Simplified Network Signaling Architecture

Lauri Liuhto

Helsinki May 14, 2014
M. Sc. Thesis
UNIVERSITY OF HELSINKI
Department of Computer Science
The wheel has been reinvented several times in signaling protocols. Most signaling protocols re-invent, e.g., their own signaling transport methods, end-point discovery, measures for reliable exchange of messages and security features. Next Steps In Signaling (NSIS) framework was created in the IETF to design a single unified framework for various network signaling needs. The signaling transport layer of NSIS, the General Internet Signaling Transport (GIST), was specified in the IETF to provide a common transport service for signaling applications. The NSIS suite also includes two signaling protocols, NSIS Signaling Layer Protocols (NSLP), one for Quality of Service provisioning and one to configure middleboxes, in particular Network Address Translators and firewalls.

The different signaling applications use GIST message delivery services through an API that consists of several operations. On top of common operations for sending and receiving data, the API also covers network events, errors and session state management. The API covers all GIST aspects, and allows application developers to have adequate knowledge of network state. However, as a result the API is very cumbersome to use, and an application developer needs to take care of non-trivial amount of details. A further challenge is that to create a new signaling application, one needs to acquire and register a unique NSLP identifier with the Internet Assigned Numbers Authority (IANA).

This thesis presents the Messaging NSLP, that provides an abstraction layer to hide complex GIST features from the signaling application. Developers of Messaging Applications can use a simple Messaging API to open and close sessions and to transfer application data from one Messaging Application node to another.

Prototype implementations of NSLP API and Messaging NSLP were created and tested to verify the protocol operation with various network scenarios. Overhead analysis of GIST and Messaging NSLP were performed, and results are compatible with earlier, third-party analysis. The Messaging NSLP can introduce up to 938 bytes of overhead to initiate a signaling session, but later signaling only introduces 78 bytes of header overhead.

ACM Computing Classification System (CCS):
C.2.0 [General]
C.2.2 [Network Protocols]
C.2.4 [Distributed Systems]
D.0 [General]
D.2 [Software Engineering]
D.2.1 [Requirements/Specification]
## Contents

1 Introduction 1

2 Signaling Protocols 3
   2.1 Quality of Service 3
   2.2 Middlebox Traversal 5
   2.3 Network Monitoring and Management 10
   2.4 Security 12
   2.5 Mobility 14
   2.6 Summary 16

3 Next Steps In Signaling Framework 18
   3.1 Overview 18
   3.2 General Internet Signalling Transport 18
   3.3 Quality of Service Signaling 25
   3.4 Middlebox Signaling 27
   3.5 Summary 30

4 Messaging NSLP 31
   4.1 Idea and Motivation 31
   4.2 Messaging Applications 33
   4.3 Messaging NSLP Messages and Messaging API 33
   4.4 Message Delivery 36
   4.5 Peer Discovery 37
   4.6 Messaging NSLP Message and Object Formats 39
   4.7 Message and Object Processing 40
   4.8 Summary 46

5 Implementation and Testing 47
   5.1 Software Implementations 47
1 Introduction

Various monitoring and signaling needs exist in current complex networks. While performance of routers and other active network devices has been growing, also Internet traffic has grown constantly [UoM09]. Monitoring and signaling needs exist also in non-conventional networks such as rapidly deployed mesh networks and sensor networks. For example, it may be needed to alter routing depending on battery statuses of battery-powered routers [MZY06].

While IP routers normally adhere to best effort policies, some sort of Quality-of-Service (QoS) characteristics have been around for years. Type of service (TOS) bits on the IPv4 datagram header[Pos81, Alm92] were not successful, and the use of these bits was redefined as Differentiated Services (DS) field[B+98]. The use of TOS and DS fields was still based on packet classification on individual routers. Integrated Services (IntServ)[BCS94] meant signaling of QoS needs between routers.

The Internet was originally built on assumption of end-to-end connectivity, any application on any Internet node could set up connection anywhere it wished. Growing number of Internet nodes and users also meant growth in malicious traffic, leading to widespread firewall deployment. Network management reasons, and later also IPv4 address space exhaustion, brought us NATs (Network Address Translators). These middleboxes, i.e. NATs and firewalls, have made end to the global end-to-end connectivity. While some systems and protocols such as Skype\footnote{See http://www.skype.com for more information.} are (notoriously) known for their success of traversing NATs and firewalls, many systems need pre-configuration of middleboxes for connectivity.

It is easy to see also security-related signaling needs. For example, personal firewall software running on desktop PC could signal office level firewall to filter unwanted traffic. After enough filtering requests, these office level firewalls could then signal corporate level firewall to filter malicious stream of data[SCB03], thus preventing it entering into corporate network. Also internal network may gain benefits when using security-related signaling. Oddly-behaving machines caught by network monitoring may be isolated, thus preventing further attacks or data leaks[Mar05].

Next Steps In Signaling (NSIS) working group of Internet Engineering Task Force (IETF) focused on signaling domain, and specified a two-layer software suite for Internet signaling needs. The upper layer, signaling applications, takes care of the given signaling logic such as QoS or middlebox configuration. Below signaling ap-
applications lies the signaling transport layer, that provides transport-related services such as signaling peer discovery and secured transport to applications of the upper layer. This split between applications and signaling transport was made to simplify development and ease deployment of new signaling applications [H+05].

