Linux Implementation of a Cooperation and Accounting Strategy for Multihop Cellular Networks

Diplomarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von:
Carolin Latze
2006

Leiter der Arbeit:
Prof. Dr. Torsten Braun

Forschungsgruppe Rechnernetze und Verteilte Systeme
Institut für Informatik und Angewandte Mathematik

Contents

Chapter 1

Introduction

Nowadays, people want to be able to go online every time and everywhere. They want to read their emails, do some bank transfers or just surf in the Internet. This leads to the need of ubiquitous Internet access. As base stations and wireless LAN access points are very expensive for an Internet service provider (ISP), there was the need to think about new possibilities. Therefore, Multihop Cellular Networks have been deployed. These networks connect different mobile ad-hoc networks (MANETs) to the Internet. MANETs are networks, which are spontaneously formed by several mobile devices, which decide to cooperate. Therefore each device acts as terminal and router at the same time. Multihop Cellular Networks (MCNs) extend this approach by a gateway node, which allows to establish connections to the Internet from inside a MANET. As MCNs first have been used for military purposes, all the participating nodes were under the same authority and therefore willing to cooperate. With the usage of such networks in civil scenarios, cooperation can not be ensured anymore. As MANETs strongly relay on cooperation, there was the need to think about possibilities to stimulate it. During the last years, some approaches have been proposed which can be divided into two categories: detection based and motivation based.

Detection based algorithms stimulate cooperation through punishment. They relay on neighborhood watch and exclusion of misbehaving nodes. Motivation based approaches motivate nodes by rewarding them for good behavior. One of these motivation based algorithms is CASHnet, which stands for "Cooperation and Accounting Strategy in Multihop Cellular Networks". CASHnet requires a sender to pay for its packet if it leaves its MANET. The receiver also has to pay if it receives a packet coming from another MANET. Furthermore all the intermediate nodes in the appropriate MANETs get rewarded since they wasted energy to forward the packets. Simulations have shown that this scheme leads to a good load of the network [1]. Therefore it has been decided to implement and evaluate CASHnet

in a real world environment using Linux notebooks, which has been done during this diploma thesis.

Chapter 2 gives a general introduction into Multihop Cellular Networks, its problems and possibilities. Chapter 3 introduced some cooperation and accounting schemes for the two different categories. Furthermore, an introduction into the CASHnet scheme is given. Chapter 4 and 5 discuss the Linux implementation of CASHnet. First, the technologies are described and then a detailed description of the implementation is given. Next in Chapter 6 the evaluation of the implementation is shown. Last, we give a summary and an outlook in Chapter 7.

Chapter 2

Multihop Cellular Networks

Multihop Cellular Networks combine the advantages of mobile ad-hoc networks with the advantages of wired networks. The basic idea is to extend a wired infrastructure by using ad-hoc networks.

2.1 The Concept of Multihop Cellular Networks

Multihop Cellular Networks consist of different mobile ad-hoc networks which are interconnected using wired links or the Internet as can be seen in Figure 2.1. This is especially useful in regions with a weak fixed infrastructure, as it is the case in disaster or low populated areas. The fixed network provides a kind of backbone interconnecting the ad-hoc networks and providing Internet access.

Another advantage of this approach is that the coverage is less visible as with other wireless networks like GSM. Wireless ad-hoc networks can work without access points equipped with big antennae, since each node acts as a small inconsiderable access point. Therefore, if a strong resistance against the setup of new antennae exists in the population, it might be easier to deploy ad-hoc networks instead of classical cellular networks.

There exist two basic types of wireless mesh networks, single-hop and multi-hop. If organized in a single-hop manner, all the nodes have to be located in the transmission range of a base station. As can be seen in Figure 2.2, all communication takes place over the access point, and there is no direct data transfer between the single nodes. In the worst case, assuming wide spread nodes, this may result in one base station per node, which is very inefficient.

The second form of organization is the multi-hop network as shown in Figure 2.3. Instead of communicating with the base station exclusively, nodes can use other nodes as relay to reach the base station. This allows a base station to virtually cover a much wider area and therefore leads to less base stations. Figure 2.3 shows how Node D can reach the base station

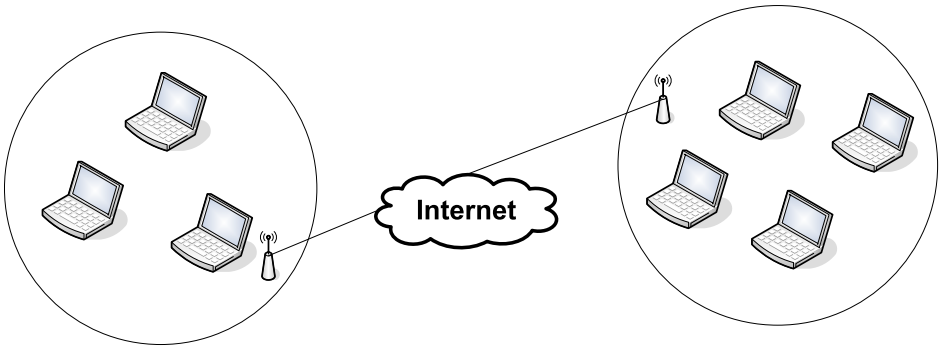


Figure 2.1: Multihop Cellular Networks interconnected by the Internet

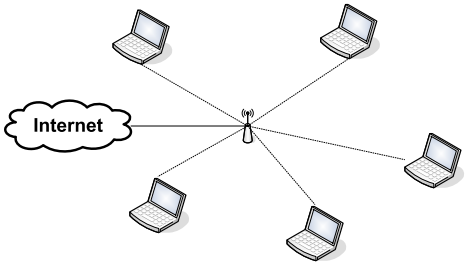


Figure 2.2: Single-hop network

and thus the Internet via Node B and A. The usage of multi-hop networks also leads to the saving of energy [2], since data has not to be transmitted to the remote base station directly, but only to a neighboring node, which will take care of forwarding the packet.

The main problem for the deployment of multi-hop wireless networks is the routing. In a wired network, we have a comparably static topology with dedicated nodes acting as routers forwarding packets. A routing protocol running on these routers allows them to exchange information about link failures and the current load. Based on that information, the routers calculate their routing tables.

In a wireless ad-hoc scenario things are not nearly as static as in wired networks. Usually no dedicated routers are deployed, but the users' mobile devices act as routers. These nodes are moving, which leads to permanent changing routes. Furthermore, mobile devices usually are not built to forward packets, but for personal use. They are usually administrated by the user itself, which causes problems for the security and the robustness of the network. An experienced user may modify the routing behavior of his node preventing packets to be forwarded. The owner of the mobile device

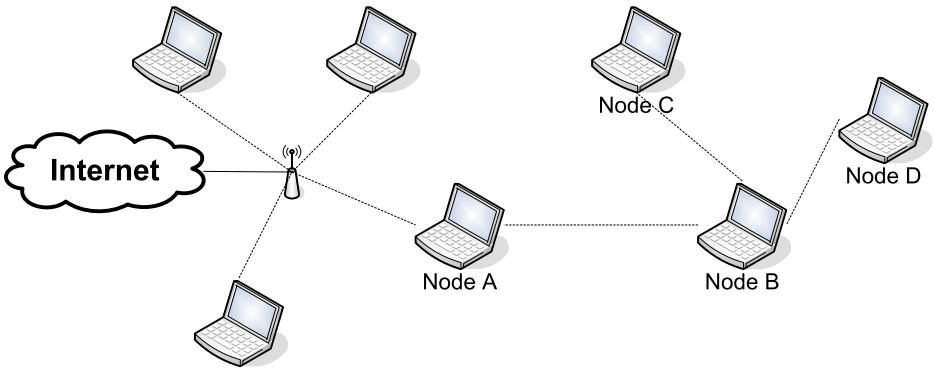


Figure 2.3: Multi-hop network

might also eavesdrop on forwarded packets, or even modify them. Another problem are the technical limitations of mobile devices. A user might have no electricity and has to run the node on battery power. Advanced router functionalities would increase the need of energy, further reducing the time the user can work with its device.

Furthermore, the wireless medium per se is less reliable than the wire. Transmissions of different nodes might interfere, even if different channels are used, the propagation properties are asymmetric [3]. Furthermore there are a large number of typical routing problems in wireless ad-hoc networks like exposed stations and hidden nodes [3].

This is why special protocols for the routing in wireless networks have been developed. The most famous aproches are Dynamic Source Routing (DSR) [4] and Ad hoc On-Demand Distance Vector (AODV) Routing [5]. In the following section, AODV will be introduced more detailed, since this protocol has been used during this diploma thesis.

2.2 The AODV routing protocol

The Ad hoc On-Demand Distance Vector (AODV) routing protocol has been developed in 2003 by Perkins, Belding-Royer and Das. It is an experimental protocol for use with mobile nodes in MANETs [5].

2.2.1 AODV messages and operation

AODV is designed for mobile ad-hoc networks as it offers fast adaption to dynamic link changes, ensures loop freedom and avoids classical problems of distance vector protocols such as the “count to infinity” [5]. The AODV routing protocol requests routes only when they are needed. If a link break

occurs, it provides possibilities to inform all the nodes, which use this link to transmit data.

AODV uses special messages to exchange information about the network state. There are four types of control messages which are transmitted via UDP on Port 654: Route Requests (RREQ), Route Replies (RREP), Route Reply Acknowledgments (RREP-ACK) and Route Errors (RERR). Besides Unicast, AODV uses broadcast (255.255.255.255) for control traffic.

On startup, a node usually sends a HELLO message to inform the others that it joins the network. A HELLO message is a RREP with the time-to-live IP header field set to one. This simple mechanism ensures that only the direct neighbors are reached and a node knows its neighbors. These HELLO messages usually are repeated periodically.

If a node within the AODV network wants to transmit data, it first has to check whether it knows a valid route to the destination. If not, a RREQ control message is sent.

If a node, who has a valid route entry to the destination, receives this RREQ, it generates a RREP to the originator. Since in most cases bidirectional communication can be assumed, a gratuitous RREP is sent to the destination. By this, both, the originator and the destination are informed about a possible route between them.

To keep track of link failures, each AODV node monitors the links to its neighbors. If a link failure is recognized, there are two options:

1. Send a RERR to inform the other nodes about that link failure

2. Try to repair this link

Link repair takes place by generating RREQ to the affected nodes. If it receives RREPs containing new routes with better or equal hop counts, it will replace the old route with the new one. If this is not possible, it will send a RERR for this route. If a node decides to repair a link during an ongoing data transmission over this link, it has to buffer the data.

Of course, this mechanism might cause a repair of routes which are not used anymore.

To prevent loops, a node keeps track of received and transmitted control messages. Because of this, a node has to wait a certain time after a reboot. This shall ensure that the node is synchronized with the other nodes in the network. Rebooting nodes which are part of active routes causes problems because the node will forget all its sequence numbers, which are important to avoid loops. Because of this the node has to wait some time after the reboot before it starts sending HELLOs and joining the network. This time has to be long enough so that the other node invalidates every route to and including this node.

2.2.2 AODV over unidirectional links

In most of the scenarios bidirectional links are assumed, but this is not a mandatory requirement for AODV [5]. In a unidirectional scenario, routes are only established in one direction, which causes problems sending the RREP. With bidirectional links, when a node receives an RREQ it will send the RREP over the same path, which will fail with unidirectional links. To solve this problem, a blacklist is introduced. If a node recognizes that the sending of its RREP failed, it blacklists the neighbor over whom it wanted to send. This leads to a new route discovery over another path. Afterwards the RREP can be sent over these paths.

2.2.3 Networks using AODV

AODV has not the constraint of building subnets with the same net-mask. It will work with any composition of IP addresses. But if we have subnets within a network which uses AODV, it is possible to minimize AODV control traffic appointing a node as subnet router. This subnet router responds to every RREQ with a destination address outside of the subnet. If another node of this subnet receives an RREQ, it should forward it to the router and never send an RREP. To protect the AODV subnet from RREQs to non-existing AODV nodes, the subnet router has to discard packets to non-existing nodes and send ICMP Host Unreachable messages.

As written in [5], an ad-hoc network using AODV can connect to external routing domains also. The mechanism is similar to the described subnet mechanism above.

2.2.4 Security

AODV does not provide any security mechanism. Neither the transmitted data, nor the control messages are protected. Of course an end-to-end data transmission can be secured using standard VPN solutions like IPSec [6] or using a solution as proposed in [7]. The control messages can be authenticated using digital signatures.

2.3 Global Connectivity (Internet Gateway Discovery)

One of the main advantages for a mobile node joining a foreign self-organizing ad-hoc network is to know how to reach the global Internet from within the ad-hoc network. There must be a mechanism allowing a node to find out which node in the ad-hoc network provides gateway functionality. This problem is not covered by most MANET proposals. An IPv6 based solution

for this has been proposed in [8], where two additional control messages are introduced:

Internet gateway solicitation message: This message is sent by a node which wants to find the next gateway to the Internet.

Internet gateway advertisement message: This message is sent by the gateway either periodical or as reply on a solicitation.

Since many MANET routing protocols are reactive, it might make sense, if the gateway also acts completely reactive and does not send advertisements.

If a node wants to send a message to the Internet, it sends an Internet gateway solicitation and receives an Internet gateway advertisement. This advertisement consists of an IPv6 prefix which helps the node to configure a routable IP address and a route towards the Internet. The node stores this information with a timestamp and starts sending to the Internet.

The Internet gateway consists of at least two interfaces: one for the MANET and one for the Internet. It does not participate in the MANET which means that the gateway only listens to routing messages but does not collect the contained information. It also should not be used as intermediate node that means, it should not forward any message within the MANET. If the gateway sends an Internet gateway advertisement, it has to use its global address as source address. The gateway keeps a table storing all received solicitation messages. This allows the gateway to learn which node is in the Internet and which node belongs to the MANET.

If the gateway receives a message from a MANET node which should be forwarded to the Internet, it has to check whether the destination is really an Internet node or not. If the destination is a MANET node, the gateway tries to redirect the message to the right node and notify the sender. If the redirection fails, the message is discarded.

Chapter 3

Cooperation Schemes

The idea behind mobile ad-hoc networks (MANETs) is that of self-organizing nodes forming a network without any fixed infrastructure. This leads to the need of cooperation. Originally MANETs have been developed for disaster areas and battle-fields. This implies that all the nodes are operated under the same authority and cooperation can be guaranteed. With the deployment of MANETs for consumers, other ways have to be found to stimulate cooperation. This implies security issues as well as social aspects.

In the following sections, we describe some of the existing schemes to stimulate cooperation in MANETs, concluding with our own approach called CASHnet.

3.1 Detection based

Detection based cooperation schemes provide mechanisms to exclude selfish nodes from a MANET. In all of these schemes, blacklists are introduced in which selfish nodes are collected. These nodes are not allowed to send any traffic anymore and they become excluded from all routes. These blacklists are based on neighborhood watch where every node is monitored and evaluated by its neighbors. This monitoring mechanism should avoid false detections since it might occur that a node does not misbehave because of selfishness, but due to link failures or something else outside his control. Furthermore there should be a possibility to reintegrate blacklisted nodes if their behavior changes, otherwise a probably wrong evaluated node would be excluded forever.

In the next section, the CONFIDANT protocol as one exponent of these detection based cooperation schemes is presented.

The CONFIDANT Protocol

The CONFIDANT protocol has been developed [9]. CONFIDANT means "Cooperation Of Nodes - Fairness In Dynamic Ad-hoc NeTworks", is based

on Dynamic Source Routing (DSR) and expects that nodes are proper authenticated and cannot pretend to be someone else.

The idea behind CONFIDANT is that if the delay between behaving well and getting rewarded is too long, there is no stimulus for nodes to conform. CONFIDANT uses strict policing to detect and exclude misbehaving nodes. In other cooperation schemes, nodes store a history of all bad experiences leading to a huge need of memory, which scales not well for large networks. Furthermore, the processing of these lists puts stress on the limited resources of mobile devices. This is why, CONFIDANT uses a method to learn from own experiences as well from other's experiences.

CONFIDANT consists of four components: the monitor, the trust manager, the reputation system and the path manager (see Figure 3.1).

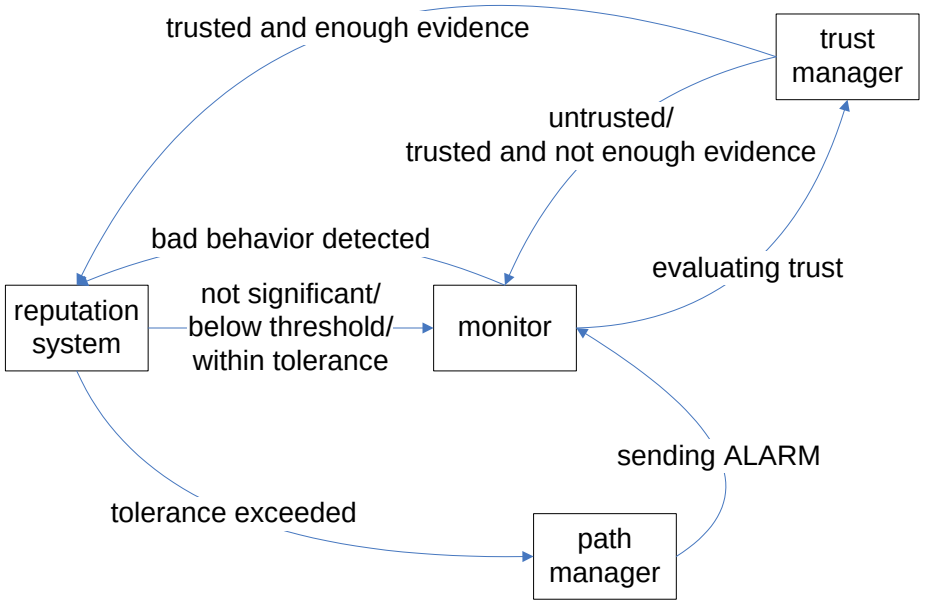


Figure 3.1: The components of CONFIDANT

The monitor The idea behind the monitor is that each node can detect misbehavior in its neighborhood listening to the transmissions or observing the routing messages. It is also possible that a node keeps a copy of a forwarded packet and compares it with the packet forwarded by the next node. By this, it is possible to detect content changes. If the monitor detects misbehavior, it calls the reputation system.

The reputation system To avoid centralized rating, every node has to maintain a so called blacklist. This list can or should be exchanged

with friends and can even be added to a route request alarming nodes on that route. Since there is the possibility that this list contains nodes which only seem to behave badly, the reputation system has to decide how bad a node behaves. Therefore it tests whether such a behavior is significant or not. This is done by updating an event count. If this event count exceeds a certain threshold, the node is rated. Only if this rating exceeds a tolerance, the path manager is called.

The path manager The path manager re-ranks paths due to the reputation of the nodes on this path. It deletes paths containing malicious nodes, ensures that requests from malicious nodes are ignored and that requests to malicious nodes are answered with an alert to the source.

The trust manager The trust manager decides whether an alert is trustable or not and if there are enough evidences for this message.

The authors claim, that a network using CONFIDANT is serviceable even if 50% of the nodes behave maliciously.

