Bachelor
im Studiengang "Angewandte Informatik"

Design and Implementation of a Scout Daemon for CASP: a General-Purpose Internet Signaling Protocol

Fabian Meyer

am Lehrstuhl für
Telematik

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen
09. Februar 2004
Ich erkläre hiermit, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 09. Februar 2004
Abstract

The CASP protocol is a general signaling protocol working on top of existing transport protocols such as TCP, UDP, SCTP or raw IP. It provides a framework for applications that need signaling. Applications include first of all QoS.

This thesis describes the design, implementation and testing of a scout daemon for CASP. The task of this daemon is to provide the main CASP daemon with a service, that can effectively discover the next CASP-aware hop on the path to a given destination.

The experimental results show that the scout protocol implementation is stable, feasible and fast.
Acknowledgements

My thanks go to my advisors that enabled me to participate in this project.

Advisors:
Prof. Dr. Dieter Hogrefe
Dr. Xiaoming Fu

Additional thanks:
My girlfriend Nadine for supporting me, even with the varying degrees of incomprehensiveness that working late at night can cause.
Emmanouil Staflarakis for his help with debugging.
My colleague Sebastian Willert for his help and suggestions.
## Contents

1. **An Overview of the CASP Protocol** ........................................... 1
   1.1. The CASP Framework ..................................................... 1
   1.2. CASP Usage by Examples ............................................. 2
   1.3. CASP Messages ......................................................... 3
   1.4. CASP Message Format .................................................. 4
       1.4.1. Length Header .................................................... 4
       1.4.2. Common Header .................................................. 4
       1.4.3. Objects ........................................................... 5
       1.4.4. Scout Request and Response Messages ....................... 6
   1.5. Scope of the Thesis .................................................... 7

2. **The Scout Protocol** ............................................................ 8
   2.1. Introduction to Scout .................................................. 8
   2.2. Scout Protocol .......................................................... 9
   2.3. Scout Security .......................................................... 9
   2.4. Comparison with RSVP ................................................ 9

3. **Design and Implementation** ................................................. 11
   3.1. Goals and Requirements .............................................. 11
   3.2. Design ................................................................. 11
       3.2.1. Pthreads ......................................................... 11
       3.2.2. Practical Considerations ..................................... 13
       3.2.3. Differences from the Draft .................................. 13
   3.3. Implementation ........................................................ 14
       3.3.1. Discovery ......................................................... 14
       3.3.2. Scout Discovery Module ....................................... 16
   3.4. Experimental Testing .................................................. 17
       3.4.1. Testbed Setup .................................................... 17
       3.4.2. Reliability and Functionality ................................ 19
       3.4.3. CPU, Memory and Time Consumption ......................... 19
       3.4.4. Responsiveness ................................................ 21

4. **Conclusions and Future Work** ............................................ 23
   4.1. Conclusions ............................................................ 23
   4.2. Next Steps ............................................................. 23
       4.2.1. General Concerns ............................................... 23
       4.2.2. Scoutd Specific Concerns ..................................... 24
       4.2.3. Summary of Future Works .................................... 25

A. **Glossary** .............................................................................. 26
1. An Overview of the CASP Protocol

To establish a general understanding of the CASP protocol, an overview is given in this chapter. First we give a description of the CASP framework. Then we illustrate CASP with an example and go further into detail about the CASP messages and message format. At the end of this chapter the scope of this thesis is discussed.

1.1. The CASP Framework

The Cross Application Signaling Protocol (CASP), specified in [1], is not signaling protocol only for Quality of Service (QoS) resource reservation, but rather a general signaling protocol. It aims to provide a framework for applications which need installing and mainenance of corresponding control states in the network. These applications can be QoS-related or anything else. Thus, an extensible, general messaging protocol is required. RSVP [3] was designed by the IETF but its complexity, mobility and modularity limit its use. CASP is a proposal developed after RSVP which addresses the limitations of RSVP.

Here, signaling means the installation and mainenance of soft states. Signaling can be end-to-end or hop-by-hop, whichever is more suitable for a given task. Signaling can be used for various tasks, one of the most prominent is QoS.

QoS signaling has been a hot topic in the internet community for more than ten years. There has not been a simple solution to the problems. While challenges for QoS signaling keep changing these years, new ideas have continuously been proposed in this field. The extension of CASP for QoS signaling is described in [2].

Recently the IETF NSIS (next steps in signaling) working group has been formed to define a framework and solutions for signaling protocols. They also try to address new challenges for QoS signaling as well as signaling for other purposes.

As a general signaling protocol developed in the NSIS working group, CASP is neither bound to a specific purpose like QoS nor to any transport protocol. It can be used with TCP, UDP or SCTP. The CASP-components can be related to the NSIS-framework directly (as seen in figure 1.1).

Scout in the CASP framework is a protocol that can automatically discover the next node which is CASP-aware. This way, CASP does not need to rely on any pre-configured databases with CASP-aware nodes because the scout can discover the next node on the data path for it.

Furthermore the discovery-process is, as seen in figure 1.1, seperated from the rest of CASP, so that the discovery can be flexibly changed to current needs.

For example, for a CASP-aware node which has to rely on a single (CASP-aware) default router
to reach any other nodes, there is no necessity for a dynamic, on-demand discovery. In this example a static discovery which just uses a pre-defined router as CASP-aware node is sufficient.

In other environments, for example where all nodes are CASP-aware, a simple routing table lookup can be enough, in that case there is also no need for a real scout.

In other environments, for example where all nodes are CASP-aware, a simple routing table lookup can be enough, in that case there is also no need for a real scout.