This thesis presents a generic messaging protocol for various network signaling needs. The original goal was to create a monitoring protocol with plug-in capabilities, but it was quickly discovered that restricting only to monitoring scenario would be an artificial limitation. The protocol seemed to be a part of a more generic signaling architecture. The Next Steps In Signaling (NSIS) framework [H+05] was chosen as a building block for the protocol because it fulfills its transport needs including peer discovery service. Using NSIS stack may also ease deployment in the future networks.

NSIS framework and its signaling transport service GIST (General Internet Signalling² Transport) are quite complex to use, even in simple cases. The new messaging protocol forms a new NSLP (NSIS Signaling Layer Protocol). The new NSLP provides a peer discovery service for messaging applications and hides as much networking complexity from them as possible. Also the peer discovery service is hidden from messaging applications that access the messaging service via a simple API.

This thesis goes along with following structure. In Chapter 2 we familiarize ourselves with Internet signaling, including challenges introduced by Network Address Translators (NATs). The NSIS framework that was recently published as a set of RFCs by Internet Engineering Task Force (IETF) is described in chapter 3. Description of the new signaling protocol, Messaging NSLP, and rationale behind it are shown in Chapter 4. Details about Messaging NSLP implementation and test results are given in Chapter 5. Finally, Chapter 6 concludes this thesis with remarks of possible future directions. The full protocol specification of Messaging NSLP is presented as an appendix A. Internet draft *Peering Data for NSIS Signaling Layer Protocols* is presented as an appendix B.

²This thesis is written using American English, but the GIST specification uses British spelling. This leads to mixed use of *signaling* and *signalling*. 
2 Signaling Protocols

Signaling protocols carry the role of supporting actor in telecommunications. Instead of carrying user data, they are used in setting up or controlling links or hardware. Several use cases for signaling protocols exist. For example setting up communication channels, adjusting communication channel or path properties (quality of service signaling), configuring devices used to forward traffic (middlebox signaling) and monitoring behavior of the communication system.

Signaling protocols have been used in setting up communication channels for ages. An example, heavily utilized in POTS (plain old telephone system) world to create virtual circuits for telephone calls, is SS7 (Signaling System 7) [DH04]. It has a counterpart in IP world, namely Session Initiation Protocol (SIP) [R+02]. While these are significant protocols, they are not essential for this thesis. However, other signaling needs including quality of service signaling, middlebox signaling and network monitoring are dealt with greater detail in this chapter.

2.1 Quality of Service

Quality of Service (QoS) signaling means messaging that is utilized to confirm that given QoS requirements are fulfilled in the network. For example, certain bandwidth is reserved for a given data stream.

Several QoS signaling protocols have been specified by different organizations [MF05]:

- **RSVP** resource ReSerVation Protocol. Discussed more below.
- **Tenet** Historic, receiver-initiated QoS protocol.
- **ST-II** Historic, hard-state QoS system with separated data transfer and control protocols.
- **YESSIR** YEt another Sender Session Internet Reservations. Designed after RSVP and uses one-way signaling with multicast support. Works on RTCP (Real-Time Transport Control Protocol) and is a soft-state protocol.
- **Boomerang** A simple QoS protocol with no multicast support. It uses ICMP (Internet Control Message Protocol) as transport protocol.
- **INSIGNIA** In-band quality of service signaling for mobile ad-hoc networks. Uses IP headers to transport QoS information.
Quality of service signaling is not the main topic of this thesis, and thus only the classic example of QoS signaling, RSVP (resource ReSerVation Protocol) is briefly covered. Application for Quality-of-Service signaling in NSIS framework is covered in Section 3.3.

**Resource ReSerVation Protocol**

Traditional best effort forwarding was shown inadequate for real-time applications at the dawn of multimedia services on Internet. Solution for co-existence of real-time and non-real-time services was outlined as IntServ (Integrated Services)[BCS94]. In contrast to better than best effort of DiffServ (Differentiated Services)[B+98], IntServ can give guarantees for QoS due to resource reservations covering the data path.

The heart of IntServ is RSVP which is used for unidirectional resource reservations along the data path. It is intended for both unicast and multicast flows. The host receiving data initiates RSVP session, and routers spread reservation messages toward the flow sender along the reverse data path[Bra97].

Reservations made with RSVP are always related to a specific data flow. The QoS specification for data flow is identified by *flow descriptor*, that consists of *flowspec* and *filterspec*. Flowspec describes the QoS properties of the given flow, and filterspec is used to match packets to the flow.

The two most fundamental message types of RSVP are *Resv* and *Path*. Resv messages are used by receiver in resource reservations, and path messages by sender to discover reverse path for reservations. Even though the RSVP is a soft-state protocol, i.e. states time out unless refreshed, both of these messages have counterparts for tear-down, namely *ResvTear* and *PathTear*. Other RSVP message types consist of error messages *ResvErr* and *PathErr*, and reservation confirmation message *ResvConf*. The confirmation message ResvConf is sent as a reply to a Resv-message carrying explicit request of response. Even when a ResvConf is received, there are no guarantees about the reservation status. Error messages may be sent as replies to respective Path and Resv messages in case of failures.

