relations

3. Each node creates a list of trusted nodes.
4. A node gets the role as master which all nodes gets informed about. (A good node to choose as master would be node **5** in figure 2.4, since node 5 already trusts most other nodes in the network.)
5. All nodes sends their list of trusted nodes including their public keys to the master.
6. The master uses the information gathered in **5** to build a trust map identifying which trust islands exist (groups of nodes that trust each other, but doesn't trust the main group that the master belongs to).
7. For each trust island (except the main trust island), the master asks one node in the island to manually create a trust relation with the master. If some nodes (12 and 13) can't do the manual trust exchange with the master, they won't be able to be part of the trusted group.
8. The master makes a two lists of the nodes the nodes that has setup trust (i.e all nodes except those in D). One list (T) contains the nodes the master has direct trust with and the rest of the nodes in the other list (N). The master then sends the keys to the nodes in N to all nodes in T.
9. The following check is performed in each node that receives the list sent by the master in **8**. For each node Y in N, check if a trust relation exists with that node. If yes, sign the set of keys found in the list received from the master and then transfer them to the trusted node. The trusted node will then return a list of nodes it trusts. Sign these keys and then forward them to the master.
10. Remove Y from N and forward N to the next node in T. When no nodes is left in N or when the last node in T has been reached. A message is sent to the master informing it that the trust setup are done.
11. When the message in **10** arrives at the master, the master sends out a complete list of all trusted nodes to all other nodes in the group of trusted nodes.

When this algorithm has been executed, a complete trust will be setup with all nodes that are welcome in the group and were able to transmit its keys to the other nodes.

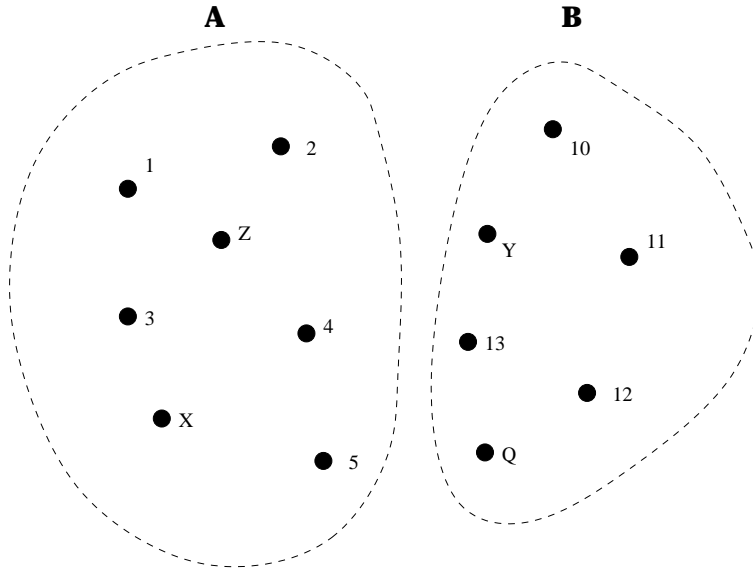


Figure 2.5: Two trust groups that should be joined.

Merging Trust Groups

Here follows a description of how the two trust groups described in figure 2.5 can be merged.

1. An arbitrary node X in group A connects to any node Y in group B. Y sends all the trusted keys in its group to X.
2. X checks if it trusts any node in the list received from Y. If any trusted node is found, X signs all its own trusted keys in A and the nodes it trusts in B and sends this information to B, jump to [7].
3. If no node in B is trusted by X, X creates a distribution list for all nodes in A where all the keys from B and the group address to B is included.
4. A node that receives the list of keys check if it trusts any of them. If yes do the same as X in [2]. Jump to [7].
5. If no trust was available, the next node in the distribution list gets the keys and it repeats the actions in [4]. This is done as long as there are nodes left in the distribution list.
6. If no trust can be found, X gets a message (from the last in the distribution list) asking it to manually create a trust relation with Y. Then jump back to [2].
7. Y receives the message sent by some node Z in A. Y identifies which node (Q) in B that Z trusts and forwards the message to Q.
8. The node Z receives a signed message which it checks for validity. If it trusts the signature, it signs the public keys of A and then multicasts them to all nodes in B. Then Z signs all the public keys of B and sends these to Q.
9. When X receives a message with signed keys it checks if it accepts the signature on the message. If it is ok, it multicasts the signed message to all nodes in A.

Adding Single Nodes to Trust Groups

Finally, here are a way of adding a single node to the trust group.

1. A node wants to join a trust group. It broadcasts its address and public key to all nodes in the local network.
2. If any node in the group trusts the new node, it signs the key and rebroadcasts the key. Then it unicasts all keys from the group to the new node.
3. If no answer arrives to the new node within a timeout, it has to manually create a trust relation with some node in the group. When the trust is setup, the new node gets all public keys from the other node. The other node sends out the new node

2.4.3 Announcement of Groups

In AODV all multicast group masters broadcasts periodical `group hello` messages. In this message the group master and the multicast address being used are included. This information are recorded at all nodes that hears the message. The lack of group hellos after a timeout will indicate a partition in the network. The information broadcasted about the groups can later be used when joining the groups and to find out if a partitioned group has reconnected.

2.4.4 Key Exchange

The public keys of the nodes need to be transfered in a secure way. There are several possible ways of doing this. The basic demand in all solutions is that no man-in-the-middle attack should be possible.

I have found two suggestions on how to transfer public keys. The first uses optical reading of a public key and the second use transmission via a physical system.

Optical Transmission

One user friendly way of transmitting public keys are the use of a optical reader [1]. This can then be used to read a public key printed in the barcode format.¹⁰ Possible ways to present the key is then to use a buisness card or a printed slip. The whole key isn't printed out, instead a shorter unique fingerprint is used to identify the whole key that is transmitted over the network. You must know for sure that the printed card or slip is genuine, but that is easier to accomplish in the physical world then in the digital one.

Physical Transmission

The main difference from the optical version here is that the key never needs to be viewable in any way. All transmission is through some electronic media. One way is to have physical contact between the two nodes [14] (via a cabel¹¹ or some similar device). You could also use infrared (IR), but then you must screen the transmission so it isn't possible to fool the system from outside.

¹⁰Today you usually find barcodes on grossery, but they can be used to print any kind of text or numbers.

¹¹Note that this cable should only be connecting the two nodes in question, a ethernet doesn't work in this case

Chapter 3

Solution

3.1 Assumptions

I assume the following in my solution.

- The system only needs to handle a small network (< 50 nodes, usually less than that).
- I use AODV (see 2.1) for routing and I take advantage of the multihop capabilities of AODV.
- There will only be communication between local nodes, no direct internet connections.¹
- Because the system uses multicast and broadcast over a error prone media, UDP with transmission security must be implemented to some extent.

3.2 Node Setup

When a node enters the zone of an ad-hoc network and wants to start communicating over the network, it needs an IP-address. This IP-address is choosen randomly, so a check has to be performed to see that the address is not already reserved by some other node. When an address has been located, the node also checks if the configured name is free to be used. If the name is occupied, the user has to choose a new name that is free. When all this is done and works, the real communication can start.

3.3 Trust Setup

To be able to communicate securely with the rest of the group, trust has to be set up between the nodes. My solution uses the algorithm in section 2.4.2. A less detailed version of the algorithm used are decribed bellow. It is only the detailed description of how the nodes are found and joined into the group that isn't included.

3.3.1 Building the Initial Trust Group

1. When a master has been appointed, all nodes send out a list of the other nodes it trusts.
2. The master gathers these lists and builds a trust tree.
3. The first time all nodes will be alone in their islands.

¹This would be easy to add, one node in the network will have to work as a tunneling proxy out to the Internet.

4. In each trust island, one node is chosen randomly by the master and a message is sent unencrypted to it, asking it to build trust with some node that is part of another trust group.
5. The nodes create the trust relation by transferring the nodes public keys over a secure media.
6. When the public keys are transferred, the keys belonging to other trusted nodes can be spread through the previously separate groups.
7. When all groups has connected in some way (direct or indirect), then all nodes in the network are trusted if they were able to setup trust with any of the trusted groups.

3.3.2 Adding New Nodes and Joining Trust Groups

Joining two groups or adding a new node doesn't differ much from building the initial trust group. One node from each group will have to exchange the trust so all nodes in the two groups will get the information. This only have to be done if no trust is available between any nodes in the two groups. Normally all other nodes in a group trusts all nodes the rest of the group trusts, so any particular actions shouldn't be needed to find this out.

3.4 Group Setup

When building the trust, a master is appointed. After the trust has been setup, the master can create a secure group. Before it creates the secure group, it creates a multicast group that will be the base for the communication inside the group. On top of the multicast group it then creates a secure group. The master (or in some cases, some other node) then adds the rest of the nodes that should be part of the communication. It is on this secure group the operations described below will be performed.

3.4.1 Public, Private and Protected Groups

There are three kinds of groups, private, public and protected. The name and participants of public groups is announced to anyone who asks any member of the group for this information. Questions about private groups are only answered when nodes that are listed as invited by the master are asking (if a node belongs to this category or not is checked by looking at the signature of the message).

Nodes can be added to public groups by any member of the group. Private and protected groups can² only be administrated by the master of the group. The main difference of private and protected groups are that a protected group is publicly announced, which a private group isn't.

Limitations to Private Groups

There are limitations to how private a group can be. It is impossible to hide the group completely. Here follows a description of what kind of limitations there are and to what level the group actually is private.

- The routing protocol will send out its group hello messages that will notify any node on the network that there is a group with that multicast address available. This makes it a lot easier to find a private groups multicast address.
- All members of a private group that sends out any messages to the group can be identified as a member of the group by anyone that listens to the traffic in the group and is part of the multicast tree. Unfortunately, it is impossible to stop a node from being part of the tree, if it wants to and is placed in such a way that traffic gets routed through it.

²This can only work as a policy, it is in reality impossible to stop a node from giving out the group key to another node.