1.2. CASP Usage by Examples

CASP can be used for QoS signaling or other signaling purposes, for example, in the area of decentralized filesharing tools or massively multiplayer games. If we look at the recent file-sharing tools, we can see that they all rely on some sort of central servers which have to coordinate the connecting peers. It should be possible to use CASP and its discovery methods to create a file-sharing tool that can automatically find peers without the need of a central server. The same idea can be used for massively multi-player games. Until recently there was only one approach to these games. There is one central server (or server farm) that all players log on to. When players want to affect the game world (e.g. build their own virtual house), they have to contact support so that their finished house can be added to the world. Again, making use of CASP and its discovery procedures it should be possible to create a huge virtual world where every player hosts its own small part of the game on his local machine. These small worlds will interconnect using CASP, forming a huge world for the players to explore. These two examples show that
CASP can be of interest to a wide range of people.

As CASP is not a request-response-protocol, a message to the destination is delivered in a more fixed way than in standard TCP or UDP. CASP works using (in the ideal case) direct connections between neighbouring CASP-nodes. If the communication start and end-nodes are not neighbouring each other there will be a chain of connections as seen in figure 1.2. In this example we assume that all nodes are CASP-aware and TCP is the underlying transport mechanism.

![Figure 1.2.: Example network topology](image)

Assume a CASP-using client in node 'a' wants to establish a connection to node 'e'. Using its scout mechanism (in GoCASP this can be either the scoutd or static discovery with fixed next hops) CASP determines the next hop (NHOP) on the path to destination 'e'. The discovered NH will be 'b'. Now CASP checks if there is already a connection to 'b' which can be reused for this new connection, otherwise the caspd will create a connection to 'b'. The connection will be established between node 'a' and node 'b's caspd's and be typically bi-directional. Now node 'b's CASP daemon will know that node 'a' wants a connection to destination 'e' and try to find it's own NH on the path to 'e'. Then 'b' will connect to 'c' (or reuse an existing connection). This is done in all nodes, until there is a connection-chain from 'a' to 'e'. It will be virtually one single end-to-end connection, and 'a' can start sending to 'e'. Different from e.g. TCP connections, a CASP association is determined in every step and has (ideally) a chain only between (link-layer) neighbouring nodes. If a node in the chain fails and is not reachable, the neighbouring nodes will detect it and try to close the gap, once again using the scoutd or some other discovery mechanism. This way there can be a (mostly) fixed path and node failures can still be signaled dynamically.

There may be CASP-unaware hops in between two as there cannot be end-to-end guaranteed features, since there is no knowledge if the unaware hops can fulfill the needs.

### 1.3. CASP Messages

CASP messages, as seen in the example, are addressed and delivered in a hop-by-hop way. In this chain every node is responsible for delivering the message to its next peer along the data path. To offer end-to-end reliability in a non-request-response-protocol CASP allows clients to send back messages to notify the sender of a message if the original message was delivered successfully or if it encountered any kind of error.

CASP incorporates the ability to re-use existing transports. If a connection to the next hop
exists it can be re-used, no matter if the connection was from the same session or another one.

Sessions, or more exactly session states, are maintained in CASP by the use of session identifiers. Besides the session identifier, there are some other informations for a single session state that needs to be maintained in CASP nodes. These informations include a a flow-identifier, the previous hop (PHOP) and the next hop (NHOP), the refresh interval and a branch identifier. By using these branch identifiers there may even be several NHOP/PHOP’s for a single state.

CASP does not require every node to support all clients, as long as all nodes support CASP itself. The data path can traverse the node without support for the given client without harm. This might destroy the reliability of some of the clients features (e.g. if the client is a QoS client, there can be no reliable end-to-end guarantees) but it does not affect CASP itself.

The seperation of signaling message delivery and discovery gives CASP a number of advantages. This thesis focuses on the scout discovery and the scout process, details can be found in the respective chapters.

1.4. CASP Message Format

A CASP-message consists of a common header and a variable number of objects. These objects can also be of variable size but have to be identified to be of a specific type.

Except for the length- and common-header, which have to appear first, the objects can be in any order. They should be in the order specified, but the receiver must be able to correctly parse messages with the objects in any order.

Even though the following specifications are identical to those in the draft, they will be included here because they are essential to the implementation.

1.4.1. Length Header

The length header is required when using stream-based transports (e.g. TCP) but must not be used when using message-based protocols (e.g. UDP).

The only field is the 32-bit long length-value of the CASP-message in bytes. It includes the length of the common and length header.

1.4.2. Common Header

The common header’s fields are all 8 Bits, except for the session identifier. The fields are:
### Flags
- **R**: reverse bit; node shall route opposite to the direction of the data-flow
- **T**: teardown bit; signals that the node shall tear down any m-layer and associated client layer sessions for this session. If not set the message refreshes (or establishes) the m-layer state.
- **D**: discovery bit; signals that the node shall perform a new discovery operation. If this bit is not set the old NHOP should be used (if possible). This bit can not be set if 'R' is set and should not be set if 'T' is set.
- **U**: unsecure bit; signals that the message has traversed a hop without channel security.

### TTD
Time To Deliver; this is just like IP’s TTL (Time To Live). The TTD-value is decreased in every hop. If the value reaches zero the message should be discarded.

### Hop Count
The value of the hop count is increased in every node.

### Type
This indicates the type of the CASP message. Right now there are 3 valid types:
- **Type 1**: CASP signaling message
- **Type 2**: CASP scout request message
- **Type 3**: CASP scout response message

### Session Identifier
denotes a signaling session. The identifier is a 128-bit, globally unique value and should be a cryptographically random integer. Alternatively a number could be generated from the MAC-address of a node. The latter method however has some issues with NAT and privacy considerations. It might nevertheless be useful for nodes that are not able to generate 128 bits of random data.