While the data receiver initiates the actual reservation process, support from the data sender is also needed. The data sender also sends RSVP Path messages toward receiver. These messages contain *router alert* IP option (RAO)[Kat97], and are used to build reverse path for the following RSVP Resv messages. Because of the RAO, each router on the path is notified about the packet even when it is addressed to
the receiver.

Messaging on RSVP is done with its own protocol number over IPv4 or IPv6, but RSVP also supports UDP encapsulation. It does not have a reliability layer on transport, and all RSVP states are soft. This means that any RSVP state that is not refreshed periodically will be automatically torn down. While IP support is needed for every host to handle or forward RSVP messages, all nodes of the data path do not need to support RSVP for successful QoS reservations. Naturally QoS is at best effort level on these non-RSVP hosts.[Kat97]

2.2 Middlebox Traversal

Traditional IP routing is mostly transparent to application protocols. A router receives a packet, makes a routing decision and sends the packet toward end-host via selected route. Middleboxes are devices that do not follow traditional IP routing functionality. They may alter addressing information or payload of IP packets, or even drop them[CB02].

Firewalls form one class of middleboxes. They are used to separate and protect networks. Actions of firewalls are governed by firewall policies. Traditional firewall packet filtering policies are based on source and destination IP addresses and ports. Also connection tracking features are common. For example an end-host initiates a TCP connection, and the firewall allows reply packets from the remote server to reach the initiator even when all other incoming packets are dropped. Firewalls of today may have sophisticated traffic analyzing features, and content-related policies may exist. Many firewall policies are very tight, and may require dynamic firewall configuration to allow legitimate traffic to traverse the firewall.

It is easy to see that firewalls cause disruptions to network connectivity. If traffic does not fit into firewall policy, it may be dropped even if it is legitimate. A classic example of a firewall-related problem is the active mode of File Transfer Protocol (FTP). A client creates the FTP signaling channel by connecting server, and server creates the data channel toward client as a new TCP connection. If the firewall is FTP-unaware, the creation of data channel fails if the firewall is set to drop incoming TCP connections.

Network Address Translators (NATs) form another common class of middleboxes. Several flavors of NATs exist, but the most common are basic NAT and Network Address and Port Translation (NAPT). The basic NAT is a one-to-one translation
of IP addresses, where port numbers are not modified. It may be used for example to provide public addresses at the edge of network for end-hosts that have addresses from private address space. While basic NATs have only limited use (address renaming), NAPT is in very widespread use. Many consumer-grade network devices such as wireless routers or broadband modems provide NAPT service. Fight against IPv4 address space exhaustion is also largely based on NAPT, because it allows several end-hosts to share a single public IP address. A box with NAPT service is also very commonly used as a very simple firewall, because end-hosts behind a NAPT are hidden and inbound connections are not generally possible. As with firewalls, the active mode of FTP does not work with NAPT middleboxes.[SH99]

A basic NAT and NAPT are shown in Figure 1. Host A connects Host C via a basic NAT. The address of Host A is translated, but port information is left intact. Also the port information is changed when the Host B connects to Host C via NAPT.

![Figure 1: Network Address and Port Translator (NAPT) and basic NAT.](image)

Several methods and protocols for middlebox traversal have been proposed. One option is to deploy Application Level Gateways (ALGs)[SH99], but it is not feasible to implement support for every possible protocol for every middlebox type. One option is to use middlebox-aware protocols, or modify existing protocol to support middlebox traversal. Symmetric Response Routing extension of Session Initiation Protocol (SIP) [RS03] is an example of protocol extension for middlebox traversal.

Developing ALG for every protocol or application, or modifying all existing protocols to work with different NAT scenarios is unfeasible. Thus a more generic middlebox
control method is needed. Miscellaneous proposals that try to enable existing and new protocols to work in a middlebox-ridden network include:

**TIST** Obsoleted Topology-Insensitive Service Traversal protocol to control middleboxes. It used RSVP to locate middleboxes from the network. [Sho02].

**MIDCOM** Middlebox communication architecture and framework and Middlebox communications protocol [S+02a, S+02b]. Next Steps In Signaling working group inherited some of midcom ideas (see Section 3.4).

**DNCA** Diameter Nat Control Application [B+12] by dime working group of IETF is based on using Diameter AAA (Authentication, Authorization and Accounting) protocol family to control NATs.

Also NSIS framework includes an application for middlebox signaling. It is dealt in Section 3.4 on page 27.

**Port Control Protocol**

*Port Control Protocol* (PCP) has quite a wide scope in middlebox traversal. It is meant to be used on almost everything from small residential NAT boxes to carrier grade NATs with different mixtures of IP versions [W+13]. Work on the PCP is still under progress in IETF, but the main protocol features are already published.

All PCP messages are meant to be replied. Thus retransmission functionality of TCP is not needed and UDP has been selected as the transport protocol of PCP. While PCP supports both IP versions, it is primarily designed for IPv6, and all addressing is done using 128 bit address fields to keep message formats simple. By using constant address lengths there is no need to include IP version information into PCP messages, but on the other hand, IPv4 addresses must be encoded using IPv4-mapped IPv6 addresses.

