A Prototype Implementation and Experimental Test of the Messaging Layer of CASP: a General-Purpose Internet Signaling Protocol

Sebastian Willert

am Lehrstuhl für Telematik

Bachelor- und Masterarbeiten des Zentrums für Informatik an der Georg-August-Universität Göttingen

9. 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 9. Februar 2004
A Prototype Implementation and Experimental Test of the Messaging Layer of CASP: a General-Purpose Internet Signaling Protocol

Sebastian Willert

Abstract

Signaling has become necessary for allowing IP-based networks to manage states in network nodes. However, existing protocols cannot deliver desired signaling services needed for large-scale deployment.

The Cross-Application Signaling Protocol (CASP), a general-purpose signaling protocol, introduces a new approach to overcome the limitations of these protocols. This thesis reports a prototype implementation of the CASP transport layer protocol using TCP as the underlying transport protocol, and studies the feasibility of the modular design. The behavior of this implementation has been analyzed through an experimental testbed. Performance results show that the memory and CPU consumption of the implementation are low even under heavy signaling loads; the round trip time of signaling messages is also acceptable. Although further work will be necessary, critical design choices in CASP have been proved to be feasible.
## Contents

1. Introduction .................................................. 4  
   1.1. Introduction to the CASP Protocol ...................... 4  
   1.2. What is Signaling? .................................... 4  
   1.3. QoS Signaling Protocols .............................. 5  
   1.4. Scope and Thesis Organization ....................... 6  

2. The NSIS Framework & CASP .................................. 7  
   2.1. The Next Steps In Signaling Working Group .......... 7  
   2.2. Problems Addressed by the NSIS WG ................ 7  
   2.3. CASP and the NSIS Framework ....................... 8  
   2.4. Rational of the CASP Design ....................... 8  

3. Implementation Design ........................................ 10  
   3.1. Implementation Goals ............................... 10  
   3.2. Threads ............................................. 10  
   3.3. Responsibilities of the Transport Layer and Clients 11  
   3.4. Connection Sharing / Multiplexing ................ 11  
   3.5. Session Identifiers .................................. 12  
   3.6. Soft States .......................................... 12  
   3.7. Scout cookies ....................................... 12  

4. From Concept to Implementation ............................ 13  
   4.1. Basic Implementation Principles .................... 13  
   4.2. Thread Overview .................................... 13  
   4.3. Class Overview ..................................... 16  
   4.4. Thread Management .................................. 18  
   4.5. Memory Management .................................. 18  
   4.6. Signal Transport .................................... 18  
   4.7. Session Handling .................................... 21  
   4.8. CASP Transport Layer Client Interface .......... 23  
   4.9. Refresh Implementation .............................. 24  

5. Testing ......................................................... 25  
   5.1. Setup of the Test-Bed ............................... 25  
   5.2. The Ping Client ..................................... 26  
   5.3. Round Trip Time Evaluation ......................... 27  
   5.4. Component Runtime Profiling ....................... 28  
   5.5. Memory Requirements ............................... 30
6. Conclusions
   6.1. Performance Evaluation .................................. 31
   6.2. Learnings about the CASP Protocol ....................... 31
   6.3. Learnings about Threading in Daemons ................... 31
   6.4. Open Issues ............................................ 32

A. Glossary ...................................................... 33

B. Bibliography ................................................ 34
1. Introduction

1.1. Introduction to the CASP Protocol

The Cross-Application Signaling Protocol [1] (CASP) is a modular protocol for establishing network control state along a data path between two nodes communicating on the Internet. The CASP framework is defined as a protocol, which includes a general purpose messaging layer (M-Layer), which supports a number of client layers for particular signalling applications (e.g. QoS, MIDCOM). In addition there is a distinct, special purpose client component for next-peer discovery.

CASP has been proposed as an Internet draft for the NSIS Working Group [2] of the IETF. Its major design principles have been accepted and incorporated into the working group protocol specification "General Internet Messaging Protocol for Signaling (GIMPS)" [3]. Many lessons learned while implementing the CASP protocol can also be applied to GIMPS.

1.2. What is Signaling?

Signaling in the Internet is very different from transport. Transport protocols only provide means of getting data from A to B while signaling protocols carry meta information about data flows which is be interpreted in certain nodes along its path.

Signaling is used to install and/or manipulate state in the network. We can distinguish two types of signaling: path-coupled and path-decoupled. Path-coupled signaling messages are routed only through nodes that are in the data path. They do not have to reach all the nodes on the data path and the route taken by signaling and data might diverge between hops that are unaware of the signaling protocol. In the path-decoupled case signaling messages are routed through nodes which are not assumed to be on the data path but most likely are aware of this path and have to interact with the hops in the data flow to manipulate their behavior.

1.2.1. Signaling and QoS

Quality of Service (QoS) denotes the performance properties of a network service, possibly including throughput, transit delay and priority.

Traditional Internet only delivers data in a best-effort way, therefore certain QoS mechanisms [4, 5] need to be introduced for the Internet to support services with performance requirements.
These mechanisms contact each relevant hop in the data path and ask for special treatment for the flow. This task comes naturally to signaling protocols. In fact, generic signaling evolved out of the requirements of QoS protocols.

1.2.2. Other Signaling Applications

Signaling is still mainly considered for Quality of Service. In recent years however, several other applications of signaling have been defined such as signaling for label distribution in Multi-Protocol Label Switching networks, Network Address Translation, firewall traversal or signaling to other types of middle-boxes. Therefore, a generic approach is desirable.

1.3. QoS Signaling Protocols

RSVP [6] (Resource Reservation Protocol) is a unicast and multicast signaling protocol, designed to install and maintain reservation state information at each router along the path of a stream of data. The state handled by RSVP is defined by services specified by the IETF Integrated Services Working Group [7]. RSVP has some major scaling problems and is difficult to be incorporated in Differentiated Services but is implemented widely for multicast streams. Another problem of RSVP arises from the soft-state approach using unreliable transport protocols. This means that quite frequent refresh cycles are needed and leads to a significant overhead in terms of bandwidth.