### 1.4.3. Objects

Each object consists of one or more 32 bit-word(s) and an one word header with the following fields:

<table>
<thead>
<tr>
<th>R</th>
<th>T</th>
<th>D</th>
<th>U</th>
<th>Session identifier (16 bytes)</th>
<th>TTD</th>
<th>Hop count</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 1.4.: Common header**
A CASP Scout Daemon

An Overview of the CASP Protocol

<table>
<thead>
<tr>
<th>Length (bytes)</th>
<th>Class-Num</th>
<th>C-Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>// object contents //</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.5.: Object header

**Length** 16 bits containing the total length of the object (including header, excluding padding) in bytes.

**Class-Num** 8 bit class identifier. Any CASP implementation must recognize the following classes:

- **FLOW_ID**: Contains information about the flow which should receive a particular client treatment. It is contained at the CASP M-layer to allow policy based forwarding and NAT devices to inspect this object without major effort. It typically contains the IP addresses of the data sender and data receiver, and possibly some additional demultiplexing information (such as protocol type, source and destination ports, SPI or flow label).
- **CASP_TIMEOUT**: Refresh interval for the m-layer state.
- **CLIENT_DATA**: Client data part of the message.
- **ERROR**: Contains an errorcode if any error occurred.
- **SCOUT_COOKIE_I**: Random cookie (cryptographically generated random number) generated by the scout initiator of a connection.

**C-TYPE** 8 bits containing the object-type. This is a value that is unique inside its class.

Each object must be padded to align on a 32 bit-word boundary, with the least number of possible additional bytes. There may be up to 3 zero-valued bytes at the end of an object. The length of the padding is not included in the length field of the header.

### 1.4.4. Scout Request and Response Messages

A scout request message is sent when the sender wants to determine which the next hop (NHOP) is. The message consists of the CASP common header and the SCOUT_COOKIE_I object. Because scout uses UDP the length header is omitted. In the common header only the field 'Type' is set to '2'. The other fields in the common header are not used yet. 'TTD' and 'Hop Count' may be used if capability discovery is added. As capability discovery is not of a high priority in the CASP specification, it is not supported in GoCASP (release 0.1.0).

The SCOUT_COOKIE objects contain the scout cookies. In CASP cookies are random numbers that are exchanged between communication partners to verify the identity of the peer.
A scout response message is sent by the node that received the scout request message. It is addressed to the node that sent the scout request and very similar to the scout request message. The response message consists of the CASP common header followed by the SCOUT_COOKIE_I object which is in turn followed by the SCOUT_COOKIE_R object. The SCOUT_COOKIE_I object will be copied from the request message; it is used to verify the identity of the responder. The SCOUT_COOKIE_R object contains another cookie that will be used when the CASP m-layer creates the sessions connection to the NHOP. It will be sent with the first message to the NHOP so that the NHOP node may in turn verify that the sender really is the one that sent the discovery message using the scout.

1.5. Scope of the Thesis

This thesis describes the prototype implementation and experimental tests for the CASP-scout daemon of GoCASP, the CASP implementation of the University of Goettingen.

The rest of the thesis focuses on the scout protocol, scout daemon and the discovery module for the casp daemon. It presents the in-depth design and implementation of the GoCASP scout daemon. After an introduction to CASP and scout, the implementation is discussed and the detailed test results are presented followed by discussions of possible further enhancements.
2. The Scout Protocol

This chapter starts with an introduction to scout before detailing about the scout protocol and scout security in the respective sections. The chapter ends with a comparison to RSVP.

2.1. Introduction to Scout

The scout protocol of the CASP framework is separated from the rest of the CASP components. This provides several advantages. For once, it provides the possibility to handle path-coupled and path-decoupled signaling in one protocol as only the discovery process is different. Furthermore there can be several different scouts so CASP can be adapted to the needs of the environment as mentioned earlier. The path-coupled discovery procedure is called scout. Its design is similar to that of the RSVP PATH message.

The CASP specification states that scout discovery messages are only necessary if the next node is more than one network hop away and if there are no other suitable means to determine the NHOP. However, in GoCASP 0.1.0 we support scout for all discoveries in all nodes. This has several reasons which will be identified later in section 3.2.3.

Other valid methods of discovery can be a static discovery with a pre-configured default next CASP-aware node (NHOP) or routing table lookup. These methods do not need dynamic discovery and are only suitable for certain environments.

To discover the NHOP to which the current CASP node wishing to send a CASP message, this CASP node first determines if there is already an existing CASP connection to the given destination. These CASP connections are maintained in CASP in m-session states. If there is a m-session state to the NHOP, caspd can re-use the connection to send the new message without needing to perform a discovery. Otherwise the NHOP has to be discovered. This can be done by one of the named procedures: scout message, routing table lookup, pre-configured default NHOP. Please note that the discovery procedure is performed using the destination IP of the message, which is the basis of determining the NHOP. After a successful discovery procedure a new m-session state will be created and the CASP-message with its payload can be delivered.

The CASP specification states that a node has to use the discovery procedure for every new signaling session as it can not generally determine the NHOP by inspecting the destination address. Since GoCASP does not support capability-based discovery yet, a node may well determine the NHOP for a given destination. We used this fact and implemented a scout cache that stores NHOPs for given destinations. This will also be explained in detail later in section 3.2.3.
2.2. Scout Protocol

The scout protocol is used to discover the next CASP node. With capability-based discovery special requirements to the node to discover can be made so that not only the next but the next suitable node may be discovered. This is, as mentioned earlier, not yet supported in GoCASP.

If there are other means available that produce less overhead and delay they are preferred. Each discovery using the scout triggers a scout request message to be sent and in most cases also a scout response message (if not there should at least be an ICMP error message). The overhead consists of network load because of the packets as well as processing time and memory in the nodes that send/receive the request/response messages. The delay is mostly introduced by the need to wait for the request message to reach its goal (in most cases that is not the destination it is addressed to) and the response to be sent back. The discovering node cannot drop or confirm the state establishment procedure before the response or an error message arrives.

The scout request messages are sent using UDP for transport. They contain some CASP objects (as seen in 1.4) and have the IP_ROUTER_ALERT option (see [4] for details) enabled. This IP option effects that any router that is capable of recognizing that option will take the packet off the wire and inspect it further. With this IP option the scout request message can simply be addressed to the destination of the triggering CASP request to find the next hop on the data-path.

As a reliability mechanism scout requests use a periodical retransmission. The time interval of the retransmission is exponentially increased and starts at 500ms.

