# Implementation and Evaluation of the Cross-Application Signaling Protocol (CASP)

Xiaoming Fu Dieter Hogrefe Sebastian Willert
Telematics Group, University of Göttingen
Lotzestr. 16-18, 37083 Göttingen, Germany
{fu,hogrefe,swillert}@cs.uni-goettingen.de

***Abstract*— In this paper, we describe implementation aspects and performance results of a novel general signaling protocol for the Internet, the Cross-Application Signaling Protocol (CASP). There has been much debate on the applicability of RSVP as a general signaling protocol for the Internet, particularly with respect to its modularity, complexity, security and mobility support. Based on a layered architecture, the CASP design intends to address these challenges, which, unlike RSVP, provides a simpler mechanism for reliability and security by reusing existing protocols for transporting signaling messages. In addition, it supports a wide range of signaling applications. While this concept is considered to be advantageous over RSVP signaling, the actual mechanisms and behaviors of the CASP implementation have not yet been explored. Our study attempts to shed light on this issue by presenting a first public CASP implementation and preliminary examination of its properties. Performance results show and analyze the round trip times and their variances of signaling messages upon different number of signaling requests and different congestion situations in the experimental setup. The memory required for a large number of signaling sessions and the CPU consumption for each routine from profiling the implementation are low. Although further work is necessary, critical design choices in CASP have been proven useful and practically feasible.**

## I. Introduction

The growth in demand and use of "middle box" functionalities such as QoS resource reservations, NAT configuration or firewall pinholes, has been evolving the Internet from a "best effort" network towards a new one with a variety of architectural changes. As a result, it has become common and necessary to install, maintain or remove certain control states in certain network nodes, which end-to-end flows traverse, known as "signaling services". In the near future, a more intensive diversification of such signaling services is envisioned to be provided based on signaling protocols. The Resource ReSerVation Protocol (RSVP) [8], [11], [44] introduced a number of features for Internet signaling, such as soft state, two-pass signaling message exchange, separating signaling from underlying routing protocols, and modularity through the use of opaque objects. Over the last decade, however, the intrinsic design of RSVP have put its scalability and complexity into question. RSVP tightly couples application semantics such as resource reservation with the delivery of signaling messages. In addition, it was designed when IP multicast was expected to widely deploy, IP mobility was in its infancy and NATs were uncommon. Furthermore, a reluctance to change working implementations often leads designers to insert new objects or redefine functionalities to RSVP rather than introduce a new architecture. Thus, there have been a number of proposals for simplifying RSVP even under other protocol names. Most of them address QoS signaling with enhancements in certain scenarios, but little investigation has been carried out concerning modularity and an attempt to accommodate various signaling services instead of single QoS signaling support. The implicit "filtering out all RSVP messages traversing a node" assumption for signaling transport often represents an additional challenge when supporting general signaling for a variety of middle box functionalities. The result is ever-increasing complexity with respect to protocol modularity and functionality.

These considerations suggest that the existing methods of modularity and transport of signaling protocols may not be sufficiently adapted to new challenges of signaling for the Internet. This lack of extensibility may be considered desirable as it forces compliance with existing standards, but in practice, it often results in short-sighted solutions that may violate existing ones. As a result, an alternative design principle is needed, if methods of modularity and signaling transport are inadequate for RSVP to use as a means of general signaling. In [34], [36] we proposed a new approach for signaling referred as Cross-Application Signaling Protocol or CASP.

According to our design, the RSVP model has been enhanced to allow for reliable transport mechanisms (such as TCP and SCTP) for signaling transport, separate signaling application semantics and next-hop discovery from signaling message delivery, and intrinsically support node mobility. The first open release of our CASP implementation has been made available through our website [1].

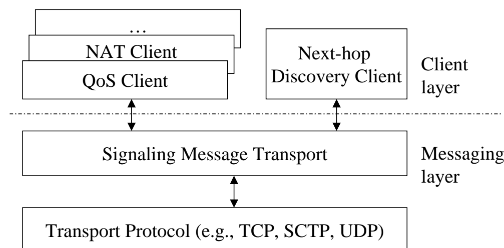
In this paper, we present design details and preliminary performance results of the CASP implementation. In Section II, we describe the CASP architecture. An extensive discussion of the CASP implementation is detailed in Section III, followed with a performance result and analysis in Section IV. After a review of related work in Section V, conclusions and future work are presented in Section VI.

## II. The CASP Architecture and Implementation Goals

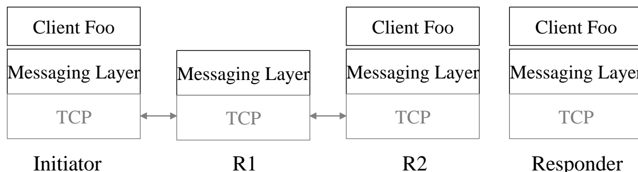
### A. Overview

As shown in Fig. 1(a), the CASP architecture consists of a generic *messaging* layer, which transports signaling messages between the initiator of the signaling session and the

responder, and a *client* layer, which consists of a *next-hop discovery* client and any number of specific signaling client protocols. The client protocols perform the actual signaling operations, such as QoS resource reservation, NAT/firewall configuration, or network diagnosis, in which client data are carried in opaque objects meaningful for various types of middle boxes. Typically, the initiator is the data source and the responder is the data sink, but CASP supports both sender-initiated actions, such as reserving resources, as well as receiver-initiated actions. *Scout protocol*, a common discovery mechanism using RSVP PATH-like message with Router Alert option, has been introduced to actively determine next CASP hop along the path without bothering any application-specific functionality. However, each node can choose its own next-node discovery mechanism, relying on manual configuration, router advertisements, link state routing protocols, scout, or – for loosely-path-coupled operations – server discovery solutions such as DNS or SLP [21].




(a) CASP signaling architecture




(b) CASP signaling example

Fig. 1. CASP – a generic Internet signaling architecture

Modern reliable transport protocols offer flow control, congestion control, message fragmentation and fast loss recovery, which are important characteristics for a generic signaling protocol. For example, public-key-based session setup messages or active network code may be quite large. Such large messages may need fragmentation and congestion control, which are contained in the functionalities of TCP and SCTP, but not within unreliable transport protocols like UDP. Therefore, the CASP messaging layer is built on existing reliable or unreliable transport protocols, such as TCP, SCTP or UDP, depending on the needs of the application (and the availability of the protocol in network nodes). Small, "one-shot" signaling messages can be embedded into the UDP or raw-IP discovery message when efficiency is the first concern, while larger messages and reliable responses then make use of a chain of reliable transport connections (TCP or – when exists – SCTP). Naturally, the end-to-end transport behavior may be determined by the weakest link. In many cases, signaling peer nodes will communicate with each other repeatedly and thus maintain long-lasting connections, avoiding the connection set up latency. As a result, the average session setup latency is, intuitively, low.