RSVP is complex and tries to cater to too many different needs. This has been learned and led to a new QoS architecture: full fledged but lightweight protocols based on Integrated Services that are deployed in outer network regions and aggregating protocols on the edges that rely on Differentiated Services and scale very well.

YESSIR [8] (Yet Another Sender Session Internet Reservations) is a lightweight resource reservation protocol that generates reservation requests by senders to reduce the processing overhead. YESSIR builds on top of RTCP, using soft state to maintain reservation states. It also supports shared reservation and associated flow merging and is backward compatible with the IETF Integrated Services model.

BGRP [9] (Border Gateway Reservation Protocol) is a sink-tree based distributed architecture for inter-domain aggregated resource reservation for unicast traffic. It scales well in terms of message processing load, state storage and bandwidth and relies on Differentiated Services. BGRP uses soft states, too, but in contrast to RSVP, refresh messages are delivered reliably, allowing for a reduced refresh frequency.

1.3.1. Problems of Existing Approaches

The RSVP QoS signaling protocol is far too complex. This complexity is reduced slightly in YESSIR/BGRP with the tradeoff that two protocols have to be supported which causes overhead in maintaining interoperability and is far from a desirable
generic solution for signaling problems. One thing missing from the existing protocols is modularity at protocol level, they are constraint to their specific signaling purposes. A further problem is that a proper scheme to handle authentication and authorization of QoS resource requests and a framework for providing signaling message security seems to be missing from most protocols. Finally, none of these existing protocols supports mobility.

1.4. Scope and Thesis Organization

Guided by Xiaoming Fu and Dieter Hogrefe, Fabian Meyer and I implemented and evaluated GoCASP [10], a basic CTLP daemon with scout discovery support. This thesis studies the basic approaches taken in CASP and the prototype implementation of the CTLP part. It also tries to answer a number design questions during the planning phase of GoCASP, such as:

- Is it feasible to use TCP as transport protocol?
- Can a modular design really reduce the complexity of an implementation?
- Can multi-threading be used to reduce the complexity of data synchronization?

They are the focuses of the thesis. Other topics, such as the work on the scout discovery protocol of CASP which has been done by Fabian Meyer, is beyond the scope of this thesis.

After giving an overview on the background of the CASP protocol in Chapters 1 and 2, I will focus on my work for the CASP project [11]. Chapter 3 summarizes the design decisions made prior to the implementation, followed with a detailed description of the implementation in Chapter 4, an overview over the testbed and the results gathered in Chapter 5 and finally a recapitulation of the learnings and a list of open issues in Chapter 6.
2. The NSIS Framework & CASP

2.1. The Next Steps In Signaling Working Group

The Next Steps in Signaling Working Group (NSIS WG) [2] of the IETF is responsible for standardizing an IP signaling protocol with QoS signaling as the first use case. This working group will concentrate on a two-layer signaling paradigm. The intention is to re-use, where appropriate, the protocol mechanisms of RSVP, while at the same time simplifying it and applying a more general signaling model.

For this purpose the NSIS WG has analyzed existing QoS solutions [12] to address their problems and identify the concepts and approaches that have proven themselves useful. The results of this analysis were used as a basis to fundamental guidelines for the design of a next generation signaling protocol. These guidelines were split over a number of papers, addressing requirements for signaling protocols [13], security threats [14] and a design framework [15].

The NSIS suite of protocols is envisioned to support various signaling applications that need to install and/or manipulate state in the network. This state is related to a data flow and is installed and maintained on protocol aware entities in the network. The basic protocol concepts do not depend on the signaling application, but the details of operation and the information carried do.

2.2. Problems Addressed by the NSIS WG

"NSIS will develop a transport layer signaling protocol for the transport of upper layer signaling. In order to support a [modular] approach, the two-layer model will be used to separate the transport of the signaling from the application signaling. This allows for a more general signaling protocol to be developed to support signaling for different services or resources [...]" [2]. This model will help reducing the complexity of the layers and also encourages the creation of modular protocol specifications and implementations.

Addressing sessions within the protocol is independent of the flow endpoints or topological network addresses. Mobility support requires this independence because endpoint addresses of flows might have changed after a hand-off but the flows may still have the same service requirement.

"Security is a very important concern for NSIS. The working group is studying the threats and security requirements for signaling [and] strive for compatibility with authentication and authorization mechanisms.” [2]
2.3. CASP and the NSIS Framework

The CASP [1] protocol and a QoS resource allocation client for CASP [16] were proposed to the NSIS WG for consideration and got a go ahead for exploration. This work is part of the evaluation of CASP protocol family and although the CASP proposal evolved into the GIMPS protocol family, it is hoped that the lessons learned from and larger parts of the implementation can be transferred to the new flavor of signaling. CASP complies with all requirements for signaling protocols set by the NSIS WG [13] and is designed to allow high levels of modularity. The CASP Transport Layer Protocol (CTLP) [1] utilizes an existing reliable transport layer protocol like TCP or SCTP for signaling message transmission and delegates discovery of neighboring CASP entities (CEs) to a specialized and exchangeable client. Transport connections between CEs are reliable channels which are shared by all signaling flows which pass through this entity. Multiplexing the flows on a channel is done transparently by the CTLP implementation. Flows are addressed via Session Identifiers (SIDs) and are not bound to specific endpoint addresses.

2.4. Rational of the CASP Design

The CASP protocol caters to the requirements of the NSIS WG but a few decisions have been made during the specification process which are not readily discernible and not anticipated by the NSIS documents. These decisions do not conflict with the NSIS documents but amend the specification where no detailed procedures were given.