2.3. Scout Security

Scout messages have their own security considerations and methods. Since the request messages cannot be secured using IPsec or TLS method because the communication partner is unknown at the beginning, scout request messages encounter some security risks. To establish some security the following considerations were made.

First of all, since scout messages cannot be protected by TLS or IPsec mechanisms, they are separated from the regular message delivery. To prevent DoS-attacks, a CASP node receiving a scout message should not establish a state (mechanisms learned from mobile IPv6 can be used to deliver such a functionality in a more secure way). A CASP node makes use of cookies (cookies as described in 1.4.4) to ensure that the response received matches the request sent. To ensure that no adversary can hijack such a connection the cookies are exchanged once again securely when a security association is established between the two nodes.

2.4. Comparison with RSVP

The scout messages are designed similar to the RSVP [3] path messages. The RSVP path messages are also addressed end-to-end rather than hop-by-hop. However the RSVP path messages carry a larger overhead because they are carrying additional soft state information. CASP scout messages carry only scout information (the scout cookies). This has several advantages over RSVP path messages. One is that the overhead in CASP is lower. Another advantage of CASP
is that the scout messages are separated from signaling message delivery. This is important because the scout messages are hard to secure (as described in section 2.3). In RSVP the path messages carry signaling information. This introduces some severe security risks, when the unprotected packets are manipulated by a malicious node. This malicious node immediately gains access to at least a subset of the signaling information of RSVP.

In summary we can assume that CASP is able to resolve some of the problems of RSVP.
3. Design and Implementation

This chapter starts with a summary of the goals that we had in mind when we implemented the scoutd of GoCASP. It continues by a detailed sections on the design and implementation. The last part of this chapter is about the experimental tests and their results.

3.1. Goals and Requirements

The GoCASP’s scoutd has been designed to be as conformant to the CASP specification as possible. The main goals for GoCASP’s scoutd were stability and reliability, fast responsiveness, conformity to the draft and a modest CPU and memory consumption.

Further discussions showed that other things had to be taken into consideration. These were in particular the need to use IP_ROUTER_ALERT option (see [4] for details) and with it the need to use raw sockets and the need for scoutd to be able to manage different data connections simultaneously. The distinct tasks for the connections are first sending and receiving scout request messages which have the IP_ROUTER_ALERT option enabled, second the communication with the mlayer (or rather the scout discovery plugin for the mlayer) using a local unix socket and third reply to scout request messages by sending a scout response message using a normal UDP socket.

3.2. Design

When designing a software, one of the first steps is to determine which programming language should be used. We discussed this topic and came to the conclusion that ANSI C with the GNU libraries would fit our needs best. We decided to use C because it is the best option when you are programming a protocol stack and want to be close to the hardware.

3.2.1. Pthreads

To accomplish the goals of modest memory consumption, fast responsiveness and stability we decided to use pthreads instead of fork(). Pthreads introduced less overhead to the memory, as only sub-processes are spawned which do not require a complete own process structure as fork’ed processes would. Besides the lower memory consumption the use of pthreads has the advantage that it is separate threads can access the same memory because they are still part of the same process. This enables the scoutd threads to access mutual data safely using mutex locks and spares the necessity of costly inter-process communication.
When we decided to use pthreads, we needed to specify a reasonable structure for the pthreads to use. After some discussion and testing, we implemented the structure seen in figure 3.1. The main method of the scoutd is just needed to spawn the three main threads. There is the mlayer-server, which handles communication with the scout discovery module in the caspd, the scout-server which handles the incoming discovery responses and the recv_discover thread which uses a raw socket to receive scout request messages addressed somewhere else but tagged with the IP_ROUTER_ALERT option.

![Scoutd pthread structure](image)

**Figure 3.1.: Scoutd pthread structure**

The mlayer-server is the only one that spawns further threads. It communicates with the scout-discovery-module in the caspd. The caspd can trigger number of tasks. For each task the mlayer-server spawns a new thread to accomplish it. The mlayer-server simply determines which task is being asked for and spawns a new thread for it, handing over the socket communication to the new thread. The new thread then deals with any further exchange of information that is necessary. This way the mlayer-server is very quickly ready to accept the next request.

There are 4 possible tasks that the caspd can trigger in scoutd. Three of these tasks are rather simple and can be done very fast. The functions are casp_scout_initialize, which is called when the caspd starts up, casp_scout_validate, which can be called by caspd to validate a given SCOUT_COOKIE_R and finally there is the casp_scout_refresh_all function that caspd triggers so that the scoutd refreshes all its states and data structures. The refresh function is triggered...
Design and Implementation

A CASP Scout Daemon

by the caspd because the scoutd does not have its own means to maintain a central time base. The scoutd does not need such a facility except when refreshing its states. But since the scoutd refreshes should use the same interval as the caspd we implemented it so that scoutd refreshes its data structures when contacted through the mlayer-server.

The last task is the most complex one, the discover function. Because it is the heart-piece of the scoutd, the whole section 3.3.1 is about it. Detailed information about the use of pthreads and the technique of discovery can be found there.

3.2.2. Practical Considerations

Some of the design decisions had to be made when we encountered problems with the implementation. These practical considerations led to some interesting design decisions.

The first thing that we encountered when implementing the function that should receive the discovery request which are tagged with the IP_ROUTER_ALERT option. To capture packets addressed elsewhere, even using the IP_ROUTER_ALERT option requires the use of a raw socket in C. When we used the raw socket to capture the packets, it quickly showed that a lot of bogus packets were caught. The problem is not only that the false packets would have to be filtered, which would have been very time and memory consuming, but also that the packets were taken off the wire by the raw socket, so that the actual receiver never got them. These false packets included everything from http data traffic to ICMP echo requests. To circumvent this problem, we introduced the use of a new protocol number for scout requests. We used 0xfa (250 decimal) as it is unused. By using this number we ensure that no bogus packets are caught, since only those with the protocol number 0xfa are caught.

