

Overhead and Performance Study of the General Internet Signaling Transport (GIST) Protocol

Xiaoming Fu, *Member, IEEE*, Henning Schulzrinne, *Fellow, IEEE*,
Hannes Tschofenig, Christian Dickmann, and Dieter Hogrefe, *Member, IEEE*

Abstract—The General Internet Signaling Transport (GIST) protocol is currently being developed as the base protocol component in the IETF Next Steps In Signaling (NSIS) protocol stack to support a variety of signaling applications. We present our study on the protocol overhead and performance aspects of GIST. We quantify network-layer protocol overhead and observe the effects of enhanced modularity and security in GIST. We developed a first open source GIST implementation at the University of Göttingen, and study its performance in a Linux testbed. A GIST node serving 45,000 signaling sessions is found to consume average only 1.1 ms for processing a signaling message and 2.4 KB of memory for managing a session. Individual routines in the GIST code are instrumented to obtain a detailed profile of their contributions to the overall system processing. Important factors in determining performance, such as the number of sessions, state management, refresh frequency, timer management and signaling message size are further discussed. We investigate several mechanisms to improve GIST performance so that it is comparable to an RSVP implementation.

I. INTRODUCTION

The Internet was designed to have simple packet forwarding nodes and complex end systems. However, over the years these design principles have been challenged by new application requirements and evolving demands on the infrastructure [1]–[4].

There is an ever increasing demand to provide configuration and maintenance of flow-specific control state in the network (i.e., signaling services) along the data path in IP-based networks. Examples include resource reservation for Quality of Service (QoS) provisioning and the configuration of various middleboxes such as stateful packet firewalls and Network Address Translators (NATs) [5]. Although the *Resource ReSerVation Protocol (RSVP)* [6], [7] has been developed, usage of RSVP has more focused on QoS reservation models – initially IntServ [8], later DiffServ [9] – and their performance [10], [11] – rather than the underlying signaling services. Apart from this, shortcomings in the RSVP design, e.g., lack of a solid security framework and mobility support have also played a role in contributing to the limited market adoption. Approaches like RSVP refresh overhead reduction

extensions [12], BGRP [13], Yessir [14], Boomerang [15], Beagle [16], MRSVP [17], Insignia [18] or RSVP Mobility Support [19] investigate QoS signaling with the goal to reduce overhead, improve performance, or extend the signaling scheme to support mobility. These extensions are based on the idea of discovering QoS-aware nodes along the data path by using end-to-end addressed messages (mostly equipped with a Router Alert option) that deliver QoS parameters and rely on a flow identifier to identify a signaling session state. In addition, protocol complexity has been a concern, especially due to the support for multicast flows [20]. These design principles have been a source of limited flexibility, security and mobility. More importantly, although these approaches individually may meet the needs of certain signaling purposes, they lack an extensible framework which allows easy extensions for future signaling applications. Thus, in 2001 the IETF formed a new working group – Next Steps in Signaling (NSIS) [21] – to investigate the architecture and protocols for generic and application-specific signaling. One pioneering work has been presented by Braden and Lindell [22], who attempted to split RSVP into a two-layer architecture allowing any type of signaling application rather than being QoS centric.

Due to the shortcomings of RSVP and its current extensions, we have presented an alternative extensible signaling approach [23], [24] – *Cross-Application Signaling Protocol*, or *CASP* – for ensuring modularity, flexibility and security without changing the conventional path-coupled signaling paradigm. There are three key ideas that underpin our proposed approach: decoupling message transport from next signaling hop discovery, reuse of existing transport and security protocols, and the introduction of a location-independent session identifier. This approach enables us to effectively support generic IP signaling that can be used for various signaling scenarios, with enhanced protocol flexibility. The NSIS working group reused many ideas from CASP and is standardizing a *General Internet Signaling Transport (GIST)*¹ [25] as the base protocol component of NSIS protocol stack to support a variety of signaling applications.

In this paper, we study the protocol overhead and performance aspects of GIST and compare them with RSVP. While some results are specific to our implementation, we believe the tests and results should approximate some common behavior in other GIST implementations. The results confirm that GIST is meeting its major design goals. Our experience has been that

X. Fu, C. Dickmann and D. Hogrefe are with the Institute of Computer Science, University of Göttingen, Germany, Email: {fu,cdickman,hogrefe}@cs.uni-goettingen.de.

H. Schulzrinne is with the Department of Computer Science, Columbia University, New York, USA, Email: hgs@cs.columbia.edu.

H. Tschofenig is with Nokia Siemens Networks and University of Göttingen, Germany, Email: hannes.tschofenig@nsn.com.

Manuscript received on December 6, 2006, revised on September 30, 2007, and accepted on December 21, 2007. This is a revised version of [65].

¹The protocol described here was known as GIMPS (General Internet Messaging Protocol for Signaling) until its final name was chosen in August 2005.

implementation details are very important to achieve all of the benefits of GIST.

The organization of the paper is as follows. Section II provides a short introduction to GIST, Section III discusses the results of a study which indicate that the additional overhead in GIST are largely due to modularity and security. Furthermore, it delineates the limitations of QoS-centric approaches in providing generic signaling services. We then elaborate our GIST implementation details and performance study in Section IV.

II. AN INTRODUCTION TO GIST

A. NSIS: A Two-Layer Signaling Framework

In order to meet the requirements for an extensible, generic signaling protocol, the design of the NSIS protocol suite separates the transport functionalities, such as reliability, fragmentation, congestion control and integrity, for signaling message transport from signaling applications. Thus, following [22], [23], signaling functions in NSIS are split into two protocol layers [26]:

- An NSIS Transport Layer Protocol or NTLP, primarily composed of a specialized *messaging* layer, denoted as GIST [25], which is used to transport the *signaling application* layer messages. The GIST layer is running over standard transport and security protocols. Examples of such protocols are UDP, TCP, SCTP [27] and DCCP [28], with or without IP security (IPsec) or Transport Layer Security (TLS) mechanisms; in the current version, usage of UDP, TCP, TLS over TCP [25] and SCTP [29] are specified.
- NSIS Signaling Layer Protocols or NSLPs, each run signaling application-specific functionality. Examples of NSLPs include the QoS NSLP for resource reservation signaling [30], the NAT/Firewall NSLP [31] for middlebox configuration, and the NSLP for Metering Configuration Signaling [32].

The different layers are depicted in Fig. 1.

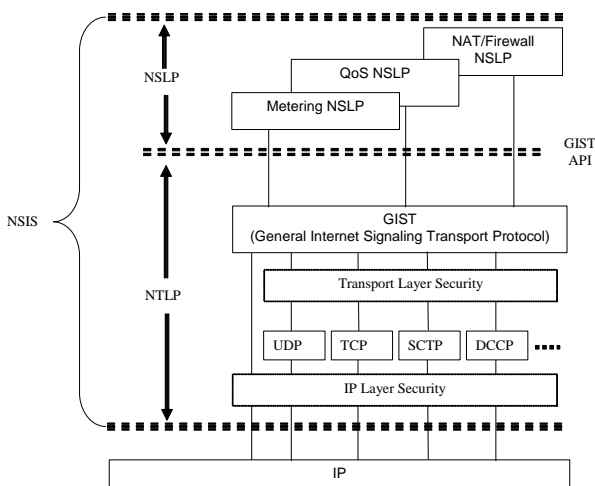


Fig. 1. NSIS: a Two-layer Signaling Framework

B. GIST Overview

The GIST protocol forms the fundamental building block of the NSIS protocol suite. The main task of GIST is to deliver signaling messages for various NSLPs between neighboring GIST nodes that support the same NSLP. The NSLP itself is responsible for pushing the signaling message from the NSIS Initiator (NI) towards the NSIS Responder (NR), typically the flow source and destination, respectively, as GIST just provides means to transport messages from one node to the next on the path. The NI and NR can, however, also be represented by proxies, e.g., to support end systems that do not themselves have NSIS capabilities.

Instead of building a new transport protocol, GIST reuses existing transport and security protocols to provide a universal message transport service. Like RSVP, GIST is a soft-state protocol. It creates and maintains two types of states related to signaling transport: a per-session message routing state (MRS) for managing the processing of outgoing messages, and a message association (MA) state for managing the per-peer state associated with connection mode messaging to a particular peer, including signaling destination address, protocol and port numbers, internal protocol configuration and state information. In addition to its neighboring GIST peer information, GIST also maintains certain message routing information, such as flow identifier (flow ID), NSLP type and session identifier (session ID), to uniquely identify the signaling application layer session for a flow.

GIST has two modes of operation; a *datagram mode* (D-mode), which uses an unreliable unsecured datagram transport mechanism, with UDP as the initial choice, and a *connection mode* (C-mode), which uses any stream or message-oriented transport protocol (currently TCP as the initial choice) and may use IPsec security or TLS. It is possible to mix these two modes along a chain of nodes, without coordination or manual configuration. This allows, for example, the use of D-mode at the edges of the network and C-mode in the core of the network.