2.4.1. Signaling Message Transport

The designers of CASP chose to delegate signal transmission to tested and true reliable transport protocols for various compelling reasons: “We [the CASP designers] chose a reliable transport since signaling requires many of its functions, such as reliability, congestion control, flow control and fragmentation. This may seem surprising since it is often assumed that signaling applications have low data rates, with small, infrequent packets. However, while this is often true, not all signaling applications are that well-behaved all the time. For example, authentication tokens and user authorization certificates for AAA can easily push the message size to several kilobytes. A CA-signed certificate including the principals public key weighs in at about 5 kB, without signed data. Such large messages will likely require fragmentation and may make congestion control advisable. Also, end systems may decide to rapidly probe for available resources if the network is busy. Since CASP nodes may need to perform time-consuming AAA operations, the processing time for each request can vary, so that flow control is needed to keep a neighboring node from overwhelming it with requests.” [17]

There are a few issues caused by the decision to go with reliable protocols. Reliable transport protocols work on a connection metaphor and such a connection has to be established between the neighboring entities in a signaling path, which takes a reas-
onable amount of time. It is a realistic assumption that the number of neighboring
CEs is quite small for each single CE in the signaling path. Thus, this performance
penalty can be avoided in most cases if CTLP implementations try to keep alive as
many connections as possible without interfering with other system duties.

An added bonus of persistent connection between CEs is the feasibility of secure sig-
nal transmission via existing security protocols on the transport layer like SSL/TLS.
The cost for the initial handshake between two nodes is high in comparison to the
cost of the signal transmission, but this cost will be amortized during the lifetime
of a signal channel. It is even imaginable to try to discover and connect to CEs
up-front so even initial signaling messages passing a node won’t be subjected to this
delay in the forwarding process.

2.4.2. Discovery

CTLP implementations are not burdened with the discovery of the next CEs for a
flow. This duty is delegated to a specialized CASP client for the sake of simplicity
and stability. Discovery of CEs in the Internet is a complex problem and requires
the use of a few technologies not needed anywhere else in a CASP implementation.
On the other hand, some approaches taken for CTLP are less useful in discovery.
Discovery can’t take advantage of a reliable transport protocol as its messages are
so small that it can be safely assumed that they won’t be subject to fragmentation
and the connection metaphor doesn’t work for discovery. You simply can’t connect
to some place you don’t know the address of. One way to account for that is to use
datagrams with the IP_ROUTER_ALERT option enabled, so any CE in the path
of this datagram is notified (see the bachelor thesis of F.Meyer for further details).
Unlike any other functionality in a CTLP implementation, raw sockets needed to
create such datagrams require root privileges in most UNIX flavors so separating
discovery and transmission has an additional practical benefit: it is possible to run
unprivileged CTLP services (just the discovery daemon has root privileges) without
the hassle of in-process privilege separation.

This full-fledged discovery is described in the CASP Scout Discovery Protocol but
the separation of discovery doesn’t limit CTLP to this approach. A static discovery
module has been implemented to demonstrate the flexibility of this approach. This
discovery module is also part of a test harness for the GoCASP implementation and
just returns the addresses of CEs which have been defined at load-time.
3. Implementation Design

3.1. Implementation Goals

3.1.1. High Responsiveness

One of the most important goals for this work was to create a very responsive CTLP implementation to meet the requirements of real-world applications. Thus GoCASP is designed to be multi-threaded instead of a multi-process daemon to minimize setup time for signal processing and reduce complexity in the handling of sessions.

3.1.2. Modularity

As CASP is a generic approach to signaling, it has been tried to keep the GoCASP implementation as modular as possible. After some exploration of possible approaches a decision to go with a plug-in architecture has been made. Plug-in architecture is used widely in GUI applications but has been successfully implemented in the Apache web server.

3.1.3. Low Memory Profile & Modest CPU Usage

The CASP should be able to be implemented on a wide variety of machines so a low memory profile and a modest demand in processing power was another albeit not primary goal.

3.2. Threads

A multi-threaded approach has been chosen to achieve a low latency, reasonable complex CTLP implementation (herein after referred to as M-Layer in accordance with the CASP draft). This allows for a central state and connection repository and fine grained synchronization on data access using established threading mechanisms like mutexes.

The threading approach of choice is a manager-worker model, which is agreeably the model most fitting to the demands of a networking daemon (see Section 4.2.1 for a longer discussion).

It is hoped to encapsulate the clients from vital control functionality as far as possible because of the plug-in architecture of GoCASP. Multi-threaded plug-in daemons are seldom implemented because of the risks that ill-behaved clients can disable the
whole service but the gains of this approach out-weight this potential drawback. Nonetheless, it has been cared for maximal containment of the client code. Clients are encouraged to access thread specific memory only, which is cleaned up by the worker threads after the client returns from processing to minimize the impact of memory leaks within the client code.

3.3. Responsibilities of the Transport Layer and Clients

The CASP specification has few provisions which functionality should reside in the CTLP implementation and which are left to the clients. Obviously we wanted as much functionality within the CTLP implementation as possible without limiting flexibility. Some tasks are natural to the CASP transport layer: connection management, meta information about flows, scheduling and timing come to mind. But what about the actual states? Can a generic signaling transport layer encompass all functionality needed for soft states? Should there be a facility for hard state management in this layer? And how do clients interact with the M-Layer?

After the careful consideration described in the last section it has been decided to let clients be part of the CTLP daemon process and let them be controlled by the worker threads. This way it was possible to offer a wide range of services to the client without the need to worry about performance penalties and additional complexity of a multi-process implementation. Hard state support is unimplemented for now (barring states with ridiculous high timeouts), as it is out of scope for this work. Basic soft state facilities were implemented as they were the main focus for the testing. A more in-depth analysis on this problem will be needed when the implementation of more complex clients is started.

3.4. Connection Sharing / Multiplexing

Connection management and multiplexing is done in a centralized facility within the M-Layer. Because of a current lack of system support, certain interactions with sockets can’t be truly multi-threaded. Therefore, those operations are currently carried out sequentially and synchronization points are placed on all socket access operations. These synchronization is done with a per connection granularity and should not cause a significant performance loss in real-world applications. Nonetheless the performance of this facility could be improved by more sophisticated management which should allow for multiple channels between two CEs and a spare connection repository. It is unclear however if this feature would increase the number of open connections beyond a tolerable limit.