The next thing we encountered was, that in the caspd and the scoutd the need for a data structure which could be searched very fast and efficiently arose. After some discussions we used binary search trees in GoCASP 0.1.0. Binary search trees are implemented in the GNU C library and can be searched very efficiently, which helps in getting faster response times.

3.2.3. Differences from the Draft

As written in 3.1 we tried to follow the proposals of the draft as closely as possible and as loosely as necessary, so we have a few minor differences to the draft.

First of all, it is difficult to assume that “Scout messages are only necessary if the NHOP is one or more network hops away”. Actually, we implemented, intended and tested the scoutd for all discoveries, even if they are less then one network hop away (neighbouring nodes).

The second difference is the use of a discovery cache. The draft proposes that there could not be such a cache, since the discovered NHOP for a given destination might differ from the one previously discovered. This can be true in two cases. The first case is that the previous NHOP has disappeared from the network (or is for some other reason unable to compute the requests). Such a case would be detected and a new discovery would be initiated, after deleting the old entry from the cache. Another situation where the NHOP in the cache might not be the correct one, is that the client uses capability based discovery. In that case, the NHOP in the cache might not be able to provide the capabilities that are needed for the new NHOP. But since GoCASP does not support capability based discovery, the second case does not matter.
With these considerations in mind we implemented a discovery cache, enabling us to save estimates up to 75-90% of discoveries. These savings largely depend on the network topology. To save inter-process communication, we later moved the discovery cache into the scout discovery module, which is loaded into the caspd at startup. This way there is no necessity to contact the scoutd if the NHOP for a given destination was already in the cache - the discovery module just replies with the address.

### 3.3. Implementation

#### 3.3.1. Discovery

The discovery procedure is undoubtedly the heart of the scoutd. It provides the functionality that defines the scoutd. The idea is very simple, just address a packet to the destination of the connection and tag it with the IP_ROUTER_ALERT option, the next CASP-aware router will recognize the packet, take it off the wire and reply. That is the overall idea put short. To provide this functionality in a reliable way, it is necessary to consider many things like network congestion and reliability (scout discovery requests are sent using UDP, so there is no reliability mechanism from the used protocol). To account for all eventualities the discovery procedure turned out to be very complex. A schematic view of the procedure can be found in figure 3.2.

![A schematic view of the scout discovery procedure](image)

Because figure 3.2 is very complex, a detailed explanation of the process follows below.

First the caspd triggers a discovery, using the scout discovery module. The figure starts when the scout discovery module is triggered by the caspd (labeled as ’trigger’). The cd_gethop function of the discovery module contacts the mlayer-server of the scoutd using the local unix socket. 
The mlayer-server recognizes the discovery request and spawns a discover thread. This thread is handed the socket to communicate with the scout discovery module. The cd_gethop function of the scout discovery module and the discover thread of the scoutd communicate directly, exchanging necessary information (like the destination of the connection, meaning the address that the discovery request packet will be addressed to).

After the discover-thread acquired the necessary data, it assembles the UDP packet to the destination and tags it with the IP_ROUTER_ALERT option. To assemble the packet it needs to generate the SCOUT_COOKIE_I and the respective casp headers. The cookie is generated using a crypto random number generator of the OpenSSL libraries. This procedure is the one that uses the most CPU time of the whole scoutd. The generated i-cookie is put into the packet, and after the packet is sent for the first time, it is also put into the awaiting-list.

The awaiting-list contains all the i-cookies that were sent and are still pending. The i-cookie is sometimes called "id" in this context.

When the packet was sent, and the i-cookie put into the awaiting list, the discover-thread enters a conditional, timed wait. The wait time is initially 500 msec and uses a multiplicator of 2 as gain, until a hard timeout of 512 seconds is reached.

If the timer wakes the thread, it is because there has not been a reply to the sent request - this indicates that the packet might have been lost. As UDP does not provide any means to ensure reliability, the discovery procedure uses a simple resending of the initial discovery request as a reliability mechanism. That means, that the scout request is resent every time the timer wakes the thread until the hard timeout is reached on which the id is deleted from the awaiting list and the discovery failed, there is no NHOP that could be discovered.

When the request was received by another scoutd in the (still unknown) NHOP, it replies by sending a scout discovery response. This response will be received in by the recv_discover thread of the initiating scoutd. The recv_discover thread checks if the i-cookie that is contained in the response packet is in the awaiting list. If it is not, the response packet is unwanted and is discarded. If the i-cookie is in the awaiting-list, the whole packet is put into a response list (called resp_list), again using the i-cookie as the key. Now the i-cookie is deleted from the awaiting-list, because it is no longer awaiting: the reply has arrived. Subsequently the recv_server broadcasts all waiting discovery threads using the conditional variable.

Every waiting discovery thread wakes up and checks if its i-cookie is still in the awaiting list. If it is, it goes back to its timed wait - it is still waiting. If the i-cookie is not found in the awaiting list, the discovery thread checks the resp_list for its i-cookie. If it is not found, there has been an error with the request, the error is reported to the scout discovery module and the thread ends. If the i-cookie is found, the discovery thread retrieves the whole response packet from the resp_list and starts extracting the necessary data (like the SCOUT_R_COOKIE which might have to be passed to the caspd). Using the packets source address, the discovery thread learns the address of the NHOP and transmits this information to the scout discovery module in the caspd using the previously established local unix socket. The cd_gethop function in the scout discovery module now stores the destination and received NHOP in the discovery cache and passes its information about the NHOP to the calling procedure of the caspd.
3.3.2. Scout Discovery Module

When we implemented the scoutd and the caspd it quickly showed, that we could benefit largely from using a plugin architecture for discovery modules. A plugin architecture provides the possibility to select certain parts of the program dynamically at startup. The basic functions and functionalities are defined in the plugin architecture. In the main program the functions use callbacks and wrappers. These wrappers pretend to be the function providing the functionality. In truth they are just empty shells that call upon the functionalities of the loaded plugin. The plugin itself is not a normal program, it only contains the functions defined by the plugin architecture. This way different means for discovery can be implemented at the same time, choosing which one is most suitable for the task at hand at startup time. The caspd implemented a plugin architecture with callbacks, defining the basic functions that a discovery module should provide. The implementation of the scout discovery module provided the opportunity to freely implement the communications between scoutd and the discovery module, freeing us of the need to depend too largely on each other to program functions that would communicate well with each other.