Differing from normal CASP messages, CASP scout request messages utilize UDP with an IP alert option for its transport (like RSVP Path), since it typically does not need to perform frequently like periodical refreshes. Although the destination of a scout request is the CASP signaling destination, the first CASP node on its transmission path will respond (with a scout response) without forwarding it on. Scout requests have their own reliability mechanism. They are retransmitted periodically, with an exponentially increasing retransmission interval, starting at 500 ms.

### B. CASP Operation Example

A CASP messaging layer session is established between an initiator and a responder, along a chain of CASP nodes, with a cryptographically random *session identifier* (session ID) chosen by the initiator. Additionally, a *flow identifier* describes the data flow the signaling message pertains to. At each node, the CASP messaging layer remembers its previous CASP node, aside from the initiator; it also determines the next node along the data path, checks if there is an existing transport connection to that node, establishes one if not, and forwards the message downstream. The node then remembers the upstream node and associates it with the session ID, a state refresh timer and a state expiration timer. This ensures that all signaling messages belonging to a same session traverse the same set of CASP nodes in both directions. While delivering generic messaging layer signaling messages, the messaging layer establishes, refreshes, or releases *states* for signaling *sessions*; it also remembers the traversed path by installing state at individual routers (stateful approach) or records a route (stateless approach).

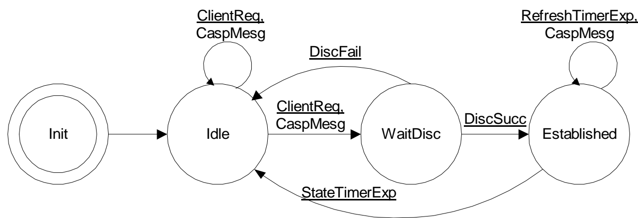
Fig. 1(b) illustrates an example of CASP signaling where TCP is used as the underlying transport mechanism (the SCTP case is similar; when UDP is used, a CASP message will be sent to a discovered next hop without checking/reusing any existing connection). A signaling client ("*foo*") requests the CASP messaging layer for delivery of its services from the initiator towards the responder. It is possible, however, that some intermediate CASP nodes (in this example, R1) do not support the requested client layer. If so, the following operations take place:

1) The initiator creates a messaging layer session ID, and determines next CASP node is R1 (e.g. via a separate scout discovery process: sending a scout request towards the signaling responder and the first node supporting scout will respond with a scout response) and there is an existing TCP connection between the initiator and

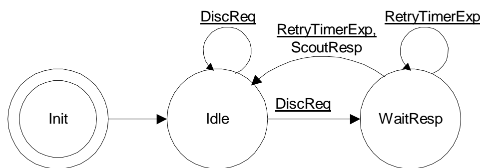
R1. It then generates a CASP message with supplied signaling client payload and delivers it to R1.

2) Upon receipt of the CASP message sent by the initiator, R1 determines it does support the requested client type, and then performs the same procedure as the initiator; additionally, it also remembers previous hop. R2 differs from R1 in that it passes its client payload on to the corresponding client protocol. After that, it establishes a TCP connection between R2 and the responder, when no TCP connection exists.
3) After the responder receives the client data, its client layer may decide to send a response message via the messaging layer to the initiator. This client response message is transparent to the messaging layer (same as other CASP messages) and follows the reverse chain of CASP nodes.

The state diagrams for the basic CASP operation are shown in Fig. 2.




(a) CASP messaging layer




(b) Scout for discovery

Fig. 2. State diagrams for a CASP messaging layer instance and a scout instance

Before describing details of the implementation, we identify the design goals of our target system in the next section.

### C. CASP Implementation Goals

CASP differs from traditional signaling protocols in a number of ways. In this paper we focus on the following two objectives which are important to a CASP implementation:

1) Modularity and efficiency: *Modularity* enables CASP client protocols and discovery mechanisms to be easily implemented, added or modified. CASP inherits RSVP's modularity on the object orientation (i.e. TLV opaque objects), and additionally, modularity in CASP further allows for different signaling transport protocols (upon different application needs and system support) as well as well-defined interfaces between components and with the surrounding environments. Modularity, however, can deteriorate the signaling performance due to costly session setup and maintenance, header processing, as well as interfacing between layers. Nevertheless, as the fundamental block of the CASP architecture, a CASP messaging layer implementation needs to be *efficient* while maintaining modularity. This efficiency may be represented by high responsiveness, low memory profile and low computation time.
2) Dynamic availability: The components can be *dynamically available* in a CASP implementation. First, the number of participating signaling client sessions may vary over time (some can join or disappear without notice). Second, due to soft state refreshes, route change notifications, or session requests of client applications, CASP messages can be generated or received in any node over time. The consequence is that the amount of computation delivered by a messaging layer session may also vary with time.

## III. Implementation

In this section we present the implementation details of GoCASP [1], the CASP implementation at the University of Göttingen. For our initial implementation we chose Linux as the operating system and C as the programming language. We will discuss how difficulties encountered and implementation objectives were resolved through a set of design choices in order to meet implementation objectives. In addition, we will report on key techniques which were introduced in the CASP implementation.

### A. Implementation Architecture

As described above, the first goal of the CASP implementation is modularity and efficiency. Modularity requires that the CASP messaging layer is independent of any signaling client protocol that may relies on it (e.g., QoS resource reservation, firewall pinhole or NAT configuration) as well as any protocol that it relies on (e.g., UDP, SCTP or TCP for signaling transport, and any specific discovery mechanism which it may use: scout, extension to routing protocols, DNS-based or other mechanism). When implementing these protocols, however, they must be incorporated in a target system. Basically, there are two methods to do this:

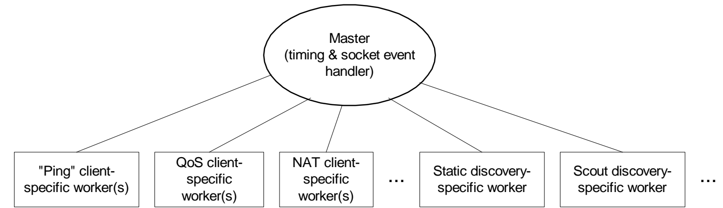
- To connect the processes of each protocol, so that a message sent by one process using the facilities of another protocol is first sent to a process implementing the second protocol. This traditional method allow us to build a hierarchy of protocols.
- To construct protocols together inside a single process, modifying them as necessary. This approach has an advantage in terms of efficiency [14]. On the other hand, it can increase the complexity of the composite mechanism.

Since the first approach requires more communication overhead and processing time, thereby reducing efficiency, the second approach was applied with special care concerning its complexity. Usually, different clients and different discovery mechanisms require different amount of time for their processing. Constructing these protocol components sequentially (i.e. FIFO) may result in unpredicted responsiveness for signaling sessions (i.e. significant variance in round-trip times). One possible solution to this problem is to introduce a scheme that manages all events in a systematic way and distributes different tasks into different handlers. We implemented this by using the *master-worker* paradigm, which has been widely used in implementing web services [20], distributed systems [37] and grid computing [19]. Unlike most existing approaches, where the master and workers are implemented as individual processes, we used *multi-threading* to avoid high communication overhead.