- The name of the group and the messages sent to the members of the group are secret.
- Theoretically, a private group would be able to hide behind the same address as a public group (sharing the master node). This would make it harder to notice that some traffic isn't to the same group as the publicly known. This only works for hiding the private group from nodes that outside of the public group.

3.4.2 Group Master Privileges

A group master node has some privileges. It may add nodes into or answer questions about private or protected groups. It is the only node that may remove nodes, and it is the only node that can update the group key.

3.5 The Nodes

In each node some information must be stored to be able to communicate within the groups it is a member of.

Info About Nodes

The following information is stored locally for all nodes and groups.

Name	Description
IP-Address	for routing and identification
Public Key	used when communicating via unicast and checking signatures.
Name	for identification by the users.

The same information about the current node could also be stored here for convenience.

Info About Groups

The nodes must store the following information about all groups that it is part of.

Name	Description
Group address	The multicast address to the group.
Group Type	Public, private or protected.
Group name	the unique name of the group.
Group master address	For administration purposes.
Group member addresses	To keep track of which nodes are part of the group at this time. Only needed for presentation and unicast transmissions.

3.6 Messages

All communication between nodes in the system is sent as messages. These messages can either be sent via broad-, multi- or unicast. All multi and unicast messages are encrypted and signed. No broadcast messages are encrypted but most are signed. Those that aren't signed are address lookups that has to be done before a trust relation has setup up to the sending node. As soon as that node has got an address, trust can be exchanged and the following messages can be signed.

3.6.1 Broadcast

Broadcast is used to send out questions to all available nodes, but also for contacting the closest nodes (one hop away) and sometime also in an incremental way (by extending the range in hops for each retransmission). Broadcast is also used in the routing layer to inform nodes about the availability of other nodes. Normally these broadcasts are one hop broadcasts sent by the AODV layer, but broadcasts from the master is different. These broadcast messages are network wide and are used to keep the group joined and to find out if any partitions has occurred. Any partitions will lead to a broken route that can't be recreated and the master will be informed of which nodes aren't available. Nodes that aren't part of the group will also hear these broadcasts and can then contact the master if they want to join the group.

Broadcast messages are sent using UDP and need acknowledgments to be able to guarantee integrity of the communication. The same holds for multicast that is described next.

3.6.2 Multicast

Multicast messages are used for sending data to the whole group. Control messages that use multicast are messages for addition of nodes and destruction of a group.

3.6.3 Unicast

Unicast is used for private discussions between nodes. An example of a private discussion that isn't pure data traffic is the transmission of a new group key.

Unicast messages are transmitted using TCP, guaranteeing the integrity of the communication.

3.7 Group Operations

As said earlier, a master node has some more privileges than an ordinary node. Here are the operations only a master node can execute (with one exception).

3.7.1 Add Node

Adding a node to a group is easy. The only thing needed to be done is to give the new node the group key and then inform all other nodes that a new node has joined the group. For a public group, this can be done by any node that is part of the group (this is the exception mentioned above). For private or protected groups, only the master is allowed to give out the group key to a new node. The other nodes can of course not be stopped from doing this anyway, but if a node that is part of the group is found out doing that, it will quite certainly be punished for it. Unfortunately it is very hard to identify who gave out the key, so it will be hard to stop it from repeating.

3.7.2 Remove Node

To remove a node, a new group key must be distributed to all the members of the group that is going to stay in the group. This is done by the master by sending out a new group key via unicast to each node that may stay in the group. The reason why only the master may remove nodes is that it will be best to have only one node that may carry out punishment (kicking out nodes) and also take care of the administrative work of sending out new groups keys. It would be too easy to harm the groups communication if any node would be able to divide the group into different parts. In public groups, the most important feature isn't high secrecy, instead transmission integrity counts higher. Under those premises it isn't motivated to allow any node to be able to destroy the secure group by removing nodes from the group that shouldn't be removed. In private and protected groups it's natural that only the master may remove nodes (it matches the add case).

One way to limit the risk of having the group destroyed by a malicious node that takes over the master role is to use voting when deciding who should be kicked out of the group. It is much

less likely that three or more nodes has been taken over. This is very suitable in a public group, and could be suitable in larger private groups. In small private groups it is not very likely that a bad node can come into the group.³

3.7.3 Handling of Group Partitioning

When a partition occurs the system realizes this and informs the application of the current status.⁴ It is then up to the application to handle the situation with missing nodes until the partitions are joined again. Two main strategies can be identified for applications, either freeze the communication. This is what is applicable when all nodes are important in a realtime application.

The other strategy is to ignore the lost nodes, and maybe store the data transmitted until the nodes return. They can then get the data in one single transmission. This is applicable in applications where realtime isn't of much interest.

3.7.4 Joining Two Groups

Joining two partitions should be quite trivial as long as the separate groups hasn't changed its group key. Then its only to choose one node to be master of all the nodes again. If any of the partitions has changed, they should be treated as two separate groups and joined in that way.

To join two partitions that has changed or two completely different groups, the applications and users will need to be involved. Joining the groups means that the nodes in one group must join another group and then the original group will be destroyed. This is done like this: the master of the group that should be joined into another first asks the other master if a join is accepted. If it is, it then sends a message to all members of its group telling them to join the new group. It is then up to the different nodes to decide if they want to join the group or not. This choice can be automatic or demand user intervention. Which one is up to the application implementor.

3.7.5 Destroy Groups

Destroying groups is quite easy. The master sends out a group destroy message and when all nodes has accepted it or when a timeout has occurred, the master removes all data about the group. A problem occurs if a node want to keep the group going. It can then choose to create a new group, or maybe take over the master role in the current group. How this is solved is left to the application, all basic operations needed to handle both scenarios are available.

3.8 Node Operations

The previous operations can be done by masters (and in the add case also by other nodes if the group is public). The following operations are executed by any node by sending messages to a master via uni- or broadcast.

Operation	Description
Join Group	Sends a request asking to be allowed to join a group.
Leave Group	Sends a request asking the master to remove this node from the group.
List Available Groups	Broadcasts a request for available groups. All available masters will answer with its public and protected groups. Private groups to which the node is invited will also be listed.
List Group Members	A list of all members of a group can be requested with this operation, if the master of the group allows this node to get that information.

³At least if the users of the group is well known.

⁴More exactly, the routing system notices a breach of contact with some nodes that can't be restored.

3.9 Scenario Handling

One of the goals in this solution has been to handle the scenarios described in section 1.4. Here I will describe how well the problems described in the scenarios are handled. I also include a short description on how the system will be used in the scenarios.

3.9.1 Group Meeting

This is the best handled scenario of the three. All participants are close to each other, they are willing to set up trust between each other.⁵ I did not see any particular problems for this scenario, and I haven't found any problem when I designed the solution either.

Usage Description

How will the system be used at a meeting? Hopefully all nodes belonging to a particular company will already trust each other. If that is the case, setting up the trust will be easy. The two (or more) participating groups of nodes only have to join via the masters, which does all the transmission of keys.⁶

One node must of course become master. Who is not very important, only that all participants accept its authority. As soon as everyone has received the group key, the communication can start. Removing and adding of nodes will be handled by the master during the meeting.

If nodes lose contact, it is up to the master to decide if the communication should continue in the group.

When the meeting is finished, the group will be destroyed, and the nodes may choose to remove the publickeys belonging to the nodes in the other company, if they don't trust them outside the meeting. They will not have much use for it anyway.

3.9.2 Information Gathering

This scenario is handled pretty well by the solution. The major problem is to handle who will have access to the group of devices. This can either be handled by a central administration that always have access to the devices and has authority to decide who will be trusted and who won't.

Another way to handle the access problem would be to configure the devices to always allow a few nodes to have constant access. Instead of configuring the devices you configure these nodes to only allow a few people to use them. This could be done with smart cards or some other security solution that limits the access to only known and trusted persons. This means that the applications in these nodes must be configured to handle these increased security demands.

Usage Description

All devices in a room or at a bed will be a group. All participants of the rounds will be a group. Depending on required level of security, the groups can either be joined or not. If the groups are not joined one node needs to work as a bridge between the groups transmitting only the necessary information. The owner of this group would normally be the doctor. If the groups are joined, only one (or maybe a few) nodes will be trusted by the stationary group. These nodes are the ones that will have to create the trust bridge between the mobile and the stationary nodes. The trust of the mobile nodes should be very limited in time. When the group is removed, no trust should be stored for future use, in order to avoid any problems with non authorized access to the equipment.

⁵At least enough to be able to communicate within the group, that doesn't mean they truly *trust* each other.

⁶If no trust has been set up yet, it will be more work to do, but that won't be too difficult.

3.9.3 Search & Rescue

As mentioned in the scenario description, battery consumption and high load on central (of the search line) nodes is a huge problems. This system doesn't solve those problems to any greater extent.

One configuration that would help in this scenario would be to have a master with stronger transmission strength then the other nodes and preferably with a stronger energy source than batteries. This would make the group more stable because every node would have at least one node they can communicate with. This is unfortunately not handled in AODV so it won't work with the current solution. Even if it would be handled in AODV, this would go against the idea behind ad-hoc groups, because you would get a central node that stands for the communication.

Usage Description

Before the search starts, all participants meet and exchange trust information. The group should be created at the same time, a protected group is probably most suitable here. When the group spreads out, the master should stay as centered as possible. The reason for this is that the master will get more requests then the other nodes, if the master would be at the end of a line, the nodes on the masters end of the line will get much higher load then the nodes on the other end of the line.

As long as the line isn't cut off, the groups should be fully functional. As a safety measure, at least two nodes should be in range in both directions. This will lower the risk of partitioning because we will have at least two routes to the next node in the line.

During the search, any node may leave or join by finding one node in the search party and get the trust information from it. The master can then add, and of course remove the node, if it doesn't have any objections.