The scout discovery module provides all necessary means to transmit the requests of the caspd to the scoutd. Every function of the scout discovery module establishes a communication with the mlayer-server of the scoutd using the predefined local unix socket. It transmits its request and necessary data if applicable. Afterwards it is either finished (if no results from the scoutd are expected) and exits, or waits for the scoutd to finish its assigned task and transmit the results.

We introduced a discovery cache to minimize the necessity of dynamic discoveries. The problem is that dynamic discovery takes time to compute and time to transmit the packets over the wire. To improve speed and reduce network traffic, we introduced this cache. We decided, after some discussions to incorporate it into the discovery module. This integration enables the discovery module to save further time. There is no inter-process communication needed if the NHOP for a destination is already in the cache. This is the main argument for including the cache there.
3.4. Experimental Testing

3.4.1. Testbed Setup

The testbed consists of 3 machines, sometimes reinforced by another two machines. These two machines are a laptop (ltfmeyer) and another machine (nodens). The topology is shown in figure. The specifications of the three testbed machines are shown in Table 3.1:

![Testbed topology](image)

Table 3.1.: Testbed specification for PCs in most setups

<table>
<thead>
<tr>
<th>Feature</th>
<th>ap28 and ap29</th>
<th>ap20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modell</td>
<td>Dell OptiPlex GX260</td>
<td>Dell OptiPlex GX260</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel Pentium 4 2.0 Ghz</td>
<td>Intel Pentium 4 2.53 Ghz</td>
</tr>
<tr>
<td>L2 cache</td>
<td>512 KB</td>
<td>512 KB</td>
</tr>
<tr>
<td>RAM</td>
<td>512 MB</td>
<td>1 GB</td>
</tr>
<tr>
<td>Harddisk</td>
<td>40 GB</td>
<td>80 GB</td>
</tr>
<tr>
<td>Network interface 1</td>
<td>3c905C-TX/ TX-M (100 MBit)</td>
<td>3c905C-TX/ TX-M (100 MBit)</td>
</tr>
<tr>
<td>Network interface 2</td>
<td>Intel 82540EM (1 GBit)</td>
<td>Intel 82540EM (1 GBit)</td>
</tr>
<tr>
<td>Operating System</td>
<td>Debian Linux</td>
<td>Debian Linux</td>
</tr>
<tr>
<td>Kernel</td>
<td>2.4.20-gx260</td>
<td>2.4.20-gx260</td>
</tr>
</tbody>
</table>

All three machines are running a Debian Linux distribution which was initially installed with
the same software and configuration (as far as the hardware allowed). The machines differ in their software selection because each member configured its own machine to its own needs.

The laptop ltfmeyer was running knoppix 3.2 which is also based on a Debian linux distribution. The machine "nodens" was running knoppix 3.2 as well. These two machines were not used for performance testing. They were used for functionality testing. The machine "nodens" for example was used to check if scout discovery works through an IP-cloud of CASP-unaware hops. These two machines’ specifications are given in Table 3.2.

<table>
<thead>
<tr>
<th>Feature</th>
<th>ltfmeyer</th>
<th>nodens</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modell</td>
<td>Toshiba Portege 7140ct</td>
<td>Custom built</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel Pentium 3 512 Mhz</td>
<td>AMD Duro 1 Ghz</td>
</tr>
<tr>
<td>RAM</td>
<td>128 MB</td>
<td>512 MB</td>
</tr>
<tr>
<td>Harddisk</td>
<td>30 GB</td>
<td>30,40 and 14 GB</td>
</tr>
<tr>
<td>Network interface</td>
<td>Belkin USB2network adapter (10MBit)</td>
<td>Realtek RTL-8139 (100MBit)</td>
</tr>
<tr>
<td>Operating System</td>
<td>Knoppix 3.2</td>
<td>Knoppix 3.2</td>
</tr>
<tr>
<td>Kernel</td>
<td>2.4.22-xfs</td>
<td>2.4.22-xfs</td>
</tr>
</tbody>
</table>

We used mainly two setups for our experiments (as shown in Figures 3.4 and 3.5)

Figure 3.4.: Setup A (e.g. used for stress test)

Figure 3.5.: Setup B (e.g. used for client wait test)
3.4.2. Reliability and Functionality

To test the reliability and stability of scoutd it has been tested by using it intensively inside our testbed. It was tested using different setups. We tested it using a pure scoutd-discovery environment in which it proved itself stable. It showed no crashes or unwanted behaviour after several hours of use with varying intensity (number of discoveries varied through the testrun).

Other tests have been made trying to discover through an IP-cloud of unaware hops using my mobile computer. The scoutd worked reliable even used from a dial-up connection over the internet. Having lots of CASP-unaware hops in between the scoutd was still able to find the NHOP without trouble.

There have also been tests with a heterogenous testbed. The nodes ap28 and ap29 were running a static discovery with fixed NHOP and PHOP values and ap20 in between those nodes was using scoutd. The other nodes still had to run the scoutd so that it could answer to scout requests but it was not used to discover the NHOP. This setup has also proved itself stable. All nodes were able to establish an CASP association from ap28 to ap29 traversing ap20. Neither discoveries nor the sending of messages have been any trouble.

We also did some stress-tests with a large numbers of discoveries per minute. These proved that scoutd is very reliable and keeps its fast responsiveness. There are detailed information about those tests and their results in 3.4.4.