Let us have a look at a standard GIST operation using an example (cf. Fig. 2), where A is QoS NSLP while B is another type of NSLP. Assume a QoS NSLP RESERVE message is generated by GIST at the NI, the flow sender. The GIST module first constructs a GIST-query message, namely a UDP datagram addressed to the flow destination and includes an IP Router Alert Option [33], similar to RSVP. The next downstream NSIS peer supporting the QoS NSLP, R2 in this case, recognizes this message and replies to it with a GIST-Response message. Upon the receipt of this response, the NI creates a message association (MA) with R2. After that, all subsequent GIST-Confirm or GIST-Data messages, i.e., GIST messages with NSLP payload, between these two peers can be sent over this MA. Upon receipt of such GIST messages in R2, NSLP payload and the flow ID are passed to its QoS NSLP processing. Note it is the responsibility of the NSLP layer to determine the action upon receipt of a GIST message. If the QoS NSLP in this node determines that it has enough resources as requested by QoS NSLP RESERVE message, it will make a tentative reservation for the session. The

QoS NSLP RESERVE message will be delivered to the next QoS NSLP node along the path, according to the procedure described above. When the NR QoS NSLP eventually receives the RESERVE message, it responds along the reverse path towards the NI with a RESPONSE message to finally confirm the establishment of the reservation. In this example, the QoS NSLP payload (e.g., for signaling IntServ would be primarily Sender.TSpec and RSpec) is delivered, examined and possibly modified in intermediate nodes, across a chain of QoS NSLP aware nodes.

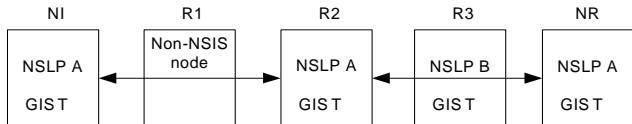


Fig. 2. An example of GIST operation

A GIST message consists of a common header and a sequence of type-length-value (TLV) objects. The common header indicates the message type such as Query/Response/etc., as well as the NSLP identifier (NSLP ID) and hop counter to avoid message loops. In addition, GIST can use query- and response-cookies for protection against spoofing and denial of service attacks.

GIST-Query messages are retransmitted with exponential backoff if a corresponding GIST-Response is not received on time. Whenever possible, re-use of existing reliable transport and security protocols is recommended via the C-mode in GIST. This is necessary with larger data objects, when a fast state setup in the face of packet loss is desirable, or when channel security is required. A querying node (Q-node) can choose to refresh the message routing state by resending a GIST-Query. Local policy can determine whether it is necessary to maintain an MA. For example, a node may choose to keep the MA open if there are sessions still in place, which might generate messages that would use the MA. If no MA exists between a Q-node and the responding node (R-node), and the Q-node desires to run over C-mode, it will send a Query with a stack proposal (Q-stackp) and stack configuration data (Q-stackcd) to negotiate the desired C-mode transport protocol, e.g., TCP or TCP+TLS, with the R-Node during the discovery phase (see Fig. 3(a)); TCP three-way handshake is required to setup the MA.

A detailed GIST protocol description can be found in [25] and its corresponding state machine operations are described in [34].

C. GIST Security

Security mechanisms for GIST try to provide the following properties:

- 1) Authentication of the two neighboring protocol peers;
- 2) Security association establishment to provide integrity, confidentiality and replay protection for signaling messages exchanged between these entities;

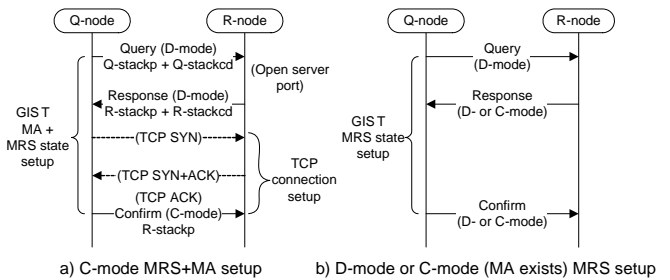


Fig. 3. GIST session setup

3) Denial of service protection;

4) Security protection for the discovery mechanism.

It is difficult to design a new security protocol to address all these issues. Existing security protocols, such as TLS or IKEv2/IPsec already provide a number of these features, such as properties 1), 2) and 3), but at the cost of considerable setup latency. The establishment of a secure channel between signaling peers to protect all signaling messages, which may belong to any signaling session, is recommended. An existing security channel saves the per-session security association setup cost in C-mode.

Authorization at the GIST layer aims to ensure that a GIST R-node only establishes communications with a legitimate GIST Initiator. It is even more difficult to ensure that the GIST Initiator sends signaling messages to the “right” GIST peer, i.e., one which supports a specific NSLP; this requires authorization information to be provided along with the authentication and key exchange process, e.g., as part of the authorization certificate. These aspects are described in [35].

Relaxing assumptions regarding the desired protection against man-in-the-middle adversaries might often be required and desired. Furthermore, in most cases it is difficult for GIST to make an authorization decision without consulting the NSLP layer.

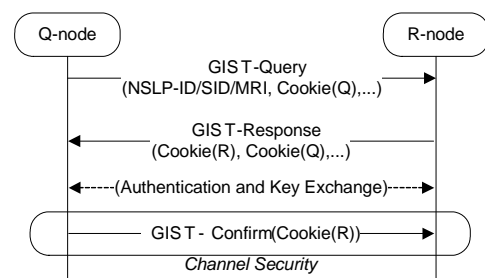


Fig. 4. Protection of the GIST discovery procedure

In order to deal with adversaries that redirect signaling messages, a cookie mechanism has been integrated into the discovery exchange. This mechanism (see Fig. 4) can be illustrated as follows. The cookies provided by the querying and responding node (Cookie(Q) and Cookie(R)), e.g., 256-bit cryptographically random nonces, are used to prevent DoS attacks, similar to those used by other protocols, e.g., SCTP or IKEv2. Cookie(Q) is included in the GIST-Response message

to prevent off-path adversaries from flooding the querying node with bogus responses. The initiator uses this cookie to match a request with a pending response. Once a security association has been established, Cookie(R) is transmitted from the querying node to the responding node. This allows the responder to verify that it has actually participated in the discovery exchange, binding the discovery procedure to the subsequent exchange. More details of authentication and key exchange as well as possible cookie construction in Fig. 4 are provided in the GIST specification.

III. PROTOCOL OVERHEAD

Every signaling protocol imposes some overhead in the form of number and size of control messages, which is indicative of the total bandwidth consumed by the signaling protocol and must be processed in signaling-aware nodes. In this section we discuss the sources and quantities of protocol overhead in GIST as opposed to RSVP ². For convenience we consider the primary signaling messages used for state setup and maintenance: GIST-Query, Response, Confirm and GIST-Data in comparison with RSVP-Path and RSVP-Resv; RSVP/GIST Error, MAHello and RSVP PathTear/ResvTear messages are omitted here for simplicity.

The detailed sources of overhead, including message and memory overhead, in each of the layers of a GIST protocol structure (based on the latest draft version of [25]) are given in the Appendix, in comparison to RSVP. Table I summarizes the overall message overhead for common message types.

With this information, we are able to analyze the overhead of the two signaling protocols, GIST and RSVP. On the one hand, layering in GIST makes it possible to provide the general functionalities required for signaling transport, namely,

- Error control: GIST makes the “channel” more reliable (by reusing reliable transport protocols);
- Flow control: GIST avoids flooding slower peer by signaling message flow control,
- Fragmentation: Dividing large data chunks into smaller pieces, and subsequent reassembly, e.g., TCP MSS fragmentation/reassembly for large sizes of NSLP payload,
- Multiplexing: Allow multiple sessions to share a single message association between adjacent peers,
- Connection setup: Handshaking with peer, e.g., by TCP three-way handshake.

More importantly, GIST provides richer security support, which makes it easier to support mobility and allows high modularity to allow any signaling applications with a comparable requirement for state repository.

On the other hand, layering and more functionality support increase message and memory overhead. For example, if C-mode is desired, there are at least two possibilities for GIST session setup with minimal security support, i.e., only the cookie mechanism is used:

- 1) There is no TCP connection. This requires a GIST-Query with stack proposal/TCP-SYN/TCP-SYNACK/Response(C)/Confirm(C) process, which

in turn imposes $180+44+44+220+188=676$ bytes message overhead, in addition to the following memory overhead:

- a new per-session GIST message routing state, which comprises a MRI (32 bytes), a NSLP ID (8 bytes), and a Session ID (20 bytes), or 60 bytes in total;
 - a new per-connection TCP control block (TCB) state [60], [61], which ranges from hundreds to thousands of bytes depending on the underlying TCP implementation;
 - and a new GIST message association state, which is implementation-specific and only a few hundred of bytes in our implementation case.
- 2) A message association already exists. This requires a GIST-Query(no stack proposal)/Response(C)/Confirm(C) process, which imposes $148+220+188=556$ bytes message overhead, in addition to the memory overhead of a new GIST message routing state (60 bytes).

Thus, for signaling session setup, GIST C-mode requires 4 (or 3, when MA already exists; same applies below respectively) messages, totally 676 (or 556) bytes overhead, for the scenario where no TCP connection exists (or there exists an MA).

With GIST D-mode, no connection setup is required, but three-way handshake in GIST layer is still needed, imposing $148+180+140=468$ bytes message overhead, in addition to the creation of per-session GIST MRS (60 bytes memory overhead).