If an application uses PCP to create NAT/Firewall mapping to be able to listen for incoming traffic from the network, it also needs to inform the other communicating party about the external address/port combination. Spreading of this addressing information is not in scope of PCP, and must be done by other means such as using some kind of rendez-vous service.
Universal Plug and Play

Universal Plug and Play (UPnP) is a family of protocols to access and control networked devices. It is mainly focused on residential networks and hardware. Enterprise class hardware usually does not support UPnP that lacks for example authentication features. Work on UPnP is governed by UPnP Forum[upn12b] that consists of massive number of hardware vendors[UPn12a].

UPnP uses two device roles, device and controller. A given physical device may represent also of both of these roles, i.e. it can control other devices and be controlled at the same time. The first step to do when a device joins a network is IP address configuration. By UPnP specification, it is mandatory to first try DHCP (Dynamic Host Configuration Protocol) auto-configuration, but if DHCP method fails, the device acquires a link local IP address. After address configuration the device enters to discovery mode, where it starts to either advertise services it provides (if it represents device role) or listen for advertisements (if it is a controller). Next step after discovery is called description, where controller downloads XML-based description of the services and functions the device provides. Successful description phase leads to control phase, where a controller may utilize services of the given device. These control messages are also XML-based, and exchanged using SOAP (Simple Object Access Protocol). The final mode is called eventing. It is a publish/subscribe based scheme where a controller may subscribe to service events published by the device. If a subscribed parameter changes, the device pushes the related data to subscribers.[UPn08]

Internet Gateway Device Protocol (IGD), a part of UPnP framework, provides a middlebox signaling service to UPnP nodes. A NAT/Firewall device implementing IGD is thus UPnP-configurable. For example, applications may add or modify NAT port forwarding rules and firewall holes using IGD.[UPn10]

STUN, TURN and ICE

The original STUN protocol (Simple Traversal of User Datagram Protocol Through Network Address Translators), nowadays commonly referred as classic STUN, was defined as now obsoleted [R+03]. The original design turned out problematic. It did not work for some NAT scenarios and also security concerns were raised[R+08]. Revised version of the STUN is defined in [R+08]. While the acronym was kept unmodified, the long version of protocol name was changed to more accurately describe
the functionality: *Session Traversal Utilities for NAT (STUN)*. It focuses now on
discovery of address and port information of the network endpoint located in the pri-
vate network, i.e. behind a NAT. A separate protocol for the actual NAT traversal,
TURN (Traversal Using Relays around NAT (TURN): Relay Extensions to Session
Traversal Utilities for NAT (STUN) [MMR10], was also introduced. TURN extends
STUN messages, and makes client behind NAT able to exchange UDP datagrams
with its peers, also possibly located behind NATs.

Another protocol, ICE (Interactive Connection Establishment), is also used together
with STUN [Ros10]. It provides a mechanism to select and test address/port combi-
inations to find a working one. A scenario for STUN, TURN and ICE is shown in
Figure 2.

Figure 2: Scenario for TURN (Adapted with modifications from [MMR10]).

A Network Address Translator sits between TURN client and server. The TURN
client with address 10.1.1.2:49721 sends a TURN message to server and the NAT
translates client address to 192.0.2.1:7000. The translated address is called
server-reflexive address, i.e. the address of client that TURN server sees.

The TURN client can create a TURN allocation, i.e. a state on the TURN server
using TURN messages. The allocation contains the server-reflexive address of the
client. Now the server remembers how to reach the client. If the client now wants to
communicate with the Peer B, it encapsulates the payload data into TURN message,
together with the address of Peer B (192.0.2.210:49191), and sends it to TURN
server. Now the TURN server extracts the payload, and sends it to Peer B using relayed transport address 192.0.2.1:50000 as a senders address. Also this address is bound to the allocation created by the client. This way the Peer B can send reply messages to TURN server using the relayed transport address as a destination, allowing the server to capture and resend the payload TURN-encapsulated to the client.

To reach the Peer A that sits behind a NAT of its own, the client needs to address the data to the server-reflexive transport address of the Peer A (192.0.2.150:32102). Naturally there must be an existing TURN allocation created by the Peer A. The TURN specification does not provide means to distribute addressing information, i.e. the client needs to learn the TURN server address and server-reflexive addresses for each peer behind NATs by some other means, such as configuration.[MMR10]

Peer C happens to be located within the same private network with the client. If the client knows the private address of the Peer C, it may be able to communicate directly with it. If the Peer C was located behind the NAT of its own, this direct communication would most likely be impossible, even if the addresses seem to belong into the same network. ICE can be used to gather and check possible addresses, i.e. find a working way for the client to communicate with Peer C. In this scenario, ICE would most likely find that direct addressing between the client and Peer C is possible. If the Peer C happened to be behind its own NAT, ICE discovery for direct addressing would fail, and most likely TURN-based connection would be selected.

STUN-based solutions described above are considering only UDP traffic, but an extension to support TCP has also been specified[PR10].