3.4.3. CPU, Memory and Time Consumption

To determine if there are memory leaks and which functions are the most costly in scoutd the tools valgrind, kcachegrind and mpatrol (part of the debian distribution) were used. The output of valgrind, a memory debugger were computed using kcachegrind. These tools were applied repeatedly while we were implementing. We used them to ensure that we have no memory leaks or similar problems. If we found such problems we fixed them immediately. As a side-effect of these checks we know that the by far most costly function in scoutd is the OpenSSL random function. It is responsible for the cookie generation. This random function takes up more CPU time than all other functions combined. In fact all other functions can be disregarded in terms of CPU usage, compared to the random function. The generation of the cookies usually starts with a duration of approximately 350μsec. Beginning from the second time a cookie is generated the values are more than ten times less as seen in figure 3.6 below.

<table>
<thead>
<tr>
<th></th>
<th>minimum</th>
<th>average</th>
<th>maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cookie generation</td>
<td>18μsec</td>
<td>25μsec</td>
<td>40μsec</td>
</tr>
<tr>
<td>Mutex locking of awaiting list</td>
<td>0μsec</td>
<td>1μsec</td>
<td>1μsec</td>
</tr>
<tr>
<td>Mutex locking of response list</td>
<td>0μsec</td>
<td>1μsec</td>
<td>20μsec</td>
</tr>
</tbody>
</table>

The most memory consuming parts of the scoutd are a number of necessary lists which are kept in binary search trees for faster access. These lists can grow quite large over a longer period of use. Some examples are the awaiting response list and the response list mentioned in 3.3.1 as
well as the discovery cache. But the discovery cache has been moved into the scout discovery module which resides inside the caspd, so that it can be disregarded when measuring the scoutd.

The most time consuming parts in real time environment might be something completely else. In the worst case, waiting for the response packets to arrive can take a long time (maximum of 512 seconds until the hard timeout is reached and the discovery signals a failure). The other thing is, that because of the use of pthreads all mutual data structures (as seen in 3.3.1) have to be accessed using a mutex lock. This way data inconsistencies can be avoided but it requires exclusive use of the data structures. Only one thread can access a data structure at a time, all others have to wait for it to finish. So they wait in a queue and are given access to the structure one after another. When there are a lot of threads, e.g. when lots of discoveries are pending, the time to wait for access to the data can grow large as well. But this is a worst-case scenario, as figure 3.6 illustrates, the average time to wait for a mutex is very short (sometimes below 1\(\mu\)sec).

In a test with many discoveries in a short period of time we gathered data that shows that in most cases the mutex-wait times are very short. We tested by timestamping with \(\mu\)sec accuracy one time just before trying to acquire the mutex and then a second time directly after the mutex was acquired. Then we calculated the difference and wrote that value to a file.

With the results from these tests we saw that the mutex locking does not represent a problem in the tested setup at least. The mutex locking times were about 1\(\mu\)sec or even so low that our test showed 0\(\mu\)sec, this happened many times. The maximum for acquiring a mutex was 20\(\mu\)sec for the response list. As this is happened only once in over 600 accesses to that list and an overall of about 3000 acquired mutexes in the whole test, we can assume that value is a measuring error.
3.4.4. Responsiveness

To determine the time it takes for scoutd to discover the NHOP under different circumstances we tested using a driver to trigger scoutd called "caspd_sim". This simulator does nothing but triggering a discovery to a given destination. The times were acquired using ethereal to capture the scout request and response packets. The time in the following graphs are the time from capturing the outgoing request packet until the incoming response packet was caught. The times are displayed in $\mu$s.

![Figure 3.6.: Request/response duration](image)

The different curves in figure 3.6 are from different tests runs. When the data for the red curve was acquired, the scoutd had been running for about 3 days. On the third day we started the stress-test with a discovery request every 10 seconds for several hours. We acquired more than 550 values.

For the green curve the scoutd was freshly started and ran for about 24 hours. This test was not intended as a stress test and only sent a discovery request every 10 minutes. Those values show the most promising results as they show a very stable response time with low variance. The response times are quite stable over a long time. In realtime the data for the green curve was acquired over a much longer period of time. There are no significant rises. The red curve is not as promising, because the response times show a higher variance.
Table 3.4.: Request/response duration

<table>
<thead>
<tr>
<th></th>
<th>minimum</th>
<th>average</th>
<th>maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>stress test</td>
<td>134µsec</td>
<td>353µsec</td>
<td>635µsec</td>
</tr>
<tr>
<td>longterm test</td>
<td>202µsec</td>
<td>243µsec</td>
<td>298µsec</td>
</tr>
</tbody>
</table>

Table 3.4 gives an overview of the results that whole set of data provide. As mentioned Table 3.5 only shows an excerpt of the data, because there are more than 550 values.

Table 3.5.: Client wait times and corresponding request/response duration

<table>
<thead>
<tr>
<th></th>
<th>minimum</th>
<th>average</th>
<th>maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>request/response duration</td>
<td>228µsec</td>
<td>352µsec</td>
<td>406µsec</td>
</tr>
<tr>
<td>client wait duration</td>
<td>7062µsec</td>
<td>12214µsec</td>
<td>80705µsec</td>
</tr>
</tbody>
</table>

The values in Table 3.5 show how much time the caspd had to wait for an answer from the scoutd. The data was acquired using the same timestamping method that is described in section 3.4.3. Even though the values are much higher than the actual waiting for packets on the net, they are still very could - we are still dealing with µseconds.

In summary the results showed that scoutd is able to provide very good response-times under all tested circumstances. We can safely assume that scoutd is able to find its NHOP very fast under these circumstances. The green curve shows that even other network traffic does not interrupt the good performance of scoutd. While the data for the green curve was acquired the machine with the discovering scoutd was providing NFS volumes for the other two nodes.
4. Conclusions and Future Work

This chapter summarizes our experiences with GoCASP its scoutd and provides details for future works.

4.1. Conclusions

In conclusion we can say, that our prototype implementation achieved most of its goals. Some issues still have to be investigated and new features considered but the initial goals have been reached. A summary of the goals and the status of them can be found in the following list.