For convenience we assume the QoS NSLP payload is the same as RSVP, namely, Sender_TSpec and FlowSpec. Per [30], the minimal QoS NSLP header length for the basic messages to deliver these payloads is: QoS NSLP RESERVE: 16 bytes (QoS NSLP common header + RSN) and

QoS NSLP RESPONSE: 16 bytes (QoS NSLP common header + INFO_SPEC)

Overall, QoS NSLP layer adds at least 32 bytes more overhead in addition to the overhead incurred by GIST.

With RSVP, on the other hand, in order to carry the signaled data (Sender_TSpec and Guaranteed/Controlled-Load Service FlowSpec) of $12+48=60$ bytes (Guaranteed Service)/ $12+12=24$ bytes (Controlled-Load Service), every session setup requires a Path+Resv pair of $64+72=136$ bytes (IPv4)/184 bytes (IPv6) message overhead, in addition to creating a new PSB and a new RSB.

According to this analysis, a comparison of overhead for GIST (D-mode and C-mode, including with and without MA) + QoS NSLP and RSVP is given in Fig. 5. It can be noted that most of the protocol overhead of GIST results from the lower levels of the protocol stack during the session setup phase, while in steady state (session refresh case), all modes of GIST operation have no significant difference in terms of overhead compared with RSVP. For instance, using the GIST D-mode together with QoS NSLP which is closest to RSVP operation, it incurs overhead of 500 bytes for session setup case and 136 bytes for session refresh case (compared with

²Note there are some small changes in this paper compared to the numbers given in [65], to count for more fair and accurate comparison for QoS signaling using GIST/NSIS and RSVP

TABLE I
OVERHEAD BY PROTOCOL LAYER (IN BYTES)

Message type\Protocol layer	IP layer	Transport layer	GIST layer	Overall overhead
GIST-Query (no stack proposal)	24 (IPv4), 48 (IPv6)	8	116 (IPv4), 156 (IPv6)	148 (IPv4), 212 (IPv6)
GIST-Query (with stack proposal)	24 (IPv4), 48 (IPv6)	8	148 (IPv4), 188 (IPv6)	180 (IPv4), 244 (IPv6)
GIST-Response (D-mode)	20 (IPv4), 40 (IPv6)	8	152 (IPv4), 192 (IPv6)	180 (IPv4), 240 (IPv6)
GIST-Response (C-mode)	20 (IPv4), 40 (IPv6)	20	180 (IPv4), 184 (IPv6)	220 (IPv4), 244 (IPv6)
GIST-Confirm (D-mode)	20 (IPv4), 40 (IPv6)	8	112 (IPv4), 152 (IPv6)	140 (IPv4), 200 (IPv6)
GIST-Confirm (C-mode)	20 (IPv4), 40 (IPv6)	20	108 (IPv4), 148 (IPv6)	188 (IPv4), 208 (IPv6)
GIST-Data (D-mode)	20 (IPv4), 40 (IPv6)	8	76 (IPv4), 122 (IPv6)	104 (IPv4), 168 (IPv6)
GIST-Data (C-mode)	20 (IPv4), 40 (IPv6)	20	72 (IPv4), 116 (IPv6)	112 (IPv4), 176 (IPv6)
(TCP-SYN/SYN+ACK)	20 (IPv4), 40 (IPv6)	24	–	44 (IPv4), 64 (IPv6)
(TCP-ACK)	20 (IPv4), 40 (IPv6)	20	–	40 (IPv4), 60 (IPv6)
RSVP-Path	24 (IPv4), 48 (IPv6)	0	–	64 (IPv4), 112 (IPv6)
RSVP-Resv	20 (IPv4), 40 (IPv6)	0	–	72 (IPv4), 108 (IPv6)

136 bytes in RSVP for both cases). The most heavy-weight situation is when a GIST C-mode is desired but MA is not yet established, GIST/QoS NSLP incurs 708 bytes for session setup; when an MA already exists, the corresponding overhead for session setup is 588 bytes, about 17% higher than D-mode operation.

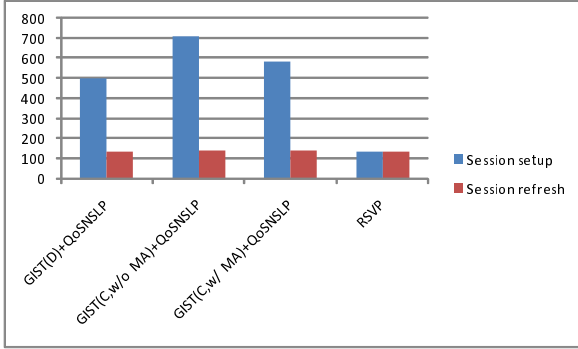


Fig. 5. Overhead comparison of GIST and RSVP

This comparison demonstrates that GIST’s rich functionality, modularity, security, and mobility support is also accompanied by certain costs. Indeed, similar to other general-purpose protocols, GIST does have its disadvantage of higher protocol overhead in terms of large messages, more message exchanges, additional parsing and processing. However, as we will confirm in Section IV-C, with some appropriate implementation considerations and optimizations, it is possible to achieve comparable performance to RSVP in terms of signaling performance of maximal number of supported sessions, CPU and memory consumption in steady state. Furthermore, it should also be noted that in many scenarios, signaling application payloads are rather large (which can easily exceed normal link MTU), e.g., certificates and active programming packets, where the transport capability of GIST becomes of greater use and the relative GIST protocol overhead becomes much less. In addition, concepts of staged timers [12], [36], state compression [37], and robust header compression [38] can also be considered which may further improve GIST performance.

IV. IMPLEMENTATION AND PERFORMANCE EVALUATION

In this section, we evaluate the quantitative performance of a GIST implementation through benchmarks, and show its performance is roughly comparable to KOM RSVP [11], and scales well with the number of signaling sessions.

A. Implementation Overview

We have implemented the GIST protocol in C++, using the Linux 2.6 kernel. Our implementation conforms to the GIST protocol and its API; currently we support draft version 14 [25]. We have developed a benchmarking NSLP application (“Ping”) [39] for testing purposes. If not otherwise mentioned, this document discusses our original NSIS implementation released as version 0.1.0. It consists of about 6,900 lines of code in total, which comprises about 1,300 lines for the core program, 2,000 lines for state machines, 700 lines for state management, 1,400 lines for message parsing and processing, and 1500 lines for the GIST-NSLP API. In addition to the original version, this document covers an improved version 0.5.2, which incorporates a new timer management, as well as many new features the original prototype lacked. The code is publicly available in [40].

The implementation architecture is shown in Fig. 6. It has been developed based on a single process approach using a main event loop based on XORP [41] library, which is used to implement socket maintenance and callbacks as well as timer callbacks. This design has no additional overhead for maintaining and synchronizing multiple threads, which results in a high throughput and a rather simple implementation.

Besides the event loop, a key component in our GIST implementation is state management. In order to support tens of thousands of signaling sessions efficiently, we used a hash table to manage the MRSs, associated with linked lists to resolve conflicts. A standard lookup takes constant time, however in the worst case, all table entries would be compared to find a given MRS.

To search the MRS table, one needs to know the associated key information, namely the session ID, the NSLP ID and message routing information (MRI). This is nevertheless subject to some limitations, e.g., it is not possible to search for all MRSs using a specific MRI. Such a search feature may be useful to find MRSs that are affected by a detected link

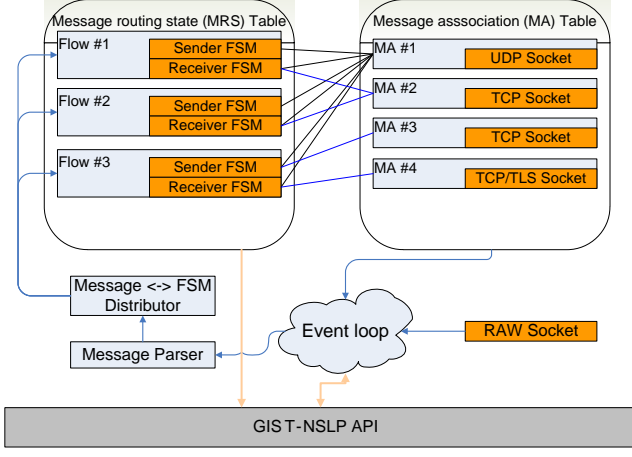


Fig. 6. GIST Implementation Architecture

failure. A possible solution is to maintain specialized hash tables for link failures, which would allow for quick searches. However, this approach would add maintenance overhead for every MRS table which usually comprise a number of tables.

In addition to managing MRSs, a GIST implementation has to manage MAs for C-mode operations. If two peers already have an MA and a new session is being established on the same path, the MA should be reused to minimize resource usage. This feature implies that there should be a way to search the MA table for an MA that can be reused for a certain session. Our implementation uses a second hash table to accomplish that goal. The upstream peer information (PI) serves as the key information. The UDP socket is treated as a “virtual” MA for the convenience of unifying the socket interface module.

Another important component of the GIST implementation is the finite state machine (FSM) to maintain states for each session. We implemented the GIST FSMs [34] based on a combination of the XORP timer class and an FSM framework that was originally written for the Linux ISDN device driver [42]. Every MRS is associated with two FSMs, one for the upstream peer and another one for the downstream peer. There is no need for a global table of FSMs, because every MRS provides pointers to the associated FSMs. In addition, every MA is associated with a list of FSMs, so that the state machines can be informed, e.g., when a loss of connectivity with its current peer takes place.