2.3 Network Monitoring and Management

Management of complex networks requires information on several levels of the network infrastructure. Lowest level consists of physical network infrastructure and network-connected hardware that may fail over time. Information about amount of traffic passing through the network is also valuable to find out possible bottlenecks and to be able to react in time to add capacity if volumes of traffic keep growing. Also other resources and environmental parameters such as battery levels or room temperatures may need monitoring. Network performance and reliability may also be monitored due to Service Level Agreements (SLAs). Network security monitoring is also important for network management. The network may have firewalls or
intrusion detection systems (IDS) on the network perimeter to filter out unwanted traffic. Statistics about filtered traffic may alert administrators about ongoing or prospective attacks.\cite{Aksoy+2002, Kam+2001}

Network monitoring may generate significant amount of traffic on network and processing load for monitored devices \cite{Xia+2004}. Network load may be optimized by collecting and preprocessing monitoring data in relaying network nodes. If data is preprocessed already in the monitoring network, also the CPU load of network management system that collects and analyzes the information may be lowered\cite{Tur+2004}. When monitoring data is collected or even preprocessed on a network node, this must not interfere with the main task of the node\cite{Aksoy+2002}.

While also other alternatives exist, the Simple Network Management Protocol (SNMP) is one of the most dominant network monitoring and management protocols around. Basic SNMP functionality is described below. Messaging NSLP could be used in network management with suitable messaging application. It is covered in Section 4.

**Simple Network Management Protocol**

As the name of the protocol dictates, the scope of SNMP (Simple Network Management Protocol) is network management. To support management functionality also monitoring characteristics are typically needed.

History of SNMP is long. While first SNMP RFCs were published already in 1988, new extensions and specification revisions have been published as lately as in 2011. Age of the protocol is also one reason for many security issues SNMP has seen during years, because security was mainly not an issue when SNMP was originally specified. Thus, the first SNMP version had catastrophic security properties at least when measured by standards of this millennium. All messaging was unencrypted, and authentication was based on plain text password (\textit{community string} in SNMP terminology) until SNMPv3 was published\cite{HPW2002, BW2002, WPM2002}.

Later specifications have targeted these security issues, and a quite complex security model\cite{Glymph+1993} was presented with SNMPv2\cite{Cros+1993}. However, legacy devices did not support the new complex security model, and soon an intermediate form between SNMPv1 and SNMPv2 was published, namely \textit{SNMPv2c}\cite{Cros+1996, Wat+1996}. The SNMP security properties were thoroughly revised finally for SNMPv3.

Today majority of SNMP installations utilize UDP, but also other transport methods such as Appletalk or IPX have been specified\cite{Pand+2002}. An experimental RFC
about SNMP over TCP has also been published [Sch02]. Support for Transport Layer Security (TLS) belongs to one of newest additions to the SNMP specification family [Har11].

Main components of SNMP architecture are managed device, agent that runs on the managed device and manager. These components and the basic SNMP functions are shown in Figure 3. In other words, managed device runs agent software that communicates with network management system via SNMP.

![Figure 3: Basic SNMP Architecture. Managed device runs SNMP agent that sends unsolicited trap messages to SNMP manager. The manager is running on the Network Management System (NMS) node. Agent also sends responses to Set or Get operations initiated by the Manager.](image)

Agent may send SNMP traps that are asynchronous events, for example alerts. It also sends SNMP responses to SetRequest or GetRequest calls initiated by manager. While other message types exists, these four are the fundamental messages of SNMP.

Parameters that are monitored or managed with SNMP (i.e. managed objects) are defined in MIB (Management Information Base) specifications. When first SNMP specifications were published, only a handful of managed objects about IP networking were defined. Nowadays when SNMP has been widely accepted, vast number of MIBs exist, specified both within and outside of the IETF.

### 2.4 Security

Complications of middleboxes were discussed in Section 2.2, and middleboxes were dealt as nothing short of nuisances. However, firewalls are deployed for a good reason, to protect the computing system. Traditional firewall traffic filtering is based on static firewall policies, but the situation has changed, and firewalls may carry advanced traffic analyzing and anomaly detection capabilities.
To take full benefit of traffic analysis and anomaly detection, communication among firewalls or between analyzer and firewalls is needed. If firewalls can communicate with each other, malicious traffic may be stopped earlier (at the network perimeter, or even nearer the source) and network resources may be conserved. Also a concept of distributed firewalls exist, in which traditional approach of single firewall at the network perimeter is changed, and we may deal with networks of several autonomous firewalls or ones that honor centralized policy server[Mer03].

Network Intrusion Detection Systems (IDS) may work together with network firewalling. If the IDS detects malicious traffic deep inside the network, it may inform the nearest firewall about the attack, and the firewall is able to dynamically change firewalling policies and stop the attack[SB99a]. If IDS contacted a firewall that was not at the network perimeter, it may be beneficial to cascade information toward the attack source, thus saving network resources and possibly protecting yet unattacked hosts[SCB03]. The co-operation of firewalls could even reach outside into Internet level. This would make possible to cut attacks near the attack source[SB99b].

A scenario of co-operating firewalls is shown in Figure 4. An organization has two separate networks using Gateways to connect to Internet. These gateways act also as firewalls, and additionally the GW/FW device at the segment 2 of the internal network supports also IDS functionality. The first step of firewall co-operation is when the IDS detects incoming malicious traffic. The IDS initially blocks the traffic, but it sends enough information how to identify the malicious data stream to the FW software that is later used to block the traffic. The second step is to spread attack information to other routers nearby, including informing the firewall of the other segment of the organization’s internal network. Thus, the attack information is signaled from GW/FW2 to node ext_r2 and toward GW/FW1. The node ext_r1 learns about the attack when the information propagates toward GW/FW1. Routers ext_r1 and ext_r2 have also local policies concerning information propagation – the router ext_r2 sends it toward some router outside the top of the Figure while ext_r2 sends the information also to ext_r3. With this system, the attack is blocked before it even had opportunity to reach segment 1 of the organization’s network. The spreading of attack information also makes it possible to block the attack nearer the attack source.