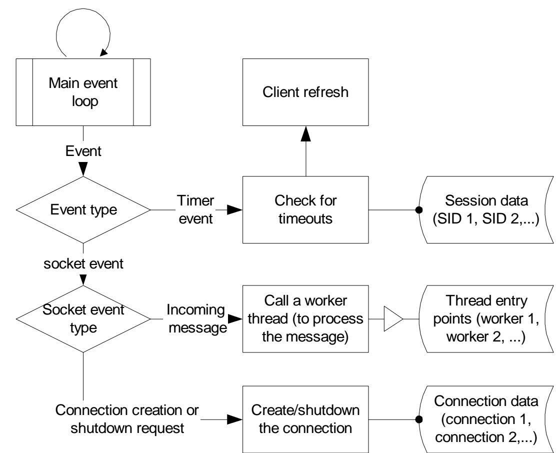
The master-worker method we used is shown in Fig. 3. A *master* thread, or "manager", is responsible for the overall scheduling of tasks, inter-system communication (monitoring sockets), as well as the creation of worker threads, as illustrated in Fig. 3(b). The master examines external requests (upper layer requests and incoming messages from TCP, SCTP, or UDP transport) and events (such as expiration of timers or TCP connection requests), partitions them and delivers the tasks to the workers. A *worker* thread can be created and deleted when necessary, and performs actual work based on its task-specific requirements. Workers do not need to be cooperative, while the master controls the concurrency of workers. The simple communication structure and the well-defined tasks each worker has to perform led to an easy but effective implementation of CASP.

A worker thread is dynamically created (by the master thread) to handle an incoming request (either from the upper layer or the incoming inter-system request). When multiple requests arrive simultaneously, they are stored in a buffer and individual worker threads are then created and processed. When a worker thread finishes the request processing, it exits. We found through our experiments that the master-worker paradigm as a basic communication structure is well suited to implement multi-thread scheduling.

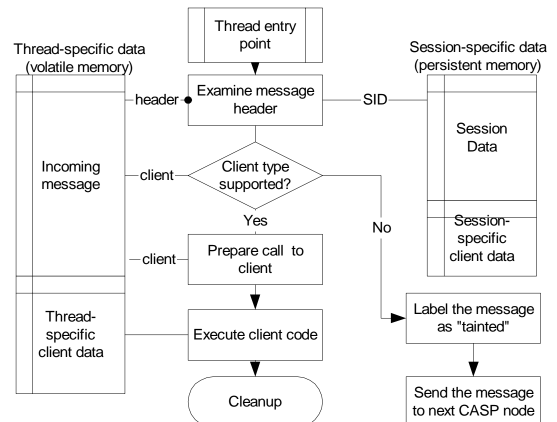
Our CASP implementation uses plug-in architecture, a well-known concept in building GUI applications, to support a wide range of signaling client protocols (i.e. codes) and discovery mechanisms (signaling clients and discovery mechanisms are viewed as the same client level). According to potential needs, we can support up to 256 client types and 256 discovery mechanisms. The master maintains the centric socket- and session-specific data, using well-established threading mechanisms such as mutexes. When the worker thread processes a message and finds the client type or the discovery type is known (i.e. has been registered), the corresponding plug-in daemon will then be executed to handle the client-specific part of data. When a new client or discovery protocol is introduced (i.e. code/finite state machine), a plug-in's entry point is registered for that protocol in the master. Although multiple transport protocols and discovery mechanisms are supported, only one discovery mechanism is currently enabled at a time in the implementation, until a criterion can be determined for dynamically choosing one among several mechanisms at runtime.




(a) The master-worker model




(b) The master thread

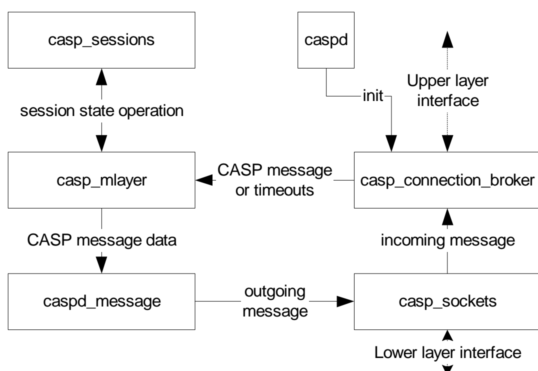



(c) A worker thread

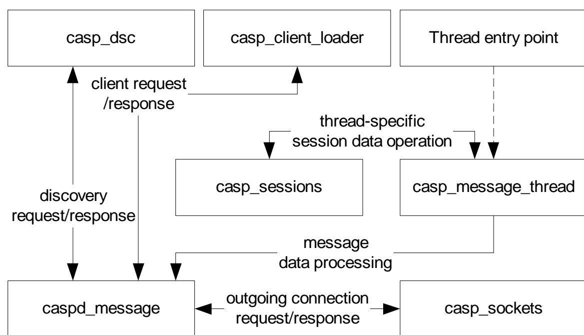
Fig. 3. The master-worker model in the CASP implementation

### B. Class Overview

Fig. 4(a) and Fig. 4(b) show the relationships between the various classes related to the master thread and a worker thread. The master thread (class *casp_connection_broker*) captures both upper requests and incoming messages, disseminates tasks (copies messages into obstack objects) for individual worker threads. When handling an outgoing or incoming CASP message, the master thread in a CASP node selects between connection-oriented (such as TCP, SCTP) and connectionless (such as UDP) transport protocols, currently depending on its configuration. Each worker thread can dynamically load a client code and a discovery mechanism for serving specific signaling purposes. Class *casp_sessions* is responsible for session management including creation, retrieval and expiration of a session. The figure indicates that *casp_sessions* uses the services of the messaging layer and *casp_message* corresponds to callbacks from *casp_mlayer*.




(a) The classes in the master thread




(b) The classes in a worker thread

Fig. 4. Class overview

Detailed descriptions for each class are given in [17].

### C. Soft State Management

Soft state management is the major functionality of a signaling protocol. It is concerned with soft state refresh, session (state) generation, session repository and retrieval.

*1) Soft State Refresh:* Refreshes in CASP differ from most (QoS) signaling protocols. When reliable transport protocols are used, soft state refresh at the messaging layer is not intended for dealing with message losses due to unreliable communications, but rather for dealing with possible changes of network situations such as node or link failures, as well as the client aliveness. Currently, the default soft state expiration timer is set at 90 seconds and the default refresh interval is set at 30 seconds. Refreshes are initiated independently in each CASP hop and each hop receiving such a refresh will re-discover its next hop, in order to keep the up-to-date route information conforming to the end-to-end flows. Once a state timer expires, CASP messaging layer checks whether there is any active client. If there is, it initiates a session refresh towards the responder, otherwise the session will be removed from the local repository. A refresh message carries as many active session data as possible, depending on the underlying transport protocol the messaging layer uses. CASP can also support local repair [11], a faster way for state recovery in a changed segment [1].

Each client can have its own soft state. Typically, its refresh interval needs to be shorter than the messaging layer refresh timer. There is a default value for each type of clients but it can be reset by high layers (aka. agents [2]) when necessary.