**Conformity to the specifications:** Successful with noted exceptions (e.g. discovery cache).

**Functionality:** Successful, all basic scout functionalities are supported.

**Reliability:** Successful, scoutd is able to work over a long period of time.

**Fast responsiveness:** Successful, scoutd is responds fast in all experiments so far

4.2. Next Steps

There are obviously many open issues in GoCASP and we are planning where to go next. There are some plans for GoCASP as well as some scoutd specific issues that should be investigated.

4.2.1. General Concerns

The next steps for the entire GoCASP include among the first the support for IPv6. There should not be too many modifications necessary to support IPv6 as well as IPv4. The expansion to IPv6 should help GoCASP to become more interesting for long-term projects and IPv6 testbeds.

Another aim for GoCASP is the port to GIMPS, the IETFs advancement of CASP. GIMPS is a merge of CASP and RSVP to form a ‘General Internet Messaging Protocol for Signaling’. But this aim has to be discussed. As GIMPS is meant to provide a merge of the best parts of CASP and RSVP, it might be beneficial for all if we continue to enhance and test a pure CASP implementation. This would help GIMPS by refining CASP.
4.2.2. Scoutd Specific Concerns

There are some further improvements that have come to mind during the implementation and experiments.

Additional testing includes tests with varying degrees of network load, e.g. from a lightly loaded network up to a congested network by introducing background traffic.

The first improvement is a more intensive testing of the flooding behaviour and consideration of the addition of a DoS detection and defense mechanisms. One very basic protection from a simple replay-attack can easily be implemented. We just have to add a simple check if the cookie of an incoming request has already been received. If this happens more than once, we can assume a replay or DoS attack is in progress and consider countermeasures like ignoring further packets from that host for a while. This is similar to the means used in the newer WLAN protocols and introduces further security risks (e.g. a malicious node could fake an IP-address and get an innocent node to be ignored). An addition of such a countermeasure should first be discussed thoroughly.

In the CASP specification is an option for capability based discovery. We can consider if we want to support this in GoCASP. This broadens the possibilities largely, but also introduces some new complications for scoutd. Apart from that it can void any considerations for a discovery cache, as discussed in section 3.2.3.

If we decide not to add capability based discovery we can consider enhancement of the discovery cache. Possible enhancements include the addition of some kind of network topology cache. This cache could be acquired parsing the routing table (e.g. on linux systems by parsing /proc/net/route) and then discovering the CASP-awareness of the nodes in that table. Such a cache in combination with a routing table lookup prior to the scout discovery might be able to cover up to 70-95% of all discovery-needs. The drawbacks include the introduction of a lot of network load whenever a CASP-aware node starts up and checks all neighbouring nodes for their CASP-awareness. Furthermore the scout security through cookies is voided. If connections are established using the discovery cache then no discovery packets are sent over the wire. This also means that no cookies are exchanged. Without the exchange of cookies there is no cookie-based scout security.

Because the cookie generation takes up most of the CPU time (as discussed in section 3.4.3), the idea of pre-generating a number of cookies arose. When scoutd generates a number of cookies in idle times the response times under a high number of discoveries can be improved largely. The idea is to implement a buffer that can store a certain number of cookies. For example tens or hundreds of cookies. The actual number should be determined by intensive tests. As an example we assume that we choose the size of 100. We now start to generate cookies whenever the scoutd is idle and the buffer is not full. We keep a variable 'used' that points to the index of the cookie last used. All cookies before this index have been used and can be replaced by fresh ones. We keep another variable 'refreshed' that points to the index of the last freshly generated number. We assume that scoutd just used cookie number 100. That means that all cookies in the buffer below index 100 have already been used. If the scoutd now enters an idle time, it checks for the variable "refreshed". We assume that no cookies have been generated, the scoutd was always busy since using the first cookie. Now as the scoutd is idle and can afford to spend more CPU time, it starts generating new cookies to insert into the ringbuffer. It increases the 'refreshed' variable whenever it enters a fresh cookie, replacing the used one. When the 'used' variable
reaches the maximum of 500, it starts again at the beginning. Since scoutd replaced the used cookies in idle times, the variable still points to fresh cookies. This way, scoutd should be able to (nearly) always have a list of 500 fresh cookies. It can use these when it needs to send a response to a scout discovery request or when it sends a scout discovery request itself. The longer response times when lots of requests come in or have to be sent out should be prevented effectively by this method. A reasonable value for the idle time should be determined by experiments as well. We could start the experiments using values of 2 or 3 minutes or even 10 minutes the best value may depend on the existing processing load of the node.

4.2.3. Summary of Future Works

In short summary, these are the future works that were mentioned in this thesis.

- IPv6-Support
- Consider port to GIMPS
- Flooding behaviour and DoS detection/protection
- Consider improvement of discovery cache (e.g. security with cache)
- Consider support of capability based discovery
- Cookie generation (e.g. pre-generation of cookies or implementation of a less costly randomize function)
A. Glossary

**CASP:** Cross-Application Signaling Protocol

**GoCASP:** University of Göttingen’s CASP implementation. When writing "GoCASP" the first release "0.1.0" is meant for the scope of this thesis

**caspd:** The CASP daemon of GoCASP 0.1.0

**scoutd:** The CASP scout daemon of GoCASP 0.1.0

**m-layer:** The messaging-layer of the caspd

**m-session state:** A signaling session maintained in the mlayer of caspd

**scout discovery module:** A plugin module that interconnects the caspd and the scoutd of GoCASP

**scout cookie:** A random number that is exchanged between hosts to verify the identity of the peers

**NHOP:** Next hop

**PHOP:** Previous hop

**CASP-aware:** A network node that supports the CASP

**DoS:** Denial of Service, an attack that aims to overload a certain service in order to render it unusable

**replay-attack:** an attack that uses replaying of captured network packets

**NFS:** Network File System
References