Messaging NSLP and GIST could be used as a platform for a messaging application that could spread attack information into firewalls at the network.
Figure 4: Scenario of co-operating firewalls. Attack is detected at the IDS of segment 2, and information about it is spread to other interested parties. This way the attack is stopped nearer the source.

2.5 Mobility

Location in IP networks is traditionally defined by IP address. When a mobile node moves within a network, it may be able to continue communications using the old IP address. However, when the mobile node moves from one network to another, it needs a new IP address (that corresponds to the new network) and existing connections are lost. Mobile IP (MIP) is a solution that allows the mobile node to move from one network to another while maintaining connections[Per10, PJA11].

In MIP terminology home address means fixed IP address for mobile node that is reached through home agent. When not at the home network, the mobile node uses care-of-address that is reachable through correspondent node. Mobility in foreign network is governed by foreign agent.[Per10]

A simple mobility scenario is shown in Figure 5. A Mobile Node (MN) is connected to network (Net 1) via an access point (AP 1). Several outcomes are possible when the mobile node moves connects to Net 2 via AP 2. It may be possible that all the networks are bridged together to form a single larger network. In this case the node may keep the IP address it has received from the AP 1, and continue communications without any configuration changes. However, if the given networks are separate and
use different addressing, the MN needs to acquire a new IP address. This means also that existing connections are typically lost.

Figure 5: Mobility scenario: A Mobile node (MN) is connected to one network (Net 1) via access point (AP 1) and moves to another net (Net 2, via AP 2). Both of these networks are connected to network 3.

When using MIP, the mobile node acquires a care-of-address from the foreign network. All traffic addressed to home address is tunneled to the mobile node via foreign network address by home agent, thus allowing external connections to MN. Communications originating from the mobile node are tunneled to home agent, and sent toward final destinations using the home address as the sender address. Foreign agent is the tunneling endpoint in foreign network. It may be provided by the foreign network infrastructure, but also the MN may implement foreign agent functionality. With MIP, the mobile node in scenario of Figure 5 could maintain the home address all the time without losing existing connections.

Network mobility is analogous to node mobility, with the difference that a network is attached to foreign network instead of single node. While MIP provides mobility for mobile nodes, it does not support network mobility. However, a network mobility solution NEMO (network mobility) has been proposed as an extension for both MIP versions (Mobile IPv6 and Mobile IPv4)[D+05, L+08].

Proxy MIP (PMIP) is another mobility solution providing network-based mobility. The goal of PMIP is to provide mobility support for unmodified (lacking MIP support) mobile nodes. Two flavors of PMIP have been proposed, one for each IP protocol version: PMIPv4 and PMIPv6. In PMIP the network unloads task of maintaining connectivity from the mobile node by tracking mobile nodes and interacting
with the home agent. [L+10, G+08]

One of the greatest shortcomings of Mobile IP is the hit on performance due to complex routing, where each transmitted packet needs to travel via home agent. This triangular routing is shown in Figure 6. Mobile node is accessing network via foreign agent (FA) and home agent (HA). Node at fixed network (FN 1) is sending data to MN. Even though FN is at network level quite close to MN, all the packages need to visit HA that is at different network.

Figure 6: Triangular routing: A Mobile node (MN) is accessing the network with help of foreign agent at network 2. Home agent (HA) for MN is forwarding traffic from fixed node (FN) to MN.

While NSIS framework and GIST are not a silver bullet for mobility related problems, mobility was taken into account during NSIS specification work [S+11]. Network change detection and adaptation is a fundamental feature of GIST.

2.6 Summary

Traditional philosophy seems to be "one problem, one protocol". Lots of signaling protocols exist, and plenty of duplicate work has been done, at least on signaling transport. Some signaling protocols reuse parts of existing transport systems, but even in these cases some additional customization is needed. Some signaling protocols have even created their own transport mechanism from the scratch. Many protocols fail when middleboxes (NATs, firewalls) are brought in, and middleboxes do not necessarily spare signaling protocols either. To make things worse, many
existing signaling protocols do not work well in mobile environment of today.

NSIS framework seems to provide adequate services for many signaling needs. It has also some degree of mobility support and it is designed to work in networks having middleboxes. Thus NSIS protocols could be used to increase cohesion in signaling systems.
3 Next Steps In Signaling Framework

Plenty of signaling protocols exist, and lots of duplicate work on signaling transport has been done. Thus, development of a common signaling transport layer to benefit also future signaling protocols seems justified. A complete signaling suite designed by IETF that consists of signaling transport layer and two main signaling applications is described in this chapter.

3.1 Overview

Next Steps In Signaling (NSIS) working group of the IETF worked years on a generic signaling framework. The framework consists of two main layers, generic signaling transport and signaling applications[H+05]. The signaling transport layer provides the following main services to signaling applications:

1. Unreliable or reliable transport
2. Discovery of next signaling application level peer
3. Security services such as message confidentiality and integrity

Signaling applications, NSLPs (NSIS Signaling Layer Protocol), work on top of the transport layer. The two initial signaling applications are for quality of service and middlebox signaling[H+05].