Client soft state refreshes are triggered by its signaling end, aside from a few exceptional cases (for example route change cases, where the detecting node's messaging layer triggers all active clients in the closest previous hop(s) to refresh). In our implementation, the plug-in architecture allows the messaging layer to notify clients about timer expiration events, upon which the latter do their specialized work (e.g. client state refresh or removal).

*2) Session creation:* A session needs to be created upon the request of a signaling client application. A 128-bit random sequence is used as the session ID. With a 128-bit length it is practically impossible for an adversary to guess its value [41]. Crypto-strength randomness is ensured by the use of the OpenSSL library. Each session uses generic *sockaddr* pointers to record addresses of neighboring CASP nodes and signaling ends, and has a mutex lock to ensure concurrency for access from different threads. The detailed description of a session data structure can be found in [17].

*3) Session repository and retrieval:* There are several requirements for the session repository implementation. To start, session retrieval should be as fast as possible, while minimizing the overhead of session creation and removal. Second, it is necessary to achieve reasonably low memory consumption and a good worst-case performance. Finally, it should not be too complex and time-consuming. As responsiveness was one of the main goals of our implementation, a limited mean and worst-case search time is a predominant requirement.

[1]The implementation for local repair and mobility support [18], [36] in CASP is planned for future work.

[2]An agent can correspond to (or act on behalf of) a user application, which requests services of one among known CASP client types, such as "Ping", QoS, or NAT. For example, a simple QoS-aware agent can feed the following information to the QoS client: (source IP, source port, destination IP, destination port, traffic specification and QoS parameters).

TABLE I
RUNTIME CHARACTERISTICS OF TWO CANDIDATE SCHEMES FOR SESSION MANAGEMENT

| Scheme | Search time | Computation | Collision | Session add/removal |
|---|---|---|---|---|
| Hash table | $O(1)$ (average), $O(n)$ (worst case) | 2 or 4 CPU cycles per comparison, plus 2 or 4 cycles per collision, plus hashing cost ($\geq$20 cycles) | possible; can be improved at the cost of more memory | $O(1)$ (average), $O(n)$ (worst case) |
| Balanced binary search trees | $O(\log n)$ | $\leq$2 cycles for comparison | none | $O(\log n)$ in modern algorithms |

Two method candidates, *balanced binary search trees* and *hash tables* were evaluated during the design process for internal management of the session repository. They deliver $O(\log n)$ and $O(1)$ (barring collisions) runtime characteristics for searching, respectively. As shown in Table I, this method is faster for searches and tree walks in comparison with hash tables, but can be a little more expensive when adding or deleting sessions. Note that even for a reasonably large amount of sessions (e.g. 1 million), $O(\log n)$ is still comparable to $O(1)$:

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

In GoCASP we implemented a simple but efficient structure for session repository and retrieval, utilizing *tsearch* functions (a C-library in any system conforming to System V or X/Open) which are implemented based on balanced binary search trees, thus providing a mean access time proportional to the logarithm of the session number.

The session repository is independent of IP versions and it is even possible for a single session to encompass IPv4 and IPv6 clouds. Each session is protected by a mutex, so as to provide an exclusive access of several possible incoming messages belonging to a same session.

### D. Memory Management

The master thread can control a number of threads. Each thread is associated with a thread slot in the master, which provides thread-specific information used for memory management and free/busy information. This is to prevent thread race conditions. If the master cannot find any free thread slots while scanning the cyclic buffer, there are two possibilities for dealing with this problem. One possibility is to block further connections while the master waits for a thread slot to become available. However, this may block the messaging layer in adverse situations (e.g. infinite loops or deadlocks in clients). Therefore, a more practical solution is to let a thread expire after a reasonably long period of execution time. We implemented the thread library using the *glibc pthread* implementation and plan to port it to the *Native POSIX Thread Library* (NPLT) provided by the latest Linux kernel version 2.6, to allow more fine-grained thread control.

### E. Messaging Layer/Socket Interface

Socket management and multiplexing is done in a centralized facility within the messaging layer. Limited by known Unix-like systems, certain operations with sockets cannot be fully multi-threaded. Instead, they are currently carried out sequentially and synchronization points are needed for all access operations with a same socket. These synchronizations are done with a per-connection (except UDP) granularity and should not cause a significant performance penalty in real-world applications. Nevertheless, the performance of this facility could be improved by a more sophisticated management scheme which allows for multiple connections between two CASP nodes and a spare connection repository. Note that such schemes increase the number of open connections and thereby making a tradeoff necessary.

It was decided that socket communications between two CASP nodes should be uni-directional. This doubles the number of connections that have to be kept open but simplifies socket management and threading by an order of magnitude. This approach is not uncommon in the engineering world: CORBA for example had no support for synchronous transfers up until version 2.3 [30], although CORBA IIOP (the transport protocol of CORBA) is a classical query-response protocol.

All sockets to neighboring CASP nodes are retrieved through calls to the *casp_sockets* object. Sockets are returned in a connected state so that it is possible to implement flow multiplexing in this class. Right now, the *casp_sockets* object checks its connection repository for open connections whenever being asked for a socket to a specific CE. If it fails to find one, a new connection is established, stored in the repository and passed back to the client.

Although a finer granularity is desirable, message sequencing is done on a per-process level, i.e. only one thread can write to an outgoing socket at any given time, by using mutex locks. This is to avoid blocking when accessing the socket.

### F. Signaling Clients and a "Ping" Client

In our design, we have tried to incorporate all client-specific control functionalities into clients. Meanwhile, the corresponding plug-in accommodates a complete client code as far as possible. Each client code is also restricted to access only its thread-specific memory as far as possible. In order to minimize the impact of memory leaks within a client code, the thread-specific memory is cleaned up by the worker threads after the client returns from processing.

A simple client module, which we call "Ping", has been developed for the purpose of testing and benchmarking the GoCASP implementation. This client is similar to ICMP Ping and allows for any CASP node to access and modify Ping client data along the path. Unlike QoS client, the Ping client has no soft state and uses sockets for local communication with the messaging layer. The structure of (and the interface

between) the ping client and entities using its service are quite simple (note the structures in these code snipplets are packed, i.e. in network byte order) and details can be found in [17].

## G. Next-hop Discovery and the CASP Scout Protocol

The plug-in architecture in GoCASP provides a flexible way to discover the CASP hop towards a given destination. Separating signaling from discovery allows different means for discovery to be implemented simultaneously, and it is then easy to choose which is most suitable for the task by changing the plug-in configuration on demand. We implemented a static discovery module in order to demonstrate the flexibility of this approach and evaluate the main portion of the messaging layer.

Furthermore, the scout protocol for next-hop discovery has proved useful. There is a discovery module located in the messaging layer which interacts with the master thread, and another separate scout daemon communicates with the discovery module through a predefined local Unix socket. Since scout requests include the IP router alert option, unlike any other functionality in our implementation, raw sockets are needed to create and intercept such messages. Therefore, separating discovery and signaling has another additional practical benefit: it is possible to run unprivileged CASP signaling services in the messaging layer (only the scout daemon needs the root privilege) without the difficulty of in-process privilege separation.