During partitions, the nodes in a partition should continue the work and the application should be able to handle this. When the partitions join, the changes should be transferred over to the other partition(s) so they get updated. If the new master of the joined group isn't a central node (i.e the former master), the master privileges should be transferred over to this new node to keep the preferred organisation.

3.10 Communication

All the data sent via the ad-hoc network should follow a certain protocol.

Here follows a description of the message format for the protocol.

Name	Datatype	Description
Datasize	integer	Size of the data section of the message.
Encrypted	1 bit	Set if packet is encrypted.
Type	byte	Type of message, see separate table for available values.
Data	binary data	The data that is being transported.
Sequencenumber	int32	Sequence number for messages. Used to order messages ⁷
Acknowledge	int32	Acknowledge that all data up to this sequencenumber has been recieved.
Sign	PGPsigSize	PGP signature of the whole message.

3.10.1 Message Transportation

In unicast traffic, TCP can be used. In this case the communication is easy. The first message sent over the connection includes the session key and then the rest of the communication will be encrypted using that key.

In multicast and broadcast traffic some steps must be taken to ensure that all data ends up at the destination.

Chapter 4

Conclusions & Future Work

This work was done under the following assumptions.

- a small network (< 50 nodes, usually less than that)
- use AODV for routing and take advantage of the multihop capabilities of AODV.
- There would only be communication between local nodes, no direct internet connections.
- Because the system uses multicast and broadcast over a error prone media, UDP with transmission integrity must be implemented to some extent.

The work has resulted in the solution described in chapter 3 and both a functional specification and a implementation specification of an API which is described in appendix B. This system should, with some further work in the implementation details, be possible to create today. There are still a lot of work to make the system give the quality standards needed for a commercial system, but in most areas the techniques are as evolved enough to cope with the demands. Exactly what needs to be looked closer at is described below under future work.

What you get if you implement this system today is the following. You can communicate securely and assume that the nodes you assume you talk to are the only nodes that will understand what you are saying. This holds under the assumption that no node that is part of the group is *evil* and routes the data out of the group. You will know who said what in the communication. As long as the application supports this, you can assume that no node will miss any data due to network partitioning, if it returns from a partition within reasonable timelimits.¹

4.1 Future Work

There are of course a lot to do from this point. First of all some kind of real implementation must be done to see how well it actually works. My prediction is that it will work acceptably well in terms of functionality. The efficiency and cost of the system is more uncertain.

There are also some areas that need further study. A particular area is security where there are still more work to do in some areas.

4.1.1 Implementations

A LAN-based demo version would be a good start. No multicast implementation of AODV is available openly today. An implementation of AODV must be performed first before a real system can be created.

¹These limits are defined by the resources available and the implementors of the application.

4.1.2 Effects on Physical Devices

Many of the techniques used in this solution puts quite high demands on the devices running the system. Especially, the amount of traffic between the nodes will probably need to be lowered. This might mean that a different routing protocol must be used. Further study of the effects on battery consumption due to a high level of transmissions and high usage of CPU-cycles is needed. This can only be done after a real implementation has been performed.

If a different routing protocol should be chosen some changes might have to be done to the solution. It would probably mean some work to compensate for the loss of some features the AODV gives. I think mainly of its ability to keep track of which nodes are available or not in the multicast group which the group hello messages is useful.

4.1.3 Security

Some security issues are handled in the proposed solution, some other are not. I have concentrated on finding working solutions for man-in-the-middle attacks and basic security like integrity, non-repudiation, confidentiality and secrecy of data. What this solutions doesn't handle is availability, physical security [19] and secrecy of group membership². It would be quite easy to shut down the communication by disturbing the communication on several layers. Especially sensitive is the routing layer which can't handle false information very well.

The membership secrecy problem that I mentioned above are the lack of possibilities to hide a group totally from nodes that can listen to the traffic. This is normally not a big problem, the data is far more sensitive then the information about the participants of the group. The worst problem with this information is that it can be used to disrupt the whole communication in the group in a denial of service attack. Nodes that isn't so active or nodes that are (physically) far from the rest of the nodes in the group are much more sensitive to attack from someone that want to take over the physical device.

Nodes beeing stolen is a serious problem in some of the scenarios, especially in the medical scenario where a device easily could be stolen and then be used to access private information about patients. The reason this is possible is that it will probably hard to keep the security around the actual devices. Passwords and/or access cards should lower the risks of theft, but I see it as very hard to avoid totally.³

A similar problem as in the medical scenario exists in the search & rescue scenario where the users are spread out and hard to keep in full control. Normally, problems are not that likely, not many want to interfere in the work to rescue people (if it doesn't belong to the military or the law enforcement).

How these problems should be solved are difficult questions, and it is definitely needed to do further work on the security issues. Some ideas on how to handle compromised nodes are discussed in [19]. It is probably possible to implement some of the ideas, for instance the demand that more then one node can guarantee that one particular action is needed. This approach should lower the risk of having compromised nodes that run havok in the group.

4.1.4 The Protocol

The communication protocol needs to be refined so it is more slim. Good protocols must be designed by someone with experience of designing protocols, especially as it should give transmission security over a unreliable media like radio.⁴

²You can't hide who is part of a group from someone that can listen to the radio transmissions under most circumstances.

³This is assume because the medical profession is very stressed, and a lot of people are involved. It is also enough with one compromised node to put a hole into the security.

⁴Of course, most of the errors will be handled by the lower layers, but there is still a big difference from a Ethernet LAN or a backbone.

Bibliography

- [1] ANDERSSON, F., AND KARLSSON, M. Secure jini services in ad hoc networks. Master's thesis, Department of Teleinformatics, Royal Institute of Technology (KTH), 2000.
- [2] BROCH, J., JOHNSON, D. B., AND MALTZ, D. A. The dynamic source routing protocol for mobil ad hoc networks. IETF Internet draft, draft-ietf-manet-dsr-03.txt, October 1999. Expires 22 April 2000.
- [3] CALLAS, J., DONNERHACKE, L., FINNEY, H., AND THAYER, R. RFC 2440: OpenPGP message format, Nov. 1998.
- [4] CORSON, S., AND MACKER, J. Mobile ad hoc networking (manet): Routing protocol performance issues and evaluation considerations. IETF RFC 2501, <ftp://ftp.isi.edu/in-notes/rfc2501.txt>, January 1999.
- [5] DROMS, R. RFC 2131: Dynamic host configuration protocol, Mar. 1997.
- [6] E.PERKINS, C., AND M.ROYER, E. Ad-hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications* (February 1999), pp. 90–100.
- [7] GEHRMANN, C. Securing ad hoc services, a jini view. Work in progress.
- [8] GUTTMAN, E., PERKINS, C., VEIZADES, J., AND DAY, M. RFC 2608: Service location protocol, version 2, June 1999.
- [9] HAAS, Z. J., AND PEARLMAN, M. R. The zone routing protocol (zrp) for ad hoc networks. IETF Internet draft, draft-ietf-manet-zone-zrp-02.txt, June 1999. Expires December 1999.
- [10] LEE, S.-J., SU, W., AND GERLA, M. On-demand multicast routing protocol (odmrp) for ad hoc networks. IETF Internet draft, draft-ietf-manet-odmrp-01.txt, June 1999. Expires December 1999.
- [11] M.ROYER, E., AND E.PERKINS, C. *Ad hoc On-Demand Distance Vector (AODV) Routing*. IETF Internet Draft, draft-ietf-manet-aodv-04.txt, October 1999. Expires 22 April 2000.
- [12] NETWORK ASSOCIATES, I. An introduction to cryptography, chapter 1. <http://www.pgpi.com/doc/pgpintro/>, 1999.
- [13] PURSER, K. Internet access through a multihop ad hoc network: An implementation study. Master's thesis, Department of Information Technology, Uppsala University, 1999.
- [14] STAJANO, F., AND ANDERSON, R. The resurrecting duckling: Security issues for ad-hoc wireless networks. Presented at 3rd AT&T Software Symposium, Middletown, NJ, USA, Sep 1999.
- [15] STAPP, M., AND REKHTER, Y. *Interaction between DHCP and DNS*. IETF Internet Draft, draft-ietf-dhc-dns-11.txt, October 1999. Expires April 2000.

- [16] TANENBAUM, A. S. *Computer Networks*, 3 ed. Prentice Hall, 1996, ch. 7 - Network Security, pp. 605–606.
- [17] TROLL, R. Automatically choosing an ip address in an ad-hoc ipv4 network. IETF Internet draft, draft-ietf-dhc-ipv4-autoconfig-04.txt, April 1999. Expires October 19, 1999.
- [18] WU, C., TAY, Y., AND TOH, C.-K. Ad hoc multicast routing protocol utilizing increasing id-numbers (amris). IETF Internet draft, draft-ietf-manet-amris-spec-00.txt, November 1998. Expires 24 May 1999.
- [19] ZHOU, L., AND HAAS, Z. Securing ad hoc networks. *IEEE Network* 13, 6 (November/December 1999).

Appendix A

Functional Requirement Specification

The API should be able provide the following functionality. This is a general overview of how the API will work. A more detailed description of how the functions should be designed can be found in the implementation specification found in appendix B.

A.1 Application Controlled Functions

These functions is called directly from the application when needed.

A.1.1 Key Exchange

Used by All nodes

Description Executes the exchange of the public keys between two nodes to build trust between the two secure groups these nodes belong to. The communication chanel to use must be given and shouldn't be an open media.

Arguments

→ Channel to send data over.

Output

The keys needed to communicate with the contacted node and all its trusted nodes.

Side effects

Trust will be set up between nodes that now are joined via the link created between the two nodes.

Demands

The transmission must be done with **no** possible way of doing a man-in-the-middle attack.

A.1.2 Address Handling

Used by All nodes

Description Each node needs a unique address and name, which is handled here.

Arguments

→ Suggested name for the node

Output

The unique address selected, or fail if the name couldn't be allocated.

Side effects

The address and the name are reserved for the node as much as that is possible under the circumstances.