A high level NSIS architecture is shown in Figure 7. A signaling application NSLP1 is sending signaling messages through the network. The nodes in the network have heterogeneous software configurations, where only nodes R1 and R5 are running NSLP1. Node R2 is NSIS-unaware, R3 runs different signaling application and while R4 is NSIS-aware, it does not run any NSIS signaling application.

3.2 General Internet Signalling Transport

In NSIS terminology, signaling application protocols are called NSLPs and the signaling transport layer is called NTLP (NSIS Transport Layer Protocol). Specification for a realization of NTLP called GIST (General Internet Signalling Transport)[SH10] was also made by NSIS working group of the IETF. To provide signaling transport service, GIST uses well-known protocols and methods. For example, TCP and UDP
are used for the actual byte transferring and TLS (Transport Layer Security) for protecting message confidentiality.

A basic NSIS stack is shown in Figure 8. Signaling applications lie on top of the GIST that uses TLS or DTLS (Datagram Transport Layer Security) to secure transport protocols such as TCP or UDP. Finally, at the bottom lies the IP layer.

**Transport modes**

Two different transport modes, *D-Mode* and *C-Mode* are used by GIST. D-mode transport is a lightweight, connectionless and unreliable transport mode, that may be used when message size or attributes do not demand using more sophisticated C-mode. Size restriction for D-mode is quite strict, minimum of pair (*known MTU, 576*). C-Mode transport is a connection-oriented transport method, that requires heavier state management, but provides reliable, reusable service. Typical transport protocols used are UDP and TCP for D-Mode and C-Mode respectively. Alternative transport protocols that may be used with GIST include DCCP (Datagram Con-
gestion Control Protocol) and SCTP (Stream Control Transmission Protocol), and both may be used with DTLS (Datagram Transport Layer Security) for securing the transport [Man07, FDC11].

When no previous routing state exists, a special variant of D-Mode, \textit{Q-Mode} is used. Thus, messages sent in Q-mode are sent without confidentiality protection and share message size limit with D-mode messages. If signaling payload does not fit into D-mode size limit, or the message attributes demand using secure communications, the Q-mode message is only used to create routing state, and the actual payload is later transported using C-Mode messages. Combination of service type/transport mode-combinations is shown in Table 1.

Table 1: GIST transport modes. Note that Q-Mode is a special encapsulation for D-Mode messages that is used in session setup phase to discover NSLP-level peers. C-Mode may be used for everything else than session setup. Even small messages that do not have need for reliability or secure transport may use C-Mode. However, NSLP can not mandate using C-Mode in these cases.

<table>
<thead>
<tr>
<th>Service</th>
<th>Q-Mode</th>
<th>D-Mode</th>
<th>C-Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large messages</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Small messages</td>
<td>(X)</td>
<td>X</td>
<td>(X)</td>
</tr>
<tr>
<td>Non-secured</td>
<td>(X)</td>
<td>X</td>
<td>(X)</td>
</tr>
<tr>
<td>Non-reliable</td>
<td>(X)</td>
<td>X</td>
<td>(X)</td>
</tr>
<tr>
<td>Secured</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Reliable</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Peer discovery</td>
<td>X</td>
<td>(X)</td>
<td></td>
</tr>
</tbody>
</table>

A signaling application can not directly control mode selection of GIST. Should the signaling application set signaling message attributes to require reliable or secure transport, GIST will always select C-mode. Respectively, if the signaling application does not require secure or reliable transport, and the message is small enough, GIST may send it in D-mode. However, GIST has freedom to upgrade sending mode from D-mode to C-mode any time, i.e. choose using C-mode also for messages that could be sent using D-mode.
Flows and message routing methods

In GIST terminology data flows are unidirectional. If a node A is sending data to node B, and the node B is responding, the response is considered a separate data flow of other direction. GIST signaling is done either downstream (same direction with the data flow) or upstream (opposite to data flow direction). Flow directions are shown in Figure 9.

Message routing methods (MRM) describe GIST level routing information. Path-coupled MRM is used with signaling messages that follow the data flow, i.e. the signaling messages visit the same network nodes. Loose-end MRM is used with signaling messages that do not necessarily follow data flow. These two MRMs are specified for GIST, but GIST may support also other MRMs.

Multicast support was originally left out from NSIS framework. However, an extension for multicast has been proposed. The multicast extension introduces also a new multicast variant of the path-coupled MRM.

Messages

GIST has six message types, Query, Response, Confirm, Data, Error and MA-Hello. The first three are handshake messages used in setting up routing state. Error messages carry information about error conditions and MA-hello messages are used to refresh messaging associations. While the main message for payload carrying is the Data message, also handshake messages may contain payload.

Query messages are used in creating and refreshing routing states, i.e. peer discovery. They are Q-mode encapsulated D-mode messages, which means that GIST
queries are sent using UDP with parameters suitable for peer discovery. Q-mode encapsulation may include using router alert IP option, or if such configured, just using given well-known GIST port. Query messages may carry NSLP data if the data size is small enough, and has no requirements for security or reliability.