To prevent DoS-attacks, a CASP node receiving a scout request should not establish a state. The use of cookies in the scout implementation ensures that the response received matches the request having been sent. To ensure that no adversary can hijack such a connection, the cookies are exchanged securely once again when a security association is established between the two nodes.

## H. Security

Currently, there are three types of CASP messages for different purposes in the CASP implementation, namely the CASP signaling messages, CASP scout request messages and CASP scout response messages.

CASP signaling messages deliver actual signaled data between CASP nodes. They therefore need to be confidentiality protected. The use of TCP/SCTP reliable transport in GoASP makes it possible to use TLS [15] for channel confidentiality and integrity. Due to limited time and resource, this feature will be implemented in the next phase.

CASP scout request and response messages are used for active discovery of next CASP hop to a given destination. Unavoidably, they face the same level of security threats as in RSVP, since a CASP node sending scout requests typically knows nothing about the next CASP hop. The usage and implementation of scout cookies (see Section III-G) provides a basic security for scout discovery. Nevertheless, because the scout process merely discovers the next hop information and does not deliver service-sensitive data, this does not present a real threat.

# IV. Evaluation

We did some preliminary performance evaluation of our implementation through repeated tests, profiling program costs, and performance in a runtime testbed environment. More comprehensive tests will soon be performed, particularly those testing more sophisticated scenarios which are important to prove the effectiveness of CASP — e.g. those involving more nodes and more signaling sessions.

## A. Testbed Setup

As depicted in Fig. 5, our testbed setup consists of 5 interconnected nodes, all being PCs with Via Eden CPU (533 Mhz), 256 MB PC 133 SDRAM, 20 GB Hitachi HDD, running Linux 2.4.26 and GoCASP-0.1.0 (except for Quartus, which is the only CASP-unaware node). All NICs are of the same type: 100Mbps Realtek RTL 8139.

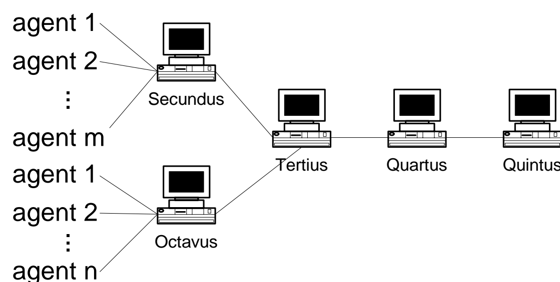



Fig. 5. Testbed setup

The performance of the CASP implementation has been preliminarily measured when running different numbers of agents, each of which requested a CASP "Ping" client service.

As a first step, we use a dedicated socket connection for each agent for its communication with the CASP Ping client module. Thus, 1000 simultaneous agents is almost the upper limit of open file-handlers (1024 per process) that the Linux kernel can support by default in the initiator side. This could be regarded an accumulated effect, since a realistic edge system will only be serving very few number of signaling users but the middle nodes may serve much more. However, Ping, as the only implemented client for the moment, is a stateless client without installing any client session data nor refreshes after a ping transaction is finished. Therefore, the tests were limited to show the behaviors of the messaging layer with a limited number of agents (rather than evaluating the performance under a significant number of, i.e. millions of, sessions). In order to perform more extensive testing of CASP performance, stateful client will be developed and tested in the next phase.

## B. Signaling Round Trip Time

We developed a multi-threaded tool called *cp_benchmark* in order to benchmark the Round Trip Time (RTT) values. *cp_benchmark* spawns $n$ agent threads that use the services provided by the ping client to initiate a ping transaction towards the responder (here, Quintus in Fig. 5). After these sessions were established, $m$ CASP messages were sent towards the destination with a randomized interval (3-15$sec$) between

two subsequent messages. RTTs were measured starting from sending the request packet to Responder until the completed reception of a response. Each agent stores its RTT values and passes them back to the main thread if all messages are sent. After all agents complete their task, the average RTT and the variance of RTTs of this run are calculated and stored. Fig. 6 depicts 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 $n \in \{1, 50, 100, \ldots, 950, 1000\}$.

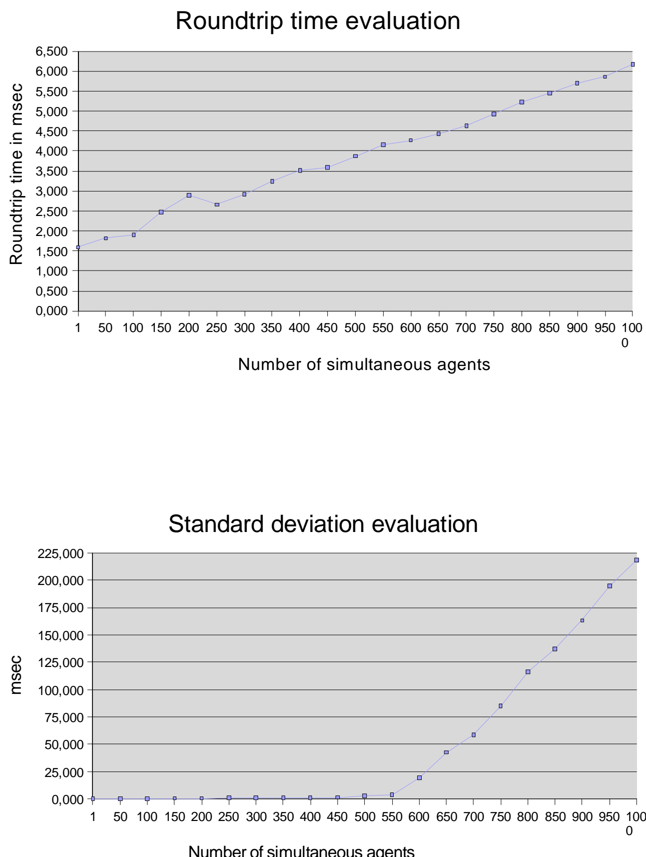



Fig. 6. Measured round trip times

The variances of RTTs can be interpreted as a consequence of multi-agent scheduling in the initiator side. Note that less than 550 agents only introduced very small values of variances; the more simultaneous agents were applied, the larger variance was observed. Also, when 200 agents was used, the RTT appeared surged. This can be caused by the overhead to manipulate open file-handlers after reaching a certain threshold. A possible way of improving this will be a more efficient way of the agent-client interface.

Note, in a multi-thread implementation, context switching, for example when the time slice of a thread (a worker thread or even an agent thread) is used up or a thread exits, Thus, the cost of context switches contributes to the results obtained above. Benchmarking this cost using the Linux Scheduler Benchmark [2] yields the results shown in Table II for the initiator of the testbed:

TABLE II
LATENCY OF CONTEXT SWITCHES

| | |
|---|---|
| Minimum context switching latency: | 98.7 $\mu s$ |
| Median context switching latency: | 101.0 $\mu s$ |
| Average context switching latency: | 119.5 $\mu s$ |
| Maximum context switching latency: | 1017.4 $\mu s$ |

These results help to get a rough understanding of the measured RTT values: much amount of the RTTs and their variance may be due to the context switches.