B. Testbed Setup and Tools

The performance experiments were carried out on six low-end PCs running Linux 2.6.8.1. They are equipped with the following hardware:

- Via Eden CPU 533 MHz
- 3 Realtek 100 Mb/s NICs
- 256 MB SDRAM PC 133
- 20 GB HDD

Fig. 7 depicts how we connected the nodes for our experiments. N_1 and/or N_2 was used as the sending host(s) – NI(s), while one or more of N_3 , N_4 , N_5 and N_6 were the flow destination(s) – NR(s). In addition to the benchmarking tool “Ping”,

we have also developed an Ethernet GIST dissector [43] for monitoring the GIST messages.

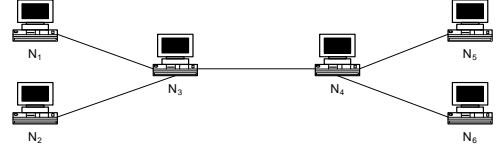


Fig. 7. Testbed Setup

The Ping tool is a light-weight NSLP protocol that sends so-called Ping messages through a GIST aware network, which emulates the ICMP Ping and Traceroute functions. The traversed nodes insert a timestamp and information about the local node (i.e., the IP address). When the message reaches its destination host, it is redirected upstream and traverses the network back to the original sender.

We use this tool in our experiments to model the scenario of a real NSLP application without introducing unnecessary overhead. Our main goal was to measure the maximum number of sessions a backbone router can maintain. In addition, the tool was intentionally designed to involve other aspects a real NSLP application would likely require, including:

- GIST layer session lookup;
- GIST layer session refreshes;
- Communication between GIST and the NSLP layer;
- NSLP layer message processing.

In order to focus on these goals, we neither maintain NSLP layer timers nor store NSLP layer state, but allow the sending node to send a Ping message for each session every 30 seconds in order to simulate NSLP behavior; this message can be regarded as a refresh message in the NSLP layer. As a result, we were able to use this tool to study the GIST performance and scalability. It is expected that a real NSLP application would have additional overhead, including timers, parsing and state management, all in the NSLP layer, resulting in some worse results in terms of round trip times and maximal number of sessions that can be maintained at a time.

A simple PHP script measures the CPU and memory utilization every second using the proc filesystem entries, in the same fashion as the popular *top* program. After completing the test, the script uses the debugging component of the GIST implementation to fetch internal statistical information like the average number of entries in the used hash table buckets.

To calculate the round trip times (RTTs), the information contained in every Ping message is saved on the sender and after the test is completed the collected timestamps are used to calculate the round trip times. As the measurements were conducted in lab environments without intervention from the background traffic, the standard deviations for the obtained values were very small, for example less than 0.3ms for the RTTs, thus the results are meant as the mean values.

C. Performance Study

1) *Scalability in Number of Sessions*: As signaling protocols maintain and manage soft state in network nodes, the most critical performance metric for GIST is the upper limit on the number of sessions a GIST node can maintain. Additionally, we would like to evaluate how the CPU load and memory consumption scale with an increasing number of concurrent sessions. Other parameters like average RTTs were collected too. We performed three experiments for this test.

In the first experiment, we used N_1 as the NI and N_3 as the NR, and let the NI first established a configured number of sessions and then emulated refreshes for all of them and measured performance of N_3 . The refresh intervals for NSLP and GIST MRS were set 30 and 180 seconds, respectively.

The results are shown in Fig. 8-10. The first observation is that the increase in CPU load and memory consumption is nearly linear. With the original implementation, the consumption of CPU time reached 70% (C-mode) – 71% (D-mode) of the whole system when serving 60,000 sessions at a same time in our test. Serving the same number of session, the improved version 0.5.2 consumed 75% (D-mode) of overall CPU time. While similar in overall performance, the new version is slightly slower than the original one. The assumption that optimizations in timer management and message composition compensate for the increased complexity in message validation and GIST logic is backed by the per-routing processing time study presented later.

The second observation is that the RTTs were very small (4.8-5.2 ms) before the session number reached 50,000. It increased to 56.2 ms when serving 55,000 sessions, then increased rapidly afterwards, reaching 7.0 seconds when serving 60,000 flows, indicating approaching the exhaustion of system resources (memory/CPU/interface) in network nodes.

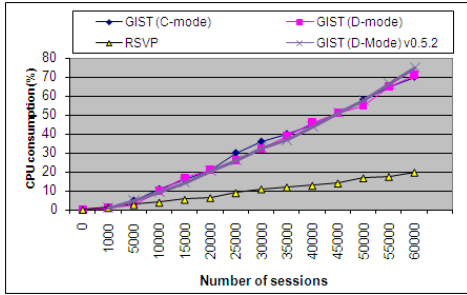


Fig. 8. Effect of concurrent sessions on CPU consumption

In the next experiment, we studied the case where two senders (N_1 and N_2), one intermediate node (N_3), and one receiver (N_4) were involved. NSLP refresh interval was 30 seconds, and GIST refresh rate was 180 seconds. We let each sender serve 30,000 sessions, so the receiver had to handle 60,000 sessions. The measured RTT turned out to be about 5.5 ms. This confirms that the bottleneck for RTT in the tests above is the sender and not the receiver.

Based on these observations, we obtain a rough estimate of the upper limit of the supported session number in a GIST node, which is at least 60,000. Note after the concurrent session number of 45,000, the average RTT increases rapidly,

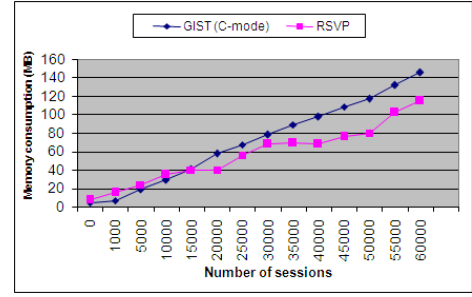


Fig. 9. Effect of concurrent sessions on memory consumption

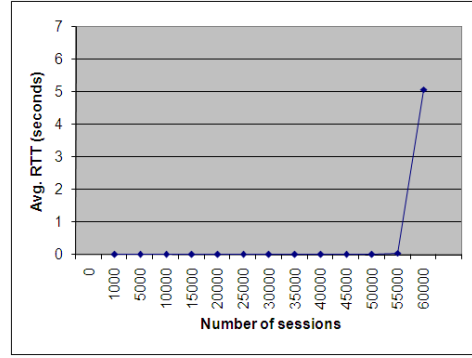


Fig. 10. Effect of concurrent sessions on average RTT

thus, the effective session number that the system can support is estimated as 45,000. This number may be improved by introducing some optimizations, such as the ones suggested in Section IV-E.

Another experiment we performed was to measure the approximate processing time required for a GIST message in an intermediate node, i.e., the time difference from incoming to outgoing message. Taking both the N_1 and N_2 as the flow receiver and using ethereal dissector, we performed tests for 20,000 and 60,000 simultaneous GIST sessions in steady state, respectively.

In the light traffic cases (20,000 sessions), the results show that the average processing time for GIST-Query and Response messages was very small, about 0.25 ms, whereas a GIST-Data (carrying Ping NSLP) message took the average processing time of 1.1 ms. This conforms to the RTT results obtained in Section IV-C.1.

In the second set (the more heavy-load traffic case), the processing time for Query/Response increased to 0.9 ms, whereas for a GIST-Data message it increased to 20 ms. This confirms our observation in the first experiment, namely when entering the heavy load traffic range, RTT is starting to be much larger than the light traffic case.

We also did performance tests of a recent RSVP implementation, the KOM RSVP engine [11] in the same testbed and PC hardware. The results are also shown in Fig. 8-10 and we could conclude that we obtained roughly comparable results. After fine tuning of the environment for running KOM RSVP, we observed that KOM RSVP grows slower for CPU consumption with session number increases: when serving 60,000 simultaneous sessions, KOM RSVP just needed

about 20% of CPU time, in comparison with 70%. This difference demonstrates certain properties of implementation-specific design and the testing environment, for example: 1) the use of XORP timer turned out to consume 50% of the overall CPU usage in our GIST implementation, while the *fuzzy timer* approach allowed KOM RSVP to manage timers more efficiently [11], 2) in order to reach high signaling loads, we did not change anything to the system environment, while KOM RSVP was necessary to be deliberately tuned, most likely due to a different development hardware/software platform the KOM RSVP developers chose. On the other hand, the required memory for KOM RSVP was found to be rather similar to that for GIST: it was just 20% less than GIST C-mode when both implementations served for 60,000 simultaneous sessions; for small numbers of sessions (less than 15,000), it required even more memory than our GIST implementation. This is due to our introduction of optimizations (see Section IV-E).

Ideally, the memory consumption in different signaling loads should be straight linear, but Fig. 9 shows that there were some turbulence over the time. This is likely caused by the non-deterministic OS scheduling regarding the receipt, queuing and delivery of each GIST/RSVP message, as both KOM RSVP and GIST were implemented as user space daemons.

2) *Analysis of Session Setup Time*: When GIST is used in a real application (not just a Ping client), a critical metric is the time required to finish the first signaling round trip (e.g., a QoS reservation). This involves the GIST three-way handshake for every hop-to-hop connection that is performed sequentially, which could result in a rather long initial setup delay. Our measurements show that this delay was between 3ms and 5ms for D-mode or C-mode scenarios when an existing message association can be reused. The number of sessions for this measurement ranged between 15,000 to 25,000.