Demands

That there are addresses available and that the name is free to be taken by this node.

A.1.3 Create Group

Used by A node that wants to become master for a new secure group.

Description Creates a secure group without any members, making the calling node the master of this group.

Arguments

→ A group name

Output

The selected group address if ok, or fail otherwise.

Side effects

Chooses a multicast address that seems to be vacant (noone answers on any questions sent to it using AODV:s RREQ¹). A group key is also generated to be used when communicating with any future members of the secure group.

Demands

That the node calling the function is strong enough² to be able to become a master.

A.1.4 Group Add

Used by All nodes part of the group

Description Adds one or more members to a secure group.

Arguments

→ Group name

→ Nodes to add

Output

A list of the nodes that actually was added

Side effects

All nodes added will be informed that they are added and when all new nodes has been added a message is sent out informing the whole group of the new member(s).

¹See 2.1.1 for details.

²In a physical/practical point of view.

A.1.5 Group Remove

Used by Master

Description Removes one or more nodes from the secure group by updating the group key. This group key is then only sent out to the nodes that should stay in the secure group (i.e the nodes not listed amount the *nodes to remove* bellow).

Arguments

- Group name
- Nodes to remove

Output

A list of the nodes that did get the new group key, and thereby are aware of the change.

Side effects

All removed nodes get a message that they are removed if there are an route to it. An acknowledgement to this message is expected, but not demanded. All nodes still in the secure group gets informed about the current member status for the group.

A.1.6 Send Message

Used by All nodes part of a secure group

Description Sends a message using uni-, multi- or broadcast.

Arguments

- Destination (uni-, multi-, or broadcast address, *or* node- or groupname)
- Message data
- Options

Output

Ok, if the message was recieved by the destination(s), list of failed nodes if not.

Side effects

The only reason a node won't recieve a message is a group partition.

Demands

That the system should try to rebuild broken routes and that a group partitioning can be handled gracefully.

A.1.7 Join Groups

Used by Master

Description Joins this members secure group into another secure group. Some nodes that is member of the old group might not want to be part of the new group.

Arguments

- Current group name
- Other group name
- The other master

Output

A description of the new group.

Side effects

All nodes part of the secure group that is inserted into the current one will choose if they want to join or not.

A.1.8 Become New Master

Used by All nodes

Description When the AODV system makes a node a master for the multicast group, this function takes care of making the node the master over the secure group also.

Arguments

→ Group name

Output

Ok, if the node is now master of the secure group, a failure message if not

Side effects

All nodes available are informed of the new master.

A.1.9 Send Group Join Request

Used by All nodes

Description Sends a request to the closest node that is part of a secure group, requesting to join the secure group. If it is a private group, send the message to the master.

Arguments

→ Group member

→ Groupname

Output

Ok if the node was accepted as member in the secure group.

Side effects

A encrypted message is sent to an appropriate node and if the node beeing asked accepts it, the node becomes a part of the secure group.

Demands

That a trust relation has been set up with the the groups members.

A.1.10 Send Group Leave Request

Used by All nodes part of a secure group except the current master

Description Sends a message to the master requesting to leave the secure group.

Arguments

→ None

Output

Ok, if the master accepts it.

Side effects

If the master accepts the leave request, all nodes in the secure group will get a new group key except this node.

Demands

That a unicast connection can be set up to the master.

A.1.11 List Group Members

Used by All nodes part of a group

Description Sends out a request to a master for a list of all members in the secure group and the identity information about these nodes.

Arguments

→ Groupname

Output

The list of members if the master accepts the request.

Side effects

The local member tables get updated.

Demands

That the node is part of the secure group it wants to get information about if the secure group is private or protected. With public groups this information is freely available.

A.1.12 Group Destroy

Used by Master

Description Informs all member nodes that the secure group they are part of will be destroyed.

Arguments

→ Group name

Output

A list of all nodes that recieved the message.

Side effects

All nodes will mark the secure group as being destroyed, and can't expect any further communication to get through.

A.2 Event Driven Functions

Alot of the activity in the system is event driven. For instance, when a message arrives or a part of the network gets partitioned a event occurs and must be handled. These functions also need to send up information to the application. This is solved by executing a function provided by the application programmer.

A.2.1 handle Availabilty Change

Used by All nodes part of a secure group

Description Takes care of updating the nodes data when the availabilty of other nodes change.

Arguments

→ A list of the unavailable nodes

Output

none

Side effects

If the application care, a loss of a node may result in a freeze of communication from the node.

Demands

That the node sending the message is part of at least one secure group.

Application function comments

Called with the list of actually available nodes in each secure group.

A.2.2 Recieve Message

Used by All nodes

Description Starts when a message has been recieved.

Arguments

→ The message in raw format

Output

The message is sent to an appropriate handler function.

Side effects

The message gets decrypted (if needed) and the appropriate counters/tables get updated.

Demands

That the message is readable by this node.

Application function comments

Gets the extracted message and the sender as argument.

A.2.3 Handle Grouplist Request

Used by Master

Description Creates a list of available secure groups to be sent to a node requesting this information. Public and protected secure groups will always be included, private groups is only listed when nodes that is invited to the secure group is asking.

Arguments

→ Node requesting group info

Output

A list of secure groups to be sent to the node.

Side effects

If the node requesting the information is known, the result is encrypted. If the node is known and is invited to a private group, this group is entered into the list.

Demands

That a unicast connection can be set up to the requesting node.

Application function comments

Gets the caller and which secure groups was delivered to it. Mainly for informational purposes.

A.2.4 Handle Master Change

Used by All nodes part of a secure group

Description Executed if a new master has appeared, only accepts if the application function says ok or if the old master guarantees that the change is legitimate. This will normally be the case if this is caused by a partition join.

Arguments

- The new masters address
- ID of the updated nodes
- The current group key
- Node information, if any changes has occurred.

Output

None

Side effects

If the application function returns true this node is joined into the new secure group and the membership of the old secure group is removed. If the result is false, the group won't join the new secure group and if no other nodes stay in the secure group it becomes alone in the secure group and without a master. (It will still use the same multicast secure group, so it won't become a master on that level.)

Application function comments

Gets the same data as the API-function. The result must be true (if join is accepted) or false (if join is not accepted.)

A.2.5 Handle Groupcomposition Change

Used by All nodes for add, master for remove

Description This function processes the add and remove message that can be sent to nodes in the group.

Arguments

- group name
- changed nodes

Output

None

Side effects

All relevant tables are updated.

Application function comments

Called with the current member list and all info about the members

Appendix B

Implementation Specification

This is a fairly detailed implementation specification for an API that gives the service *secure group communication*. An even more detailed specification will be needed to do the real implementation. The reason I don't write that kind of specification is that I don't want to lock the implementor into using a specific coding solution. The real implementation will be very dependant on any libraries for routing and encryption that will be available. It should be fairly simple to add those details for those that do the actual implementation.

B.1 Datastructures

Several datastructures will be needed to store and transport the data used to store the state of the system.

B.1.1 GroupInfo

In this structure information about a group should be stored.

Name	Type	Description
groupname	string	A name that is unique for this master and multicast address.
groupaddress	ip-address	The multicast address to the group
master	ip-address	Address to the master, not used if this node is the master.
groupmembers	list of nodeids	A list of the datastructures describing all the members of the group (except this node).
currentlyAvailable	list of nodeids	A list of those members that are currently available for communication

B.1.2 NodeID

Name	Type	Description
Organization	string	The organization this node belongs to. Must be unique ¹ in the network.
User	string	The name of the user having this node, if it isn't a user, a department or node type could be used here instead. Must be unique within the organization.
Node name	string	The name of the node. Must be unique for the user.
Address	IP-address	The address to the node.

¹Two organizations may not have the same name.

B.1.3 Message

A message is always signed by the sender. The data between the lines are encrypted if the **Encrypted** flag is set.

Name	Type	Description
Sign	PGPsigSize	PGP signature of the whole message.
Encrypted	1bit	Set if packet is encrypted.
Datasize	int	Size of the data section of the message.
Type	byte	Type of message, see separate table for available values.
Data	binary data	
Sequencenumber	int32	Sequence number for messages. Used to order messages ²
Acknowledge	int32	Acknowledge that all data up to this sequencenumber has been recieved.

Here are the possible message types listed.

Code	Expected datafields	Description
MSG_RESPONSE	Variable	Used when the message is a response to a request from another node.
MSG_ACK	none	Used when the message is a acknowledgement of a previous message.
MSG_FAIL_RESPONSE	Reason id, Reason text	Used when the message is a failure response. The data is always a reason why the request failed, both in integer and text form.
MSG_JOIN	Group name	A join request.
MSG_LEAVE	Group name	A leave request.
MSG_FINDNAME	Full node name (all parts)	Lookup a node name.
MSG_NEWGROUPKEY	Group name, new key	A message containing a new group key.
MSG_NEW_MEMBERS	Group name, member info	For announcing new members.
MSG_NEW_MASTER	Group name, [transfer certificate]	Message indicating that there is a new master for a group. A transfer certificate could be included that certifies that the new master is approved by the old master.
MSG_REMOVED	Group name	Recieved when beeing removed from group
MSG_GROUP_DESTROYED	Group name	Informing the node that the group has been destroyed.
MSG_GROUP_PARTITIONED	Group name	Sent by the new master of a group partition.
MSG_DESTROY_GROUP	Group name	Should group be destroyed
MSG_GROUP_LIST	Group name	List of groups request
MSG_GROUP_MEMBERS	Group name	Request for list of members in the group.
MSG_JOIN_NEW_GROUP	New and old group name, new master	Instructs a node to join a new group instead of a group that it currently is a member of.

B.2 Communication

A important part of the system is of course the communication. When doing unicast this is easy, but when doing multicast and broadcast, it gets more tricky. Unicast uses standard TCP, which gives all guarantees we need. When doing multicast and broadcast we must use UDP. UDP lacks the transmission security feature that TCP gives us so we must implement some method that gives at least limited integrity of transmissions.