Another decision that has been made was to use sockets to other CEs in a one-way fashion only. This doubles the number connections that have to be kept open but simplifies socket management and threading by an order of magnitude. This approach is not uncommon in networking: CORBA for example hasn’t had support for synchronous transfers up until version 2.3 [18] even though CORBA IIOP (the transport protocol of CORBA) is a classical query-response protocol.
3.4.1. TLS Extension

As GoCASP is based on TCP, it should be quite easy to add secure TLS transport for signals without authorization and accounting. This would reduce the danger of injected signals, flow takeover, eavesdropping and a few other security threats detailed in the NSIS WG paper Security Threats for NSIS [14]. Right now there is no protocol or specification concerning AAA in CASP so this area wasn’t investigated further.

3.5. Session Identifiers

A 128 bit sequence is used as Session Identifier (SID) according to the CASP specification. This supersedes the Flow Identifier used in previous signaling protocols. Crypto-strength randomness is ensured by the use of the OpenSSL library. This conforms to the procedure proposed for confidentiality protected session identifier (which we will implement through the use of the proposed TLS extension): “It is obvious that the session identifier must be chosen in a way which does not allow an adversary to guess it. One possibility is to choose the value for the Session Identifier randomly with each session. It must be ensured that the identifier is sufficient large (e.g. 128 bits).”[19]

3.6. Soft States

Efficient state management is a major concern for signaling services. Instead of reinventing the wheel GoCASP was implemented using the `tsearch` function family provided by any System V or X/Open conforming C library, which provide a mean access time proportional to the logarithm of the number of session. These functions are implemented using balanced binary search trees which are fast for searches and tree walks, but tend to be expensive when adding or deleting sessions (see Section 4.7.1 for details).

Although it is possible to assign tree maintenance to a separate thread which constantly walks the tree and eliminates expired sessions the implementation of this has been put of to the next version of GoCASP as the test results for this work should be unaffected.

3.7. Scout cookies

Preliminary support for scout cookies is in place and has to be amended for full conformance with the CASP specification. Right now there are no checks for valid cookies in M-Layer and this feature has been delayed until the behavior in case of an error is specified.
4. From Concept to Implementation

4.1. Basic Implementation Principles

GoCASP is implemented in ANSI C which is the lingua franca of the people involved in research and development in the telematics field. Although ANSI C is not capable of many object oriented design features (notably polymorphism and proper encapsulation) it has been tried to use object oriented design principles wherever possible to improve the clarity of the implementation. Even if it is technically incorrect the terms object and class are used when referring to functions operating on a specific set of data.

Multi-threaded processing of application requests for CASP services was not completed in time and has not been included into the release of version 0.1.0 of GoCASP. For the sake of clarity this functionality will be excluded from discussion in this chapter.

4.2. Thread Overview

4.2.1. Threading Model

In the manager-worker model [20] (also known as master-slave) there are two types of threads:

Manager This is normally a single thread, although it can sometimes be split into several ones dependant on the system. It is responsible for the overall computation as well as the creation of worker threads, and as necessary the marshalling of the workers to complete the overall objective.

Worker There can be any number of worker threads. They are created and deleted as required to provide the necessary functionality based on specific requirements.

Typically the Manager will respond to external requests or events creating workers to satisfy these needs. Workers in this model are not considered to be cooperative, they perform tasks in isolation and are coordinated by the manager thread who controls allocation and resources as required.

This model is unlike the work-group model [20], another well-known model for thread inter-operation. In a work-group model all threads are considered to be equal partners. The ethics of the work-group is to collaborate to satisfy the overall goal of
the system requirement. If the overall system has independent tasks that can be created and executed concurrently or pseudo concurrently then one can build the model based on requirements and map each element of the overall goal to either one or more distinct individual threads.

As this feature is not necessary in the handling of CASP messages the manager-worker model was chosen.

4.2.2. The Manager Thread

![Manager Thread Overview](image)

The main responsibility of the manager thread is to supervise the main event loop. This event loop is implemented as a select loop monitoring all open sockets for a change in status (e.g. connections requests or incoming messages) and a timeout that doubles as internal timer. The timeout interval is currently set at one second. This is not very fine-grained but more frequent timing events are not necessary; these events are mainly used to refresh states and removed them when a time out occurred.

The manager thread is also the instance that handles connection requests. Requests from foreign CEs are accepted here and the established connection handle is stored in the repository until the remote entity closes the connection or the process exits. There is no notification to connected clients when a connection shuts down.

An overview of classes interoperating in the manager thread is given in Figure 4.3.
4.2.3. The Worker Thread

![Diagram of Worker Thread]

Each incoming CASP message is processed by a dedicated worker thread that analyzes its content and modifies client independent-session information. When a worker thread is created from within the manager it expects a copy of the CASP message and potential client data objects to be available in its thread specific memory segment. The worker determines if this CASP entity supports the desired client protocol and passes the message up to the respective client module. If unsupported client data is encountered, the message is labeled as tainted and forwarded to the next CASP entity without further processing.

An overview of classes interoperating in the manager thread is given in Figure 4.4.
4.3. Class Overview

**caspd** The loader which initializes client and discovery modules and then hands over control to the management thread namely the connection broker.

**casp_connection_broker** The main event loop, which listens for new incoming requests from other CEs and applications. It also keeps track of and manages associated sockets (e.g. removes them from its listening stack when the remote end closes the connection). After identifying the socket it asks the M-Layer or the respective client to receive and process them.

**casp_mlayer** This class takes control when the connection broker detects an incoming CASP message. The byte stream of this signal is read in its whole length (if possible), a message is created and the data passed to it for parsing. Control is passed on to an idle message thread afterwards.

**casp_message_thread** This class cares for thread management and implements the message processing threads. When a thread object is called with an arriving message, the message object gets copied into the memory slot of an idle thread, which handles the message thereafter.

**casp_message** A *casp_message* object represents a single CASP message and provides accessors and mutator functions for message data. It also provides functions to create a message object from an octet stream and to assemble and send an encoded CASP message to another CASP entity.

**casp_sockets** The connection repository. Objects that need a connection to another CASP entity are required to retrieve a connected socket through this class.

**casp_session** The session repository. This class is responsible for session storage and retrieval. It also implements session repository maintenance and removal of timed-out sessions.