3) *Impact of GIST Message Routing State Refreshes*: The main responsibility of GIST is to manage the MRSs and MAs which are used in delivering NSLP messages from one peer to another, where both states are soft states. We study the effect of MRS state refreshes since MA state refreshes by periodically GIST-Hello messages are not necessary if there are some active signaling messages between the peer pair.

We chose 30 seconds as NSLP refresh interval and ran tests under different refresh intervals for an overall number of 15,000 GIST sessions between N_1 and N_3 , all links operating on C-mode.

The measured CPU load in N_3 are summarized in Table II.

This indicates a small refresh interval at GIST level only introduces CPU load. Given the reliability properties of C-mode, a relatively long refresh interval (e.g., 180 sec) at GIST level for MRS maintenance which impose limited CPU overhead should be enough, especially where route changes are not frequently experienced.

We performed some more tests with similar results where all the six nodes in the testbed were involved. A low CPU load in intermediate nodes was observed when the GIST MRS refresh interval was set about 180 sec (also the reason why we selected this value as default refresh interval in other tests).

TABLE II
IMPACT OF GIST MESSAGE ROUTING STATE REFRESH INTERVAL ON CPU LOAD

Refresh interval (sec)	% of CPU load used by GIST
30	56%
60	47%
90	43%
120	42%
150	41%
180	40%
210	40%

4) *Per-routine Processing Time*: In order to study the bottlenecks of the implementation, we profiled each routine in the GIST code, using the *gprof* tool. Table III shows the profiling results for each routine's contributions to the overall system processing. It reveals that the XORP library consumes over half of the total running time, mostly for managing XORP timer facilities. The reason is that XORP uses a sorted heap to structure the timers – a more detailed profile shows that maintaining this heap consumes up to 38 percent of the overall runtime of our implementation. This is due to the fact that, while adding and removing a heap element imposes a time complexity of $O(\log(n))$, the heapify algorithm costs $O(n \log(n))$, where n is the total number of timers.

TABLE III
RUNTIME PROFILES OF THE IMPLEMENTATION

Code component	% of total running time	
	v. 0.1.0	v. 0.5.2
1. XORP / Own implementation	53%	10%
1.1 Timer Management	42%	5%
1.2 Socket Management	10%	5%
2. Receiving incoming message	8%	30%
2.1 Receiving and distribution to FSM	4%	13%
2.2 Message parsing	4%	17%
3. Message composing and internal reading	17%	16%
4. Hash tables (MRS and MA)	8%	18%
5. Finite state machine	7%	15%
6. NSLP level processing (ping)	1%	6%
7. Miscellaneous	6%	5%

Table III also confirms that version 0.5.2 is slower than the original version due to additional overhead spent on validation during message parsing, as well as more complexity in the GIST state machine. These kinds of performance penalties are a common phenomenon of maturing software, caused by more and more corner cases being detected and handled properly.

5) *C-mode versus D-mode*: GIST is capable of operating in both C-mode and D-mode. so that the difference in CPU load between both modes of operation is of interest. We implemented C-mode in both TCP and TLS/TCP but the evaluation here focuses on using TCP as transport.

Fig. 8 shows the CPU load for a different number of maintained sessions in C-mode and D-mode. From this figure we can conclude that the CPU load does not make much difference from each other.

Given that TCP offers a number of transport features desired for signaling protocols, as outlined in Section III, the above

result suggests that C-mode should be used as much as possible instead of D-mode for GIST message transport.

D. Bucket-based Timer Management

The results obtained during our initial performance study clearly showed that the XORP timer management was a major bottleneck in our implementation. Hence, we decided to switch to a different, much more efficient mechanism. As already discussed, XORP uses a heap to organize all active timers, which requires to run the complex heapify algorithm for each addition and removal of individual timers. While this is reasonable for a diverse set of timers, it is very inefficient in GIST, where many timers share the same structure: The resolution of GIST timers is in seconds instead of milliseconds and the firing interval for GIST timers is not diverse, i.e. many flows are likely to share the same refreshing interval. Thus, we decided to group timers based on the combination of two properties: The interval and the starting offset. The offset is defined as $offset = time_since_startup \bmod interval$. For example, a timer which is created 50 seconds after the start of GIST and which is supposed to fire every 30 seconds, will have an offset value of 20 seconds. Every interval/offset combination corresponds to a *bucket* which uses a linked-list to store any number of timers.

The total number of buckets GIST has to manage is drastically lower than number of timers. Imagine an optimal case, where all intervals are the same. In our case we used an interval of 180 seconds for GIST refreshes, which means that there are no more than 180 buckets (i.e. one for every possible offset). In order to insert a new timer, the matching bucket needs to be found and the timer needs to be added to the linked-list. To check which timers have to be fired, the system needs to look at every bucket and check if $time_since_startup \bmod interval = offset$ holds. If the condition holds, all timers contained in the bucket need to be fired and otherwise the bucket is skipped. Execution of timers is only done once per second, while adding new timers is done many hundred times per second in heavy loaded GIST nodes. Therefore, we decided to further optimize the lookup of buckets matching a certain interval/offset combination. This is done by organizing the buckets in a hash table. As we need to walk through all buckets when firing timers, the hash table should be densely populated to avoid checking hash values which do not contain any bucket. Hence, we decided to use a hash table size of 60. Using the example from above (refresh interval of 180 seconds), we end up with approx. 3 buckets per hash value.

Table III shows that our new timer management is much more efficient for managing GIST refreshes than the general purpose heap-based XORP timers. While XORP timers used to consume over 40% of overall CPU time, the new timers consume less than 10% CPU time in our most recent implementation. Please note, that the two measurements are not entirely fair, as the new numbers are obtained with version 0.5.2 of our implementation, while the XORP numbers were measured with the original 0.1.0 release. As seen in Section IV-C.1, due to increased complexity in GIST handling, the overall CPU

consumption of the 0.5.2 version is slightly higher compared to the original one. Nevertheless, the performance gain due to switching to the new bucket-based timer management is significant.

E. Performance Optimizations

During the performance experiments we introduced several optimization techniques and thus were able to significantly reduce the CPU load of our implementation. The first optimization was to reduce data copying between processing routines. When designing an object oriented implementation, the tendency is to design every class with its own copy of the data to ensure integrity. Network protocol implementations, however, cannot afford to waste CPU and memory resources. As a result, ideally there should be just one copy of every incoming and outgoing packet and all code parts should use pointers to the part they want to use. The zero-copy approach, which has not been fully implemented in our code, reduced CPU load by about 20 percent.

Another performance bottleneck was found to be a poor design of the implemented hash table – initially we used the standard hash function, where 1 byte array as the hash key and dense size in rehash turned out to consume much computation resources. The hash function used now is still simple but efficient: The key is treated as a 4-bytes array and the hash value is the sum of the values in the array reduced modulo the hash table size. Let k_1, k_2, \dots, k_n be the values of the integer array and p be the hash table size. Then the (current) hash function is given by:

$$hash(k) = (k_1 + k_2 + \dots + k_n) \bmod p$$

This results in a possible output range of values from 0 to $2^{32} - 1$. The original hash function based on an array of 1 byte values, in contrast, results in a very limited range of output values: because all the k_i are just in the range of 0 to 255 and a typical number of bytes is 16, the range of the hash function was 0 to 4080. This means that a huge part of a large hash table was never used and so the distribution along the range was not uniform.

The hash table is rehashed with a higher hash table size whenever the load factor exceeds a certain limit (i.e., 0.5). The load factor is given by:

$$load\ factor = \frac{stored\ elements}{hash\ table\ size}$$

Originally, the list of supported hash table size was dense, which resulted in the need to rehash very often. The solution was to rapidly increase hash table sizes exponentially (i.e. the hash table size is more than doubled from one value to the next) to quickly achieve the necessary size while minimizing rehashing turns, which turned out very effective.

By optimizing the hash table, the average number of items in one hash table bucket was reduced by one magnitude and the overall GIST performance increased by approximately 20%.

The most important optimizations discussed above were also accompanied by less significant changes. Some functions were called several million times within a few minutes of operation,

which resulted in a large amount of overhead. Using the *inline* statement to integrate the function body directly into the calling code reduced this overhead and the performance gain was up to 10 percent of overall performance. In the current implementation, some small code optimizations such as making common functions inline and replacing memcpy() calls by direct assignments, which reduce readability but improve performance, were carried out in frequently used code sections.

These optimizations cut CPU load by half by incorporating the well-known principle of zero-copy and optimizing central data structures and frequently used code parts. As already mentioned in the above subsections, further optimizations in memory management and introduction of thread pooling ought to contribute to more promising results.

V. RELATED WORK

Over the last decade, various issues for signaling in the Internet, especially for QoS resource reservation, have been widely investigated. They have ranged from soft state modeling [45], [46], scalability enhancements (e.g., by reservation aggregation and more efficient refreshes) [13], [47]–[49], to complexity [14]–[16], [20] and applicability [50]–[52]. A lot of works have attempted to simplify or extend RSVP (even under other protocol names). For example, today there are 44 RFCs with the word “RSVP” in their titles, while the index of Internet drafts lists 16 documents with “RSVP” in their titles. These works employed either a server-based or a router-based approach. A 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., [53]), generic signaling support has acquired little focus. Furthermore, the dominant way of using the Router Alert Option and coupling discovery with discovery have lead to a number of security and complexity problems [20], [54]. Derived from RSVP concepts, the Label Distribution Protocol (LDP) [64] was standardized by the IETF for distributing MPLS labels (later for several other signaling purposes), but it does not address the next signaling hop discovery problem nor adequate security, leaving them for the administrators’s concern.