B.2.1 Multi- and Broadcast

When sending out a multicast message, it isn't guaranteed that the message arrives at the destination. AODV reports broken routes, but that only tells us that the message possibly didn't arrive. Therefore a system with acknowledgments should be implemented. This is supported in the message format where the receiver of a message can respond to. The receiver should then make sure it sends back an ACK, either as part of a response, or if no response will be sent in time, a pure ACK package.

This isn't really a concern for the API, so another layer must be implemented between the API's send/receive functions and the IP layer which is routed by AODV.

I'm quiet uncertain of the best way of doing this part of the implementation. Feel free to change this if a better solution can be found.

B.2.2 createConnection

Description This creates a uni-, multi- or broadcast connection to a specified destination.

Arguments

→ **address** – The address to the destination.

Output

← A connection datastructure.

Algorithm

1. Create a socket and a connection with the OS systemcalls.
2. Make sure there is a route to the requested address (only applicable for multicast addresses.)
3. If there is, create a datastructure (of type **connection**) where the socket and some other data will be stored.
4. Return the datastructure.

Used Functions

socket – For creating a socket.

connect – For setting up a connection.

Used in

sendMessage – When setting up a connection.

Side effects

A route will be set up to the destination if not already available.

B.2.3 senddata

Description Used to send data as multi- or broadcast.

Arguments

- `connection` – A multi- or broadcast socket connection.
- `message` – The message

Output

- ← Ok, if message was delivered, error code if not.

Algorithm

1. Insert the sequence number and if needed an ACK.
2. Create signature of message and store it in the message structure.
3. Send the data in a UDP-message via the connection.
4. Start timer, waiting for response.
5. When timer is interrupted, check if ACK has arrived (`recvMessage` will fix this).
6. If interrupted due to a route error, call `AODVrouteRequest` to find a new route to the destination.
7. If `AODVrouteRequest` fails, return an error code, otherwise, go back to 3.
8. If interrupted for some other reason than route error, go back to 3.
9. If the timer is finished and the message isn't ACKed, resend (messages should be resent up to `MAX_SEND_RETRIES` times).
10. If timer is finished and the message is ACKed, return true.
11. If message has been resent `MAX_SEND_RETRIES`, return error code.

Used Functions

- `send` – The system call send.

Used in

- `sendMessage` – For sending a multicast message.

Side effects

- The message will (hopefully) be delivered to the destination.

B.2.4 recievedata

Description Recieves a UDP message and informs the send function what data has been ACKed.

Arguments

- `connection` – A multi- or broadcast connection.

Output

- ← The recieved message or a failure message.

Algorithm

1. Check the message queue for the connection, wait for some data to arrive.
2. If an error arrives update the connection information so `senddata` knows that the transmission failed.

3. Extract the sequence number and the ACK from the message and update the connection information.
4. Return the message.

Used Functions

recv – The system call `recv`.

Used in

recvmessage – When receiving a multi- or broadcast message.

Side effects

The send function will get information about the connection it sent out data on.

Comments

It is up to **recvmessage** to check if the message is correctly signed, **recievedata** doesn't have that capabilities.

B.2.5 listener

Description Used to catch multi or broadcast messages directed to the node. It's this function that informs **recvMessage** about the message that has arrived. This function will normally never return.

Arguments

→ **recvMessage** – A pointer to a function pointer. This function will be called when a message arrives.

Output

← Error code if setup failed.

Algorithm

1. Create a listening connection on the standard port.
2. Call **recv** to catch the messages to the node.
3. Call the **recievedata** function.
4. Go to 2.

Used Functions

recv – For receiving data.

Used in

recievedata – Indirectly

Side effects

For each message that arrives, the receive message function will be called with the content of the message.

B.3 Addressing

B.3.1 allocateIPAddress

Description Allocates randomly an IP-address in the link local address space and sends out an ARP request for this address.

Arguments

→ *no arguments*

Output

← The allocated address.

Algorithm

1. Randomly select an address between 169.254.1.1 and 169.254.254.254³
2. Send out an ARP request for the selected address.
3. If no answer arrives in UNIQUE_ADDRESS_CHECK milliseconds, the address can be used for further communication.
4. If address *is* found, go back to 1
5. If *not*, instruct the system to use the address for further communication.

Used Functions

none

Side effects

One or more ARP requests is sent out checking for the address. When a suitable address is found, the system is instructed to use it for all further communication via the current interface.

B.3.2 storeNodeName

Description Stores the nodes name into the datastructure, unless it is already taken.

Arguments

- **organization** – The organization this node belongs to.
- **username** – The name of the user, if not applicable, use some generic name here.
- **nodename** – A local name for the node, only unique for the user.
- **address** – The address of the node.

Output

← A datastructure describing the node, or fail if address is taken.

Algorithm

1. Call **composeMessage** to do a *findNodeByName* to see if name is taken.
2. If no user was found, use the address.
3. If a response from a node who is not the owner of the address comes back. Do a **AODVrouteRequest** for that address, if a route is found, assume name is not used anymore.

³This is the freely available link local addresses.

4. If owner of name responds, report to the application that the name is taken and a new must be choosen before trying again.

Used Functions

composeMessage – For finding a node by name instead of address.

AODVrouteRequest – To see if a node is available.

appNameOccupied – Requests that the node changes it name.

Side effects

If the name was recognized, a request is sent to the listed address to see if the node is active.

B.4 Group Handling

B.4.1 createGroup

Description Allocates an available multicast address and creates all the needed datastructures to handle a group.

Arguments

→ **groupname** – The name of the new group.

Output

← A datastructure of type **groupInfo** containing the multicast address for the group, or if name was taken a fail message indicating a new name must be choosen.

Algorithm

1. Check if the name is available by doing a *incremental* broadcast, asking the nodes if they know any group by the sugested name and which node is master of it.
2. If no nodes recognize the name, the name is free to use. Go to 4.
3. If the name was taken, a new name must be choosen to avoid conflicts.
4. Select a multicast address randomly.
5. Do a lookup for the selected address.
6. If the address was taken, go back to 1.
7. Otherwise store the address and the name in a **groupInfo** datastructure and return it.

Used Functions

AODVrequestRoute – Used to check if the address is free to be used.

Side effects

All nodes on the connected network will be aware of the new multicast group when this node sounds out a group hello for the newly created group.

B.4.2 addGroupMember

Description Adds members to the group and informs all old members about the new ones. Only nodes that this node trust will be allowed to join. If the group is private or protected, only the master may do this, for public groups all members may add new nodes.

Arguments

→ **groupInfo** – The current status of the group.

→ **newMembersList** – List of addresses to the new members.

Output

← The new group info structure and status info about the add procedure.

Algorithm

1. For each member to add do the following:
 - (a) Check with the user if the node is allowed to join by calling **appAllowNodeToJoin**.
 - (b) Add node to list of nodes part of the group.
 - (c) Send out an encrypted message to the node, giving it the group key and a complete member description.⁴
 - (d) Wait for ACK from node for **NODE_ADDED_ACK_TIMEOUT** milliseconds.
 - (e) If receiving ACK before timeout, update the group info structure to have the nodes as active.
 - (f) If a timeout occurs, retry from 1 (d) for **NODE_ADDED_ACK_RETRIES** before ignoring the node (it will have to retry later).
2. Send out multicast message informing all nodes in the group about the new nodes.
3. Record any error responses from the multicast.
4. Return the updated datastructure.

Used Functions

sendMessage – When sending to nodes and to the group.

appAllowNodeToJoin – Checks with the application if a node is allowed to add.

Side effects

All nodes will get updated group informations. The list of available nodes may change (old nodes may be unavailable).

Comments

Even if a non master node does add a node to a private or protected group, the member info should be ignored by the other nodes. The master should also take steps to stop the misbehaving node.

B.4.3 removeGroupMembers

Description Removes nodes from a group by giving all nodes except the removed ones the new group key.

Arguments

- **groupInfo** – The current status of the group
- **removeNodes** – List of the nodes to remove and why the node is removed and flag telling if the node should be trusted in the future.

Output

← The current group info.

Algorithm

1. For each node in remove list, send out a message informing it that it is removed.⁵

⁴This way the node will be informed that it is accepted into the group.

⁵Any failed transmissions must be remembered and retransmitted when the node reappears.

2. To all nodes that should be still remaining in the group, send out a new group key using **sendGroupKey**. Also update the nodes member info and when doing this, inform the nodes why the removed nodes were removed. If a node was kicked out, all nodes should know this, so they don't let it in again without knowing that it was kicked out by the master.
3. If the node was kicked out, the certificate should be revoked because it should reasonably not be trusted anymore.
4. If any node is marked to not be trusted any more, remove the trust using **removeKeys**.
5. Record any nodes that didn't get the new key and return this information.

Used Functions

sendMessage – For sending messages to the removed nodes.

sendGroupKey – For informing remaining nodes about the new group key.

Side effects

All nodes that are still part of the group gets their group info tables updated.

B.4.4 sendGroupKey

Description Sends a group key to all nodes that should be left in the group, using unicast.

Arguments

- **groupInfo** – The group information for the group to change key for.
- **excludeMembers** – List of members that should not get the group key.

Output

- ← New trust relations.

Algorithm

1. Create a new key.
2. Create a message with the group name, the group key and the current status of the members in the group.
3. Send out the message to each node part of the group except those listed in **excludeMembers** using unicast. The message should be marked for guaranteed delivery.⁶

Used Functions

sendMessage – To send out the key.

Side effects

All nodes still in the group will get a new key and a current state of the group.

Used in

handleJoinRequest – Calls this function when a join request has been processed and granted.

⁶Explanation of this can be found in section B.6.

B.4.5 handleNewGroupKey

Description When a message containing a new group key has been recieved, this function updates the tables accordingly, if the sender of the key is trusted to update the group key, i.e is the master of the group.

Arguments

→ **message** – The message with the new key and current state.