Since CONFIDANT allows the exchange of experiences, it is faster than other approaches. On the other hand, wrong observations which come from other nodes can destabilize the network. To avoid a wrong ranking, CONFIDANT uses several thresholds in the reputation system.

3.2 Motivation based

In contrary to the detection based cooperation schemes, motivation based schemes do not base on blacklists. The idea is not to punish selfish nodes but to make it more attractive to cooperate by gaining a benefit from it. All these schemes are based on the fact that one can earn money by forwarding packets and has to pay for sending. Motivation based cooperation schemes can be subdivided in centralized and decentralized accounting schemes. In the centralized approaches, the accounting is done by a supervisor, whereas in the decentralized accounting, it is managed by the nodes themselves. There is also one hybrid approach called CASHnet, which combines the advantages of these two approaches.

3.2.1 Decentralized Charging

The Nuglet Scheme

In 2000, a new cooperation scheme for mobile ad hoc networks called Nuglet Scheme [10] has been presented. The idea was to motivate nodes to cooperate with the possibility to deal with a kind of money by paying for sending

and earning for forwarding. This money is called nuglets [1]. In their first approach they proposed two mechanisms:

Packet Purse Model In the packet purse model, the originator of a packet has to put enough nuglets for each forwarding node into the packet. Every forwarding node takes one or more nuglets (depending on the distance the packet has to pass) out of the packet. This leads to the need of a very good estimation of the needed nuglets on the originator. If it sends less nuglets than needed, the packet will be dropped on the node, which cannot take a nuglet anymore and the originators looses its packet and its money. If it sends more nuglets than needed, the originator looses money.

Packet Trade Model This model requests the nodes to trade with the nuglets. Therefore the packet is resold on every forwarding node until it reaches the destination. Every node requests a higher price than it payed so that the destination has to pay for the whole path. In this scenario, the source pays nothing. This can lead to an overloading of the network since the source has no limitations. To avoid this, a mechanism could be introduced allowing the nodes to decide whether they want to buy a packet or not. This would result in a sort of back pressure which may prevent users from sending too much traffic.

As a user should not be able to manipulate its nuglets, a security module, such as a smart card, was added which is responsible for all the security critical functions. Furthermore, the system includes a public key infrastructure which should guarantee the packets integrity.

The security module includes the following functions:

The nuglets The amount of nuglets are increased if a node forwards a packet and decreased if it sends an own packet. Furthermore, the amount is decreased on the receiver side if the packet trade model is applied.

Key storage As Nuglet requires a public-key infrastructure, it must be ensured that nobody can manipulate its private key. Therefore the key is stored in the security module. This module also contains a certified public key and a set of root certificates.

The routing protocol The Nuglet scheme does not require any special routing algorithm. The only requirement is that the routing algorithm runs in the security module.

[1] In the first approach, this money was called beans.

The original approach has been extended as written in [11]. The packet purse and trade models have been replaced with a secure counter of the sent and forwarded packets on each node. The counter is increased by one if a node forwards a packet and decreased by the number of estimated hops if a node sends a locally generated packet. This counter has to remain positive. If a node wants to send a packet and has less credits than the number of estimated hops, the node is not allowed to send packets until it has forwarded enough other packets. There is the possibility to buffer packets until enough credits are available. However several applications such as real time voice transmissions cannot handle delayed packets very well. So, this approach is limited.

Since a forwarding node should only earn credits if it really forwarded the packet, the earned credits are counted on the next node in the forwarding chain as shown in Figure 3.2.

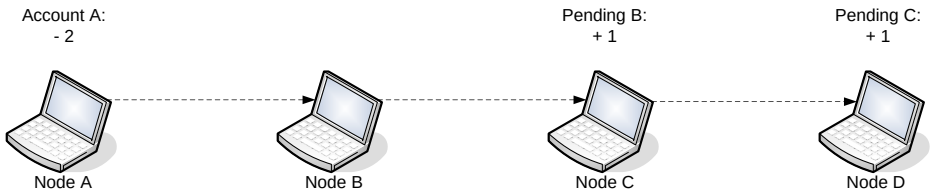


Figure 3.2: Nuglet payment process

The whole approach depends crucially on the security model. If a node wants to send a packet, it has to pass this packet to its security module, which decreases the credits counter and attaches a security header. If there are not enough credits, the packet is dropped. Afterwards the packet is send to the next node. The forwarding node also passes the packet to its security module, which verifies the security header and tests whether the packet comes directly from the source or from a forwarding node. If this packet comes from a forwarding node, the actual node increases the pending credits counter for the forwarding node and forwards the packet. Otherwise, it would only forward the packet. After a certain time, the nodes run the credit synchronization protocol, which updates the credits on every node, using their pending credits (see Figure 3.3). If a node has moved away from its position in the forwarding chain before the credits have been synchronized, its pending credits are lost.

The Nuglet scheme causes some overhead due to the cryptographic functions and the establishment of the security associations. The first one should not be a problem, since there are efficient Hash algorithms. The second point can be avoided integrating this establishment with the neighbor discovery protocol [11].

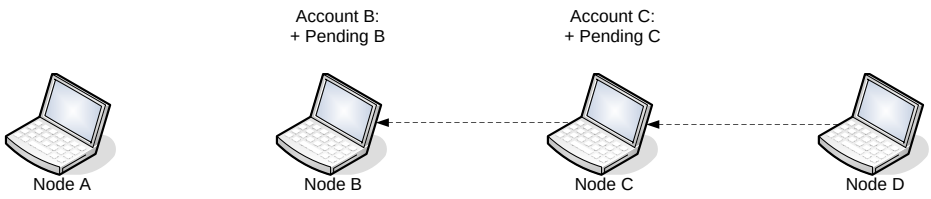


Figure 3.3: Nuglet rewarding process

3.2.2 Centralized Charging

Node Cooperation in Hybrid Ad Hoc Networks

In 2003, a new accounting scheme was proposed [12]. A system is assumed which consists of a set of mobile nodes and several base stations connected to a high speed backbone. Any end-to-end communication between mobile nodes might take place over multiple other nodes. However, it must be ensured, that at least one of these nodes is the base station. Since all the communication passes a base station, there are suboptimal routes, if the nodes are neighbors or close to each other but far from the base station. Figure 3.4 shows two nodes S and D, which would be able to communicate directly, but have to communicate using the base station. On the other hand, this reduces the routing complexity since the nodes only have to know a route to the base station. Since every packet has to pass at least one operator maintained base station, the accounting can easily be accomplished. The accounting in detail works as follows:

Ad-hoc only traffic If a node wants to send a message to another node in the same cell, it has to send this message to the base station which forwards it to the destination node as shown in Figure 3.4. The upstream forwarding nodes are rewarded when the message passes the base station, whereas the downstream nodes are only rewarded if the message arrives at the destination. As the base station does not know whether a transmission was successful or not, the destination node has to send an acknowledgment. Since the destination node could probably decide not to send this acknowledge to save energy, the node is charged by the base station for a small amount when the message is passing. If the acknowledgment has arrived at the base station, the destination node gets reimbursed this small amount. To save energy and bandwidth, it is sufficient to send a single acknowledgment for multiple packets (e.g. for a whole TCP transmission).

Traffic between two ad-hoc cells If the source and the destination node are located in different ad-hoc networks, the source sends its message

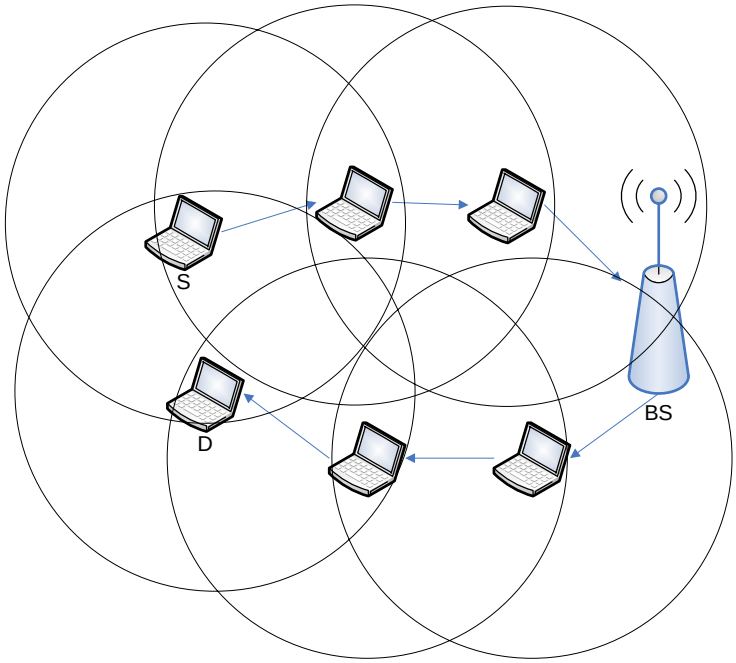


Figure 3.4: Ad-hoc only traffic

to its base station which sends the message over the backbone to the destination's base station which forwards it to the destination node as it is shown in Figure 3.5. The accounting is similar as for the ad-hoc only traffic. The transmission over the backbone is not charged.

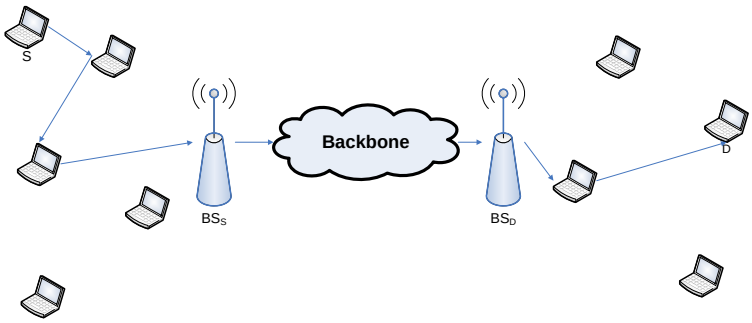


Figure 3.5: Traffic between two ad-hoc networks

In order to reward them, the base stations needs to know which forwarding nodes are located on a route. Therefore, source routing is required.

To protect against fraud, secret key cryptography is deployed. If a node wants to send a message to another node, it has to establish an authenticated path to the destination. During this establishment, every node on the route has to authenticate itself with the base station using its long-term secret key and every node obtains a symmetric session key. Afterwards the communication can start. The originator calculates a message authentication code (MAC) over the packet using its session key and encrypts the packet using a stream cipher with its session key as input. Each forwarding node also encrypts the packet using its session key. The base station recomputes all the stream ciphers which were applied by the forwarding nodes to decrypt the whole packet and verifies the MAC. If this verification was successful, the accounting is done and the packet is sent towards the destination. Before the packet leaves the down-stream base station, the station applies the stream cipher for each forwarding node on this route. If the packet passes a forwarding node, it is decrypted until it reaches the destination.

The authors do not mention how the rewarding has to be done if sender or receiver is located in the Internet.

Charging Support for Ad Hoc Stub Networks

In 2003, a new centralized accounting scheme has been proposed, which is similar to the Nuglet scheme [13]. This approach differs in the way the network is organized. They integrate the ad-hoc network in a fixed infrastructure using an access point AP (see Figure 3.6). In contrary to the idea proposed in [12], ad-hoc only traffic which does not pass the AP is allowed. This architecture is called 'Stub Ad hoc network to an ISP', since the Internet service provider (ISP) also gains benefit and manages the accounting. The motivation behind the approach is that it would be reasonable for an ISP to motivate its users to form ad-hoc networks. According to [13], this reduces the communication over the access points and allows to extend the access point's range. It is argued that since the ISP does the management and distributes the money, it is allowed to take money for it. The authors find that this scenario would be more likely in the future than purely self-organizing networks.

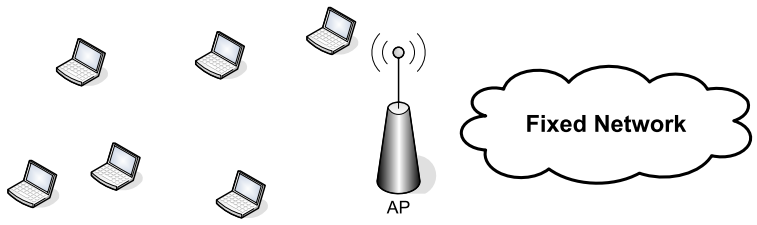


Figure 3.6: Ad-hoc stub network

For the security, an initial authentication of each participating node with the ISP is required. As the accounting is done at the ISP, the nodes are not interested in changing their identity to earn more money.

To realize the accounting, a source routing scheme is required, which traces the route. The accounting scheme works as follows: The source node digitally signs a message and each intermediate node verifies it and computes a new hash. If the packet arrives at the destination node, it generates a receipt of the received amount of data and sends it digitally signed to the last intermediate node, which informs the access point AP. The AP verifies the information and assigns the charges and rewards. If the verification fails, no intermediate node gets rewarded.

3.2.3 CASHnet - the first hybrid accounting scheme

Hybrid accounting schemes try to combine the advantages of centralized and decentralized schemes. The idea is to give the operator control over the cash-flow but allow the ad-hoc nodes to deploy distributed routing, accounting and rewarding schemes.

The accounting scheme

In 2004, a new accounting scheme called CASHnet has been proposed [14]. CASHnet is short for "Cooperation and Accounting Strategy for Hybrid Networks". It is based on a topology as shown in Figure 3.7. There is at least one service station in each mobile ad-hoc network and a gateway which connects the MANET to the Internet.

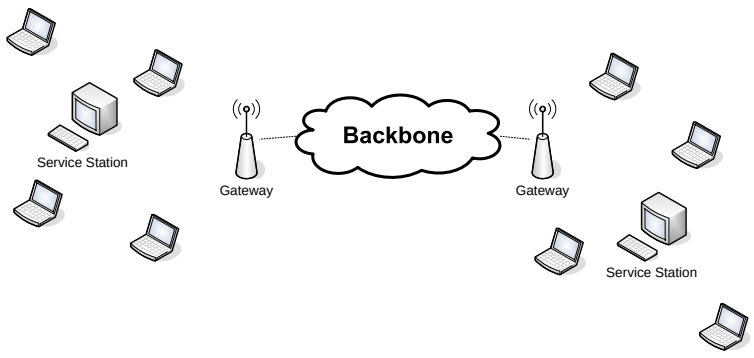


Figure 3.7: CASHnet topology

The author proposes a scheme which supports initiator- and receiver-based payment. The cost-sharing between sender and receiver leads to a better fairness since we have no sessions which leads to the fact that sender and receiver change their roles according to the direction of the transmission.

For instance in asynchronous transmission like downloads from the Internet, the receiver is the one who has the whole benefit, so it should (also) pay for it. Another advantage of this cost-sharing is that costs are covered where they occurred. Furthermore ad-hoc only traffic does not require any payment since the operator is not involved.

In the CASHnet scheme, there are two types of currencies: the Traffic Credits and the Helper Credits. Every time a node wants to send a non-ad-hoc-only message it has to pay Traffic Credits according to the number of hops in its network. If a node forwards a non-ad-hoc-only message, it receives one Helper Credit. The author introduced service stations which are maintained by the operator and located within the ad-hoc networks. A user who wants to exchange his Helper Credits into Traffic Credits has to go to a service station. It is also possible to buy additional Traffic Credits on these service stations. This allows for selfish nodes, since it is either possible to cooperate or to pay. But as a node earns no Helper Credits if it does not forward anything, cooperation would be a gainful alternative to selfishness. Another advantage of buying Traffic Credits is that it prevents starvation of nodes which cannot forward data for any reasons.

Since the only information required from the routing protocol is the hop-count towards the destination, CASHnet is more flexible concerning the routing protocols than most of the other schemes. CASHnet works with AODV or DSR like protocols. This is an advantage especially for large networks, where DSR does not work well. CASHnet requires a sufficient amount of processing power and memory on each participating node and the existence of a tamper resistant device to ensure the security of critical information. The processing power is necessary since the traffic has to be signed to ensure data integrity and data origin authentication.

The tamper resistant device (e.g. a smart card) should contain the two accounts for the Traffic and the Helper Credits and the keys needed to sign and verify data.

CASHnet consists of the following phases:

1. **Setup Phase:** The Setup Phase has to be passed before a user can join a CASHnet enabled mobile ad-hoc network. The user has to obtain a smart card from its provider which contains

   - the node N's unique identifier
   - the node N's public/private key pair K_N/KP_N
   - the node N's certificate $CERT_P(ID_N, K_N)$
   - the providers public key K_P

   Afterwards the user has to update its certificate at the service station, if necessary and to load its Traffic Credits account paying with real money.

2. **Initial Authentication Phase:** If a node wants to start a transmission, it first has to authenticate itself to the other nodes on the route. Therefore it sends an authentication request (AUTHREQ) to the destination node containing its certificate. Every node on the route stores the certificate and forwards the request. This does not generate Helper Credits and costs nothing. The destination replies with an authentication reply (AUTH) containing the destination's certificate. This is also stored on the forwarding nodes. As every intermediate node has also to be authenticated, authentication requests are sent periodical to every one hop neighbor. This leads to a better performance when forwarding a packet.

3. **Packet Generation Phase:** If a node wants to send a self generated message, it has to decide, whether it is ad-hoc only traffic or not. Ad-hoc only traffic does not cost anything. Therefore it only has to be signed and sent. If the message should leave the actual ad-hoc network, the node has to determine the costs and charge the Traffic Credits account if possible. If there are not enough credits, the packet is dropped. Otherwise the Traffic Credits are decreased by the costs and the packet is signed and sent.

4. **Packet Forwarding Phase:** If an intermediate node receives the message, it has to verify the signature of the last intermediate node (if there was one) and the originator's signature. If one of the verifications fails, the packet is dropped. Otherwise and if the message originator is located in another ad-hoc network than the receiver, the node has to store a tuple of the next node's id and its own signature of the message in a so called reward list. Then the packet is sent to the next node. Afterwards, the node sends an acknowledgment (ACK) to the last intermediate node (if there was one), which contains its own identity and the last node's signature of the message. If the message is ad-hoc only, the node only has to do the verification and forward the message if possible.

5. **Packet Reception Phase:** If a node receives a message which is addressed to itself, it first has to check the signature of the last intermediate node (if there was one) and the originators signature. If one of the verifications fails, the packet is dropped. Afterwards it has to check whether the sender is in the same ad-hoc network or not. If it is, the node only accepts the message. If the sender is located in another ad-hoc network, the receiver is charged by the costs, which arose in its own ad-hoc network and sends an acknowledgment to the last intermediate node (if there was one) containing its own ID and the last node's signature over the received message.

6. **Rewarding Phase:** If an intermediate node receives an ACK, it deletes the matching pair of ID and signature in its reward list and increases its Helper Credits account by one.

7. **Refill Phase:** After a certain time, the user has to go to a service station, put its smart card into the station and exchange Helper Credits against Traffic Credits. It is also possible to buy additional Traffic Credits and to update its certificate, if necessary.

If sender and receiver are located in different MANETs, the gateway on the sender side removes the forwarding signatures and sends the message containing the original payload and the senders signature over the backbone towards the second MANET as shown in Figure 3.8. The receiver side gateway acts as a normal first forwarding node and signs this message as written above. If one of the nodes is outside the MANET, the packet has to be send without any signatures since CASHnet packets are not compatible to normal IP.

Figure 3.8 shows the detailed messages exchanged in a CASHnet enabled MANET.