## C. Memory Consumption

Due to the simple but efficient way of state management, memory consumption of states is kept rather low (an analysis is given in Table III): a session only needs 128∼256 bytes plus 32∼128 bytes plus a thread memory page (4k bytes by default). This has also been proven by profiling of the runtime system. In other words, theoretically 10,000 signaling sessions running in a node simultaneously would consume about $42MB$. In case of a much more significant amount of signaling sessions, aggregation of the signaling sessions (e.g. similar to [7], [31], [45]), or based on destination prefixes may be necessary.

TABLE III
MEMORY CONSUMPTION FOR SESSION STATES

| | |
|---|---|
| **Session repository** | 128 to 256 bytes (without client data) |
| | *depending on the bus architecture of the host* |
| **Neighboring address** | 32 bytes for IPv4 |
| | 128 bytes for IPv6 |
| **Thread overhead** | 1 memory page per thread |
| | *(in most systems) default size is 4k bytes* |

## D. Profiling

Evaluating program performance in Linux is a difficult task using the legacy profiling tools like gprof, especially when using library routines and dynamically spawned plug-ins. If there is an obvious performance problem, one has to estimate the location of the bottleneck or resort to handcrafted measurement routines. Recently, Valgrind [5], a framework for memory debugging and program execution analysis under GPL, has proved useful for this purpose. In order to get an overview of the performance characteristics of our CASP implementation, we examined the costs attributed to functions or even libraries, by using one of Valgrind tools "*calltree*". Since multi-thread programs are generally very hard to analyze, the initial step in the evaluation process was to profile a single-thread version of the CASP implementation. Calltree tracks the amount of time spent (in terms of the cycles of the synthetic X86 CPU) for calling each routine and subroutines as well as shared libraries. Table IV gives the details of the profiling results.

It can be inferred from the table that generating a session ID is almost the most costly operation for Initiator, while the behaviors of Forwarder and Responder look very similar, due to the cryptographic computation of a random number. In a realistic implementation, however, it is possible to rely on

TABLE IV
RUNTIME PROFILES OF THE CASP IMPLEMENTATION IN DIFFERENT ROLES

| Code section | Init-iator | Forw-arder | Resp-onder |
|---|---|---|---|
| 1. Processing incoming messages | 76% | 60% | 55% |
| 1.1 Receiving messages (*glibc*) | <5% | 9% | 9% |
| 1.2 Session ID generation (*OpenSSL*) | 58% | - | - |
| 1.3 Session state creation/retrieval | 15% | 30% | 34% |
| 1.3.1 Session state retrieval | <5% | 15% | 15% |
| 1.3.2 Session state creation | 11% | 11% | 15% |
| 2. Sending messages (*glibc*) | 9% | <5% | <5% |
| 3. State refresh | 10% | 32% | 30% |
| 3.1 Removal of expired sessions | - | 14% | 15% |
| 4. Polling the event loop | <5% | 5% | 5% |
| Total cost (in million CPU cycles) | 1,826 | 626 | 576 |

hardware random number generator with computing devices available today, e.g. Intel RNG [23].

Second, we observed that soft state refresh requires a large amount of the rest operations (except session ID generation), about 30%-32% of the overall costs in Forwarder and Responder in our test.

Third, upon receipt of an upper request or refresh message, a node needs to search session repository, update the corresponding entries and forward the refresh on if necessary. Session retrieval is trivial (<5%) in Initiator, and more expensive but still of reasonably low cost (15%) in the other two types of nodes. The overall session creation or retrieval operations cost about 30%-34% of the overall costs in Forwarder and Responder in our test.

Finally, it has been found that sending and receiving messages require reasonably low costs, about 9% and less than 5% for both Forwarder and Responder, respectively. The cost for polling the event loop was shown to be relatively low (up to 5% for all nodes).

### *E. Scout Performance*

The most costly part in scout daemon was found to be the cookie generation, which was measured as about 25$\mu$s on average within the whole processing time of a scout message. This yields to the same reason as session ID generation in the messaging layer: random number generation is the most computation-intensive part of the whole process. We expect some improvements could be made, for example by computing such cookies in advance in a buffer or by using hardware-based random number generators.

While the average RTT for two-way scout messages was higher when a scout request was sent every 10 seconds for 2 hours was higher versus sending a scout request every 10 minutes for 24 hours ((353$\mu$s v.s. 241$\mu$s), the standard deviation of the former was much higher than that of the latter (30$\mu$s v.s. 1$\mu$s). Clearly, the processing packets with an IP router alert option imposes performance penalty to the system.

### *F. Performance under Different Congestion Situations*

When TCP is used for signaling transport support in the CASP implementation, it is important to assess its performance under different congestion situations. Signaling tasks were initiated by agents and operated from an Initiator (Secundus or Octacus) towards a Responder (Quintus), traversing another CASP-aware node (Teravus) and a CASP-unaware node (Quartus). We used MGEN 4.2b4 [3] to generate different UDP traffic loads (ranging from 0 to maximal bandwidth of 100Mbps) from Quartus to Tertius, in order to emulate different network congestion situations in a segment of the signaling path. Because these tests were mainly designed to test CASP performance under different network congestion situations, a lower number of simultaneous agents was used (100) here to minimize the overloading effect in the initiator, as opposed to Section IV.B. Congestion situations were emulated by using a burst pattern of MGEN, namely periodically sending UDP packets (currently each with a fixed size of 1024 bytes).

We plotted the RTT values measured in cp_benchmarking signaling; each was a steady-state value obtained in a given background traffic load in the segment (between Tertius and Quartus) of the signaling path. Fig. 7 shows that in light-loaded scenarios, RTTs were quite small (below 3ms for the situation where background traffic occupied up to 50% of bandwidth); when the network becomes overloaded, the RTT becomes larger than light-loaded cases. Nonetheless, it still yields a limited value (up to 10ms) even when background traffic approaches 100% of the link bandwidth. This can be attributed to the method of emulating congestion situations by bursty UDP traffic. The interaction between CASP signaling and other possible network situations will be investigated in the next phase.