Output

← none

Algorithm

1. Check if the message came from the node that is master of the group in question.
2. Store the new key in the group info structure.

Used Functions

appGroupChanged – Calls an application function that updates the group state in the application.

Side effects

No special

Used in

recieveMessage – Calls this function when a new group key arrives.

B.4.6 handleNewNodes

Description When a node adds a new node to a group, this function should be called to update the local datastructures. In a master, this function should also try to stop a node that tried to add new nodes to a private or protected group without permission.

Arguments

- **sender** – Identity of the sending node.
- **groupName** – The name of the concerned group.
- **newMembers** – A list of new nodes.

Output

← none

Algorithm

1. If the group is public, check if the sender is a member of the group, if so go to [5]. If it isn't, ignore message.
2. If the group is protected or private, check if sender is master of the group, if so go to [5]. If it isn't, go to [3].
3. If this node is master of the group, ask the application if the sending node should be removed from group (as punishment), check also if the new node should be allowed to stay in the group.
4. If the offender should be removed, call **removeGroupMembers** with the offending node. If the new node should be added, call **addGroupMember** (so a legitimate add will be performed).

5. Add the new nodes to the group info.

Used Functions

removeGroupMembers – When a master removes a offending node.

addGroupMember – For adding a real node.

Used in

recieveMessage – When a new member message arrives.

Side effects

A member of the group may be removed and new nodes may be added.

Comments

The adding and removing of nodes should be optimized so as few messages as possible will be needed.

B.4.7 joinGroup

Description Tries to join a group that it knows exists.

Arguments

→ **groupName** – The name of the group it should try to join.

Output

← True if group is joined, fail otherwise.

Algorithm

1. Join the multicast group, by calling **AODVjoinGroup**.
2. Try to join the group by sending a **joinGroup** message with **composeMessage**.
3. If join was accepted, return true.
4. If join wasn't accepted, leave the multicast group and return false.

Used Functions

AODVjoinGroup – To join the multicast group.

composeMessage – For trying to join a group.

AODVleaveGroup – To leave the multicast group if group join wasn't accepted.

Side effects

A group and the corresponding multicast group may be joined (if accepted) and a route to this group will be set up.

B.4.8 leaveGroup

Description Leaves a group and the corresponding multicast group.

Arguments

→ **groupName** – The name of the group to be left.

Output

← True if everything went ok, false otherwise.

Algorithm

1. Send a `leaveGroup` request using `composeMessage`.
2. When the request has been granted, or a timeout has been passed, call `AODVleaveGroup` to leave the multicast group.
3. Return the status of the operation.

Used Functions

composeMessage – For leaving the group.

AODVleaveGroup – For leaving the multicast group.

Side effects

The node will leave the group.

B.4.9 composeMessage

Description This function is used to compose and send a message that wants a response from the receivers, not to do any processing of the data. Several different messages can be sent and the response will be returned to the caller of this function.

Arguments

- **recipient** – The address(es) to the node(s) that should get the message.
- **messageType** – The type of message that should be sent.
- **data** – The data needed to send the message of the type defined in **messageType**.
- **options** – Options used to define how a message should be sent and responses received.

Output

- ← The response received from the node(s).

Algorithm

1. Identify the type of message and extract the data that message needs from the **data** argument.
2. The different message types need the following data:

Type	Arguments	Description
<code>findNodeByName</code>	The full name	Tries to locate the node with the given name. It does a <i>incremental</i> broadcast to locate any owner of the name. This message will not be encrypted.
<code>joinGroup</code>	The name of the group to join	Tries to join a group by doing a <i>incremental</i> multicast to the group. If it is a private or protected group, only the master will respond.
<code>leaveGroup</code>	The group name	Informs the master that this node wants to leave the group.
<code>groupDestroy</code>	The group name	Informs all nodes that the group is destroyed.
<code>listGroups</code>	None	Sends out a broadcast, requesting information about all available groups.
<code>groupMemberList</code>	Group name	Asks the master of the group which nodes are currently members of the group.

3. If it is a unicast, wait `NODE_RESPONSE_TIMEOUT` for an response from the node.
4. If it is a multicast, wait `GROUP_RESPONSE_TIMEOUT` milliseconds for responses from all the nodes in the group. Report back all nodes that didn't respond.

5. If it is a broadcast, wait `BROADCAST_RESPONSE_TIMEOUT` milliseconds for answers. The option `broadcastResponses` defines how many answers should be received before giving up.
6. Return the answer(s) in a form suitable for the type of request.

Used Functions

sendMessage – For sending the message to the recipients
receiveMessage – For receiving the responses from the recipients.

Side effects

Messages are sent to other nodes and answers may arrive. The messages may also change the state in other nodes.

B.4.10 handleJoinRequest

Description Accepts join requests and process them accordingly.

Arguments

- **nodeAddress** – The address of the node
- **request** – The encrypted request for join to a group (or several).

Output

← None

Algorithm

1. Check if message is legal (it comes from a known source and asks for groups controlled by this node).
2. If message ok, add node to requested groups.
3. Send group key to new node.
4. Send out message to all nodes in the group informing them about the new node.
5. Store any availability changes.

Used Functions

sendMessage – For informing the nodes about the new group.
appGroupChanged – Informs the server application that a join has been handled.

Side effects

Sends out a message to all nodes in the groups that are changed, informing them about the new node.

Used in

receiveMessage – Calls this function when a join request arrives.

B.4.11 `handleGroupLeaveRequest`

Description If no objections is noted, the master sends out a group remove request for this code.

Arguments

→ `leaveMessage` – Message from the node

Output

← none

Algorithm

1. Check if message is legal.
2. Check if node may leave by calling `appCheckLeaveAcceptance`.
3. If ok, send acknowledgment to node.
4. Call `sendGroupKey` with all nodes except this one.
5. Update the `groupInfo` datastructure to show the currently known state of the group.
6. Call `appGroupChanged` to update the applications view of the state.

Used Functions

`sendMessage` – Sending the acknowledgment.

`sendGroupKey` – For updating the other group members.

`appGroupChanged` – Informs the application that a node has left the group.

Side effects

The local group info structure gets updated and all nodes except the one requesting to leave will get updated.

Used in

`recieveMessage` – Calls this function when a request for leaving the group arrives and this node is a master.

B.4.12 `handleGroupDestruction`

Description Executed when the master sends out a destroy message, informs the application about it and removes the group info datastructure.

Arguments

→ `message` – The destroy message.

Output

← none

Algorithm

1. Check if the message comes from the master.
2. If ok, ask the application using `appAcceptGroupDestroy` what to do. Possible alternatives is accept and remove group, or keep on going, sending out a question to all (possible) members left in the group if they want to keep on going. If so, this node will become master.
3. If the group should be destroyed, call `A0DVleaveGroup`.

4. Call **appGroupDestroyed** to update the application and then remove group info datastructure.

Used Functions

sendMessage – Send out the question.

doGroupReconstruction – Reconstructs the group if destruction is not accepted.

appAcceptGroupDestroy – Asks the application if group should be destroyed.

appGroupDestroyed – Informs the application that the group has been destroyed.

Side effects

The group info may be destroyed. A new group may be created.

Used in

recieveMessage – Calls this function when a request for destruction of the group arrives.

B.4.13 handleGroupAvailabilityChange

Description Executes when AODV finds a change in the availability of nodes in the group. It updates the datastructure and informs the application about the change.

Arguments

→ **data** – The data about the availability of nodes.

Output

← none

Algorithm

1. The AODV system calls this function with information about which nodes are available/not available.
2. Store the information in the group info structure.
3. If this node is a master and the change includes nodes that has become available within that group, call **handleReestablishedConnections**.
4. If **handleReestablishedConnections** succeeded, inform the application about the change by calling **appGroupAvailabilityChanged**.

Used Functions

appGroupMemberStatus – For passing the information up to the application.

Side effects

None

B.4.14 handleReestablishedContections

Description When the lost connection to a node has been reestablished, any missed administrative messages must be replayed to make the reestablished node aware of changes that occurred during its absence. It is up to the functions that send the messages to store it for later transmission, and it is up to this function to make sure they arrive at the missing nodes when they reappear.

Arguments

→ **nodes** – The nodes that are currently available.

Output

← Status of the updating (which nodes has been updated to the current state, and which is still not updated).

Algorithm

1. For each node that has reappeared, do the following:
2. Check if any administrative messages are in the message queue.
3. If there are any, send them in FIFO order.
4. If message gets ACKed (it is enough that the underlying transport layer ACKs the message, responses are unusable and will be thrown away⁷), remove it from the queue.
5. If connection is broken to node (again), skip the rest of the messages and go to **1** to do the next node.
6. When all messages to a node has been sent, go to **1** to do the next node.
7. When all nodes has been processed, return status to the caller.

Used Functions

sendMessage – For sending the stored messages.

Used in

handleGroupAvailabilityChange – When a node has become available.

Side effects

Depending of what messages are stored for a node, its status may change.

Comments

Only messages where the response data isn't needed by the caller may be stored for later updating. It is very complicated to take care of the response from the nodes, because there isn't any way to give the data to the correct function. This is also why the system is only usable for administrative messages, and not as a way of storing normal communication must be done in the application.

B.4.15 handleGroupListRequest

Description Takes care of requests for group lists. The master of a group hears the request and returns a message with the groups it handles. If the signature matches a node it wants to have in a private group, it includes the private group together with the public groups.

Arguments

→ **message** – Message from a node requesting grouplist.

Output

← none

Algorithm

1. Check if the node is invited to a private group, include it in the list if so.
2. List all public this node is a master for.
3. Send response to node.
4. Inform the application about the request by calling **appGroupListRequested**.

⁷It isn't possible to give the response to the correct function anyway.

Used Functions

- appGroupListRequested** – Informs the application that a node has requested a group list.
- sendMessage** – For sending the response.

Side effects

The application reacts to the request, normally by logging the action.

Used in

- recieveMessage** – Calls this function when a request for a list of available groups comes in and this node is a master.