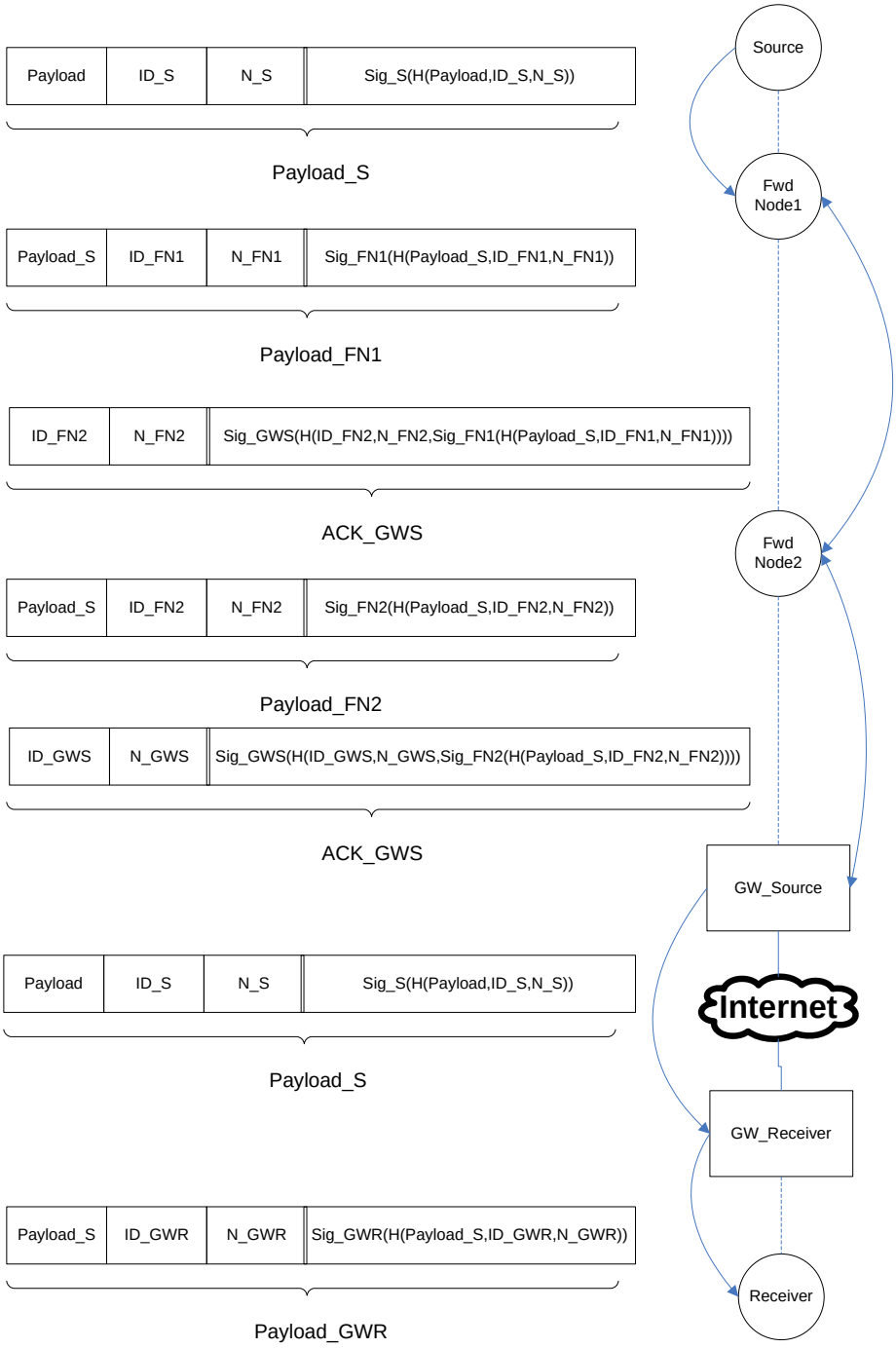


Figure 3.8: CASHnet payload

Chapter 4

Implementation Environment for the CASHnet scheme

In this chapter, the implementation requirements of CASHnet in a Linux environment will be described.

4.1 Design requirements

Independent from any special CASHnet requirement we had to decide which operating system should be used. It has been decided to use Linux since this operating system is widely used for prototyping. Furthermore, Linux is open source and very well documented.

There are several requirements, which have to be fulfilled in a CASHnet enabled MANET. First of all, CASHnet requires an ad-hoc routing protocol which takes care of the route changes and provides information to CASHnet. We decided to use AODV since we want to show that CASHnet does not require a source routing protocol like DSR like other cooperation schemes. Furthermore, AODV performs well for large networks and we do not want to limit the size of CASHnet enabled MANETs because of the routing protocol.

Further, CASHnet requires special security functions, which should be very fast. Additionally, it should not be possible to manipulate these functions. This can be ensured by tamper resistant devices like smart cards In the first implementation of CASHnet we did not use any smart card to avoid additional complexity. Therefore we decided to use a library called RSAREF, which is fast in the calculations and introduces little overhead. As it is a must, that a final version of CASHnet provides smart card based security, two types of cards have been evaluated. But none of them can be used in CASHnet due to the huge delays as will be shown in Chapter 6.

In addition, the security mechanism of CASHnet requires the modification of every data packet. Therefore, we needed a possibility to access these packets. A possibility is the Netfilter framework provided by the Linux kernel. It provides functionality to redirect data packets to the user space or modify them in the kernel space. Netfilter is more or less kernel independent, which means that CASHnet will work with different kernel versions. Furthermore, Netfilter allows packet modifications in the user space. The advantage of a user space program is that the usage of CASHnet does not require a recompilation of the kernel. Therefore we decided to use the Netfilter user space library even if it slightly reduces performance.

For the programming language we decided to use C++, since there is a kernel interface for this language. Furthermore a fast and efficient language was needed, which can be fulfilled by C. The C++ structure was needed to unitize the program. This modular design was needed to separate the different functionalities of CASHnet. Using this modularization, we were able to integrate all the cryptographic function into one single module. This allows for example to substitute parts of the software implementation with a smart card.

4.2 Netfilter

Netfilter is a framework integrated in the Linux kernel which allows packet filtering and modification. To enable it, the following kernel options have to be specified (for kernel version v2.6.8):

```
Device Drivers ---> Networking Support ---> [*] Networking
Support, Networking options ---> Network packet filtering
(replaces ipchains) ---> IP: Netfilter Configuration --->
<*> IP tables support (required for filtering/masq/NAT), <*>
Packet filtering
```

A packet filter is designed to filter packets according to certain header fields and to apply actions to that packet, such as ACCEPT or DROP. A user or an application can specify rules, which consist of matching critera and an action that should be applied to packets matching these critera. Therefore, Netfilter provides a tool named `iptables`. This tool allows adding and deleting rules from the kernel's packet filtering table. It has to be mentioned that these rules are not permanent, which means that any modification will be lost at reboot. To avoid this, the rules have to be included into an initialization script.

Each network protocol defines a set of hooks which are well defined points in the network stack on which the protocol will call the netfilter framework. For IPv4, there are five hooks: PREROUTING, INPUT, FORWARD, POSTROUTING and OUTPUT as shown in Figure 4.1. `iptables`

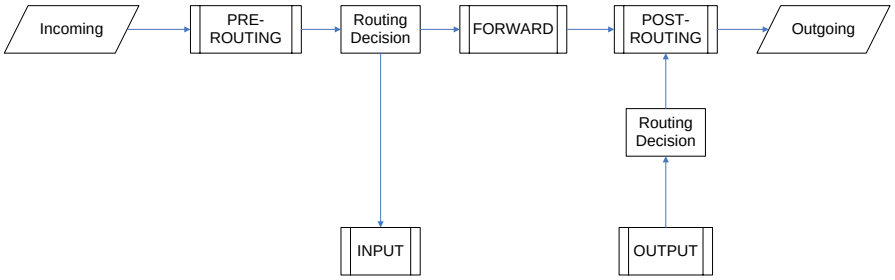


Figure 4.1: IPv4 hooks

provides several tables, which can work on a subset of hooks. For example the filter table, which is used in CASHnet, only knows the INPUT, FORWARD and OUTPUT hook (see Figure 4.2). It is possible to define several rules per hook. The list of rules at a hook is also called chain. If a packet passes a hook, it is checked against every rule in the chain.

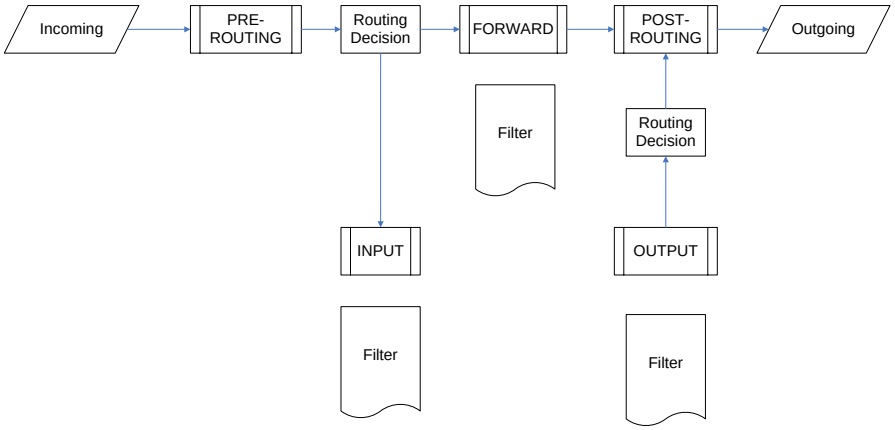


Figure 4.2: Tables on the IPv4 hooks

We will now concentrate on the table called filter, which consists of three hooks: INPUT, FORWARD and OUTPUT. The chains can be listed using `iptables -L`:

```
bash-3.00# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
TCPMSS     tcp  -- anywhere             anywhere            tcp flags:
    SYN,RST/SYN TCPMSS set 1000
DROP       all  -- anywhere             anywhere            MAC
```

```
    00:90:96:BE:9B:9E
DROP       all  -- anywhere             anywhere            MAC
    00:90:96:BE:9B:D5
DROP       all  -- anywhere             anywhere            MAC
    00:90:4B:B3:68:4A
DROP       all  -- anywhere             anywhere            MAC 00:0E
    :35:60:50:56
QUEUE      all  -- anywhere             anywhere

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
QUEUE      all  -- anywhere             anywhere

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
TCPMSS     tcp  -- anywhere             anywhere            tcp flags:
    SYN,RST/SYN TCPMSS set 1000
QUEUE      all  -- anywhere             anywhere
bash-3.00#
```

In general, when a packet arrives on a machine, the kernel decides whether it is addressed to itself or not. If the actual node is the recipient, the packet passes the INPUT chain which means, that it is checked sequentially against every rule. If one rule matches, the packet is processed according to the specified target. For example, if the target was DROP, the packet is dropped. If no rule matches the packet is handled according to the specified policy, e.g. ACCEPT. If it should be ensured that absolutely no packet arrives on the actual host, the policy has to be DROP. If a packet is generated by the actual node, it goes through the OUTPUT chain. In case the packet is neither generated by the actual node nor addressed to it and the IP forwarding is enabled, the packet goes through the FORWARD chain. If IP forwarding is disabled, such a packet is dropped.

Considering the listing above, a packet will be checked against at least one rule independent from the hook it passed. Most of these rules are applied to every packet independent from the protocol used. This can be seen in the "prot" column. Only the first rules in the INPUT and the OUTPUT chain are only applied to TCP packets. Furthermore there is no rule which filters packets according to a special source IP or destination IP address. The first column named "target" indicates what to do with a matching packet. TCPMSS means that the maximum segment size (MSS) of TCP will be modified. Therefore, further options are given. The idea is to specify a special TCP MSS at TCP connection setup. The rules with the DROP target lead to a dropping of packets, which come from a specified source MAC address. Thus, these rules realize a MAC filter. Furthermore, there is

one rule in each chain which uses the QUEUE target. As mentioned later in this section, packets, which match these rules, are redirected to the user space.

In most of the cases the filter rules should only be applied on packets which pass a specified interface. This can be done using the `-i` (input interface) or `-o` (output interface) options. If a packet goes through the INPUT chain, it does not have an output interface and vice versa. Only packets which pass the FORWARD chain have both.

In addition to the targets DROP and ACCEPT, there are another four targets, called LOG, REJECT, RETURN and QUEUE. LOG realizes a kernel logging of the matching packets. REJECT works nearly like DROP but it causes an ICMP 'port unreachable' error message. RETURN forces the packet to leave the actual chain. If a packet matches a rule using the RETURN target, the packet will not be tested against other rules in this chain. But as it is possible to nest chains (generate new chains in existing ones), this packet will be checked against the rules in the outer chain(s). QUEUE sends the matching packets into the user-space for further processing.

In case it is desired to handle incoming packets in the user-space, the QUEUE target has to be set. Furthermore, a queue handler and a user space application which processes the packet have to be installed. The queue handler is responsible for the communication between kernel and user space. The standard queue handler is the `ip_queue` module, which has to be loaded before specifying any rules with this target. Furthermore, Netfilter provides a user space library called libipq which can be used from applications to receive and process queued packets. The application then may just check the packets for their content or modify them and set a target like ACCEPT, DROP or REJECT. The target is called verdict in the libipq.

A packet read out of the user space queue consists of the whole Ethernet frame. The structure which holds the frame provides some more information as shown in the following listing:

```
typedef struct ipq_packet_msg {
  unsigned long packet_id;        /* ID of queued packet */
  unsigned long mark;             /* Netfilter mark value */
  long timestamp_sec;             /* Packet arrival time (seconds) */
  long timestamp_usec;            /* Packet arrvial time (+useconds) */
  unsigned int hook;              /* Netfilter hook we rode in on */
  char indev_name[IFNAMSIZ];      /* Name of incoming interface */
  char outdev_name[IFNAMSIZ];     /* Name of outgoing interface */
  unsigned short hw_protocol;     /* Hardware protocol (network order) */
  unsigned short hw_type;         /* Hardware type */
  unsigned char hw_addrlen;       /* Hardware address length */
  unsigned char hw_addr[8];       /* Hardware address */
  size_t data_len;                /* Length of packet data */
  unsigned char payload[0];       /* Optional packet data */
} ipq_packet_msg_t;
```

This allows to modify packets easily. But it has to be mentioned that if modifying the packet's payload, the checksum has to be recalculated. Since Netfilter does not check this, the kernel will silently drop the packet if the checksum is invalid.

The status of the user space queue may be checked via `cat /proc/net/ip_queue`. The queue's maximum length can be shown via `cat /proc/sys/net/ipv4/ip_queue_maxlen`. It is also possible to modify the length this way.

4.3 AODV-UU

AODV-UU is an AODV implementation done by Uppsala University [15]. The routing protocol has been implemented as an user space daemon. Furthermore, AODV-UU is based on Netfilter to send packets to the user space. Although a kernel implementation would be faster, the authors decided that stable operation, which is simpler to achieve in user space, has a higher priority than performance. AODV-UU complies with RFC 3562 [5].

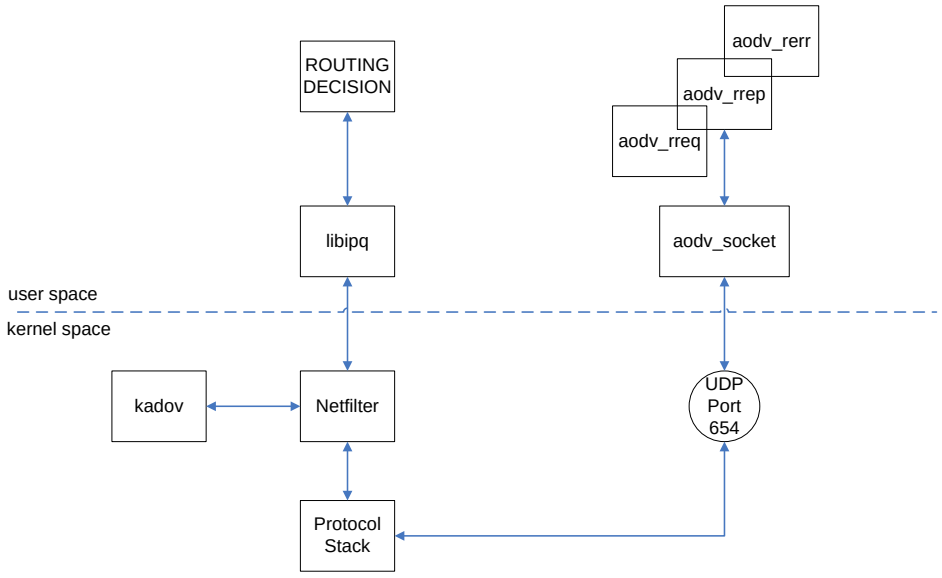


Figure 4.3: The modules of AODV-UU

AODV-UU consists of a kernel module called `kaodv` which sends the data packet to the user space using Netfilter (see Figure 4.3). In the user space, the packet is passed to the application (called "ROUTING DECISION" in Figure 4.3) via the libipq library. This applies to data packets. For AODV control messages, UDP port 654 is used, where a module called `aodv_socket` receives them.

Additionally, AODV-UU provides gateway support which allows MANET nodes to communicate with nodes in other networks such as the Internet. The implementation even provides multiple gateway support.

Multiple gateways in a MANETs may cause problems due to the mobility of the nodes. Assume a MANET with two gateways. A mobile node which wants to send TCP packets to the Internet might be near one gateway. It starts communication over this gateway. Therefore, this gateway creates an entry for the network address translation (NAT) for this node and manages the connection. But as the node is mobile it might move in the neighborhood of the other gateway. Therefore, the routing is changed and the node now sends over the second gateway. But as this gateway has no NAT entries for this node, the connection breaks. To avoid this, AODV-UU introduces tunneling. Packets which leave the MANET are tunneled to the gateway which is the nearest when the communication starts. Therefore, there are no route changes even if the node moves to another gateway.

4.4 The RSAREF library

In 1994, the RSA Laboratories implemented a free RSA[1] reference implementation called RSAREF [16]. This implementation should serve as a portable and educational implementation and therefore was designed for non-commercial use and is highly transparent. In their license they also forbade the usage of RSAREF for people, which do not live in the USA or Canada as RSA and DES [2] are export-controlled technologies. These restrictions are no longer valid since the RSA patent expired in 2000.

RSAREF supports many cryptographic functions like RSA encryption and key generation, MD2 [17] and MD5 [18] message digests, DES, Diffie-Hellman key agreement and Triple-DES.

RSAREF is well documented and comes with certain demo programs to have a better understanding of this implementation. It is written in C and can be used in every non-commercial application.

4.5 Smart cards

CASHnet as proposed in [19] requires a tamper resistant device for the critical functions and informations. As a user should not be able to modify its Helper and Traffic Credits, these accounts should be stored on such a device. Furthermore CASHnet requires every data packet to be signed which should also be unaccessible for a user. This leads to the need of signing the packets on a tamper resistant device. This device should also store the needed keys and certificates.

[1] Rivest-Shamir-Adleman security algorithm
[2] Data Encryption Standard

A smart card is such a tamper resistant device, which is very popular in our days. A smart card is a small plastic card consisting of a microprocessor and a programmable memory [20]. There are also so called non-smart chip-cards, which have no CPU.