**casp_client** This class implements the dynamic client loader/unloader and is intended to become a full-fledged client adapter.

**casp_dsc** Loader and adapter class for the discovery module of choice.

**casp_ctlp** Service provider for client modules. Each M-Layer function available to CASP clients is available through this class.
Prototype Implementation and Experimental Test of the Messaging Layer of CASP

Figure 4.3.: Classes interoperate in the manager thread

Figure 4.4.: Classes interoperate in the worker thread
4.4. Thread Management

The manager thread can control a given number of threads (defaulting to 100 but configurable at build time). Each thread is associated with a thread slot in the manager, which provides thread specific information used for memory management (see Section 4.5) and free/busy information. This facility is used to prevent thread race conditions. If a manager thread can’t find any free thread slots while scanning the ring buffer, it blocks further connections while it waits for a threads to become available. This many disable the M-Layer in adverse situations (e.g., infinite loops or dead-locks in clients) but will at least save the system that runs the CASP service. The thread library used right now is the glibc pthread implementation but the implementation may gain from a switch to the Native POSIX Thread Library (NPLT) provided by the new Linux kernel version 2.6 as this kernel becomes more prevalent. Such a switch will significantly increase the portability of this implementation and allow for more fine grained thread control.

4.5. Memory Management

While implementing and testing the first iterations of the GoCASP M-Layer we noticed a significant increase in memory consumption after prolonged execution time. In-depth analysis of this problem showed that no memory leaks occurred and left only one possible explanation of this problem: many asynchronous memory allocations of different size led to a significant address space fragmentation. We decided to solve this problem through the use of thread specific obstacks. An obstack is a pool of memory containing a stack of objects. Freeing one object automatically frees all other objects allocated more recently in the same obstack. [21]

A pointer to an obstack is assigned to each thread slot and used for all non-permanent memory allocations within worker threads. The first object in this stack is the incoming message which the thread has to processes and which doubles as obstack guard. It is pushed there by the manager thread before the worker thread is started. The manager thread relies on the worker thread to free this object, which it does right before exiting, but this is potential dangerous and will be changed in further versions of GoCASP.

4.6. Signal Transport

GoCASP version 0.1.0 supports only IPv4-based connections but IPv6 has been kept in mind while designing the implementation. Generalized sockaddr pointers have been used wherever feasible to ease porting to IPv6 later on and (with very few exceptions) we were able to keep the places where changes are needed reasonably small. The message transport functions of casp_message and the connection management of casp_sockets are affected and it is probably advisable to reorganize these functions into a new casp_transport_layer class.

Although a finer granularity is desirable, signal sequencing is done on a per process
level, i.e. just one thread can write to an outgoing socket at any given time. However, this didn’t influence our test results as we operated only on one signaling channel (which is the finest possible granularity).
4.6.1. Multiplexing

All sockets to neighboring CEs are retrieved through calls to the `caspsockets` object. Sockets are returned in a connected state so it was possible to implement flow multiplexing in this class. Right now, the `caspsockets` object checks its connection repository for open connections whenever being asked for a socket to a specific CE. If it can't find any, a new connection is established, stored in the repository and passed back to the client.

4.6.2. Error Recovery and Handling of Malformed Data

**Network Errors**

Given the fact that we operate on a reliable transport protocol and the CASP specification doesn't advise on details for error handling we decided to forego handling of physical network errors. Many provisions like `route change` needed for mobility in CASP will be beneficial to recovery from physical network errors. It has been decided to postpone this issue until mobility specifications and support are in place.

**Malformed Signals**

Besides physical network errors like unreachable hosts in the signaling path another set of possibly devastating communication errors has been noticed. Invalid length headers or not completed signal transmission can cause following signals to be misinterpreted. From the perspective of the M-Layer a signal channel is just a stream of octets. This stream could only be cut into meaningful chunks due to the length information in each signal header. If a length header misrepresents the actual byte count of a signal, the M-Layer is likely to miss the next header. The random bytes read from the stream afterwards are interpreted as a new signal and the M-Layer tries to read a new header which includes a new length header. This could render a whole signaling channel unusable.

**Malicious Signals**

The behavior described in Section 4.6.2 opens the possibility to Denial of Service attacks against a signal channel: when a malicious source injects signals into the channel that claim to be of arbitrary length but end immediately after the client data header, a whole bunch of following signals (up to 64k bytes of data) is interpreted as client data of the malicious signal and lost without any means of recovery. One possibility to avoid such malicious packets is to rely on heuristics to figure out if a valid signal is read (which could be easily circumvented by attackers) and shut down the whole channel if suspicious data is encountered. The CE on the other end of the channel is required to reestablish the connection and resend the current signal but everything in between may have been lost without traces.

This danger will be mitigated when TLS transport is in place because injecting...
malicious packets into a secured channel is nearly impossible (at least in transport layer and below). Up until then, it is deemed ill-advised to deploy GoCASP daemons in a production environment. As a further conclusion it is advisable to make secure signal transport mandatory for CASP and signaling protocols that use flow multiplexing in general.

4.7. Session Handling

Session handling is of major importance for signaling protocol daemons. A session repository implementation has to cater for different needs. First of all, the retrieval of any session information should be as fast as possible while keeping the cost of creating and deleting new session stores reasonably small. A reasonable overhead in memory consumption (memory needed for other task than storing session information) and good worst-case behavior are also desirable. Finally it should not to be too complex and too costly to implement and fit the usage scenarios specified.

As with any design decisions it was needed to weight in potential drawbacks of an algorithm against its desired properties. As responsiveness was on of the main goals for the GoCASP implementation, a good mean and worst-case search time was a predominant requirement.

4.7.1. Balanced Binary Search Trees

Balanced binary search trees and hash tables were evaluated for internal management of the session repository. They deliver $O(\log n)$ versus $O(1)$ (barring collisions, see below) runtime characteristics for searching.