B.4.16 handleGroupMembersListRequest

Description Answers to questions about which nodes that are members of a group. Only nodes part of a private or protected group will get a answer to a question about that group. All nodes get a reponse for public groups. A fail message will be returned to nodes that aren't allowed to get an answer.

Arguments

- **groupName** – The request
- **sender** – Identity of the sending node.

Output

- ← none

Algorithm

1. Check group is public, if yes, return the membership list.
2. If group is protected, check if this node is a master and if the questioning node is member of the group. If yes, go to **4**. If no, return a **information protected** error response.
3. If group is private, check if this node is master of the requested group and the node asking for the information is member of the group. If yes, go to **4**. If no, ignore the message totaly.
4. Responde with the membership list for the group.

Used Functions

- sendMessage** – For sending the response.

Used in

- recieveMessage** – Calls this function when a membership list request arrives.

Side effects

The requesting node gets an answer to its question if it isn't an unallowed request to a private group.

Used in

- recieveMessage** – Calls this function when a request for a list of available groups comes in.

B.4.17 `handleGroupPartition`

Description When a group gets partitioned, a new master must be appointed. This is done in the AODV layer, but the API and the application must take measures also, this function takes care of making a node that has become master at the ADOV layer to also become master at the group level.

Arguments

- `newMaster` – The new master node, it may be this node.
- `availableNodes` – A list of the nodes that currently are available.

Output

← none

Algorithm

1. When a partition occur, the AODV system will kick in and select a new master. When this has happened this function will be called in the new master.
2. Multicast a message to all nodes that are still available, informing them that this node has been become a new master.
3. When the nodes available has responded, inform the application that this node has become the new master due to network partitioning.
4. Inform the application about the current state of the group.

Used Functions

- `appIsMaster` – informs the application that this node has become a master.
- `appGroupChanged` – informs the application about what nodes are currently available.
- `composeMessage` – For informing the other available nodes.

Used in

- `AODVhandlePartitioning` – When this node has become the new master.

Side effects

The available nodes will be informed about the new master and the datastructures will be updated according to this.

B.4.18 `handleNewMaster`

Description This function is called when a new master has been selected after a group partition (this is identical to the situation when a master node fails and disappears), or when the master privileges has been transfered using `transferMaster`.

Arguments

- `groupName` – Name of the group that has got a new master.
- `master` – Identification of the new master.
- `certificate` – (might not be included) a certificate telling that the old master agrees to the give the new master the privileges.

Output

← none

Algorithm

1. When a new master message arrives, check if a transfer certificate is included.
2. If certificate is included, update the master information for the group.
3. If certificate is missing, ask the application if to accept the new master.
4. If the application thinks it is ok, update the master information for the group.
5. If the application doesn't accept it, it can say two things: *wait* or *leave*.
6. If the answer is *wait*, wait for `FIND_OLD_MASTER_TIMEOUT` milliseconds⁸ to see if the old master comes back. When timeout occurs, ask the application what to do again.
7. If the answer is *leave*, exit the group by sending a *group leave* message to the new master.
8. Inform the application that it is no longer member of the group.

Used Functions

groupLeaveRequest – For exiting a group that got a new master that this node doesn't accept.

appAcceptNewMaster – Used to check if a master should be accepted or not.

Used in

recieveMessage – When a *new master* message arrives.

Side effects

The node may leave the group.

B.4.19 transferMaster

Description Transfers the masters rights to a new node, then informs all nodes in the group that it has done this so they can trust the new node instead.

Arguments

→ **nodeAddress** – Address to the node which we want to transfer to

Output

← Status of the transfer

Algorithm

1. Send message to the node, asking it if it wants to be the master of the group.
2. If node accepts, send all master specific data to the new node (current member data). Together with the group data a certificate saying that the new master *is* the new master is included.
3. A message informing all nodes about the change is expected from the new master. When it arrives, the node may assume it is free of its responsibilities, after waiting `MASTER_CHANGED_TIMEOUT` milliseconds for requests from nodes that has missed the change. Forward any requests to the new master.
4. When the timeout has passed, this node may leave the group if it wants to.

Used Functions

⁸This timeout must be longer then a `AODV_GROUP_HELLO` intervall.

sendMessage – For sending messages to the new master

recieveMessage – For recieving messages to the new master

Side effects

The new master informs all nodes about the change, which leads to updates of their group info structure.

B.4.20 **handleMasterTransfer**

Description Takes care of messages generated by transferMaster and updates the datastructures accordingly. The application is informed about the change for presentation reasons.

Arguments

→ **message** – Message asking for group change.

Output

← none

Algorithm

1. When the message asking the node to become the new master arrives, the application is asked if it can be accept this. This is done by calling **appBecomeNewMaster**.
2. If ok, send a accept message back to the master.
3. Recieve group data and certificate and store it appropriately.
4. Send out the message informing about the master change to the whole group. Wait for ACKs and NACKs from all nodes.
5. Exclude all NACKed nodes from the group, using **removeGroupMembers** and keep on going.
6. Inform the application that this node now is master. Also update the application with the current membership list.

Used Functions

sendMessage – For sending data to the master.

recieveMessage – For recieving data from the master and nodes.

sendGroupKey – For sending out a new group key.

AODVcreateGroup – Create a multicast group.

appBecomeNewMaster – Check if the application accepts the master change.

appIsMaster – Informs the application that it now is the master of the group.

appGroupMemberStatus – Informs the application which nodes currently is members.

Side effects

All nodes will be informed about the new master, which will lead to updating of their group info. Some may also leave the group because of this.

B.4.21 joinGroups

Description When two partitions reconnects, the master that lost control of the multicast group (this is the node that doesn't have the lowest IP-number) asks the new master if it accepts this group to join the new larger group. Before it does this, it checks with the application if it really should try to join the groups. If the application wants to and the master accepts, this node sends out a message to all its member nodes, telling them to join the new master if they want. This function can also be called explicitly (and not implicitly as with the joining of the partitions), to join two different groups.

Arguments

- **newGroupName** – Name of the group the group to add the nodes to.
- **otherMaster** – The address to the other master
- **otherGroup** – Name of the group to take members from.⁹

Output

- ← The new group info.

Algorithm

1. Ask the application if a join is wanted.
2. If ok, send message to new master, asking if a join is accepted.
3. If master accepts, send out a message to all nodes in the partition, telling them to join the new group.
4. Join the group.
5. When the new master announces the new group members, destroy the old group if everything looks ok.

Used Functions

- sendMessage** – For sending messages to the members in the other group.
- appJoinGroups** – Checks with the application if a join should be performed.

Side effects

The group will probaly be destroyed.

B.4.22 handleJoinGroups

Description Will be called when the master asks the nodes of the groups to join a new group instead of the current one.

Arguments

- **currentGroupName** – Name of the group to exit from.
- **newGroupName** – Name of the new group to join (could be the same as **currentGroupName**).
- **newGroupMaster** – The master of the new group.

Output

- ← none

⁹This will normaly the same as the newGroupName, because we usually join two group partitions, but we could join two separate groups.

Algorithm

1. When a *join this group* message arrives, check with the application if this is ok.
2. If it is ok, send a join message to the new master node.
3. After node has joined the group inform the application about it.

Used Functions

appJoinThisGroup – For asking if this node should join a particular group.

composeMessage – For joining a group.

appGroupChanged – For informing the application that a group has been joined.

Used in

recieveMessage – When a *join this group* message arrives.

Side effects

The node changes group if the application allows it.

B.5 Key Handling

Functions for transmitting keys between nodes that wants to setup trust between each other.

B.5.1 sendKey

Description Transfers the public key to another node via a special interface that is secure. All trusted nodes will be transfered to the new node, and new trusts will be recieved.

Arguments

→ **interface** – The interface that the key will be transfered over.

Output

← The keys from all nodes that now trust this node.

Algorithm

1. Send a list of all trusted nodes to the master, this message should be signed but not encrypted.
2. Wait for instructions from master on who to contact for trust setup.
3. When instructions arrive go to [4]. If trust data containing the master arrives from a already trusted node, go to [9].
4. Find the appointed node and connect to it. If this step is done via a one-way media, skip this step. Ask the node if it is ready to recieve a key via the **interface**.
5. If ok, send over the public key and wait for the other keys (at least one).
6. When the keys has arrived. Send over all public keys that are known to this node signed by this node to indicate trust.
7. Inform the application that a key has been recieved.
8. If the master isn't among the trusted nodes, repeat from [1].
9. If master is trusted now, send out trust information to all nodes that hasn't recieved the full data yet.¹⁰

Used Functions

appTrustWasSetup – Informs the application which nodes that are trusted now.

Side effects

Trust will be set up between the group of nodes that now are joined via the trusted link created between the two nodes.

B.5.2 receiveKey

Description Recieves a public key from a node, giving its trust to it.

Arguments

→ **key** – The interface to listen to

Output

← The keys to the new trusted nodes.

Algorithm

1. Start to listen to the interface.
2. When another node asks over the interface if this node can recieve a key, respond that it accepts.
3. When a key and the other information arrives, store it.
4. Transfer all info about the nodes already trusted.
5. Wait for the new nodes trusted keys.
6. Send out the new trust information to all previously trusted nodes.

Used Functions

appTrustWasSetup – Informs the application which nodes that are trusted now.

Side effects

Trust will be set up between nodes that now are joined via the link created between the two nodes.

B.5.3 removeKeys

Description Used to revoke certificates and remove the keys for nodes that isn't trusted anymore.

Arguments

→ **addresses** – Addresses to the nodes which isn't trusted any more.

Output

← Ok if nodes are removed, error code if not.

Algorithm

1. For each node in the address list
2. Create a revocation certificate and send it to all nodes in the group using multicast.
3. When this is done, remove the public key that has been revoked.

¹⁰A good optimization would be to keep track of what data that has been transmitted already to avoid redundant transmissions.

Used in

removeGroupMembers – If a node should be removed more permanently.