Smart cards are mostly used as a secure storage medium for cryptographic keys and other critical information. Furthermore, most of them are able to compute signatures and encrypt data. It is more secure to do this on a smart card since the critical information such as the private key does not leave the card.

4.5.1 The Linux smart card interface

Most of the smart cards follow the standard ISO7816. This standard describes the card dimensions, the PIN layout, allowed signals as well as the transmission protocol.

There are two other important standards which are needed to implement applications using smart cards. The first one called PKCS#11[3] is about the communication with the card and the second PKCS#15 defines the information format on the cards.

PKCS#15: PKCS#15 is a standard about how to store information such as keys and certificates on a smart card. The whole name is "PKCS#15: Cryptographic Token Information Syntax Standard". Version v1.1 has been published in 2000 [21]. This standard ensures the interoperability of smart cards from different vendors.

PKCS#11: PKCS#11 is called "Cryptographic Token Interface Standard" [22]. The last version v2.20 has been deployed in 2004. PKCS#11 provides applications a logical view of the smart card. It follows a object-oriented approach. This standard deals with slots, tokens, objects, sessions and operations [23], where

- slots are the place where the smart card is put, usually a reader
- tokens are the smart cards
- objects are for instance keys and certificates
- sessions have to be used to communicate with the card. At first, a session has to be opened
- operations are the provided methods which can be used such as `C_Login`, which is used to login to a smart card.

Figure 4.4 shows the Linux smart card stack which is needed to build applications with smart cards.

[3] PKCS is short for Public-Key Cryptography Standards

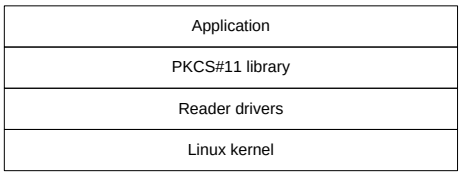


Figure 4.4: Linux smart card stack [24]

To extend a PC with a smart card, a so called reader is needed. This reader realizes the communication with the card. Our smart card can be inserted into small USB readers to attach them to a node. As shown in Figure 4.4 the Linux kernel does not provide drivers for these readers. Therefore a framework which includes drivers has to be installed. We used the OpenCT framework [24].

OpenCT As mentioned above, OpenCT provides drivers for smart card readers. It only covers the communication with a smart card reader. For the communication with a special smart card, further libraries are needed such as OpenSC.

OpenCT is an open source lightweight equivalent of the popular PC/SC standard. PC/SC provides an interface between applications, a resource manager and drivers for smart card readers [23].

OpenCT provides some tools to communicate with the card on the command-line. For instance, the `openct-tool` can be used to list the status of a card:

```
bash-3.00# openct-control status
No.   Name                          Info
================================================
  0   Schlumberger E-Gate           slot0: card present
bash-3.00#
```

Furthermore, as can be seen in Figure 4.4 a PKCS#11 library is needed to communicate with the card. One popular library for Linux is OpenSC.

OpenSC OpenSC provides a library sitting on top of OpenCT which allows application to communicate with a smart card. It allows to select and read files on every smart card, but for encryption and decryption, a PKCS#15 compliant smart card is needed. The basic library of OpenSC is called `libopensc`. Furthermore there are some advanced methods e.g. to create a RSA key pair. For the Cryptoflex cards we used, OpenSC supports signing, decrypting and initialization [23].

OpenSC also provides a tool named `opensc-tool` which allows to communicate with the smart card on the command-line similar to the tools provided by OpenCT. It is for instance possible to show the name of a smart card:

```
bash-3.00# opensc-tool -n
Cryptoflex 32K e-gate
bash-3.00#
```

On the top of the stack as shown in Figure 4.4, there is an application which is now able to integrate a smart card.

Chapter 5

Implementation of CASHnet under GNULinux

To implement CASHnet in a modular design, the functionality has been split into four main modules. The first one is responsible for the Accounting. This module manages the two credit accounts for the Helper and Traffic Credits.

Second, there is a module called UDPsock, which is responsible for the UDP CASHnet control messages.

The third main module is the security device. This module provides functions to sign and verify data. Furthermore it manages the stored certificates from other nodes. In the first CASHnet implementation, this functionality is implemented in software. But in the future, this module can be replaced with a tamper resistant device.

Last, we have a module which integrates the functionality provided by the other modules. This module is called Filtering. Filtering reads the packets out of the Netfilter user-space queue and processes them. It decides when an ACK has to be sent and manages the signing and verification. Filtering is the most powerful module in this CASHnet implementation.

Furthermore some helper modules were introduced.

5.1 Data flow

In this section, the data flow in the CASHnet implementation is described. When speaking about "accepted" packets, it is meant, that the NF_ACCEPT verdict is set. Then the packet is processed by the network stack. Sometimes, the NF_DROP verdict has to be set. Then it is written, that packets are dropped.

If a packet comes in using a Netfilter hook, the `acfi` class recognizes an event on the user space queue and calls the process method of the `filtering` class. In filtering we first check, if this packet is an AODV control message, which has to be always accepted (see Figure 5.1). If this is not an

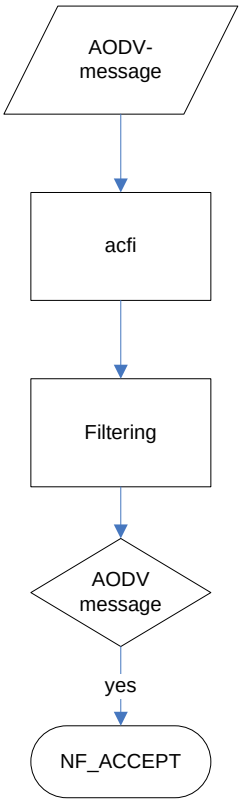


Figure 5.1: Handling of AODV messages in the CASHnet algorithm

ADOV message we check for CASHnet control messages. CASHnet control messages can be identified because of the transport protocol (UDP) and a special port number (19810). Afterwards the type of the control message has to be checked. This can be done by checking the type numbers in the control message headers. The three types of control messages are processed seperatly as shown in Figure 5.2:

1. **AUTH:** If an AUTH message arrives, the certificate has to be stored and possible pending AUTHREQs for this certificate have to be discarded. Furthermore, the now available certificate causes the triggering of outstanding packets. Afterwards the packet has to be dropped if it was addressed to us and accepted otherwise.

2. **AUTHREQ:** If an AUTHREQ is received, it has to be checked, if the actual node is the destination of this message (which would also be true, if this message is a periodical broadcast) or not. If the node is the destination, an AUTH has to be sent containing its certificate. Fur-

thermore the certificate included in this AUTHREQ has to be stored if not already done. Last, the message has to be dropped. In case, another node is the destination node of this AUTHREQ, the incoming certificate has also to be stored if not already done and the message has to be accepted.

3. **ACK:** If an ACK arrives, which is not addressed to the actual node, it is accepted without any further processing. In case the actual node is the destination of this ACK, the node first has to verify the signature. If the required certificate is missing, it has to be checked whether there is already an AUTHREQ pending. If not, a new one has to be generated. In case, the certificate is available, the signature is checked and the ACK is booked. Afterwards the message is dropped.

In case the packet is no AODV and no CASHnet control message, it is a normal data packet. In this case, there are three options (see Figure 5.3):

1. **Destination node:** If the actual node is the destination of a data packet, it first has to be checked, whether the needed certificates are available or not. If there are some certificates missing, an AUTHREQ has to be sent if not already done. Furthermore, the packet has to be queued for further processing. In case all the certificates are available, the signatures have to be checked. The packet is dropped and no further processing is done if at least one signature is invalid. Otherwise it has to be checked whether the packet comes from outside the actual MANET or not which is important for the accounting. If the packet comes from inside the actual MANET, the signatures are removed and the NF_ACCEPT verdict is set. Otherwise, it has to be tested if it passed more than one hop so far to check if there are one or more forwarding nodes. This is needed because in case of at least one forwarding node, an ACK has to be sent. Afterwards the signatures have to be removed and the packet is accepted.

2. **Forwarding node:** In case the actual node forwards a packet, as written above, it first has to be ensured, that there are all the certificates available which are needed to check the signatures. In the case of invalid signatures, the packet has to be dropped. If the packet only travels through the actual MANET, it simply has to be signed again and accepted. If it will leave the MANET or comes from a node outside the MANET, it has to be checked whether there was already at least one other forwarding node. In case there was one, an ACK has to be sent. Afterwards, the message's footprint has to be stored in the rewarding list. Furthermore, the message has to be signed again and accepted.

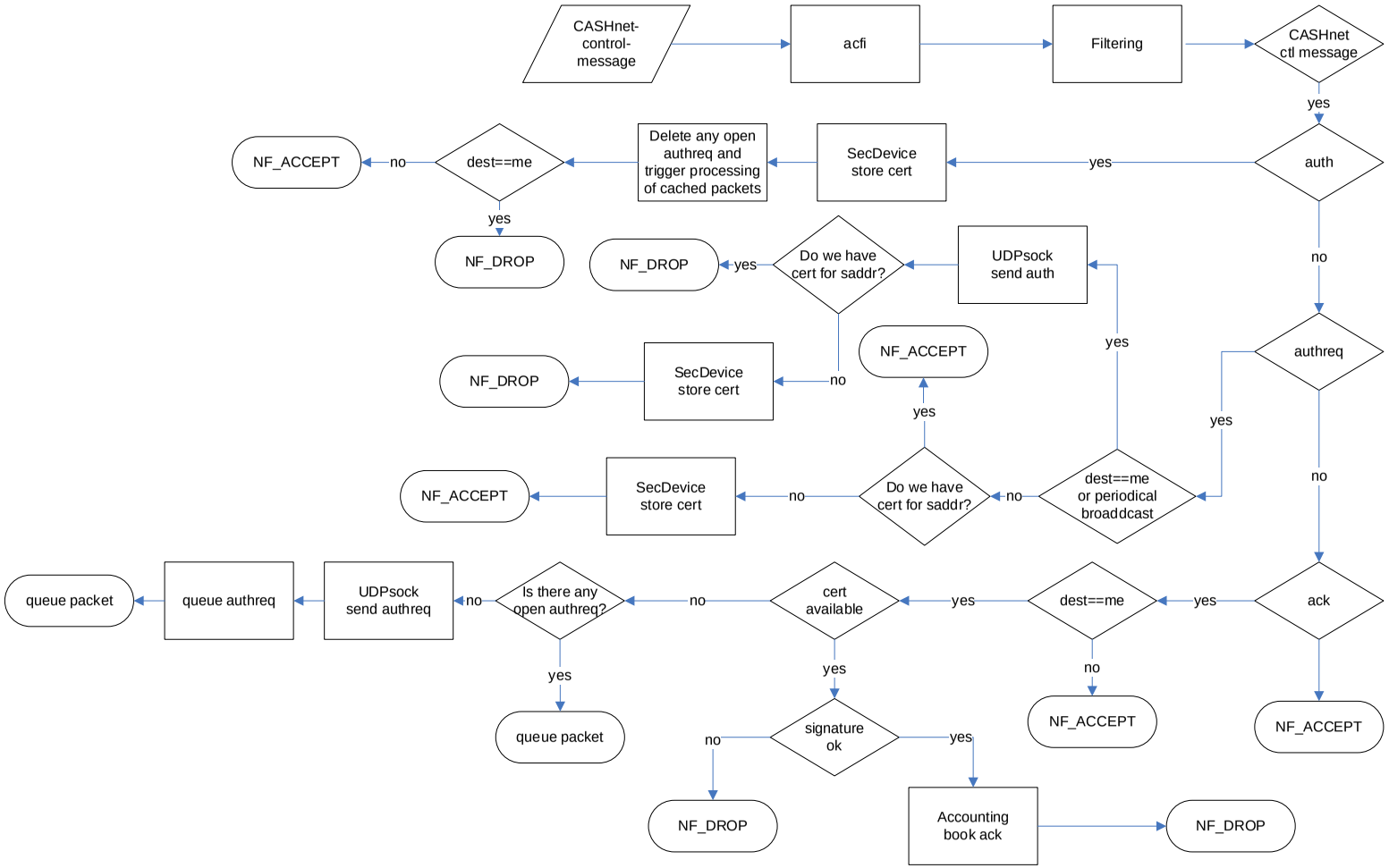


Figure 5.2: Handling of CASHnet control messages

3. **Source node:** If the actual node is the source of a message, it has to be checked if this message is adhoc-only or not. In case it only passes the actual MANET, it has to be signed and accepted. If it is not adhoc-only and there are not enough traffic credits, the packet has to be dropped. Otherwise the node gets charged, signs the message and sends it.

In case the incoming message has been tested against all the criteria written above and it was not possible to identify it, an error occurred.

Example Scenario

In this section, we want to describe an example scenario and how some specific packets are handled. A detailed overview of the exchanged messages can be found in Figure 5.5 on page 40.

Assuming a scenario as shown in Figure 5.4. Node A and B are direct neighbors, B and C too as well as C and the gateway to the Internet. The nodes dispose of an amount of traffic and helper credits as specified in Table 5.1. All the nodes have routes to each other node and to the Internet. Furthermore, due to the periodical AUTHREQs all the nodes have the certificates of their direct neighbors (see Table 5.1).

Node	Traffic Credits	Helper Credits	Certificate for
A	10	0	B
B	3	5	A,C
C	5	3	B,Gateway
Gateway	-	-	C

Table 5.1: amount of credits and certificates on the nodes

Now, Node A wants to send a data packet to a host in the Internet. Therefore, an application generates a packet and passes it to the network stack. The packet travels to the INPUT hook and gets redirected into the user-space queue. At the end of this queue, our CASHnet program reads out the packet. It asks the accounting module if there are enough traffic credits for the route the packet wants to take. The accounting module will find out that this packet will travel over the gateway which leads to the need of three hops in the actual MANET. As Node A still has ten Traffic Credits, it gets charged with three credits. Further, the Node's ID, a nonce and a signature over the whole packet including ID and nonce are added. This signature has been computed using the security module. Afterwards, CASHnet sets the NF_ACCEPT verdict and the packet is sent out of the node. The next node in the forwarding chain is Node B. The packet on Node B will travel through the FORWARD hook which leads to a redirect

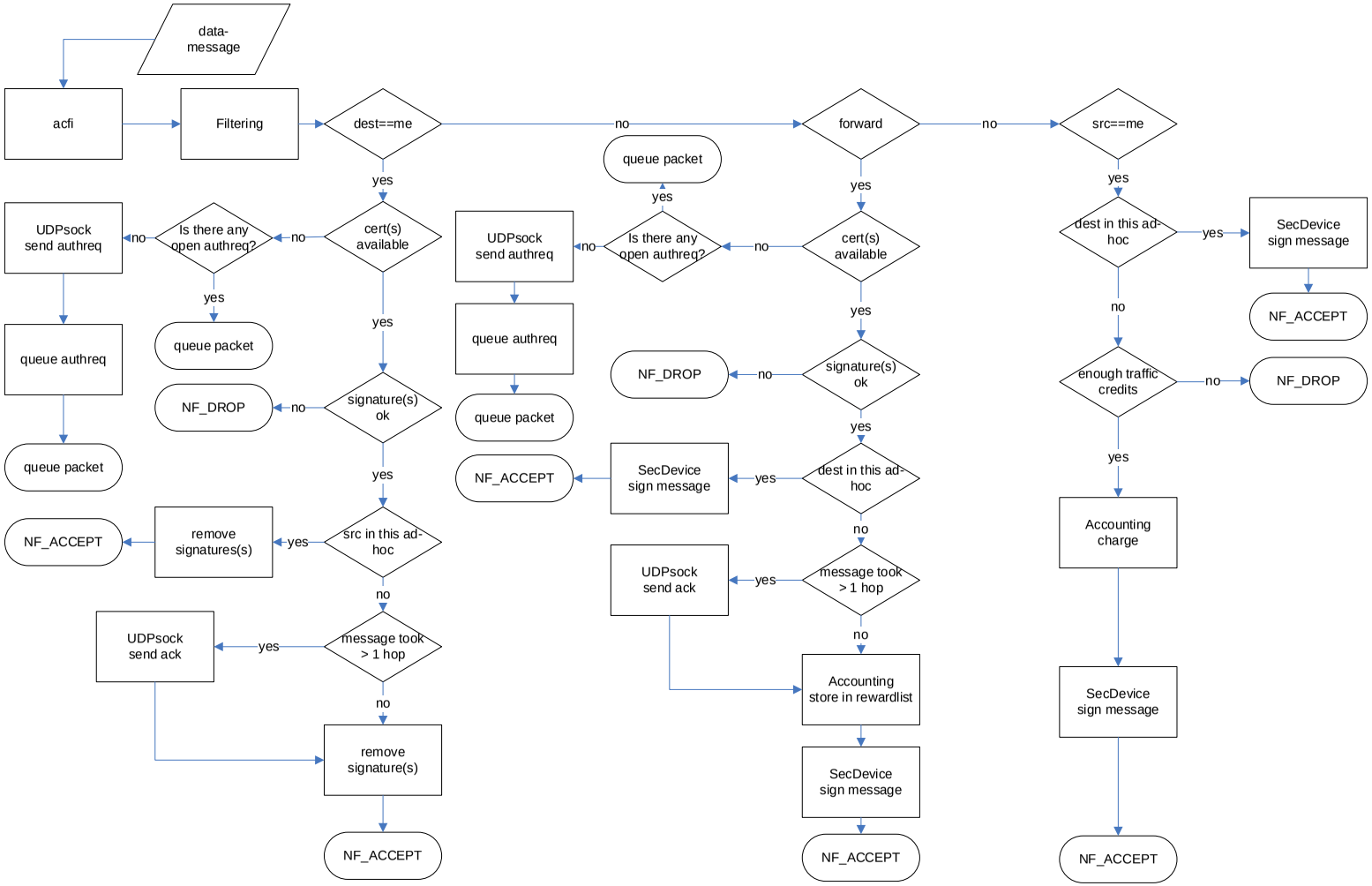


Figure 5.3: Handling of data messages in the CASHnet algorithm

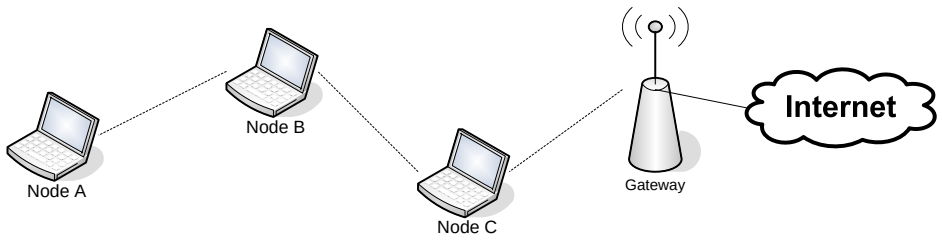


Figure 5.4: Example scenario

to the user-space. The security module on Node B is able to verify Node A's signature immediately as Node B already has this certificate. Furthermore, Node B stores the packet's message ID and its source address (= Node A's IP) in its reward list in the accounting module. Afterwards it adds its ID, a nonce and the signature over the whole packet it received from A including its own ID and nonce. Then, the NF_ACCEPT verdict is set and the packet comes to Node C. C is able to verify and remove B's signature. But as C does not have Node A's certificate, it has to queue the packet and to generate an AUTHREQ using the UDPsock module. This AUTHREQ message is addressed to A sent via Node B. B snoops the message and recognizes, that it has the needed certificate. Therefore, B drops the AUTHREQ and asks the UDPsock module to generate an AUTH containing A's certificate. Node C receives this AUTH and releases the queued packet. Now, Node C's security module is able to verify Node A's signature. Furthermore, C has to generate an ACK using the UDPsock module addressed to B, which includes the packet's message ID, the source address and B's signature over the packet. Following, C replaces Node B's ID, nonce and signature with its own. Furthermore, C has to store the packet's message ID, the source address and its own signature over the packet in the reward-list maintained by the accounting module. Afterwards C sets the NF_ACCEPT verdict. Last, the gateway receives the packet. There is no problem for the gateway to verify Node C's signature but the signature of Node A cannot be checked due to the missing certificate. Therefore, the gateway sends an AUTHREQ towards Node A via C and B. Node C, which snoops this AUTHREQ will drop it and generates an AUTH containing Node A's certificate. Afterwards, the gateway can also verify Node A's signature. Then, the gateway generates an ACK containing the packet's message ID, its source address and C's signature over the packet. This ACK has to be addressed to C. Following, the gateway removes all the signatures, IDs and nonce and sends the packet into the Internet towards the destination node.