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 a per-hop reliability mechanism [12] and general signaling support [22]. This approach removes the QoS- and multicast-specific processing burden from the original RSVP, and has the advantage of better compatibility with existing protocol and implementations. Nonetheless, issues concerning security, congestion control and fragmentation of signaling messages may be more 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 message delivery with next-hop discovery. Variations of the RSVP-based approach have been described in [22], [55]. The

latter proposal suggests a decoupled 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. Both proposals, however, leave the actual mechanism undefined. The present GIST design has followed many ideas of the CASP-based approach and reuses RSVP concepts wherever possible [25]. Nonetheless, the tradeoff between performance, security, complexity and modularity is still an issue in both approaches. Fault recovery, especially in dealing with re-routing [56] remains one major concern in the layered architecture.

These studies have been accompanied by some researchers on performance evaluation, in particular with RSVP. For example, Chiueh *et al.* [10] 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. Pan *et al.* [14], [36], [57] extensively studied processing performance and scalability issues of RSVP and possible protocol improvements. Karsten *et al.* [11] implemented a user-level RSVP protocol engine (which allows multi-threading processing) in Linux C++, evaluated its performance to find out the upper limits of the reservation requests and profiled the system for different parts of protocol operations.

After we developed an open source CASP implementation and evaluated its running properties [58], the present paper elaborates the overhead study and performance results of the evolved IETF GIST protocol through a detailed evaluation. To our knowledge, this is the first empirical study of the GIST protocol.

VI. CONCLUSIONS

This paper presented the overhead, implementation and performance study of GIST, a generic IP signaling protocol being developed by the IETF. In contrast to traditional methods, GIST provides a modular architecture to support any application (NSLP) requesting signaling services, and reduces complexity by relying on existing security and transport protocols for achieving signaling functionalities. The modularity design of the GIST implementation provides a flexible way for state management and message processing. The result is improved extensibility, security, and transport properties at the cost of additional overhead. The implementation performed efficiently when serving a number of sessions (at least 60,000) and the profiling shows the detailed processing and round-trip times for different numbers of signaling sessions. C-mode is concluded to be preferred to D-mode due to its richer functionality despite slightly higher overhead during the session setup.

The focus of this paper has been on GIST properties, such as protocol overhead, scalability and other performance issues. Composing signaling application protocols (NSLPs) and its effect on overhead and performance will certainly pose imminent concerns once the overall system has materialized, which will also effect its deployment.

In addition, a number of issues were encountered when investigating the GIST protocol, which went beyond the scope of this study. It is clear to say that further study will be necessary with respect to a more sophisticated network topology, as well as the interaction with underlying transport and security protocols (effects of applying IPsec/TLS and different TCP variants in particular). In addition, studies are being carried out on other issues connected with GIST/NSIS, such as mobility support [25], [59], fault handling and route change, as well as the QoS and NAT/Firewall NSLPs under standardization [30], [31], and a comprehensive performance evaluation of the whole NSIS protocol stack in comparison with RSVP.

ACKNOWLEDGMENT

We would like to thank Bernd Schlör, Henning Peters and Andreas Westermaier for their assistance in the implementation, as well as Elwyn Davies, Cedric Aoun, Tseno Tsenov, Fabian Meyer and Sebastian Willert for their contributions. We would also like to thank anonymous reviewers and members of the IETF NSIS working group for their helpful comments, and Martin Karsten for sharing his experience and kind support for KOM-RSVP experiments.

REFERENCES

- [1] D. Clark, "The Design Philosophy of the DARPA Internet Protocols," in *Proc. of SIGCOMM 1988*, Stanford, CA, Aug. 1988.
- [2] R. Braden, D. D. Clark, and S. Shenker, "Integrated services in the Internet architecture: an overview," Internet Engineering Task Force, RFC 1633, June 1994.
- [3] B. E. Carpenter and S. Brim, "Middleboxes: Taxonomy and Issues," Internet Engineering Task Force, RFC 3234, Feb. 2002.
- [4] M. Blumenthal and D. Clark, "Rethinking the design of the Internet: The end to end arguments vs. the brave new world," *ACM Transactions on Internet Technology*, vol. 1, no. 1, pp. 70–109, Aug. 2001.
- [5] J. Kempf and R. Austein, "The Rise of the Middle and the Future of End-to-End: Reflections on the Evolution of the Internet Architecture," Internet Engineering Task Force, RFC 3724, Mar. 2004.
- [6] L. Zhang, S. Deering, D. Estrin, S. Shen, and D. Zappala, "RSVP: A New Resource ReSerVation Protocol," *IEEE Network*, vol. 7, no. 5, pp. 8–18, Sept. 1993.
- [7] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification," Internet Engineering Task Force, RFC 2205, Sept. 1997.
- [8] J. Wroclawski, "The use of RSVP with IETF integrated services," Internet Engineering Task Force, RFC 2210, Sept. 1997.
- [9] S. Blake, D. L. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated service," Internet Engineering Task Force, RFC 2475, Dec. 1998.
- [10] T. Chiueh, A. Neogi, and P. Stirpe, "Performance Analysis of an RSVP-Capable Router," in *Proc of IEEE RTAS*, Denver, Colorado, USA, June 1998.
- [11] M. Karsten, J. Schmitt, and R. Steinmetz, "Implementation and Evaluation of the KOM RSVP Engine," in *Proc of IEEE INFOCOM*, Anchorage, Alaska, USA, Apr. 2001.
- [12] L. Berger, D. Gan, G. Swallow, P. Pan, F. Tommasi, and S. Molendini, "RSVP refresh overhead reduction extensions," Internet Engineering Task Force, RFC 2961, Apr. 2001.
- [13] P. Pan, E. Hahne, and H. Schulzrinne, "BGRP: A Tree-Based Aggregation Protocol for Inter-domain Reservations," *Journal of Communications and Networks*, vol. 2, no. 2, pp. 157–167, June 2000.
- [14] P. Pan and H. Schulzrinne, "YESSIR: A Simple Reservation Mechanism for the Internet," in *Proc of ACM NOSSDAV*, Cambridge, UK, July 1998.
- [15] G. Feher, K. Nemeth, M. Maliosz, I. Cselenyi, J. Bergkvist, D. Ahlrad, and T. Engborg, "Boomerang – A Simple Protocol for Resource Reservation in IP Networks," in *Proc of IEEE RTAS*, Vancouver, British Columbia, Canada, June 1999.
- [16] P. Chandra, A. Fisher, and P. Steenkiste, "A Signaling Protocol for Structured Resource Allocation," in *Proc of IEEE INFOCOM*, New York, NY, USA, Mar. 1999.
- [17] A. Talukdar, B. Badrinath, and A. Acharya, "MRSVP: a Resource Reservation Protocol for an Integrated Services Network with Mobile Hosts," *Wireless Networks*, 7(1): 5–19, 2001.
- [18] S. Lee, A. Gahng-Seop, X. Zhang, and A. Campbell, "INSIGNIA: An IP-Based Quality of Service Framework for Mobile Ad Hoc Networks," *Journal of Parallel and Distributed Computing, Special issue on Wireless and Mobile Computing and Communications*, 60(4): 374–406, 2000.
- [19] W.-T. Chen and L.-C. Huang, "RSVP Mobility Support: A Signaling Protocol for Integrated Services Internet with Mobile Hosts," in *Proc of IEEE INFOCOM 2000*, Tel-Aviv, Israel, Mar. 2000.
- [20] J. Manner and X. Fu, "Analysis of Existing Quality-of-Service Signaling Protocols," Internet Engineering Task Force, RFC 4094, May 2005.
- [21] The IETF Next Steps in Signaling (NSIS) Working Group. [Online]. Available: <http://www.ietf.org/html.charters/nsis-charter.html>
- [22] B. Braden and B. Lindell, "A Two-Level Architecture for Internet Signaling," Internet draft (draft-braden-2level-signaling-01), work in progress, Oct. 2002.
- [23] H. Schulzrinne, H. Tschofenig, X. Fu, and A. McDonald, "CASP – Cross-Application Signaling Protocol," Internet draft (draft-schulzrinne-casp-01), work in progress, Mar. 2003.
- [24] X. Fu, H. Tschofenig, and D. Hogrefe, "Beyond QoS Signaling: a Generic IP Signaling Framework," *Computer Networks*, 50(17): 3416–3433, Dec. 2006.
- [25] H. Schulzrinne and R. Hancock, "GIST: General Internet Signaling Transport," Internet draft (draft-ietf-nsis-ntlp-15), work in progress, Feb. 2008.
- [26] R. Hancock, G. Karagiannis, J. Loughney, and S. Van den Bosch, "Next Steps in Signaling (NSIS): Framework," Internet Engineering Task Force, RFC 4080, June 2005.
- [27] R. Stewart, "Stream Control Transmission Protocol," Internet Engineering Task Force, RFC 4960, Sept. 2007.
- [28] E. Kohler, M. Handley, and S. Floyd, "Datagram Congestion Control Protocol (DCCP)," Internet Engineering Task Force, RFC 4340, Mar. 2006.
- [29] X. Fu, C. Dickmann, and J. Crowcroft, "General Internet Signaling Transport (GIST) over SCTP," Internet draft (draft-ietf-nsis-ntlp-sctp-04), work in progress, Feb. 2008.
- [30] J. Manner, G. Karagiannis, and A. McDonald, "NSLP for Quality-of-Service signaling," Internet draft (draft-ietf-nsis-qos-nslp-16), work in progress, Feb. 2008.
- [31] M. Stiernerling, H. Tschofenig, C. Aoun, and E. Davies, "NAT/Firewall NSIS Signaling Layer Protocol (NSLP)," Internet draft (draft-ietf-nsis-nslp-natfw-16), work in progress, Feb. 2008.
- [32] A. Fessi, G. Carle, F. Dressler, J. Quittek, C. Kappler, and H. Tschofenig, "NSLP for Metering Configuration Signaling," Internet draft (draft-dressler-nsis-metering-nslp-05), work in progress, Mar. 2007.
- [33] D. D. Katz, "IP Router Alert Option," Internet Engineering Task Force, RFC 2113, Feb. 1997.
- [34] T. Tsenov, H. Tschofenig, X. Fu, C. Aoun, and E. Davies, "GIST State Machine," Internet draft (draft-ietf-nsis-ntlp-statemachine-05), work in progress, Feb. 2008.
- [35] C. Aoun, E. Davies, and H. Tschofenig, "Securing Middlebox Discovery for Path-Directed Signaling in the Internet," in *IEEE ASWN 2005 Workshop Proceedings*, Paris, France, July 2005.
- [36] P. Pan and H. Schulzrinne, "Staged Refresh Timers for RSVP," in *Proc of IEEE Global Internet*, Phoenix, AZ, USA, Nov. 1997.
- [37] L. Wang, A. Terzis, and L. Zhang, "A New Proposal for RSVP Refreshes," in *Proc of IEEE ICNP*, Toronto, Canada, Nov. 1999.
- [38] L.-E. Jonsson, G. Pelletier, and K. Sandlund, "The RObust Header Compression (ROHC) Framework," Internet Engineering Task Force, RFC 4995, July 2007.
- [39] C. Dickmann, I. Juchem, S. Willert, and X. Fu, "A stateless Ping tool for simple tests of GIST implementations," Internet draft (draft-juchem-nsis-ping-tool-02), work in progress, July 2005.
- [40] OpenNSIS Implementation, University of Göttingen, <http://user.informatik.uni-goettingen.de/~nsis>
- [41] The eXtensible Open Router Platform (XORP). [Online]. Available: <http://www.xorp.org/>
- [42] P. Marques, "Kernel ISDN subsystem and device drivers." [Online]. Available: <http://kernel.org/pub/linux/kernel/people/marcelo/linux-2.4/drivers/isdn/>
- [43] Ethereal Dissector for GIST. [Online]. Available: <http://user.informatik.uni-goettingen.de/~nsis/ethereal.html>