While $O(1)$ seems more appealing, one has to take into account that a usage scenario for CASP does not exceed 1 million sessions so $O(\log n)$ is not too far behind:

$$\log(1,000,000) \approx 13.82$$

This said, hash table lookups need a far larger constant time than binary comparisons: an optimized SID comparison (which is the unit of measurement for balanced binary search trees) needs a maximum of 2 machine word comparisons on a computer with an 64-bit bus architecture while a hash table lookup needs to calculate a hash value (which may be trivial for even large integers as long as they are randomized and evenly distributed) and has to do some collision checks afterwards which will at least cost 2 machine word comparisons for every subsequent collision.

Because of collisions, there is no good measure of expected runtime of hash tables as it depends on the size and the average fill-rate of the table (which influences the probability of collisions). Even if the expected runtime is quite low, the worst-case runtime with collisions is still $O(n)$. If it is taken into account that a malicious CASP entity can easily force collisions in neighboring hash tables when a trivial hash function is chosen, this becomes quite a concern. Using a non-trivial hash function (such as the one proposed by Wang [22] but modified to support 128 bit integers) solves this problem (non-trivial or real hash functions are collision free) but
using one significantly increases the real cost of hash table lookups so they become far more costly than balanced binary search trees.

Although balanced binary search trees may be a bit problematic with updates, there are algorithms that exhibit \(O(\log n)\) behavior even with this operations. As the tsearch family of functions that has been chosen for the implementation of GoCASP (see Section 3.6) is based on red-black trees, it is very possible that updates are carried out with this complexity, as long as these functions apply modern deletion algorithms. The search operation comes with \(O(\log n)\) naturally even in the most simple implementations. [23]

A major drawback is that it is needed to lock the whole tree when operating on its structure (which includes session retrieval). This is okay for a limited number of transactions but causes noticeable delays when removing large amounts of sessions at once. This occurs, when all sessions that timed out during a tree walk are removed simultaneously and can lock the tree for a discernible amount of time. Adding and removing sessions manually is done one at a time and does not account for this problem.

Figure 4.5 shows the internal structure used to store session information. It uses generic sockaddr pointers to retain addresses to neighboring CEs and flow endpoints. Therefore the session store is independent of the underlying IP protocol and it is even possible for a single flow to encompass IPv4 and IPv6 nodes. Each session is protected by a mutex so subsequent procession of signals is guaranteed and it is impossible for two signals of the same session to be processed in parallel by different threads.
/* magic number to protect session data structures */
#define session_magic 0xf74dc003

typedef struct {
  int magic;            /* session data structure guard */
  sidc_t *sid;          /* session id */
  struct sockaddr *next; /* socket address of next hop */
  struct sockaddr *prev; /* socket address of previous hop */
  struct sockaddr *src; /* socket address of CASP initiator */
  struct sockaddr *dst; /* socket address of CASP responder */
  time_t stamp;         /* last time this session was used */
  time_t timeout;       /* timeout interval */
  pthread_mutex_t lock; /* mutex to ensure only one message of a session is processed at time */
  void *client_state;   /* pointer to simple memory block allocated by client */
  /* to hold protocol specific information */
  void *client_info;    /* pointer to information about the client module */
  /* data structure has to be determined so this is void by now */
} casp_session_t;

Figure 4.5.: Data structure used to store session information

4.8. CASP Transport Layer Client Interface

Clients don’t interact with the transport layer directly but through a wrapping layer that delegates requests to the proper facilities within the transport layer, thus reducing the exposure of code not needed for the clients.

Interaction between M-Layer and clients is two-fold: clients use M-Layer facilities through the wrapping layer describe in Figure 4.6 and the working threads of the M-Layer call clients via call-back functions registered with the M-Layer at client load time.

One notable exception is the call to clients dispatched by local applications. This is done through a socket interface provided by the client. Right now, these calls do not cause a new thread to be spawned but are executed in the scope of the management thread. This has some unfortunate implications and can be improved by introducing a cleaner solution with specialized worker threads for this type of connection. Details
4.9. Refresh Implementation

As discussed in Section 3.6 the GoCASP implementation diverges from most signaling or QoS service implementations in the way refresh intervals are handled. There is no need for hard timeouts or reliable refresh intervals in the M-Layer as long as some constraints are adhered to. Given a session timeout of 90 seconds and a refresh interval of 30 seconds there is no need for pin-point accuracy in the timing of the refresh. With this assumption it becomes possible to implement refresh events not by system timers, which would increase the complexity of the M-Layer significantly. It has been decided to use a tree walk algorithm on the session tree instead. This will lead to a slight inaccuracy in the M-Layer induced refresh (< 3sec) but the gain in simplicity and performance is well worth this drawback. It is planned to allow client modules to set their own timers, however, to allow clients that might have time critical refresh cycles to work with the GoCASP implementation. Another eligible feature would be the provision of call-backs to notify clients about upcoming and reached timeouts to allow them to react to those events.
5. Testing

5.1. Setup of the Test-Bed

The test-bed consists of 3 nodes within 3 physical networks that are interconnected with 2 switches. The hardware specification of the nodes is as follows:

Machine 1 - Initiator:
- Intel Pentium 4 2.0 GHz
- 512 MB RAM
- Debian GNU/Linux 3.0
- Linux Kernel 2.4.20

Machine 2 - Intermediate Node:
- Intel Pentium 4 2.53 GHz
- 1 GB RAM
- Debian GNU/Linux 3.0
- Linux Kernel 2.4.20

Machine 3 - Responder:
- Intel Pentium 4 2.0 GHz
- 512 MB RAM
- Debian GNU/Linux 3.0
- Linux Kernel 2.4.20

Agents:
- Facilities using the services provided by a CASP client module

Figure 5.1: Setup of the test-bed
5.2. The Ping Client

A specialized client module has been developed for the purpose of testing and benchmarking the GoCASP implementation. This client has been modeled after ICMP Ping and does not utilize client states other than sockets for local communication. As shown in Figures 5.2 and 5.3 the data exchanged between the ping client and entities using its service is very simple (the structures in these code snippets are packed, i.e. in network byte order).

```
typedef struct {
    ulong4 dst_addr; /* IPv4 destination address */
    ushort2 dst_port; /* IPv4 destination port */
    ushort2 reserved; /* padding */
} p_ping_client_header_t;

typedef struct {
    char data_p[128]; /* arbitrary chunk of data bytes */
} p_ping_client_cnt_t;

typedef struct {
    p_ping_client_header_t header;
    p_ping_client_cnt_t content;
} p_ping_client_request_t;

Figure 5.2.: Request packet send by agent