When Node B and C receive their ACKs, it is the task of the accounting module to book it. First, the module verifies the signature with the help

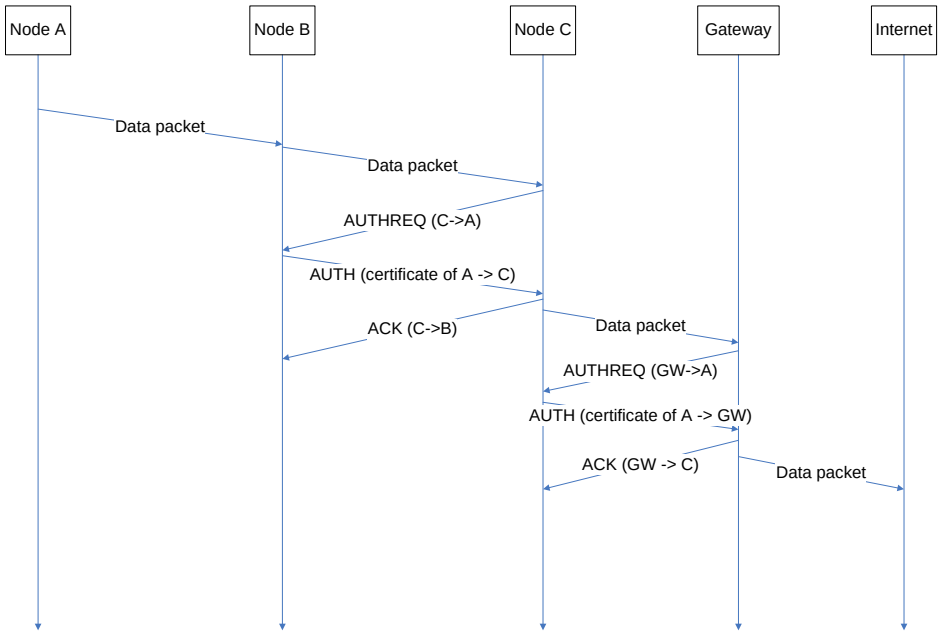


Figure 5.5: Path-time diagram of the exchanged messages in the example scenario

of the security module. This works without delay since all the nodes know their neighbor's certificate. Furthermore, the matching entry in the reward list has to be searched and the helper credits have to be increased by one.

Table 5.2 shows the amount of Traffic and Helper Credits and the available certificates after the packet has been successfully transmitted.

Node	Traffic Credits	Helper Credits	Certificate for
A	7	0	B
B	3	6	A,C
C	5	4	B,Gateway,A
Gateway	-	-	C,A

Table 5.2: Amount of Credits and Certificates on the Nodes after the Transmission

5.2 CASHnet PDUs

The security mechanisms in the CASHnet scheme require an exchange of the certificates. This can be done either by periodical announcements or

by requests. To implement these features, a new message format had to be defined. As shown in Figure 5.6 such an authentication messages consists of a 16 bit type field, which indicates, whether it is an authentication request (AUTHREQ) or an authentication reply (AUTH). Furthermore this message contains a certificate. In case of an AUTHREQ, the message contains the originator's certificate. If it is an AUTH, the certificate belongs to the AUTH's sender.

As we used a RSA key of 1024 bit and own generated certificates in our prototype, the certificate consists of 1216 bit

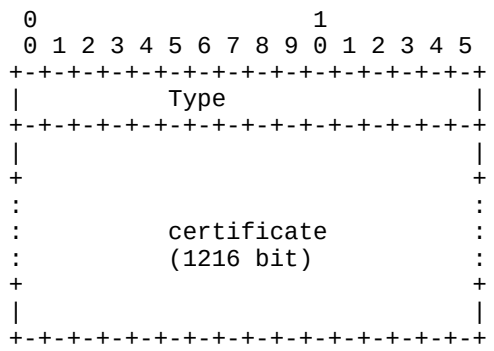


Figure 5.6: CASHnet authentication messages

Furthermore due to the charging, CASHnet requires data packets to be acknowledged. This leads to a new message format for such acknowledgment messages (ACKs). Figure 5.7 shows this new format. Like the authentication messages, an ACK consists of a 16 bit type field. Furthermore, there is a 32 bit field containing the Node ID of the node which generates this ACK. Afterwards a 32 bit nonce has to be set to avoid replay attacks. The 32 bit saddr and 16 bit MSG ID are the source address and message ID of the data packet to acknowledge. This is needed to identify the data packet uniquely, which is needed to reward the forwarding node. The whole packet and the signature of the data packet to acknowledge are signed by the originator of this ACK.

Both, authentication and acknowledgment messages are sent as payload of a normal UDP message using a special destination port.

5.3 CASHnet components

In this section, the CASHnet components according to Figure 5.8 are described.

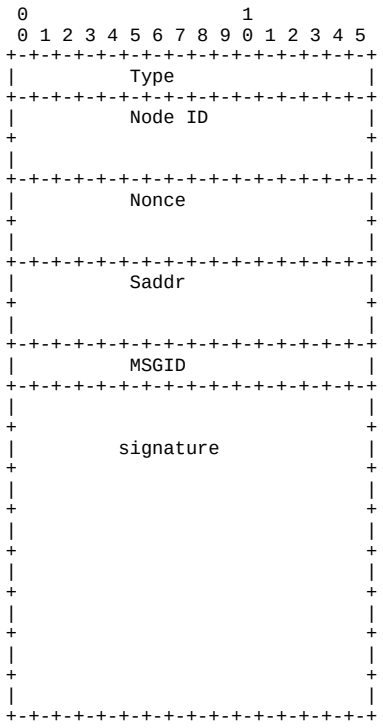


Figure 5.7: CASHnet acknowledgment

The accounting module

The accounting class ensures the accounting according to the CASHnet scheme. This means, that one has to pay for every packet which leaves the ad-hoc net. Therefore one Traffic Credit has to be paid for every hop in the ad-hoc network which is done by the generated packet. If a packet is forwarded, one Helper Credit is gained. Therefore a footprint of every forwarded packet is stored in the reward-list. If a matching ACK arrives, this entry has to be deleted. Furthermore, the accounting module decides whether a node has enough Traffic Credits or not.

In the CASHnet algorithm, the accounting is hop-based, which means, that the hop-count to the destination or the gateway is needed. As CASHnet assumes a routing protocol, which sets the metric in the routing table, this can be read out of this table. To do so `/proc/net/route` is opened and browsed for the correct wireless interface specified in the configuration file.

A node has two possibilities to get more Traffic Credits:

1. Exchange Helper Credits into Traffic Credits
2. Buy new Traffic Credits

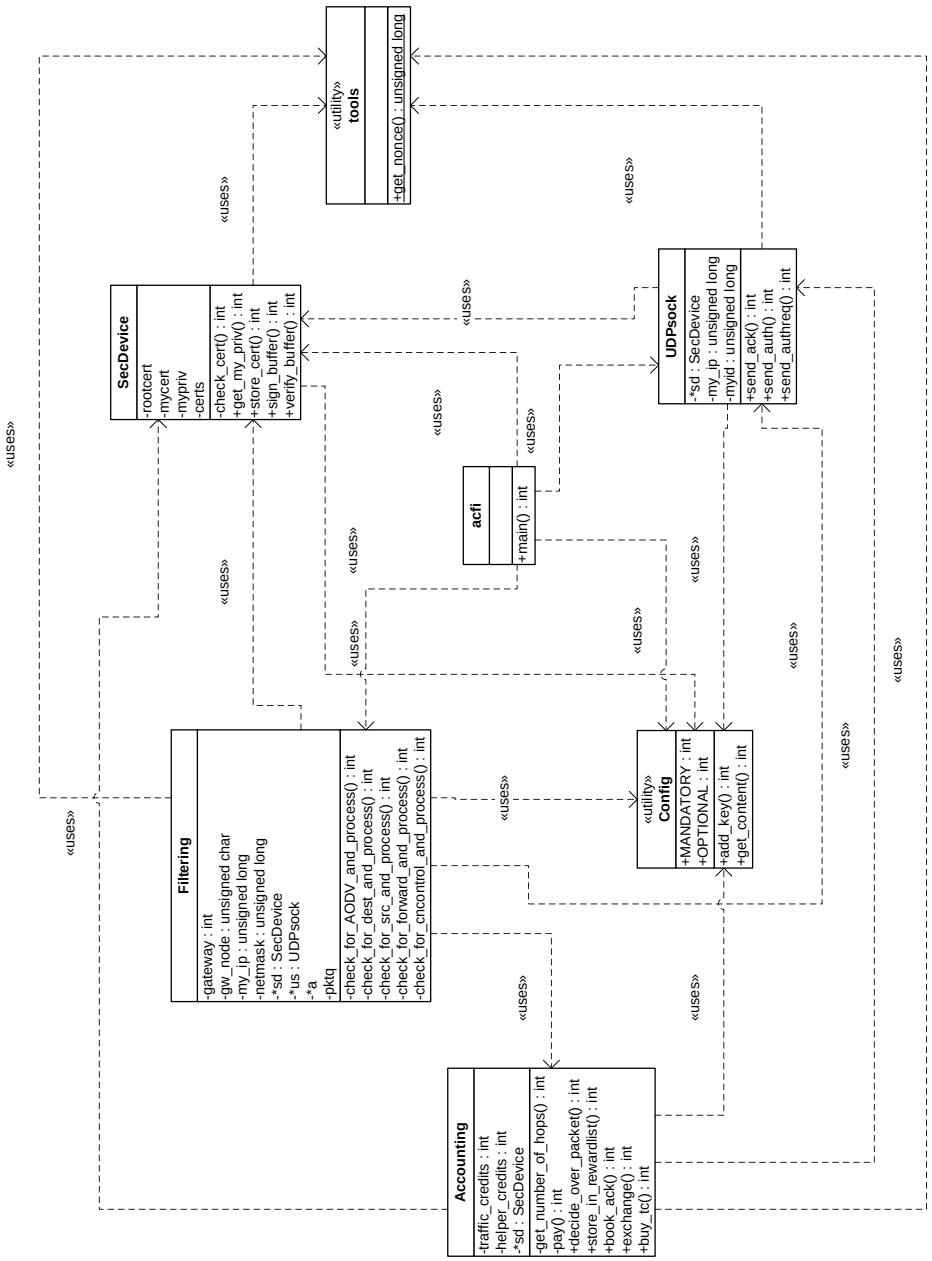


Figure 5.8: UML diagram

Our CASHnet implementation supports both.

Accounting also has to know UDPsock since it is possible that an ACK arrives which cannot be verified due to a missing certificate. If this happens, this ACK has to be queued and an AUTHREQ has to be sent.

The accounts for traffic and helper credits should be stored on a tamper resistant device. In our prototype, these accounts are stored as simple counters in the program.

UDPsock

The module called UDPsock opens an UDP socket and handles CASHnet control traffic, which consist of authentication request (AUTHREQ), authentication reply (AUTH) and acknowledgment (ACK). UDPsock is mainly used for the sending of control messages, since receiving is handled with the `filtering` module. This is faster since they come to the `filtering` module before they reach the UDP socket.

As ACKs have to be signed and authentication messages contain certificates, UDPsock has to know the security device.

The security device

This class provides methods to sign and verify buffers. It also stores the own certificate and private key, and the certificates.

Due to the UDP problems mentioned in section 5.4, an optimized certificate structure was needed. Therefore, we built an own one. This certificate contains the owner's id, the certificate's lifetime, the owner's public key and is signed using the root key.

The filtering module

Filtering is the class which is responsible for the interaction between Netfilter and our program. The CASHnet accounting and security mechanism requires some modification on the IP packets. First of all the sender's ID has to be added. Furthermore, a nonce is required which should avoid replay attacks. Last, the whole packet (including the ID and nonce) has to be signed. The first forwarding node also adds its ID and nonce and signs the whole packet. The next forwarding nodes have to replace the signature, ID and nonce of the last forwarding node with their own. Therefore, a packet has at maximum two additional IDs, two nonce and two signatures.

We defined a structure called ATT_BLOCK:

```
struct _ATT_BLOCKs {
  u_int32_t pattern;
  u_int32_t id;                 // this is the IP
  u_int32_t nonce;
  u_int8_t orig_proto;          // the original protocol number
  unsigned char flags;
  char sig[MAX_SIGNATURE_LEN];  // signature
} ;
```

This structure contains the information which has to be appended to a packet in our CASHnet: the ID of the actual node, a nonce and the signature over the ID, nonce and the whole payload of the incoming packet. Furthermore, this structure contains the protocol number of the original packet since we tunnel packets using protocol 250, a flag which indicates whether a packet comes directly from its sender or from a forwarding node (this is needed for the signing).

The tunneling was needed since TCP packets are modified which would lead to corrupted TCP frames on the network. Therefore an own protocol has been defined.

Furthermore, a packet queue has been defined, which contains all the packets which cannot be verified due to missing certificates. Afterwards an authentication request has been sent. When the reply arrives, the packets are released and processed.

Filtering is the most powerful class in our program. It handles the packets which pass our hooks. There are three cases if a data packet arrives:

1. A packet is addressed to us
2. A packet is originated by us
3. We only forward a packet

As the processing of these data packets depends on the locations of sender and receiver, Filtering has to know the actual node's netmask.

It is also possible to receive AODV or CASHnet control messages. AODV messages are always accepted since it has to be ensured, that the routing protocol is able to work. CASHnet control messages are processed and dropped if they are addressed to us, accepted otherwise. This means, that a CASHnet control message will never arrive at the specified UDP port even if it is addressed to the actual node.

The main class

As shown in Figure 5.8, this class contains the `main` method. Furthermore, the Security Device (SecDevice), Filtering and UDPsock are initialized. In this class the configuration files (see Section A) are read to configure CASHnet. Furthermore this class listens for any incoming packet or signal on the different file descriptors. There are three file descriptors, which have to be monitored. First, there is the user-space queue of Netfilter. If there is an incoming packet on this queue, the process method of the Filtering module has to be called. Second, there is the UDP socket opened by the UDPsock module. A packet on this file descriptor causes a call of the process method of UDPsock. Third, it is possible, that an user or another application writes something to standard-in which should be process by CASHnet. In the first

prototype, this option has been used to build a small graphical user interface for demonstration purposes. Furthermore, there are some tasks which should be done periodically. There are two periodical tasks. First, all the queues in the program have to be cleaned periodical. This is necessary to avoid the needless allocation of memory. Second, the CASHnet requires the periodical sending of AUTH message to all the one-hop neighbors.

Helper modules

There are also two helper modules, which are called tools and Config. The tools module provides some methods for logging.

Furthermore, it was desired to configure CASHnet using configuration files instead of hardcode the option. To integrate these configuration files into CASHnet we introduced a helper class called Config. A configuration file build conforming to this class can consist of mandatory and optional options. These options are called keys and their content is the value we want to assign to that option.

Netfilter Hooks

As we want to process data packets in the user space, every incoming and outgoing packet has to be redirected using the Netfilter API. Every Node which wants to participate in a CASHnet enabled MANET has to execute a script which sets the rules to send every packet on the wireless ad-hoc interface into the user space queue. A sample listing of these rules is given in Section 4.2 on page 24. In the user space, the program takes packet per packet out of the queue, processes it and sets a verdict, which tells the kernel what to do with this packet.

5.4 Review of the CASHnet scheme

5.4.1 Improvements

First reviews of the CASHnet scheme showed some improvement which will be explained in the following paragraphs.

Snooping AUTHREQs First, the original CASHnet only allows AUTHs to be sent by the destination of an AUTHREQ. Furthermore these AUTHs have to be signed. We decided to allow to snoop AUTHREQs and reply to them if possible. Snooping means that a node analyzes forwarded AUTHREQs for their destination. If the destination's certificate is available on the node, it drops the AUTHREQ and sends an AUTH. This will reduce the amount of CASHnet control messages in the network and reduces the time between sending an AUTHREQ and receiving an AUTH. Assuming a

scenario, were Node A wants the certificate of Node C and communicates over Node B. B has already the certificate of Node C. Node A will generate an AUTHREQ addressed to Node C. Node B, which forwards the messages recognized that A wants to have a certificate, B already has. Therefore, instead of forwarding the message to Node C, B generates an AUTH containing C's certificate. Since certificates are signed by a root authority, this feature does not introduce insecurity. Furthermore, as shown in the evaluation Chapter 6.2.4, the snooping leads to significant improvements.

AUTH on demand Second, the initial CASHnet approach requires a periodical sending of the AUTHs. This has to be done using a broadcast with a TTL of one and should ensure that every node has the correct certificates of all its one-hop neighbors for the whole time, it participates in the MANET. This feature causes a huge network load (for details, see Chapter 6). Furthermore, as it is usually desired to have reactive protocols in MANETs to save energy, it would be a disadvantage of CASHnet to be active. Therefore, we decided, to send AUTHs only on demand. Furthermore due to the snooping of AUTHREQs the time between sending an AUTHREQ and receiving and AUTH decreases significantly (see Chapter 6.2.4). Hence, there is no need anymore to send certificates before they are needed.

5.4.2 Problems

Furthermore, during the implementation process, several problems arose which will be described in the following paragraphs.

Fragmentation First of all, CASHnet requires the signing of every packet. This leads to fragmentation problems if we do not limit the packet size for the applications. Netfilter allows to limit the maximum segment size for outgoing TCP messages, but there is no possibility to do this for UDP. Because of this, our implementation only works for TCP or small UDP packets.

Signed traffic in the Internet Furthermore, the original CASHnet required signed traffic over the Internet in case sender and receiver are located in different MANETs (see Figure 3.8). If we want to realize this feature, the gateway in the sending node's MANET has to know all the other existing MANETs to check if the receiver is located in one of them. But as the idea behind MANETs is that they are built spontaneously, it would be probably not possible to know all of them. Therefore, we decided to send unsigned traffic over the Internet independent from the receiver's location.