- [44] G. Varghese and A. Lauck, "Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility", in *Operating Systems Review, Special Issue: Proceedings of the Eleventh Symposium on Operating Systems Principles*, Austin, TX, USA, 21(5):25-38, Nov. 1987
- [45] S. Raman and S. McCanne, "A model, analysis, and protocol framework for soft state-based communication," in *Proc. of SIGCOMM*, Cambridge, MA, USA, Aug. 1999.
- [46] P. Ji, Z. Ge, J. Kurose, and D. Towsley, "A Comparison of Hard-state and Soft-state Signaling Protocols," in *Proc. of SIGCOMM*, Karlsruhe, Germany, Aug. 2003.
- [47] 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," Internet Engineering Task Force, RFC 2998, Nov. 2000.
- [48] F. Baker, C. Iturralde, F. L. Faucheur, and B. Davie, "Aggregation of RSVP for IPv4 and IPv6 reservations," Internet Engineering Task Force, RFC 3175, Sept. 2001.
- [49] 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 *Proc. of ACM SIGCOMM*, Stockholm, Sweden, Aug. 2000.
- [50] 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," Internet Engineering Task Force, RFC 2208, Sept. 1997.
- [51] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow, "RSVP-TE: extensions to RSVP for LSP tunnels," Internet Engineering Task Force, RFC 3209, Dec. 2001.
- [52] A. Terzis, J. Krawczyk, J. Wroclawski, and L. Zhang, "RSVP operation over IP tunnels," Internet Engineering Task Force, RFC 2746, Jan. 2000.
- [53] D. Mitzel, D. Estrin, S. Shenker, and L. Zhang, "An architectural comparison of ST-II and RSVP," in *Proc. of IEEE INFOCOM*, Toronto, Ontario, Canada, June 1994.
- [54] T.-L. Wu, S. F. Wu, Z. Fu, H. Huang, and F.-M. Gong, "Securing QoS: Threats to RSVP messages and their countermeasures," in *Proc. of IWQoS*, London, UK, June 1999.
- [55] M. Shore, "The NSIS Transport Layer Protocol (NTLP)," Internet draft (draft-shore-ntlp-00), work in progress, May 2003.
- [56] S. Nelakuditi, S. Lee, Y. Yu, and Z.-L. Zhang, "Failure insensitive routing for ensuring service availability," in *Proc. of IWQoS*, Monterey, CA, June 2003.
- [57] P. Pan and H. Schulzrinne, "Processing Overhead Studies in Resource Reservation Protocols," in *Proc. of International Teletraffic Congress (ITC)*, Salvador, Brazil, Sept. 2001.
- [58] X. Fu, D. Hogrefe, and S. Willert, "Implementation and Evaluation of the Cross-Application Signaling Protocol (CASP)," in *Proc. of IEEE ICNP*, Berlin, Germany, Oct. 2004.
- [59] T. Sanda, X. Fu, S. Jeong, J. Manner, and H. Tschofenig, "Applicability Statement of NSIS Protocols in Mobile Environments," Internet draft (draft-ietf-nsis-applicability-mobility-signaling-09), work in progress, Feb. 2008.
- [60] J. B. Postel, "Transmission Control Protocol," Internet Engineering Task Force, RFC 793, Sept. 1981.
- [61] J. Touch, "TCP control block interdependence," Internet Engineering Task Force, RFC 2140, Apr. 1997.
- [62] H. Balakrishnan and S. Seshan, "The Congestion Manager," Internet Engineering Task Force, RFC 3124, June 2001.
- [63] R. Braden and L. Zhang, "Resource ReSerVation Protocol (RSVP) – Version 1 Message Processing Rules," Internet Engineering Task Force, RFC 2209, Sept. 1997.
- [64] L. Andersson, P. Doolan, N. Feldman, A. Fredette and B. Thomas, "LDP Specification," Internet Engineering Task Force, RFC 3036, Jan. 2001.
- [65] X. Fu, H. Schulzrinne, H. Tschofenig, C. Dickmann and D. Hogrefe, "Overhead and Performance Study of the General Internet Signaling Transport (GIST) Protocol," in *Proc. of IEEE INFOCOM*, Barcelona, Spain, Apr. 2006.

APPENDIX – SOURCES OF PROTOCOL OVERHEAD IN GIST IN COMPARISON WITH RSVP

Here we give the details on how each of the layers of GIST and RSVP protocol structures contributes to their overall protocol overhead.

1) *IP*: Both RSVP and GIST messages need an IPv4 or IPv6 header, which is 20 bytes or 40 bytes without options, routing, fragmentation and security headers. For GIST-Query and RSVP-Path messages, the IP layer requires additional 4 bytes (for IPv4) or 8 bytes (for IPv6) in order to accommodate the IP Router Alert Option.

2) *Transport Layer*: GIST-Query messages are encapsulated using UDP, thus the transport layer overhead is 8 bytes. Other GIST messages can use either D-mode (UDP) or C-mode (TCP by default), resulting in a default transport layer overhead of 8 bytes (UDP header) or 20 bytes (a minimum TCP header). Note that C-mode messages in GIST require additional transport layer messages to accomplish the transport functionality, such as connection setup and reliability. Under normal circumstances (e.g., no loss, non-congested, no fragmentation), a TCP connection setup requires an additional TCP SYN, a SYN+ACK message and a TCP ACK message, whereas each GIST-layer message exchange needs an underlying TCP ACK message. SYN or SYN+ACK messages carry an MSS option (4 bytes) in addition to the normal TCP header, thus their overall overhead is 24 bytes plus IP header. The overhead of TCP ACK is 20 bytes plus IP header.

By default, RSVP messages are encapsulated directly using IP, so normally there is no transport layer overhead in RSVP. (Note the use of UDP for RSVP signaling is not discussed here.)

3) *GIST*: The GIST layer overhead can differ from one GIST message type to another, from one NSLP to another. Firstly, for D-mode messages it attaches a magic number (4 bytes) which is the first 32-bit datagram payload. It also relies on the used lengths of query-cookie and response-cookie as well as peer identity (PI, part of the NLI – the network layer information) and message routing method (MRM, used for managing message routing state) [25]. In our work we choose 36 bytes as the length for both query-cookie and response-cookie objects. We use the peer's IP address as the PI, thus a PI object length is 8 bytes (for IPv4) or 20 bytes (for IPv6). Among the optional fields of a basic path-coupled MRM, we choose to use only destination port (2 byte) for IPv4 and only flow label (3 bytes) for IPv6, which is suggested for usage by some other protocols as well, e.g., [7], [23]. All the mandatory fields are used in below discussions.