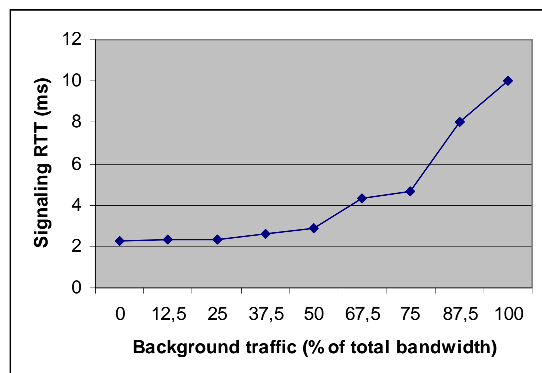



Fig. 7. Signaling RTTs under different network congestion situations

## V. RELATED WORK

Over the last decade, various issues for signaling in the Internet, especially for QoS resource reservation, have been widely investigated, ranging from soft state modeling [22], [33], scalability [7], [9], [31], [45], to complexity [12], [16], [26], [32] and applicability [6], [25], [40], using either a server-based or a router-based approach to simplify or extend RSVP (even under other protocol names). For example, the current index of Internet drafts lists 28 documents with the word “RSVP” in their titles, while there are 29 RFCs with

"RSVP" in their titles. Server-based approach relies on centralized entities (known as "bandwidth brokers") to perform admission control, while the router-based approach installs packet filters either on a per-flow or aggregated basis in a hop-by-hop way. Although there has been much focus on modularity for specific QoS or multicast models (e.g. [28]), little focus was placed on the support for generic signaling. Furthermore, the dominant way of using "router-alert option" and coupling discovery with discovery cause a number of security and complexity issues [26], [43].

Recently, several authors have addressed modular design, using either an RSVP-based or a CASP-based approach. In RSVP-based approaches, RSVP has been extended with an extra reliable mechanism and general signaling support. This approach removes the QoS- and multicast-specific processing burden from RSVP, and has the advantage of better compatibility with existing protocol and implementations. However, securing, congestion control and fragmentation of signaling messages may be complex. No simple solution is available and RSVP still has to deal with these issues, since RSVP encapsulates its messages using raw IP or UDP, and couples PATH signaling with next-hop discovery. Variations of the RSVP-based approach have been described in [10], [38]. The latter proposal suggests a decomposite system where a signaling message is just sent to next CASP hop (discovered by some next-hop discovery mechanism) using an existing transport protocol which provides capabilities such as fragmentation, congestion control, and easier security when desired. However, both proposals leave the actual mechanism undefined. Recently, the IETF Next Steps in Signaling Working Group [4], which is responsible for standardizing a general IP signaling protocol where QoS signaling is the first use case, has decided to follow the CASP-based approach and reuse RSVP concepts whenever possible [35]. Nonetheless, the tradeoff between performance, complexity and modularity is still an issue in both approaches. Fault recovery, especially in dealing with re-routing [29] remains one major concern in the layered architecture.

Another related issue comes out of IP mobility support. Node mobility is expected to become common in the near future, so that signaling in changed network paths and tunneling after a handover must be supported, which are typical characteristics of IP mobility. However, it is difficult for RSVP to support these scenarios due to its design [26], in particular with the use of flow identifier for state indexing, local repair and tunneling support [27]. Recently there have been a few proposals trying to address this issue but most of them are far from being mature. Mobility support in CASP-based approaches has been considered in [18], although further study is still needed with respect to detailed interactions within the two-layer architecture and mobility routing interfaces.

Besides these studies, there have been some work on performance evaluation of RSVP implementation. For example, Chiueh *et al.* [13] reported an empirical study of RSVP, which measured performance of a Cisco RSVP-capable router, including RSVP control packet latencies (under loaded and unloaded cases) and throughput impact delivered for QoS objectives. Karsten *et al.* [24] implemented a user-level RSVP protocol engine (which allows multi-threading processing), evaluated its performance to find out the upper limits of the reservation requests and profiled the system for different parts of protocol operations. In comparison, we also use a multi-threading model to improve efficiency and performed system profiling, but our study more focuses on proving the fundamental concepts and running properties of CASP, a different design principle for general-purpose signaling rather than QoS signaling in RSVP. To our knowledge, the work presented in this paper is the first empirical study of the CASP protocol.

## VI. Conclusions and Future Work

This paper presented the design, implementation and performance analysis of CASP, a generic signaling framework for the Internet. In contrast to traditional methods, CASP provides a modular architecture to support any client applications requesting signaling services, and reduces complexity by relying on existing reliable transport protocol for signaling transport. Due to the reuse of existing transport connections, on average, the performance of signaling transport is low. The modularity design of the CASP implementation provides a flexible way for session state management, message processing, and any type of next-hop discovery mechanism. The implementation performed efficiently via plug-in architecture and a master-worker multi-threading paradigm, also when serving for a number of agent requests. The use of obstack to handle the messages reduces the need for frequent defragmentation of memory pieces by the operating system. Balanced binary searching tree allows a logarithmic session searching time and a session. The cost of processing signaling messages depends on the volume of requests for signaling services and the refresh frequency. Our CASP implementation economizes on most of these components. Firstly, when we allow routers to use CASP for signaling with a wide range of client applications and discovery mechanisms, they can be registered on demand and seamlessly configured without disturbing the entire system. This reduces the necessity of applying different protocols and updating the protocol stack with changed needs and conditions. Secondly, by efficient soft state management and session maintenance, CASP improves the signaling responsiveness. Thirdly, CASP typically needs less frequent refreshes than RSVP does, for the following reason: RSVP signaling messages are transmitted unreliably and thus must be repeated at about three times the state expiration timer, while CASP refresh messages can be transferred reliably hop-by-hop when TCP or SCTP is used for signaling transport. The mechanisms described above reduce the complexity of CASP and the number of CASP messages. Not only does this reduce the burden on the routers to process these messages, it also reduces the link bandwidth cost to transmit these messages.

Our current design is based on a Linux platform and certain design choices have been optimized for it. However, we expect that the need for new designs will arise for a realistic implementation as other ways of composing client protocols and the overall system materialize. This brings up the question of how to add new signaling clients which can be deployed.

One possibility would be to allow each client type to run an individual process instead of thread. It is uncertain, however, how much this will influence performance. Furthermore, a number of open issues were encountered and/or considered during the design and implementation of the CASP protocol, and the evaluation of the protocol was just preliminary. In particular, performance and scalability need further investigation with respect to more sophisticated network topology (e.g. that with more nodes, a large number of signaling sessions in the middle and a small number of agents in the end, or with different signaling loads). Some improvements in CASP message processing are currently underway to enhance more flexible signaling transport support and certain issues of security. Finally, studies are being carried out on other issues connected with CASP, such as fault handling, route change and mobility support [18], incorporation with the usage of RSVP-like signaling while discovery (as in the NTLP [35]), as well as the QoS and firewall signaling client protocols under standardization [39], [42], and performance comparison between CASP and RSVP.

## Acknowledgement

We would like to thank Henning Schulzrinne and Hannes Tschofenig for fruitful discussions with them. We would also like to acknowledge Cornelia Kappler, Rene Soltwish and in particular, anonymous reviewers for their comments, and Fabian Meyer for his contribution in the GoCASP project.