AODV-UU tunneling CASHnet required a routing protocol which provides information about the hop-count. AODV-UU fulfills this requirement. But as mentioned in Section 4.3, when a node wants to send a packet out of the actual MANET, AODV-UU tunnels the packet to the gateway using an own protocol with number 55. This is needed because of the multiple gateway support. If they would not use tunnels and a node changes the gateway during a transmission, the session would break due to the missing NAT entries on the new gateway. Therefore, they establish a tunnel to one gateway and use it for the whole transmission. This leads to problems when signing the packets. On the sender, a normal IP packet will be signed. But the forwarding nodes will try to verify the tunneled packets, which must fail. Therefore, AODV-UU has not been used for tests, where we sent messages out of the MANET.

Authentication on connection setup Further, the CASHnet theory as written in [19] requires a sender to send AUTHs before starting a transmission to another node. This feature requires the nodes to listen for connection setup messages. In case such a message has been identified, it has to be queued and the authentication process has to be started. This would work for TCP as this protocol is connection oriented. But if an application uses UDP, there are no connection setup messages. Therefore, this feature has not been implemented. As can be seen in Chapter 6, authentication is very fast even without this feature.

Matching between acknowledgments and reward-list entries In [19], it is written, that a forwarding node saves the next hop's identity and the actually computed signature of the forwarded message in a reward-list as these attributes will assign an incoming ACK uniquely. In our implementation we had the problem that we were not able to know the next node's identity as a node does not know anything about the topology. Therefore we decided to store the packet's source address and the packet's message ID instead of the next node's identity. If the next node generates an ACK, it also includes these two attributes, which will identify an entry in the reward-list uniquely.

CASHnet packet format Last, in the CASHnet theory, a node inserts its ID and nonce before the TCP/UDP payload and appends the signature at the end of the packet. Since this leads to a destruction of the original TCP/UDP packet, we decided to append all these attributes at the end of a TCP/UDP packet. By this, we only modify the IP packet. Furthermore, we assign our own protocol number to this modified packet. Therefore, we avoid the destruction of the original packet by tunneling.

Chapter 6

Evaluation of the CASHnet implementation

In this chapter, it will be described how our CASHnet implementation has been evaluated. A testbed has been built which consists of several wireless nodes and one wired node which emulates the Internet. Furthermore several scenarios using different versions of the CASHnet implementation have been tested.

6.1 The Testbed

The testbed consists of seven test machines, which have been used in three different compositions. The first scenario is shown in Figure 6.1. We used the three notebooks Node A, B and C and the desktop PC Node I, which emulates the wired backbone. Node C acts as a gateway between the MANET with the ESSID "truemobile" and the backbone. Therefore, it has two IPs.

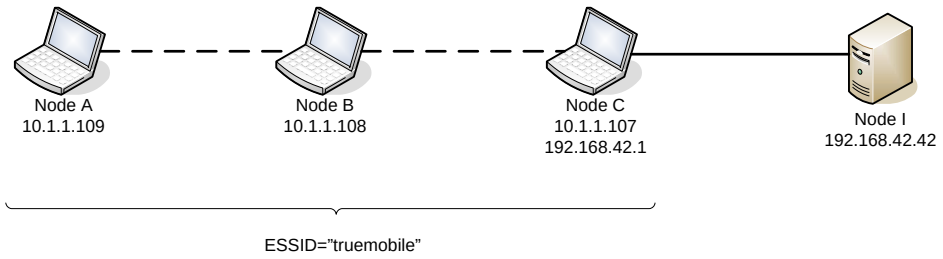


Figure 6.1: Testbed consisting of 3 mobile nodes and one desktop PC

Node A and B are Dell Latitude 100l notebook with an Intel Celeron CPU with 2.4GHz and 128KB cache. They contain a Broadcom Corporation BCM94306 802.11g wireless network controller. Node C is a Dell Latitude

D600 notebook with an Intel Pentium M processor with 1.4GHz and 1024 KB cache. It consists of a Broadcom Corporation BCM94306 802.11g wireless networks controller. As all the notebooks are also connected to the Internet (for remote programming) we equipped Node C with an additional 3Com Corporation 3c575 PCMCIA Ethernet card to connect it to Node I. Node I is an Ultra Sparc 5 with an TI UltraSparc IIi cpu. It has two Ethernet cards, one Sun Microsystems Computer Corp. Happy Meal, which is connected to Node C in this scenario and one Realtek Semiconductor Co., Ltd. RTL-8139 which will be used later. Node I and C are connected via a crossover cable and Node I has no connection to the Internet. On Node I we have installed a Debian Linux [25] version 3.0 ('woody') with kernel version 2.4.18. As Node I should only work as backbone, no special configuration is needed. The three notebooks run Slackware Linux [26] version 10.1.0 with kernel 2.6.8. We decided to use a Linux with small footprint which allows a suitable backup solution. For Node I Debian Linux was the easiest way. We put the machines side by side on a table which would cause a MANET topology where everybody sees all the other participants. Therefore we had to use a MAC-filter which helps us constructing more interesting topologies. We filtered the MAC addresses using Netfilter. To do so, we wrote an init script consisting of commands like `/usr/sbin/iptables -I INPUT -m mac --mac-source <MAC address> -j DROP`. Using these scripts, we built the topology shown in Figure 6.1 - 6.3.

All the notebooks run `iptables` version 1.2.11 and AODV-UU version 0.9 as routing protocol. Although a newer AODV-UU version was available when we started testing, we had to use the old version since the new caused failures when we used gateway support.

The first scenario with three nodes was very useful to test AODV-UU and the first CASHnet versions. But as CASHnet requires special signing on the forwarding nodes, which means, that a second forwarding node has to replace the signature of the first one with its own, we needed a second forwarding node to show and test this functionality. Therefore the testbed has been extended by one additional mobile node (see Figure 6.2).

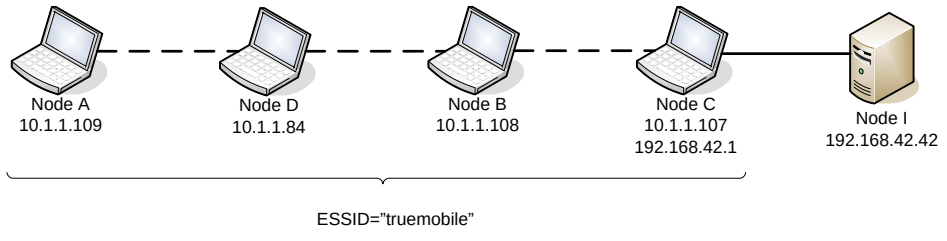


Figure 6.2: Testbed consisting of 4 nodes

Notebook D is an IBM Thinkpad T43 with an Intel Pentium M processor with 1.86 GHz and 2048KB cache. Like Node C, D consists of a Broadcom Corporation BCM94306 802.11g wireless networks controller. It runs the same Linux installation as the other notebooks.

Furthermore, we had to test the whole accounting functionality of CASHnet which requires a second MANET. Therefore we used another two notebooks Node E and F as shown in Figure 6.3.

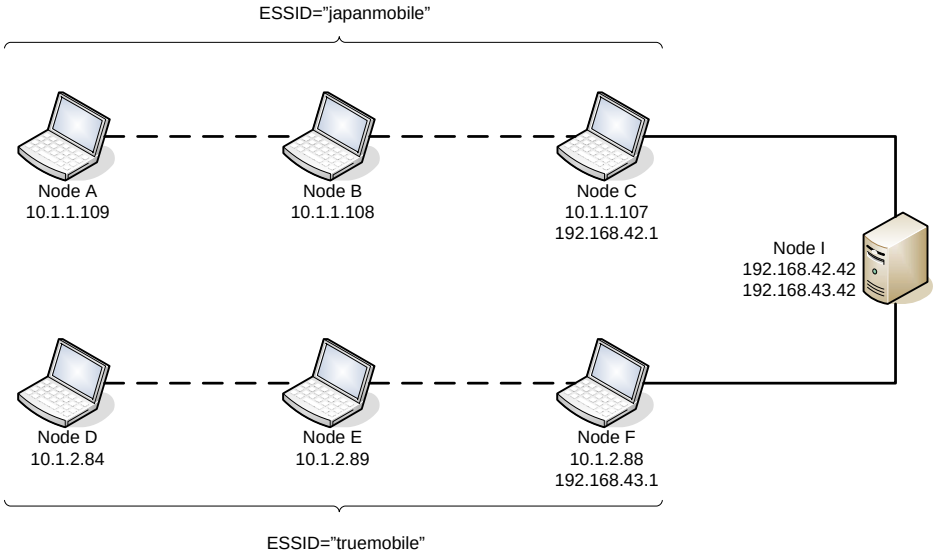


Figure 6.3: Testbed consisting of 6 nodes

Node E is a Dell Latitude C610 notebook with an Intel Pentium III Mobile processor with 1.2GHz and 512KB cache. As it was not possible to install the internal wireless card under Linux without flashing the firmware, we decided to use an external Level One WPC-0300 PCMCIA wireless card. Node F is a Dell Latitude D505 with an Intel Pentium M processor with 1.7GHz and a cache of 2048KB. It consists of an Intel PRO/Wireless 2200BG wireless controller and an additional 3Com 3c589 PCMCIA Ethernet adapter which is connected to Node I via a crossover cable.

We also had to reorganize the network. Now, the first MANET with ESSID "japanmobile" consists of Node A, B and C where Node C is the gateway. The second MANET has the ESSID "truemobile" and consists of Node D, E and F with Node F as gateway node.

As written in Section 4.3, AODV-UU establishes tunnels if a packet leaves a MANET. This makes the signing inside CASHnet problematic since we first sign a normal packet and then try to verify the tunneled packet which

leads to wrong results. We found no way to avoid this, so we decided to use static routes in the scenario with two MANETs. As CASHnet requires the routing protocol to set the metric, we also have to set the metric for the routes inside the MANET. This can be done using the route command as follows: `/sbin/route add -host <ip address> metric <number of hops> <interface>`.

6.2 Test Scenarios and Results

In this section, the test scenarios will be explained. Furthermore the results will be shown.

First of all we will show the general CASHnet overhead and afterwards provide a more detailed analysis of the reasons for this overhead.

6.2.1 Packet Delays

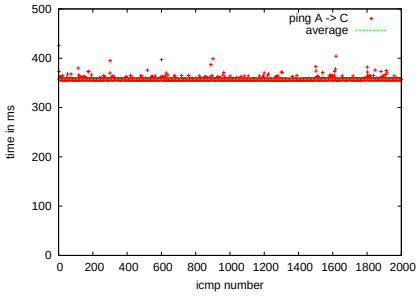
To determine the CASHnet overhead, we assumed a scenario with four wireless nodes as shown in Figure 6.2 .

For the measurements, `ping` has been chosen to determine the additional delay introduced by CASHnet. Therefore we sent 2000 pings from Node A to Node C. A packet loss of 0% has been achieved.

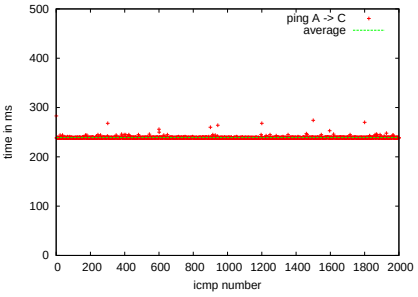
CASHnet requires a significant amount of additional computation which leads to a comparably high delay when pinging over two hops as shown in Figure 6.4(a). The average round trip time is 357ms.

As can be seen in Figure 6.4(b) and 6.4(c), this huge overhead is mainly caused by the signing of the packets. If we send unsigned ACKs, the average time is about 239ms, which is a reduction of more than 30%. If we avoid any signing, we get an average of 3ms which is very close to the setup without CASHnet. But even if the signing causes this huge delay, we cannot use CASHnet without any security mechanism since this would lead to misuse of the accounting scheme. Therefore, we have to think about methods to weaken this delay. First we could reduce the RSA key length. Till now, we use a key length of 1024 bit which is very long. As CASHnet requires short lifetimes for the certificates [19], it should be possible to reduce the key length without getting an insecure CASHnet. In Section 6.3, we will show the influence of the key length.

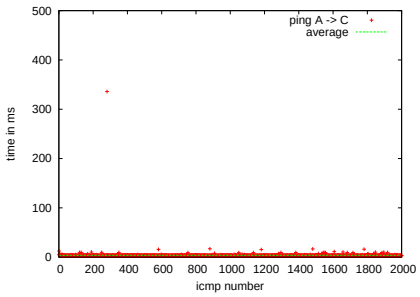
Second, we tried to reduce the delay by reducing the number of ACKs. In Figures 6.5(a) - 6.5(d) we show the ping delay when we do not acknowledge every data packet but every fifth, tenth, fifteenth and twentieth. In this case, a forwarding node is not rewarded with one helper credit, but with five, ten, fifteen or twenty credits. Doing so, we can reduce the delay by 30%. But there is no significant difference between a threshold of five or twenty. With a threshold of five, we get an average round trip time of 259ms, with twenty we have 245ms. From these measurements, we can gather that the ACK



(a) complete functionality



(b) unsigned ACKs



(c) no security mechanisms

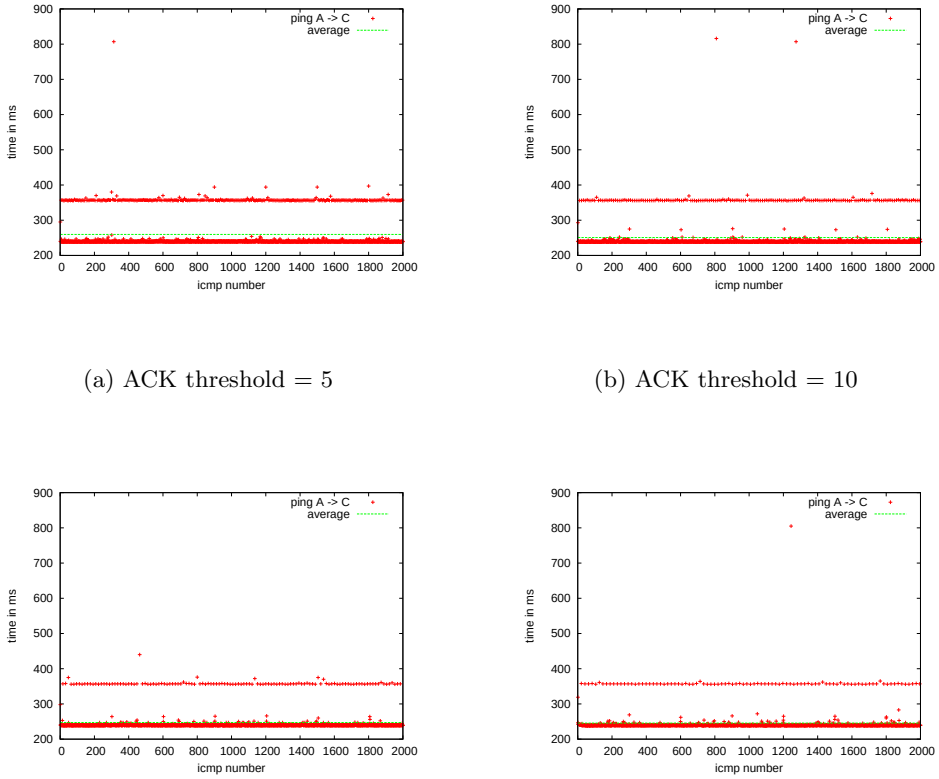
Figure 6.4: round trip time in a CASHnet enabled MANET

messages do not have a huge impact on the delay even if there is an obvious influence.

We also measured the delay introduced by the AODV routing protocol to ensure that we get no bad results caused by this protocol. As shown in Figure 6.6, the delay is pretty good. There are higher values in the beginning because the sender has no route to the destination yet. Afterwards the delays are becoming smaller and are near to the optimum.

Conclusion

The measurements with ping have shown that CASHnet adds a huge additional delay, which is mainly caused by the security mechanisms. The acknowledgment process which is a must for the accounting does not highly



(a) ACK threshold = 5

(b) ACK threshold = 10

(c) ACK threshold = 15

(d) ACK threshold = 20

Figure 6.5: round trip time in a CASHnet enabled MANET

influence the delay.

6.2.2 Processing time

We also wanted to show the time needed for the processing of the different types of packets on the nodes. We experienced that the processing time depends extremely on the processing power of the different nodes. As test scenario we used the one with four nodes as shown in Figure 6.2.

Time needed to process acknowledgment messages

In this section we will determine the time needed to process an ACK. First of all, we had a look at the incoming ACKs, which are generated by other

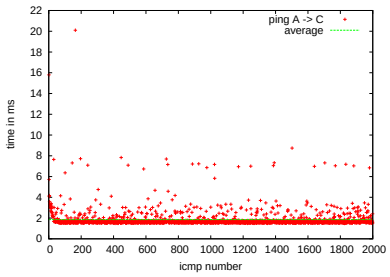


Figure 6.6: Delay introduced by AODV

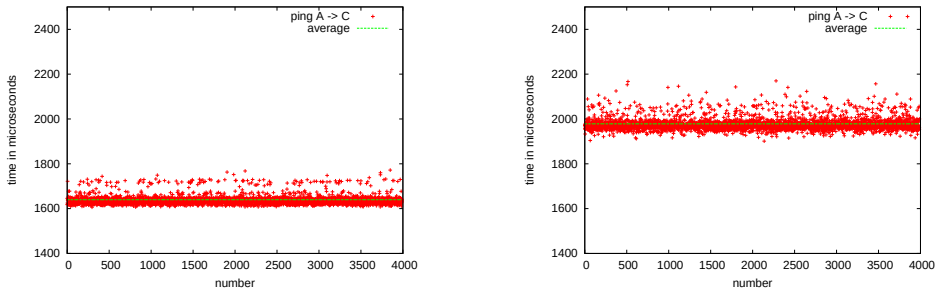
nodes to reward us. As ACKs are only generated to reward somebody, we only have to analyze the forwarding nodes.

If a node receives an ACK, it has to check whether it is addressed to it or not which is done using a simple comparison of the IP addresses in an if-condition. Afterwards we check for the needed certificate which is done by a linear search in a list. If the certificate is available, we check the signature by building a new hash over the packet and comparing it with the encrypted one. Afterwards we search for the matching data packet footprint in the reward-list, which can be done in linear time. If we do not have the certificate, we have to walk through the queue containing the open AUTHREQs if there is the need to send one and only if we do not find any, we generate a new AUTHREQ, send and queue it. The search over the queue can also be done in linear time. At the end, we set a verdict or queue the message.

For the measurements, we took the whole time an ACK was processed, which means that we summed up the time till a message was queued if we had no certificated and the verification and accounting time if the certificate arrived.