Side effects

The datastructures holding the trust will be changed.

B.6 Message Handling

Here are the functions for transmitting messages. The communication protocols used are described in section B.2.

B.6.1 sendMessage

Description Sends a message to the given addresses.

Arguments

- **data** – The data to send
- **recipients** – The addresses to send the message to.
- **flags** – The flags for the request (possible flags is guaranteed, no_encrypt, incremental).
- **maxhop** – Maximum number of hops allowed

Output

- ← Status of the action.

Algorithm

1. Create a message structure.
2. Insert the data and the flags.
3. Encrypt the message with the key matching the recipients (broadcast messages will not be encrypted).
4. Determine what kind of transmission is expected (uni-, multi- or broadcast) and if it is supposed to be incremental or not.
5. Send the message to the listed recipients using the method determined in the previous step. If it is supposed to be incremental, this step will be repeated with an increasing maxhop counter. The sending will continue until the max hop limit (this is defined in INCREMENTAL_MAX_HOPS) has been reached or **recieveMessage** has signaled that it don't want any more answers.
6. If message is marked for guaranteed delivery¹¹, store the recipient and data parts of the messages if a node can't be reached. This can later be delivered when the connection to the node is reestablished. This only works for uni- and multicast, in broadcast you can't know which nodes should be available or not.

Side effects

none

Comments

The implementation of incremental transmission can be a bit tricky. It will be hard to make **recieveMessage** stop the transmission of new messages if this isn't in the same function. I leave it up to the implementor to decide how to solve this because it depends to much on the actual implementation. An idea is to use a flag that this function checks before it sends another batch of messages. This flag can then be set by **recieveMessage** when it has recieved enough answers.

¹¹ This can only be guaranteed if a missing node returns to the group some time before the group dissolves.

B.6.2 recieveMessage

Description Receives a message and decrypts it, checking if the sender is who he says he is.

Arguments

→ `message` – The message that has been recieved.

Output

← The message data (if called explicitly).

Algorithm

1. (If called explicitly) Check the message queue, if there are any messages in it, return the first.
2. If a message arrives the message is decrypted if the node is known, discarded otherwise. Decryption key is selected by looking first at the destination address (multicast messages will be decrypted with the multicast key) and then the sender (unicast messages will be decrypted using the senders public key). Broadcast messages are not encrypted, but may be signed (which is handled on a lower level).
3. The flags is checked, if it is a special type of message, the appropriate function is called, the function is not expected to return anything. The functions are called with the content of the message divided into its components¹².
4. If it is a standard message and this function is explicitly called (poll mode), the content of the message is returned to the caller for further examination.
5. If it is a answer message, and the function isn't in pollmode the message gets stored for later retrieval.

Side effects

A function will be called that handles the message and changes data and does communication.

B.7 AODV

The following functions must be available via the AODV system. These functions should be built directly on top of the AODV routing system. These functions are highly dependant on the routing algorithm used.

B.7.1 AODVavailabilityChange

Description When a route error is sent out informing about nodes that has gone missing, this function kicks in and informs the API about the change.

Arguments

→ `routingError` – The error message from the routing system.

Output

← none

Algorithm

1. When the route error arrives, make a list of the missing nodes and send it to `handle-GroupAvailabilityChange`.

¹²Which these are is decided by looking on the type of message recieved

2. If the master of the group is missing, make this node the new master (if another node hasn't become new master already). Call `handleNewMaster`.

Used Functions

handleGroupAvailabilityChange – Makes the API aware of the change.

Side effects

The state of the groups this node is part of will be updated.

Comments

This function will not be called if `AODVrequestRoute` is called, this is to make that function work properly.

B.7.2 AODVrequestRoute

Description Generates a route request to a particular node or group.

Arguments

→ `address` – Address to find route to.

Output

← Status of the request, true if a route was found, false otherwise.

Algorithm

1. Create a route request using the address (see 2.1.1).
2. Send out the request and wait for a `route reply` or a `route error`.
3. Return the result to the caller.

Used in

allocateIPAddress – When checking if a address is taken.

sendMessage – When finding a route to a node or a group.

createGroup – When checking if a group address is taken already.

Side effects

A route will be set up to the node or group in question.

B.7.3 AODVcreateGroup

Description Creates a group by sending out a group hello with the join flag set. The group shouldn't exist (this should have been checked before calling this function).

Arguments

→ `groupAddress` – The address the group should be on.

Output

← True if the group was created, fail if the group existed or couldn't be created at all.

Algorithm

1. Create a route request with the join flag set (see 2.1.2).
2. Send out the request and wait for a `route reply` or a `route error`.
3. Return the result (ok, exists, failure) to the caller.

Used in

createGroup – When creating a new group.

B.7.4 AODVjoinGroup

Description Joins a multicast group if it exists.

Arguments

→ **address** – The address to the multicast group.

Output

← True if operation succeeded.

Algorithm

1. Send out a route request with the join flag set (see 2.1.2).
2. Wait for a route reply or a route error, return false if a error occurs.
3. If any reply arrives, select the best route found and activate that route and return true.

Used in

joinGroup – When joining the multicast group corresponding to a group.

Side effects

A route will be set up to the multicast group and this node will become a part of the multicast tree.

B.7.5 AODVleaveGroup

Description Makes the calling node leave a multicast group, it might still be part of multicast tree, but it will no longer listen to messages to the group.

Arguments

→ **groupAddress** – The address of the group to leave.

Output

← True if node has left the group, fail if it for some reason failed¹³.

Algorithm

1. Create a route message with the prune flag set (see 2.1.2).

Used in

leaveGroup – When leaving a multicast group after leaving the group using the multicast group.

Side effects

The node will leave the multicast group, but not necessarily the multicast tree.

B.8 Application Functions

The following functions will be called from the API when events occur. They should must be defined, but they don't necessarily have to do anything except returning the default behavior.

¹³Leaving a group which the node isn't a member of will fail.

B.8.1 appAcceptGroupDestroy

Description Used by nodes that get a group destroy message. The application may if it chooses to not accept the destruction and then a new group may form by calling the routines for creating a new group.

Arguments

→ `groupName` – The name of the destroyed group.

Output

← True (*default*) if application accepts destruction, false otherwise.

Used in

handleGroupDestruction – checking if the application accepts the destruction of the group.

B.8.2 appAllowNodeToJoin

Description Is called when a node wants to join a group this node is a master for.

Arguments

→ `nodeDescription` – Address and name of node.

Output

← True (*default*) if node should be accepted to join, false otherwise.

Used in

handleJoinRequest – when a administrative application should decide if a node may join a group.

B.8.3 appBecomeNewMaster

Description This is called when this node is asked to become a new master. The application must accept or deny this request.

Arguments

→ `groupname` – The name of the group in question.

Output

← True (*default*) if the node accepts to become a master or false if not.

Used in

handleNewMaster – When a new master gets announced (also when this node becomes master.)

B.8.4 appCheckLeaveAcceptance

Description Checks with the application if a node may leave the group.

Arguments

→ `nodeId` – Identification of the node

Output

← True (*default*) if ok to leave the group, false otherwise.

Used in

handleGroupLeaveRequest – When checking if a node may leave.

B.8.5 appGroupChanged

Description Updates the state of the group. Will be called if nodes come or go, both permanently and temporarily (network partitions).

Arguments

→ `nodeList` – List of changed nodes with current status (new, available, not available, gone).

Output

← none

Used in

handleGroupAvailabilityChange – when a new group composition has been registered.

handleJoinGroups – when the two groups (or partitions) has been joined).

Comments

If nodes are missing (due to a partitioning) the application might want to wait before it continues its execution. This to limit the amount of data the partitioned nodes will miss.

B.8.6 appGroupDestroyed

Description This function is called when the group has been destroyed. This function should clean away the group information.

Arguments

→ `groupName` – Name of the group that got destroyed.

Output

← None

Used in

handleGroupDestruction – When the group has been destroyed.

B.8.7 appGroupListRequested

Description Informs the master application that a node has requested a group list, this is normally only interesting for logging and in administration tools.

Arguments

→ `nodeId` – Id of the node that requested the group list.

Output

← `none`

Used in

handleGroupListRequest – when a group list has been requested.

B.8.8 appHasTrustSetup

Description Informs the application about new trust relations.

Arguments

→ `trustInfo` – The new trust info.

Output

← `none`

Used in

sendKey – When getting back the receivers trusted keys.

receiveKey – When getting the senders trusted keys.

B.8.9 appIsMaster

Description Called when node has become a master of a group.

Arguments

→ `groupname` – Name of the group that this node is master of.

Output

← `none`

Used in

handleMasterTransfer – When the node has become a new master.

handleGroupPartition – When the node has become a new master for the group partition.

B.8.10 appJoinGroups

Description Asks the application if two groups should be joined. This is used when two network partitions rejoins (if two *real* groups should be joined, the application already knows and accepts it, so this function is not relevant). If the two network partitions still use the same group key, then a join is trivial. In this case, a join should, in almost all cases, be accepted immediately.¹⁴

Arguments

- `groupName` – The name of the group that is joining its partitions.
- `otherMaster` – The master of the other partition.
- `sameGroupKey` – True if the two partitions still have the same group key.

Output

- ← True (*default*) if groups should join, false if not.

Used in

`joinGroups` – When handling join of partitions.

B.8.11 appJoinThisGroup

Description When a node is asked to join a group, the application will be questioned if it wants to join the group.

Arguments

- `currentGroup` – The current group that the node is asked to leave.
- `otherGroup` – The other group that should be joined with the own one¹⁵.
- `otherMaster` – The master of the other node.
- `sameGroupKey` – True if the two partitions still have the same group key.

Output

- ← True (*default*) if groups should join, false if not.

Used in

`handleJoinGroups` – called when deciding if a join should be accepted.

¹⁴This would probably be best to make configurable by the user, but that is of course up to the implementor of the application.

¹⁵It could be the same name, but it will be another master node