```
5.3. Round Trip Time Evaluation

Benchmarking of Round Trip Time (RTT) values is driven by a multi-threaded tool called `cpbenchmark`. `cpbenchmark` spawns \( n \) agent threads that use the services provided by the ping client to initiate a session to a specific destination (Machine 3 of the test-bed). After these sessions are established, \( m \) signals are send to the responder with a randomized interval (3 – 15 sec) in between each signal to avoid congestion. Round trip timings are measured from sending the request packet to the initiator to the completed reception of a response. Each agent stores its RTT values and passes them back to the main thread if all signals are send. After all agents complete their task, the average RTT and the variance of RTTs of this run are calculated and stored. Figure 5.4 shows the average RTT values with the number of simultaneous agents on the x-axis and average RTTs and variance on the y-axis. The values were measured with \( m = 100 \) and \( n \in [1, 50, 100, \ldots, 950, 1000] \).

![Figure 5.4.: Round trip time evaluation results](image)

1000 simultaneous agents come near the limits of the Linux kernel as it does not allow for more than 1024 open file-handles per process and each agent needs a dedicated socket connection for communication with the CASP client module.

While analyzing these result the cost of context switches have to be taken into account. Benchmarking the costs of context switches with the Linux Scheduler Benchmark\(^1\) yields the results shown in Table 5.1 on Machine 1 of the test-bed:

This results put the RTT values into perspective: much of the variance suffered and even a significant amount of RTT might be due to the context switches needed to accommodate all agents. Further testing will be needed in this direction when more resources become available to setup a reasonable test network featuring network taps.

\(^1\)http://www.atnf.csiro.au/people/rgooch/benchmarks/linux-scheduler.html
Table 5.1.: Latency of context switches

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum context switching latency</td>
<td>98.7 µs</td>
</tr>
<tr>
<td>Median context switching latency</td>
<td>101.0 µs</td>
</tr>
<tr>
<td>Average context switching latency</td>
<td>119.5 µs</td>
</tr>
<tr>
<td>Maximum context switching latency</td>
<td>1017.4 µs</td>
</tr>
</tbody>
</table>

and dedicated benchmarking systems.

5.4. Component Runtime Profiling

Another important point for evaluating an implementation is component runtime profiling to discover possible performance bottlenecks in the code. Detailed profiles of all three use cases for a GoCASP entity have been gathered using the call-tree skin for valgrind. “valgrind is a flexible tool for debugging and profiling Linux-x86 executables. The tool consists of a core, which provides a synthetic x86 CPU in software, and a series of skins, each of which is a debugging or profiling tool.”[24]

The test run utilizes the same network topology as described in Section 5.1 but the test harness is notably different. Profiling was done on a single-threaded version of GoCASP as profiles of multi-threaded programs are notoriously hard to analyze. A simple Perl script that created 100,000 sessions through another simplistic client was used as driver for this test. Measurement is done only in the CASP daemon so the performance of the driver is not an issue.

The following sections illustrate the processing overhead for each functionality of the M-Layer that collaborates to meet the challenges posed by signaling transport. The percentage values in the tables therein denote the fraction of total CPU time used by the listed part of M-Layer functionality during the whole test run. The cost in CPU cycles per run below the tables refers to cycles used in the synthetic x86 CPU provided by valgrind.

5.4.1. CASP Initiator

The M-Layer has to accomplish more tasks in the role of a session initiator than in any other role. The generation of crypto-strength session identifiers is quite costly as can be seen from the table. It is not surprising that this leads to a higher CPU cycle consumption than any other role.

---

2 http://valgrind.kde.org/
3 not unlike the Ping client discussed in Section 5.2 but without a response to the agents: the received response is simply ignored
Table 5.2.: Profile of the CASP initiator

<table>
<thead>
<tr>
<th>Task</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing incoming agent packets</td>
<td>85%</td>
</tr>
<tr>
<td>Session creation</td>
<td>70%</td>
</tr>
<tr>
<td>SID generation (OpenSSL)</td>
<td>58%</td>
</tr>
<tr>
<td>Session store creation</td>
<td>11%</td>
</tr>
<tr>
<td>Sending signals (glibc)</td>
<td>9%</td>
</tr>
<tr>
<td>State refresh</td>
<td>10%</td>
</tr>
</tbody>
</table>

Total cost: 1,826 mil. CPU cycles

5.4.2. CASP Intermediate Node

Processing in an intermediate node is the easiest from the perspective of the M-Layer. The only fact that raises its total CPU requirements above that of the responder is that each signal passes this node twice.

Table 5.3.: Profile of the CASP intermediate node

<table>
<thead>
<tr>
<th>Task</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing incoming signals</td>
<td>60%</td>
</tr>
<tr>
<td>Session creation &amp; retrieval</td>
<td>32%</td>
</tr>
<tr>
<td>Session store creation</td>
<td>11%</td>
</tr>
<tr>
<td>Receiving signals</td>
<td>9%</td>
</tr>
<tr>
<td>State refresh</td>
<td>32%</td>
</tr>
<tr>
<td>Removal of timed out sessions</td>
<td>14%</td>
</tr>
<tr>
<td>Receiving signals (glibc)</td>
<td>11%</td>
</tr>
<tr>
<td>Polling the event loop</td>
<td>5%</td>
</tr>
</tbody>
</table>

Total cost: 626 mil. CPU cycles
Cost per signal: 313 mil. CPU cycles

5.4.3. CASP Responder

In terms of functionality the role of the responder does not diverge much from that of the intermediate node, so it is not readily discernible why the CPU requirements are higher per signal than those of the intermediate node. The major reason for this is that the intermediate node has to create a session store only half of the times it receives a signal, as the session is already known the second time.
### Table 5.4.: Profile of a CASP responder