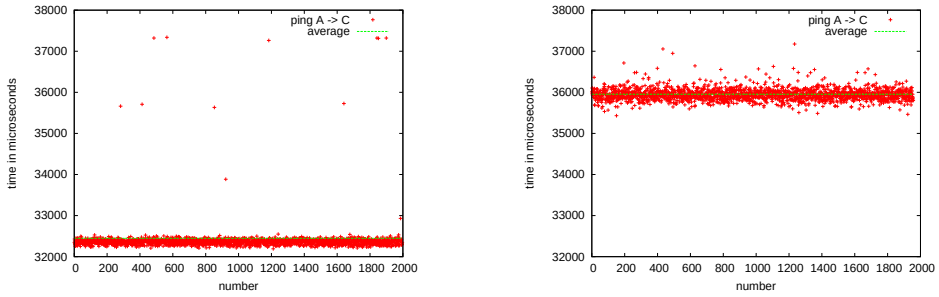
Figure 6.7(a) and 6.7(b) show the processing time on Node D and B. We can see that there is significant difference between these two nodes, which can be explained by the processing power on the two nodes. Node D is the faster IBM machine. Anyway, the processing times are not negligible since this has a significant influence on the round trip time as can be seen in Section 6.2.1. But as written in Section 6.2.1, there is a strong need to sign ACKs and the only possibililty is to reduce their number.

Figure 6.8(a) and 6.8(b) have shown the time needed to process own generated ACKs. Here we have to build a helper structure containing the last node's signature of the data message to be acknowledged and calculate the signature over this structure which is very expensive. Afterwards we



(a) Node D (b) Node B

Figure 6.7: processing time for received ACKs



(a) Node D (b) Node B

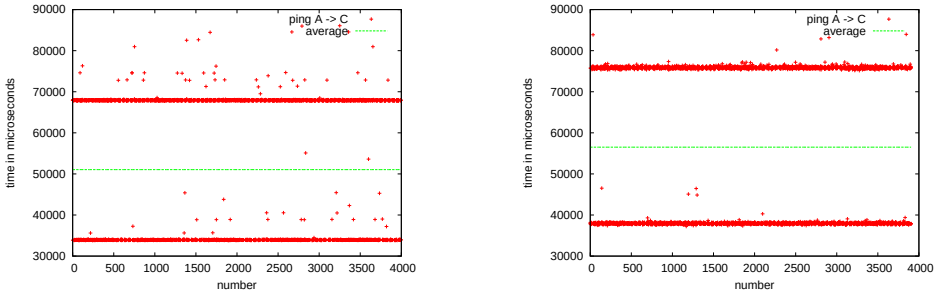
Figure 6.8: processing time for self generated ACKs

build an ACK message and send it using UDP. Most of the delay shown in the figures is caused by the signing using RSA. These measurements also make clear that there is a strong need for a reduced number of ACK messages otherwise the nodes have much work with signing ACKs.

Time needed to forward a data packet

Now, we want to analyze the processing time of forwarded packets on the two forwarding nodes Node D and Node B. We used also the testbed with four nodes as shown in Figure 6.2.

To understand the results of these measurements, we first have to explain what a node has to do if it forwards a message. It first has to check for the needed certificates and send an AUTHREQ if there are some certificates missing. Afterwards it has to check all the signatures included in a message. Last, it has to send an ACK if needed and store a footprint of the data message in its reward-list. The part of the ACK has already been measured and shown in Section 6.2.2.



(a) Node D

(b) Node B

Figure 6.9: processing time for a forwarded packet

As can be seen in Figures 6.9(a) and 6.9(b), we have a dichotomy of the results. There is simple explanation for this phenomenon. If a node is the first one in a forwarding chain, it only has to verify the originator's signature and to append its own. There is no need for sending an ACK or additional verifications. But if a node is the second or a later forwarding node, it has to verify two signatures, append its own one and to generate a signed ACK, which takes much more time than only verifying and signing once. In our scenario, Node D is the first forwarding node for the echo request messages of Node A and B is the first for the echo reply messages of Node C. Therefore we get this dichotomy. This is also a very good visualization of the huge CASHnet overhead which should be reduced.

Again we see the influence of the processing power of the nodes.

6.2.3 Messaging Overhead

In this section we want to describe the percentages of data messages, acknowledgements and authentication messages. Obviously, the goal should be to maximize the amount of data messages and minimize the other ones. For all the measurements we took the scenario with four nodes as shown in Figure 6.2. We count the messages on Node D.

Under “messaging overhead” we understand the additional amount of needed messages when using CASHnet compared to a MANET without CASHnet. This additional amount consists of the authentication and acknowledgment messages.

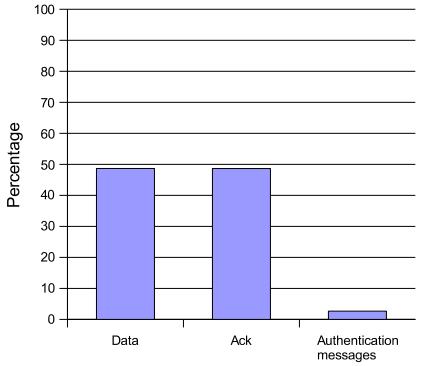
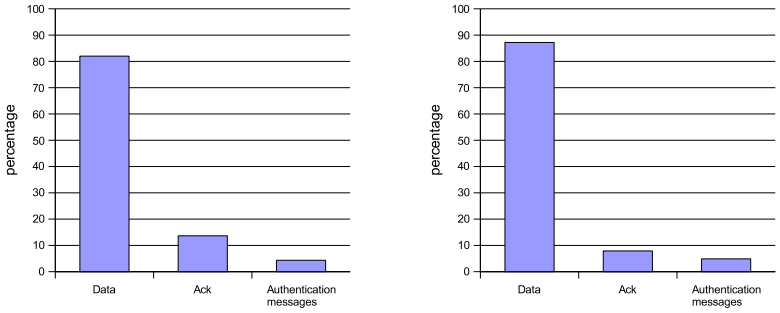


Figure 6.10: amount of the different types of messages on Node D, reference scenario

First, we made a reference measurement which should be taken as initial point for all the improvements. For this initial setup, we use an ACK threshold of one, which means that we acknowledge every data packet and send an AUTHREQ every time we need one (without queuing them). The results can be seen in Figure 6.10. We see that data messages and acknowledgments are well balanced as they both have a portion of nearly 50% of the total sum. Authentication messages are negligible as they have a percentage of less than 5%. What we can see here is that it is desirable to reduce the number of ACKs. We tried this setting an ACK threshold of five as shown in Figure 6.11(a). Doing so, we can reduce the percentage of the ACKs to nearly 15% which enlarges the percentage of the data messages to more than 80%. This are already pretty good values.

As shown in Figure 6.11(b) to 6.11(d), increasing the ACK threshold leads to better percentages until we reach a maximum of more than 90% for the data messages if we have an ACK threshold of twenty. We also achieve a percentage of the acknowledgments which is lower than the percentage of the authentication messages.

These results show us the need to reduce the amount of authentication messages too, even if they have a very low percentage. First steps have been done by snooping the AUTHREQs from other nodes and answer them with an AUTH message if the actual node has the required certificate. The results with an ACK threshold of one are shown in Figure 6.12(b). We see that comparing to Figure 6.12(a), we could halve the percentage of the



(a) ACK threshold = 5

(b) ACK threshold = 10

(c) ACK threshold = 15

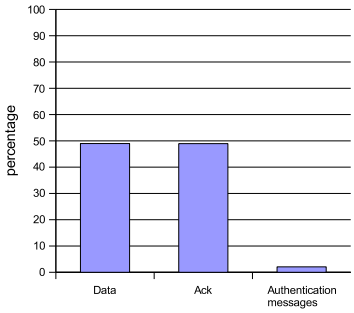
(d) ACK threshold = 20

Figure 6.11: amount of the different types of messages on Node D

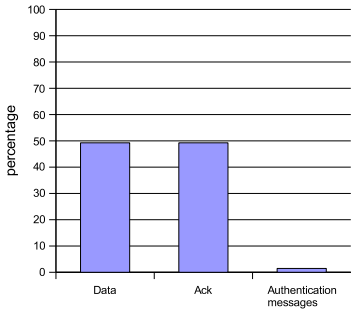
authentication messages.

In Figure 6.13(a) we show the percentages if we have an ACK threshold of one and if we queue open AUTHREQs and send our own certificate within the AUTHREQ. In the measurements above, we only sent an empty AUTHREQ which lead to an additional AUTHREQ on the forwarding and receiver nodes if they want to verify the originators signature. To avoid this additional AUTHREQ we send our own certificate in our AUTHREQ. Queuing the open AUTHREQs should avoid sending redundant AUTHREQs as done in the measurements above. In Figure 6.13(a) we see that we achieved no further improvement compared to Figure 6.12(b), but there is also no deterioration. Probably, our testbed is too small to see any improvement of these little changes.

The CASHnet theory requires the periodical sending of AUTHREQs which has also been tested. The measurements above have been done without this periodical AUTHREQs. Figure 6.13(b) shows the influence of peri-

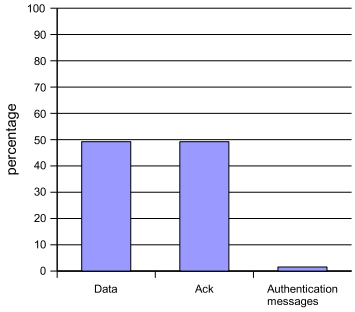


(a) no snooping of the AUTHREQ messages

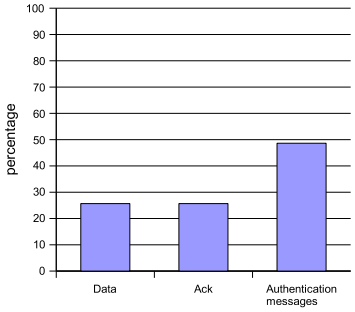


(b) snooping of the AUTHREQ messages

Figure 6.12: amount of the different message types on Node D



(a) an AUTHREQ contains the sender's certificate and gets queued



(b) AUTHREQs are sent periodical

Figure 6.13: amount of the different types of messages on Node D

odical AUTHREQs. In this scenario we send a periodical broadcast with a TTL of one every 150 seconds. As can be seen in Figure 6.13(b), this has a huge influence on the percentage since we have now a percentage of nearly 50% for the authentication messages. This shows that it would be probably better to send AUTHREQs only on demand. To avoid the usage of outdated certificates it would be a better idea to store a certificate just for a short time. If a node needs the certificate after this timeout, it can send a new AUTHREQ; otherwise there would be no need to ask for an update.

6.2.4 Response time for AUTHREQs

This section shows briefly the response time for AUTHREQs this means that we measure the time between sending an AUTHREQ and receiving the matching AUTH. For these measurements we used the scenario with four nodes as shown in Figure 6.2.

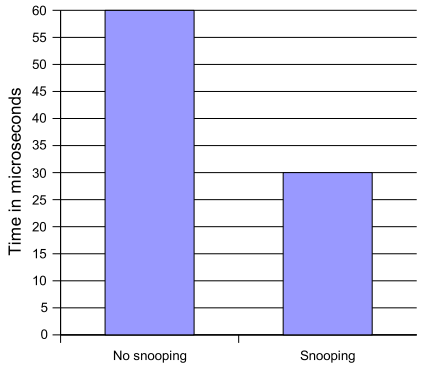


Figure 6.14: Time between sending an AUTHREQ and receiving the matching AUTH

As shown in Figure 6.14, in our reference setup we get a mean response time of 0.06ms. To improve this, we snoop AUTHREQs on every node and reply with an AUTH if possible. In the worst case, the destination of the AUTHREQ has to answer, otherwise a node on the forwarding chain is able to reply which is much faster. As shown in Figure 6.14 we can reduce the response time to nearly 50% of the reference scenario.

6.2.5 Further CASHnet properties

This section deals with further CASHnet properties which have to be mentioned for the sake of completeness. First of all we also check the state of the Netfilter user space queue to check if CASHnet processes the packets fast enough. Otherwise this could cause congestion. This queue has a

maximum size of 1024 packets and can be made longer if needed. But our measurements have shown that there are maximal 2 packets in this queue if we sign every data message and every ACK and if we acknowledge every packet which is the worst case. Therefore there is no need to enlarge the packet queue.

Furthermore we checked the accounting as it is proposed in the CASHnet scheme even under mobility. In the Nuglet scheme, the pending nuglets are only synchronized over one hop which means that a node will loose its pending nuglets if it moves before the synchronization has been done. In the CASHnet scheme, there is no constraint for ACKs. They can even be forwarded over the backbone, if a node has changed the MANET. As long as the sender of an ACK has a route to the ACK's destination the rewarding will be successful. This functionality has been successfully tested with the scenario containing six mobile nodes as shown in Figure 6.3.

6.2.6 Bandwidth Measurements

When participating in a network usually a user is interested in as much bandwidth as possible. Therefore, it has been checked how CASHnet influences the available bandwidth.

For the bandwidth measurements, the setup with three notebook as shown in Figure 6.1 had been taken. A web-server had been installed on Node I and Node A and B had to download a file from there. In the meantime, the cpu load on the three notebook had been measured.

For the web-server, `thttpd` [27] has been taken. This is a small, fast and secure web-server which supports CGI, URL-traffic-based throttling and basic authentication. Furthermore it performs very well under high load, which means, that this web-server will not be the bottle-neck in our measurements. `thttpd` is available as Debian-package [27].

To download a file from the web-server, `curl` [28] has been used. `curl` is a tool to download data from a server using for instance HTTP, HTTPs, FTP and FTPs. `curl` prints statistics to stdout (for an example see Listing 6.1), which was needed to analyze the results.

Listing 6.1: curl sample output

```
bash-3.00$ curl sun/debian-31r0a-i386-binary-1.iso > debian-31r0a-i386-
    binary-1.iso
  % Total    % Received % Xferd Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  3  640M    3 19.5M    0     0   9003        0 20:42:30  0:37:55 20:04:35 12971
bash-3.00$
```

Bandwidth measurements without CASHnet

First of all, a reference scenario was needed. Therefore, the bandwidth has been measured without any CASHnet functionality. Figure 6.15 shows the results. As can be seen, the highest bandwidth can be achieved on the Node, which directly connected to the web-server (called "direct" in the figure) using a crossover cable. Then the bandwidth decimates with increasing number of hops.

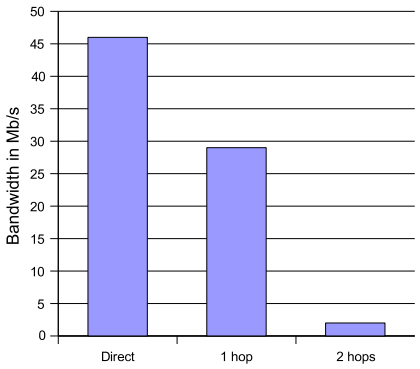


Figure 6.15: Maximal achieved bandwidth without CASHnet functionality

The results in Figure 6.15 are strongly influenced by the wireless medium. Therefore, another test without any wireless medium has been done to show the difference. Node C is connected to the web-server like before. But know, Node B and C are interconnected using a crossover cable. The leads to much higher bandwidth for one hop as shown in Figure 6.16.

This little experiment shows that the bandwidth is already significantly reduced due to the wireless medium before using any CASHnet functionality.

Furthermore, in CASHnet enabled MANETs, the TCP MSS has to be limited (see Section 5.4 on page 46). To show the influence of the limited of the MSS of 1000 Bytes, another test has been done without CASHnet. As can be seen in Figure 6.17, for one hop, this limitation reduces the bandwidth by more than 50%. But for 2 hops, the limitation's influence is negligible.

Bandwidth measurements with CASHnet

In Figure 6.18, the resulting bandwidth when using CASHnet is shown. It decreases dramatically compared to Figure 6.15, where we spoke about Mb/s whereas now, we speak about kb/s. This is caused by the processing time for packets on CASHnet enabled nodes. As shown in Section 6.2.2 in Figure 6.9(b), Node B needs about 55 ms to process a packet. Therefore,

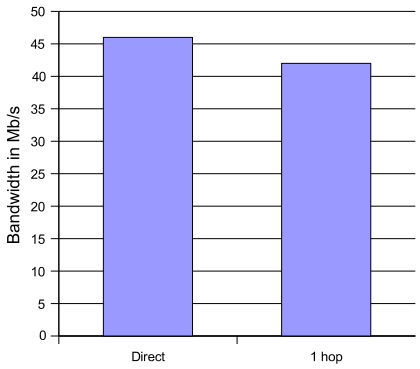


Figure 6.16: Maximal achieved bandwidth without CASHnet functionality and without any wireless connection

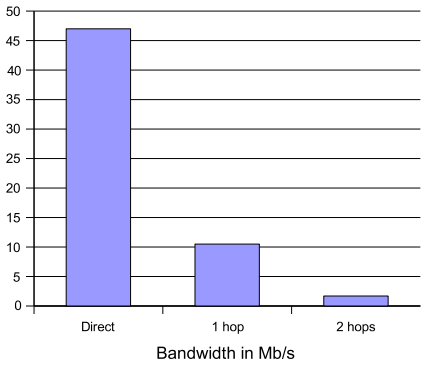


Figure 6.17: Maximal achieved bandwidth without CASHnet functionality and with a limited TCP MSS

this Node is able to process around 18 packets per second. Due to the limited MSS (see Section 5.4), one packet consists of at maximum 1000 Bytes. This leads to a maximum throughput of 144 kb/s for one hop, which corresponds to the results in Figure 6.18.

Further, some measurements with limited bandwidth have been done to show the influence of the bandwidth to the load of the nodes. Limiting the bandwidth can be done turning on the throttling option in the `thttpd` web-server and specifying the desired bandwidth. We decided to make measurements for 72, 40 and 8 kb/s where 72 kb/s is the highest available

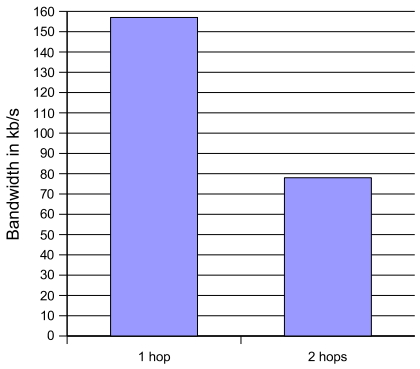


Figure 6.18: Maximal achieved bandwidth with CASHnet functionality

bandwidth over two hops in a CASHnet enabled MANET. The file on the web-server has been downloaded over two hops.

To determine the actual cpu load, `uptime` was used. This command prints the average load of the last one, five and fifteen minutes. The load determines the number of cpus which could be fully loaded by the actual system.

As shown in Figure 6.19, for 72 kb/s, the load is higher than for 40 or 8 kb/s. For 40 and 8 kb/s, the node has not reached the maximal number of processed packets yet. There are still some reserves. Furthermore, for more than 40 kb/s, the system would perform better on a two processor machine [29].

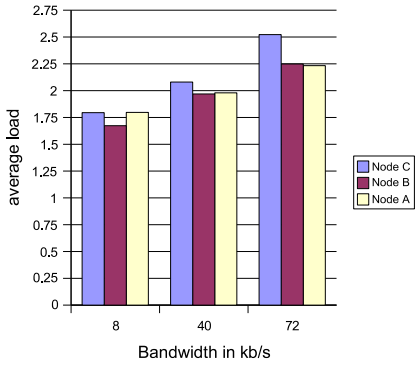


Figure 6.19: The influence of the bandwidth to the load on the nodes

6.3 The influence of the key length

In the first prototype, a key with a length of 1024 bit was used. It is clear, that this has a bad impact on the processing time on the nodes. Furthermore, there is no must for such a huge key as CASHnet requires certificates with a short lifetime [19]. Therefore, we also tested the time needed with a key length of 512 bit. We only measured the time, the RSAREF library needs to calculate a MD5 message digest using keys with two different lengths as this will suffice to estimate the influence on the CASHnet algorithm.