## References

[1] *GoCASP*. URL: http://user.informatik.uni-goettingen.de/∼casp/.
[2] *Linux Scheduler Benchmark*. URL: http://www.atnf.csiro.au/people/rgooch/benchmarks/linux-scheduler.html.
[3] *Multi-Generator (MGEN)*. URL: http://mgen.pf.itd.nrl.navy.mil/.
[4] *NSIS*. URL: http://www.ietf.org/html.charters/nsis-charter.html.
[5] *Valgrind*. URL: http://valgrind.kde.org/.
[6] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow. RSVP-TE: Extensions to RSVP for LSP Tunnels. RFC 3209, December 2001.
[7] F. Baker, C. Iturralde, F. Le Faucheur, and B. Davie. Aggregation of RSVP for IPv4 and IPv6 Reservations. RFC 3175, September 2001.
[8] L. Berger, D. Gan, G. Swallow, P. Pan, F. Tommasi, and S. Molendini. RSVP Refresh Overhead Reduction Extensions. RFC 2961, April 2001.
[9] Y. Bernet, P. Ford, R. Yavatkar, F. Baker, L. Zhang, M. Speer, R. Braden, and B. S. Davie. A Framework for Integrated Services Operation over Diffserv Networks. *RFC 2998*, November 2000.
[10] B. Braden and B. Lindell. A Two-Level Architecture for Internet Signaling. Internet draft (draft-braden-2level-signaling-01), work in progress, October 2002.
[11] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. RFC 2205, September 1997.
[12] P. Chandra, A. Fisher, and P. Steenkiste. A Signaling Protocol for Structured Resource Allocation. In *INFOCOM'99*, 1999.
[13] T. Chiueh, A. Neogi, and P. Stirpe. Performance Analysis of an RSVP-Capable Router. In *RTAS'98*, 1998.
[14] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols*, Philadelphia, PA, September 1990. ACM.
[15] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246, January 1999.
[16] G. Feher, K. Nemeth, M. Maliosz, I. Cselenyi, J. Bergkvist, D. Ahlard, and T. Engborg. Boomerang: A Simple Protocol for Resource Reservation in IP Networks. In *RTAS'99*, 1999.
[17] X. Fu, D. Hogrefe, and S. Willert. Implementation and Evaluation of the Cross-Application Signaling Protocol (CASP). Technical Report No. IFI-TB-2004-001, Institute for Informatics, University of Goettingen, Goettingen, Germany, April 2004.
[18] X. Fu, H. Schulzrinne, and H. Tschofenig. Mobility Support for Next-Generation Internet Signaling Protocols. In *VTC'03-Fall*, 2003.
[19] J.-P Goux, S. Kulkarni, J. T. Linderoth, and M. E. Yoder. Master-Worker: An Enabling Framework for Applications on the Computational Grid. *Cluster Computing*, (4):63-70, 2001.
[20] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley. XCAT 2.0: Design and Implementation of Component based Web Services. Technical report, Department of Computer Science, Indiana University, June 2002.
[21] E. Guttman, C. E. Perkins, J. Veizades, and M. Day. Service Location Protocol, Version 2. RFC 2608, June 1999.
[22] P. Ji, Z. Ge, J. Kurose, and D. Towsley. A Comparison of Hard-state and Soft-state Signaling Protocols. In *SIGCOMM'03*, 2003.
[23] B. Jun and P. Kocher. *The Intel Random Number Generator* (white paper), 1999.
[24] M. Karsten, J. Schmitt, and R. Steinmetz. Implementation and Evaluation of the KOM RSVP Engine. In *INFOCOM'01*, 2001.
[25] A. Mankin, F. Baker, B. Braden, S. Bradner, M. O'Dell, A. Romanow, A. Weinrib, and L. Zhang. Resource ReSerVation Protocol (RSVP) – Version 1 Applicability Statement Some Guidelines on Deployment. RFC 2208, Internet Engineering Task Force, September 1997.
[26] J. Manner, X. Fu, and P. Pan. Analysis of Existing Quality of Service Signaling Protocols. Internet draft (draft-ietf-nsis-signalling-analysis-04), work in progress, May 2004.
[27] J. Manner, A. Lopez, A. Mihailovic, H. Velayos, E. Hepworth, and Y. Khouaja. Evaluation of Mobility and QoS Interaction. *Computer Networks*, 38(2):137-163, February 2002.
[28] D. Mitzel, D. Estrin, S. Shenker, and L. Zhang. An Architectural Comparison of ST-II and RSVP. In *INFOCOM'94*, 1994.
[29] S. Nelakuditi, S. Lee, Y. Yu, and Z.-L. Zhang. Failure Insensitive Routing for Ensuring Service Availability. In *IWQoS'03*, 2003.
[30] Object Management Group. *CORBA/IIOP Specifications*.
[31] P. Pan, E. Hahne, and H. Schulzrinne. BGRP: A Tree-Based Aggregation Protocol for Inter-domain Reservations. *Journal of Communications and Networks*, 2(2):157–167, June 2000.
[32] P. Pan and H. Schulzrinne. YESSIR: A Simple Reservation Mechanism for the Internet. In *NOSSDAV'98*, Cambridge, UK, July 1998.
[33] S. Raman and S. McCanne. A Model, Analysis, and Protocol Framework for Soft State-based Communication. In *SIGCOMM'99*, 1999.
[34] H. Schulzrinne, X. Fu, C. Pampu, and C. Kappler. Design of CASP – a Technology Independent Lightweight Signaling Protocol. In *IPS'03*, Salzburg, Austria, February 2003.
[35] H. Schuzrinne and R. Hancock. GIMPS: General Internet Messaging Protocol for Signaling. Internet draft (draft-ietf-nsis-ntlp-03), work in progress, July 2004.
[36] H. Schuzrinne, H. Tschofenig, X. Fu, and A. McDonald. CASP – Cross-Application Signaling Protocol. Internet draft (draft-schulzrinne-nsis-casp-01), work in progress, March 2003.
[37] G. Shao. *Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources*. Ph.D. thesis, University of California, San Diego, 2001.
[38] M. Shore. The NSIS Transport Layer Protocol (NTLP). Internet draft (draft-shore-ntlp-00), work in progress, May 2003.
[39] M. Stiemerling, H. Tschofenig, M. Martin, and C. Aoun. NAT/Firewall NSIS Signaling Layer Protocol (NSLP). Internet draft (draft-ietf-nsis-nslp-natfw-03), work in progress, July 2004.
[40] A. Terzis, J. Krawczyk, J. Wroclawski, and L. Zhang. RSVP Operation Over IP Tunnels. RFC 2746, January 2000.
[41] H. Tschofenig, H. Schulzrinne, R. Hancock, A. McDonald, and X. Fu. Security Implications of the Session Identifier. Internet draft (draft-tschofenig-nsis-sid-00), work in progress, June 2003.
[42] S. Van den Bosch, G. Karagiannis, and A. McDonald. NSLP for Quality-of-Service signaling. Internet draft (draft-ietf-nsis-qos-nslp-04), work in progress, July 2004.
[43] T.-L. Wu, S. F. Wu, Z. Fu, H. Huang, and F.-M. Gong. Securing QoS: Threats to RSVP Messages and their Countermeasures. In *IWQoS'99*, 1999.
[44] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, 7(5):8–18, September 1993.
[45] Z.-L. Zhang, Z. Duan, and Y. H. Hou. Decoupling QoS Control from Core Routers: A Novel Bandwidth Broker Architecture for Scalable Support of Guaranteed Services. In *SIGCOMM'00*, 2000.