GIST-Query message comprises a magic number (4 bytes), common header (8 bytes), an MRM object (24 bytes for IPv4, 52 bytes for IPv6), a session ID object (20 bytes), a query-cookie object (36 bytes) and a network layer information object (24 bytes for IPv4, 36 for IPv6). For a node desiring C-mode operation, the Querying node's stack proposal object (12 bytes) and stack configuration data object (20 bytes) are also added. Therefore, the overall GIST layer overhead of a GIST-Query message is as follows:

$4 + 8 + 24 + 20 + 36 + 24(+12 + 20) = 116(+32, \text{ if stack proposal exists})$ bytes for IPv4, and

$4 + 8 + 52 + 20 + 36 + 36(+12 + 20) = 156(+32, \text{ if stack proposal exists})$ bytes for IPv6.

A GIST-Response message echos the query cookie and stack proposal objects, and additionally adds a response cookie object (36 bytes) to the received query message. Thus, the overall GIST layer overhead of a GIST-Response (C-mode) is 152 (+32 with stack proposal) bytes for IPv4 and 192 (+32 with stack proposal) bytes for IPv6.

A GIST-Confirm message differs from a GIST-Query in that it contains a response cookie object instead of a query cookie object (but of the same length), and removes the attached stack configuration data, besides the NSLP payload. Therefore, the overall GIST layer overhead of a GIST-Confirm is the same as Query.

A GIST-Data message comprises a common header, MRM, session ID and network layer information objects, excluding NSLP payload. GIST-Data message overhead is then as follows:

$8 + 24 + 20 + 20 = 72$ bytes for IPv4, and

$8 + 52 + 20 + 36 = 116$ bytes for IPv6.

4) *RSVP*: A minimum RSVP-Path message contains the IP layer (with overhead of 24 bytes for IPv4, 48 bytes for IPv6 including router alert option), common RSVP header (8 bytes), a session object (12 bytes for IPv4 and 24 bytes for IPv6), TIME_Values object (8 bytes) and a RSVP_HOP object (12 bytes for IPv4, 24 bytes for IPv6), in addition to the actually signaled data, namely the SENDER_TSpec (12 bytes [8]). Therefore, a minimum RSVP-Path message requires the following overhead for carry signaled data of 12 bytes:

$24 + 8 + 12 + 12 + 8 = 64$ bytes for IPv4, and

$48 + 8 + 24 + 24 + 8 = 112$ bytes for IPv6.

A minimum RSVP-Resv message for FF style (i.e., unicast) contains the IP header, common RSVP header, a session object, a RSVP_HOP object, a STYLE object (8 bytes), and a Filter_Spec object (of 12 bytes length for IPv4, or of 24 bytes length for IPv6), in addition to the embedded signaling data, i.e., a FlowSpec object (of 48 bytes length for GS, the Guaranteed Service, or of 12 bytes length for CLS, the Controlled Load Service [8]). This indicates that a minimum unicast RSVP-Resv message imposes the following overhead for carrying signaled data of 48 bytes (GS) or 12 bytes (CLS):

$20 + 8 + 12 + 12 + 8 + 12 = 72$ bytes for IPv4, and

$20 + 8 + 24 + 24 + 8 + 24 = 108$ bytes for IPv6.

5) *Memory Consumption*: Different from stateless protocols (e.g., IP and UDP), TCP, GIST layer and RSVP layer introduces memory requirements to store their layer-specific states, besides their protocol engine repository. As the exact presentation of these states is not part of the standards and may differ from one implementation/computer architecture to another, we estimate them below and validate them in the evaluation (see Section IV-C).

In the TCP layer, each TCP connection maintains a data structure for its state (TCP Control Block or TCB) [60], which includes a combination of parameters, such as connection state, current round-trip time estimates, congestion control information, and process information. A TCB connection state can vary in size between 256 bytes or less and more than

1 kilobytes. In GIST, TCP connections are recommended to be shared across signaling sessions between the same GIST pairs, where TCP Control Block Interdependence (TCBI) [61] or Congestion Manager [62] may be used in order to reduce connection state size, e.g., up to 512 bytes. Use of such multiplexing techniques allows a rather low memory consumption for per-peer GIST state management.

The GIST layer in D-mode maintains a per-session state, namely the message routing state. A minimum MRS state entry contains MRI (e.g., 1-byte method identifier for “path-coupled”, and 10-byte 5-tuple flow ID for IPv4 or 35-bytes 3-tuple flow ID for IPv6 comprising flow label, flow sender’s address, flow receiver’s address), 16-byte session ID, 1-byte NSLP ID, response direction (e.g., flow sender’s address, 4 bytes for IPv4 and 16 bytes for IPv6) and query direction (e.g., flow receiver’s address). This indicates that such an MRS entry is 36 bytes (IPv4) or 85 bytes (IPv6) in size, in addition to a validity timer.

In addition to the per-session state MRS (same as in D-mode), GIST layer in C-mode also maintains a per-peer state MA, which includes the GIST messages pending transmission (the number can be zero) and MA active timer, which is rather small in size when serving for a number of MRS sessions.

In contrast, each RSVP node maintains a per-session Path State Block (PSB) and a Resv State Block (RSB) [63], each with a validity timer and refresh interval. A minimum PSB includes information about session (8 bytes for IPv4 and 20 bytes for IPv6), Sender_Template (8 bytes for IPv4 and 20 bytes for IPv6), Sender_TSpec (12 bytes), previous hop’s IP address (4 bytes for IPv4, 16 bytes for IPv6) and logical interface handle (4 bytes), remaining IP TTL (1 byte), and several flags (assuming 1 byte), in total 38 bytes for IPv4 and 74 bytes for IPv6. A minimum RSB includes session (8 bytes for IPv4 and 20 bytes for IPv6), next hop IP address, Filter_Spec (12 bytes for IPv4 and 24 bytes for IPv6), style (4 bytes), and FlowSpec (36 bytes for CLS), in total 64 bytes for IPv4 and 90 bytes for IPv6. This represents 82 bytes for IPv4 and 164 bytes for IPv6 in overall RSB and PSB excluding management overhead and timers. This conclusion (i.e., slightly higher than GIST memory consumption) does not appear surprising, since unlike GIST states, RSVP states also include IntServ parameters.



Xiaoming Fu is professor and head of Computer Networks Group at the University of Göttingen, Germany. He received his Ph.D. Degree in Computer Science from Tsinghua University, China in 2000. He was a research staff at Technical University Berlin, before moving to Göttingen as assistant professor in 2002. His research interests include network architectures, protocols, mobile communications and service overlays. He has served as TPC member/session chair/chair for several networking conferences such as INFOCOM, ICNP, ICDCS, and

MobiArch. He is currently editorial board member of Elsevier Computer Communications Journal and a guest editor of IEEE Network Special Issue on Implications and Control of Middleboxes in the Internet.



Henning Schulzrinne received his Ph.D. from the University of Massachusetts in Amherst, Massachusetts. He was a member of technical staff at AT&T Bell Laboratories, Murray Hill and an associate department head at GMD-Fokus (Berlin), before joining the Computer Science and Electrical Engineering departments at Columbia University, New York. He is currently chair of the Department of Computer Science. Protocols co-developed by him, such as RTP, RTSP and SIP, are now Internet standards, used by almost all Internet telephony and

multimedia applications. His research interests include Internet multimedia systems, ubiquitous computing, mobile systems, quality of service, and performance evaluation. He is a Fellow of the IEEE.



Hannes Tschofenig received his Diploma degree from the University of Klagenfurt, Austria. He joined Siemens Corporate Technology in 2001 and is currently a senior standardization specialist at Nokia Siemens Networks and part-time researcher at the University of Göttingen. His research interests lie on network architectures, protocols, services and related security issues. He is currently co-chair of the IETF Emergency Context Resolution with Internet Technologies (ECRIT), Provisioning of Symmetric Keys (KEYPROV) and Diameter Maintenance and

Extensions (DIME) working groups, as well as secretary of the NSIS working group. He is a co-author of several standard track RFCs and Internet drafts, and contributed to EU funded projects including SHAMAN, Ambient Networks and ENABLE.



Christian Dickmann received his bachelor's degree (with honors) in Computer Science in 2005 and is working towards his master's degree at the University of Göttingen. He was an intern at BMW Car IT and Siemens AG. In 2007, he was a visiting research scholar at Columbia University, New York.



Dieter Hogrefe received his Diploma degree and Ph.D. from the University of Hannover, Germany. His research activities are directed towards Computer Networks and Protocol Engineering. In these fields he has published several books and numerous papers on Internet technology, analysis, simulation and testing of formally specified communication systems. After years of research positions in Siemens, he held professorships at the Universities of Dortmund, Berne and Luebeck. Since 2002 he is Professor for Telematics at the University of Göttingen.

Since 1996 Prof. Hogrefe is chairman of the Technical Committee Methods for Testing and Specification at the European Telecommunication Standards Institute, ETSI.