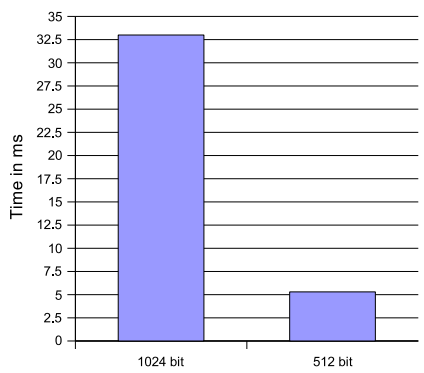


Figure 6.20: Time needed to calculate a MD5 message digest using different key lengths

Figure 6.20 shows the results of these measurements. Using a key of 512 bit reduce the time to the sixth part of the time with a key of 1024 bit. Therefore, the processing time on the forwarding nodes will be reduced to approximately 11ms per node. This leads to an available bandwidth of about 800 kb/s, which is far away from the bandwidth without CASHnet as shown in Section 6.2.6. These results show that there is no easy way to reduce the huge delay introduced by CASHnet as proposed in [19].

6.4 Evaluation of the smart cards

The most popular smart cards are the cards provided by Axalto [30]. There are especially two types of cards we tested for use with CASHnet: the Cyberflex Access card and the Cryptoflex card. The first type has been evaluated in [31]. The Cryptoflex cards have been evaluated during this diploma thesis.

Cyberflex Access Cards Cyberflex Access smart cards from Axalto [30] are ISO7816 compliant smart cards which are programmable using JavaCard [32]. The cards consist of a 32 or 64 Kbytes EEPROM and have a cryptographic co-processor. They support RSA, DES, 3-DES and SHA-1 hashing [33]. Furthermore there runs a JavaCard Virtual Machine on the card which allows the execution JavaCard applets.

JavaCard is subset of the Java programming language. Due to the limited card memory and processing power it is not possible to use a language as powerful as Java itself.

A Cyberflex Access card can contain several applets which can be identified uniquely as every applet has a unique application identifier (AID).

There is also the possibility to write a client application on the host computer in another language. In our project work [31], we used Java.

It is not possible to use these cards with OpenSC since they have no PKCS#15 structure [34].

Cryptoflex Cards Cryptoflex smart cards from Axalto [30] are non-programmable cards. They are also compliant with ISO7816. The cards consist of a 32Kbytes EEPROM and have also a cryptographic co-processor. Cryptoflex cards provide RSA, DES, 3-DES and SHA-1 hashing [35].

These cards are PKCS#15 compliant which means that they can be used with OpenSC. To create the PKCS#15 structure on these cards, OpenSC provides a tool named `pkcs15-init`. Further this tool can be used to create keys and certificates and to set PINs. There is another tool named `pkcs15-crypt` which can be used to sign and encrypt on the card. Additionally a tool to list the cards content is provided. It is called `pkcs15-tool`. For example, the `-c` option shows the certificates which are available. We created one certificate on our card:

```
bash-3.00# pkcs15-tool -c
X.509 Certificate [Certificate]
        Flags    : 2
        Authority: no
        Path     : 3F0050154545
        ID       : 45

bash-3.00#
```

The client application can be written in various languages. We use C as programming language since the libraries mentioned above are written for C.

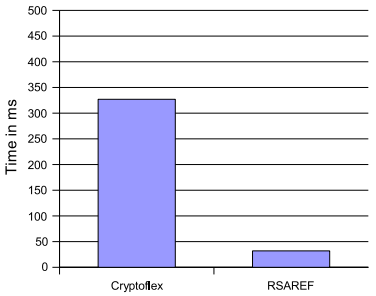


Figure 6.21: Comparison of the Cryptoflex card and RSAREF using MD5

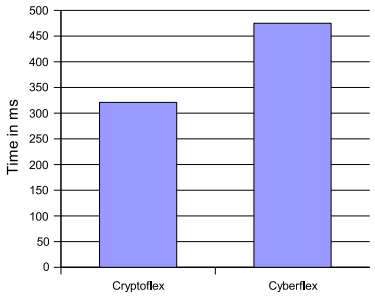


Figure 6.22: Comparison of the Cryptoflex and the Cyberflex card using SHA-1

6.4.1 Results

To evaluate the smart cards, they had to be checked against RSAREF, the library used for the actual CASHnet implementation. Furthermore the two types of smart cards had to be compared to each other.

First of all, the Cryptoflex card has been compared to the Cyberflex card. Therefore, both cards had to calculate a SHA-1 digest over a message and encrypt this hash using RSA with a key length of 1024 bit. As can be seen in Figure 6.22, the Cryptoflex card was significant faster than the other card. Signing a message took around 320ms on the Cryptoflex and 470ms on the Cyberflex card. Using these algorithms on one of these cards would introduce a huge delay on the nodes, which is unacceptable for the CASHnet scheme.

Furthermore, the Cryptoflex cards had also been compared to the RSAREF library. As this library only supports MD5 hashing, doing this test, the smart card also used MD5 to calculate a message digest. This digest then had been encrypted using RSA with a 1024bit key. Figure 6.21 shows that RSAREF needs less than 50ms to sign a message whereas the smart cards needs around 320ms.

As the Cyberflex card did not support MD5 when it has been evaluated [31], it was not possible to compare it directly to the RSAREF library. But as the delays for SHA-1 and RSA were such high, there is no reason to expect that the card will be much faster with MD5.

The measurements with the smart cards have shown that the actual cards are not adapted for time-critical operation such as per packet signing.

Chapter 7

Summary and Outlook

Nowadays, it becomes more and more important to be online every time and everywhere. To achieve this, the approach of Multihop Cellular Networks has been introduced. The networks relay on the fact that every node acts as router which means that everybody joining such a network has to cooperate. As cooperation is not a natural property of human beings, there have to be some algorithms to stimulate it. There are two types of such algorithms. First, there are some ideas to stimulate cooperation using punishment. This leads to cooperation due to fear. The second type of algorithms tries to motivate the nodes using rewards. During this diploma thesis, CASHnet, one of these motivation based algorithms has been implemented and evaluated in a Linux environment.

CASHnet requires a node to pay so called Traffic Credits if it wants to send a packet out of its actual MANET. The receiver, if it is located in a MANET also has to pay if it receives a packet coming from outside its MANET. Furthermore, all the intermediate nodes in the networks get charged with so called Helper Credits, which is a very good stimulus to forward packets. Traffic Credits can be bought with real money or earned by exchanging Helper Credits. Therefore, cooperation will save money.

With the Linux implementation done in this diploma thesis, we gave a proof of concept. This means that it has been shown that it is possible to implement CASHnet in a real world environment with most of its features. Furthermore, as most of the other cooperation schemes require source routing, it has been shown that it is possible to implement such a scheme without source routing. CASHnet only requires the hop count, which can also be provided by other routing algorithms such as AODV [5].

However, this work also showed the limitations of CASHnet. The evaluations showed that the performance is bad due to the proposed security scheme, which requires every data packet to be signed. This is a fact which is unacceptable in a possible end product. Therefore, for future work, one has to think about a new security scheme, which should be probably not directly

integrated in the data stream. For instance, a sender could insert periodical signed CASHnet control messages into its data stream which could be the base for the accounting. A sender is then charged according to the needed messages for a transmission and a forwarding node is rewarded based on the same principle. These control messages could be very small so that they have nearly no influence on the ongoing transmission and would therefore not cause any significant additional delay. As this separated control stream also has to be secured, one could think about methods like neighborhood watch or e2e sessions with the gateway.

Furthermore, one has to think about an adaptation to UDP. Our prototype does not support UDP due to fragmentation problems caused by the security scheme. If thinking about another security scheme, the UDP problem probably gets also solved. But if it is desired to keep a scheme similar to the existing one, UDP transmissions are still problematic. And as Internet telephony becomes more and more important CASHnet has to support UDP, otherwise it will never be usable.

Last, as security becomes more and more important, there is still the need for the integration of tamper resistant devices. Either the long time for signing on a smart card is less important in a new security scheme or it has be thought about faster devices.

Appendix A

CASHnet configuration

To configure CASHnet, we decided to introduce configuration files. There are two configuration files: a general CASHnet configuration file and a node specific file. In the general configuration, we set the port for the CASHnet control messages, the protocol number used to tunnel the packets, the granularity of the lists, the initial traffic credit amount, the number of maximum non-booked rewards, several timeouts and the period for the broadcasted AUTHREQs. In the future, the initial amount of traffic credits should be set on the tamper resistant device.

In the node specific configuration file (see an example in Listing A.1), only node specific option are set like the ID, the IP address, the netmask, a flag which indicates if the node is a gateway or not and the locations of the own certificate and private key as well as the root certificate. The gateway flag is needed in our test scenario since a normal notebook acts as gateway. Usually gateways should be provided by an ISP. Therefore this option would never be used in a real life scenario. The locations of the certificates are needed since in the first prototype, we did not use a tamper resistant device. The root certificate is needed to check the certificates the other nodes send. This has to be done to ensure that all the participating nodes use valid certificates.

Listing A.1: surveyor.conf

```
# configfile for acfi
#
ip-address: 10.1.1.107
netmask: 255.255.255.0

id: 10.1.1.107

#
# the UDP port we use for auths, authreqs and acks
#
acfi-port: 19810
```

```
#
# which IP-protocol number is used to "tunnel" the ip packets
#
protocol-number: 250


#
# enables gateway mode. This should be set to 1 on the gateway
#
gateway: 1

#
# The certificates and the root key
#
 root_certificate : /home/latze/cashnet/src/root.cert
 my_certificate : /home/latze/cashnet/src/10.1.1.107.cert
my_private_key: /home/latze/cashnet/src/10.1.1.107.priv
```

Appendix B

Demonstrator

CASHnet has been extended by a small graphical user interface (GUI) for demonstration purposes (see Figure B.1). This GUI shows the actual amount of traffic and helper credits and allows exchanging helper credits and buying traffic credits. It has been written in Tcl/Tk and communicates with CASHnet using file descriptors.

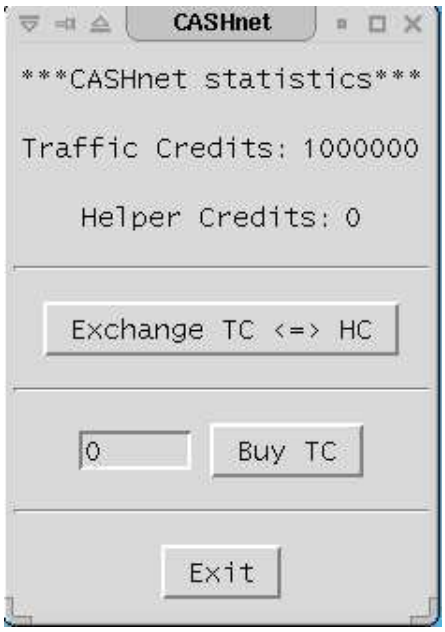


Figure B.1: graphical user interface for the CASHnet algorithm

Tcl is short for "tool command language" has been developed in the early 1980s [36]. It is a scripting language which allows the fast implementation of small programs. Today, Tcl is a powerful programming environment.

Tk is a extension to the Tcl language, which is very popular. Tk adds command to write GUIs and allows binding Tcl scripts to them. This allows interaction between user and GUI.

CASHnet and the GUI have to exchange some information. The GUI for instance has to know the actual amount of traffic and helper credits which has to be given by CASHnet. Furthermore, it is possible to change helper credits into traffic credits and to buy additional traffic credits using the GUI. This information has to be given to CASHnet. Therefore, the output of the GUI has to be piped into CASHnet, whereas CASHnet writes into a file, which can be read by the GUI.

Bibliography

[1] T. Staub. *Implementating a Cooperation and Accounting Strategy for Multi-hop Cellular Networks.* Master's thesis, University of Bern, 2004. URL `http://www.iam.unibe.ch/~rvs/research/publications.php`. Last visited 10.12.2005.

[2] Ananthapadmanabha R., B.S. Manoj and C.Siva Ram Murthy. Multihop Cellular Networks: The Architecture and Routing Protocols. In *Proceedings of 12th IEEE International Symposium on Personal, Indoor, Mobile Radio Communications (PIMRC)*, pp. 78–82. San Diego, CA, USA, Sep.-Oct. 2001.

[3] G.Anastasi, M.Conti, and E.Gregori. IEEE 802.11 Ad Hoc Networks: Protocols, Performance and Open Issues. In S.Basagni, M.Conti, S.Giordano, and I.Stojmenovic, ed., *Mobile Ad Hoc Networking*, chap. 3, pp. 69–116. Wiley-IEEE Press, 2004.

[4] D. B. Johnson, D. A. Maltz, and Y.-C. Hu. The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR), Jul. 2004. URL `http://www.ietf.org/internet-drafts/draft-ietf-manet-dsr-10.txt`. Last visited 10.12.2005.

[5] C. Perkins. Ad hoc On-Demand Distance Vector (AODV) Routing, Jul. 2003. URL `http://www.faqs.org/rfcs/rfc3561.html`. Last visited 16.12.2005.

[6] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol, Nov. 1998. URL `http://rfc.net/rfc2401.html`. Last visited 16.12.2005.

[7] K. Sanzgiri, D. LaFlamme, B. Dahill, B. N. Levine, C. Shields, and E. M. Belding-Royer. Authenticated Routing for Ad Hoc Networks. In *IEEE Journal on Selected Areas in Communications*, vol. 23(3), pp. 598–610. Mar. 2005.

[8] R. Wakiwaka, J. T. Malinen, C. E. Perkins, A. Nilsson, and A. J. Tuominen. Global connectivity for IPv6 Mobile Ad Hoc Net-

works, Jul. 2005. URL http://www.ietf.org/internet-drafts/draft-wakikawa-manet-globalv6-04.txt. Last visited 25.09.2005.

[9] S. Buchegger and J.-Y. L. Boudec. Performance Analysis of the CONFIDANT Protocol (Cooperation Of Nodes - Fairness In Dynamic Ad-hoc NeTworks), Jun. 2002. URL http://icwww.epfl.ch/publications/documents/IC_TECH_REPORT_200201.pdf. Last visited 29.09.2005.

[10] L. Buttyán and J.-P. Hubaux. Enforcing Service Availability in Mobile Ad-Hoc WANs. In *Technical Report No. DSC/2000/025*. Swiss Federal Institute of Technology - Lausanne, 2000. URL http://cwww.epfl.ch/publications/documents/IC_TECH_REPORT_200025.pdf. Last visited 29.09.2005.

[11] L. Buttyán and J.-P. Hubaux. Stimulating Cooperation in Self-Organizing Mobile Ad Hoc Networks. In *Technical Report No. DSC/2001/046*. Swiss Federal Institute of Technology - Lausanne, 2001. URL http://icwww.epfl.ch/publications/documents/IC_TECH_REPORT_200146.pdf. Last visited 29.09.2005.

[12] N. Ben Salem, L. Buttyán, J.-P. Hubaux, and M. Jakobsson. A Charging and Rewarding Scheme for Packet Forwarding in Multi-hop Cellular Networks. In *MobiHoc'03*. Annapolis, Maryland, USA, Jun. 2003. URL http://www.informatics.indiana.edu/markus/papers/routingpay2.pdf. Last visited 29.09.2005.

[13] B. Lamparter, K. Paul, and D.Westhoff. Charging support for ad hoc stub networks. In *Elsevier Journal of Computer Communications*, vol. 26(13), pp. 1504 – 1514. Aug. 2003.

[14] A. Weyland and T. Braun. CASHnet - Cooperation and Accounting Strategy for Hybrid Networks, May 2004. URL http://www.iam.unibe.ch/~rvs/research/publications.php. Last visited 29.10.2005.

[15] Uppsala University. AODV-UU, Aug. 2005. URL http://core.it.uu.se/AdHoc/AodvUUImpl. Last visited 16.12.2005.

[16] RSA Laboratories. RSAREF(TM): A Cryptograhic Toolkit - General Information, Mar. 1994.

[17] B. Kaliski. The MD2 Message-Digest Algorithm, Apr. 1992. URL http://www.faqs.org/rfcs/rfc1319.html. Last visited 3.01.2006.

[18] B. Kaliski. The MD5 Message-Digest Algorithm, Apr. 1992. URL http://www.faqs.org/rfcs/rfc1321.html. Last visited 3.01.2006.

[19] A. Weyland. *Cooperation and Accounting in Multi-Hop Cellular Networks*. Ph.D. thesis, University of Bern, Nov. 2005. URL `http://www.iam.unibe.ch/~rvs/research/publications.php`. Last visited 16.12.2005.

[20] C. Eckert. *IT-Sicherheit. Konzepte-Verfahren-Protokolle*, chap. 10.3, pp. 387 – 397. oldenbourg-verlag, 2003.

[21] PKCS#15 v1.1: Cryptographic Token Information Syntax Standard, Jun. 2000. URL `http://www.rsasecurity.com/rsalabs/node.asp?id=2141`. Last visited 10.12.2005.

[22] PKCS11#11 v2.20: Cryptographic Token Interface Standard, Jun. 2004. URL `http://www.rsasecurity.com/rsalabs/node.asp?id=2133`. Last visited 10.12.2005.

[23] The OpenSC Framework. URL `http://www.opensc.org/files/doc/opensc.html`. Last visited 10.12.2005.

[24] The OpenCT Framework. URL `http://www.opensc.org/openct/`. Last visited 10.12.2005.

[25] Debian Linux. URL `http://www.us.debian.org/`. Last visited 5.12.2005.

[26] Slackware Linux. URL `http://www.slackware.com/`. Last visited 5.12.2005.

[27] thttpd webserver. URL `http://packages.debian.org/stable/web/thttpd`. Last visited 13.12.2005.

[28] curl. URL `http://curl.haxx.se/`. Last visited 13.12.2005.

[29] Determine the system CPU load using uptime. URL `http://www.tech-recipes.com/solaris_system_administration_tips20.html`. Last visited 16.12.2005.

[30] Axalto: smart card solutions and point of sales terminals. URL `http://www.axalto.com/access/index.asp`. Last visited 10.12.2005.

[31] C. Latze. *Deployment and Performance Analysis of JavaCards in a Heterogenous Environment*. Computer science project, University of Bern, Jan. 2005. URL `http://www.iam.unibe.ch/~rvs/research/publications.php`. Last visited 10.12.2005.

[32] Java Card(TM) 2.1 Platform API Specification. URL `http://java.sun.com/products/javacard/htmldoc/`. Last visited 10.12.2005.

[33] Cyberflex Access 32k. URL `http://www.cyberflex.com/Products/cards_access.html`. Last visited 10.12.2005.

[34] H. Seudie. *Anwendung von Digitalen Identitaeten an einer Hochschule*. Master's thesis, Technische Universitaet Darmstadt, 2005.

[35] Cryptoflex. URL `http://www.axalto.com/access/smartcards/cryptoflex.asp`. Last visited 10.12.2005.

[36] P. Raines and J. Tranter. *Tcl/Tk in a Nutshell. A Desktop Quick Reference*. O'REILLY, 1999.