<table>
<thead>
<tr>
<th>Activity</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing incoming signals</td>
<td>55%</td>
</tr>
<tr>
<td>Session retrieval</td>
<td>34%</td>
</tr>
<tr>
<td>Session search</td>
<td>15%</td>
</tr>
<tr>
<td>Session store creation</td>
<td>15%</td>
</tr>
<tr>
<td>Receiving signals (glibc)</td>
<td>9%</td>
</tr>
<tr>
<td>State refresh</td>
<td>30%</td>
</tr>
<tr>
<td>Removal of timed out sessions</td>
<td>15%</td>
</tr>
<tr>
<td>Polling the event loop</td>
<td>5%</td>
</tr>
</tbody>
</table>

Total cost: 576 mil. CPU cycles

### 5.5. Memory Requirements

Memory consumption of states can be quite easily calculated from the source because of the simple but efficient way, state handling is implemented. Other requirements are negligible.

### Table 5.5.: Memory requirements

<table>
<thead>
<tr>
<th>Description</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>each active session</td>
<td>128 to 256 bytes (without client data) depending on the bus architecture of the host</td>
</tr>
<tr>
<td>additional requirements for addresses per session</td>
<td>32 bytes for IPv4</td>
</tr>
<tr>
<td></td>
<td>128 bytes for IPv6</td>
</tr>
<tr>
<td>each thread slot used</td>
<td>1 memory page most frequently 4k bytes</td>
</tr>
</tbody>
</table>
6. Conclusions

6.1. Performance Evaluation

All in all the results of the tests are quite promising for further development of the GoCASP M-Layer. The analysis of component runtime showed no real bottlenecks and proved that the search tree implementation of choice scales well with the numbers of session. The costs of searching, inserting and deleting sessions is comparable, even under the load of 100,000 sessions which is a good indicator that these operations are all in the $O(\log n)$ complexity class.

The RTT values gathered are not as ideal as has been hoped but after weighting in the overhead caused by the testing method (see the discussion about context switches in Section 5.3) it is unclear if they are really significant or caused by a systematic error. Further testing with suitable hardware and testing methods with less impact on the results are desirable and needed for an in-depth analysis of the performance of TCP.

6.2. Learnings about the CASP Protocol

The CASP protocol showed some very nice properties. It provides extensibility through the use of modular clients and discovery modules. The choice of a reliable transport protocol for signal transmission allows for a very clean implementation design. The use of flow multiplexing reduces the overhead of this protocols although it makes the protocol susceptible to DOS attacks using malformed headers. This can be mitigated by the use of transport layer security which is made possible by the concept of persistent signal channels between neighboring entities. The usage of secure transport channels is advisable even without this threat and can be easily accomplished within the CASP protocol.

6.3. Learnings about Threading in Daemons

The implementation of a manager-worker thread model can be considered a viable option for a CASP daemon. Through the use of this model GoCASP is able to deliver very good response times and provide a clean and efficient code base. Synchronization of session data is not an issue in multi-threaded implementations as long as the precautions needed to avoid data race conditions are in place.
6.4. Open Issues

- **QoS signaling client**
  
  QoS signaling is the first and most important use case for signaling protocols. A QoS signaling client for GoCASP will also allow for more in-depth testing of session management performance and a direct comparison with RSVP implementations.

- **TLS**
  
  TLS support is a natural approach to secure CASP messages between CEs as long as reliable transport protocols (e.g., TCP or SCTP) are used. Furthermore, TLS support is necessary to protect the CASP infrastructure from the Denial of Service attack discussed in Section 4.6.2.

- **Error handling**
  
  Error handling for network errors, e.g. *ICMP unreachable* error messages, is nearly unimplemented and needs future study.

- **Adaptive thread management**
  
  GoCASP is currently using a static approach to thread and memory management. Responsiveness could be further increased with pre-initialized threads and no hard limit on thread count.

- **Better connection management**
  
  Connection management could be largely improved with connection keep-alive messages and spare connections for frequently used channels.

- **GIMPS**
  
  Further work has to be done to provide support for the GIMPS protocol [3] which has been adopted as NSIS draft recently.
A. Glossary

Agent  An entity that use the services provided by a CE to initiate a session and to request properties for a data flow.

CASP Entity (CE)  The function within a node, which implements an CASP protocol. In the case of path-coupled signaling, the CE will always be on the data path.

CASP Forwarder (CF)  CASP Entity between a CI and CR, which may interact with local state management functions in the network. It also propagates CASP signaling further through the network.

CASP Initiator (CI)  CASP Entity that starts CASP signaling to set up or manipulate network state.

CASP Responder (CR)  CASP Entity that terminates CASP signaling and can optionally interact with applications as well.

Flow  A traffic stream (sequence of IP packets between two end systems) for which a specific packet level treatment is provided. The flow can be unicast (un- or bi-directional) or multicast. For multicast, a flow can diverge into multiple flows as it propagates toward the receiver. For multi-sender multicast, a flow can also diverge when viewed in the reverse direction (toward the senders).

Data Path  The route across the networks taken by a flow or aggregate, i.e. which domains/sub-domains it passes through and the egress/ingress points for each.

Signaling Path  The route across the networks taken by a signaling flow or aggregate, i.e. which domains/sub-domains it passes through and the egress/ingress points for each.

Path-coupled signaling  A mode of signaling where the signaling messages follow a path that is tied to the data packets. Signaling messages are routed only through nodes (CEs) that are in the data path.

Path-decoupled signaling  Signaling with independent data and signaling paths. Signaling messages are routed to nodes (CEs) which are not assumed to be on the data path, but which are (presumably) aware of it. Signaling messages will always be directly addressed to the neighbor CE, and the CI/CR may have no relation at all with the ultimate data sender or receiver.

Service  A generic something provided by one entity and consumed by another. It can be constructed by allocating resources. The network can provide it to users or a network node can provide it to packets.

SID  The 128 bit long, crypto-random sequence used by CASP as session identifier.
B. Bibliography