Responses are sent to acknowledge a received query message. They are sent using D-mode (UDP), unless a possibly existing messaging association is reused. In case of messaging association reusing, the response is sent using the given messaging association in C-mode. Also response messages may carry NSLP data. In case of using D-mode responses, data must be small enough, and have no security or reliability requirements. These limitations do not affect responses sent in C-mode.

Confirm messages finalize the three-way handshake of GIST, and they may be sent using C-mode or D-mode, depending on the routing state type. In case of reused messaging association, confirm messages are optional. Confirm messages may carry NSLP data with the same preconditions as with response messages.

Data messages are used to carry NSLP data and they have no effect on routing states. Data messages may be sent using D-mode or C-mode.

Error messages are used to pass information about error conditions. Both the C-mode and D-mode transport may carry error messages.

MA-hello messages are used to refresh messaging associations. They are always sent using C-mode.

Routing states and Messaging Associations

**Routing states** in GIST are identified by triplet MRI (message routing information), SID (session identifier) and NSLPID (NSLP Identifier). Message routing information contains addressing information such as signaling source and destination addresses together with port numbers. Session identifier is an identifier for the given signaling session, and it is created by the signaling application. Finally, applications are identified by NSLPID. Routing state creation is described below in Peer discovery.

Messaging association (MA) describes a communication channel between two GIST nodes. The information stored into MA includes protocol and port numbers, MA state information and any other information the used protocol needs. Messaging associations are not bound to any NSLP-level sessions, and they may be reused. Messaging associations are internal to GIST, thus NSLP has no direct knowledge of used MA.
Peer discovery

Signaling application starts a signaling session by creating a MRI object, filling it with addressing information and finally allocating a new SID. This information is given to GIST together with the signaling payload. GIST sees that it does not have a corresponding routing state, because at least the SID is different when compared to possible existing routing states. Now GIST starts the routing state creation process, or from the viewpoint of the NSLP, the peer discovery process by sending a Q-mode encapsulated message toward the signaling endpoint stated in the MRI.

The GIST peering process is shown in Figure 10. Signaling application \textit{NSLP1} at node \textit{R1} passes message to GIST to be sent toward \textit{NSIS Responder}, i.e. signaling endpoint \textit{Re}. Because there is no existing routing state, GIST sends a query message toward \textit{Re}. Router \textit{R2} is not running GIST, and thus it does not intercept the query message.

![Figure 10: GIST Peering. NSLP at router R1 is messaging toward router Re.](image)

Router \textit{R3} runs GIST together with NSLP1. It intercepts the query message (either the Router Alert IP Option, or port-based interception is used). Now the NSLP at \textit{R3} accepts the peering request from GIST. If the message was small enough, and had no reliability or security requirements, the query already may have carried the message payload. If the payload was present with the query, it is delivered to the NSLP. Because the NSLP at \textit{R3} sees that it is not the signaling endpoint, it passes the message back to GIST, to be sent toward \textit{Re}.

The next router on path (\textit{R4}) does not run the given NSLP, and thus it does not create peering relationship with \textit{R3}. The message is re-injected to the network for \textit{R5}, where handling is identical to one at \textit{R3}. Finally, at the router \textit{Re} the NSLP
sees that it is the signaling endpoint, and does not pass the message any further.

**Security**

As any other protocol, GIST needs to assert its own security properties. Besides its own security\(^3\), GIST provides security related services to NSLPs. When a signaling layer protocol needs secured transport, it sets transfer attribute `secure` to a properer value such as `integrity` or `confidentiality`. GIST takes care of securing the actual communication by, for example, setting up TLS session to the neighboring NSLP level peer. Secured communications implies using C-mode encapsulation, because D-mode is always unsecured.

**Extensions**

Overall description of using and extending the NSIS protocol suite is given in [D+10]. In addition to describing deployment issues, it also covers adding features such as new MRMs, transport protocols, NSLPs or security mechanisms into GIST. A few extensions to NSIS suite have been proposed, and the most interesting ones regarding this thesis are briefly explained below.

Authorization for NSIS Signaling Layer Protocols[M+11] is a method to provide a generic authorization capability to NSLPs by introducing `authorization object`. Another published extension to NSIS framework, a specification for running GIST over Stream Control Transmission Protocol (SCTP) and Datagram TLS[FDC11] is an example of an additional transport mechanism.

A method to encode peering information data into GIST messages was proposed in [M+08]. The sending signaling application may include peering information data into GIST routing state setup messages to ease the responding signaling applications decision whether it wants to peer the sender or not. The University of Helsinki GIST implementation[Var08b] supports peering information data.

**Application Programming Interface**

A high level description of GIST application programming interface (API) is given as an appendix of [SH10]. The API consists of six `primitives`, that are used to send

\(^3\)Several attack scenarios such as man in the middle attacks, replay attacks and denial of service attacks were identified in [TK05], and they are not handled in this thesis.
information between signaling application and GIST.

**SendMessage** is a primitive to send data from a signaling application to GIST to be sent toward signaling endpoint.

**RecvMessage** primitive is sent by GIST to signaling application to inform that data from remote NSLP has arrived.

**MessageStatus** is a notification that may be received from GIST considering an earlier call to SendMessage. For example, about failed message sending.

**NetworkNotification** is received from GIST if something related to the NSLP in network has changed. For example, a NSLP peer has disappeared.

**SetStateLifetime** may be used by the signaling application to inform GIST about lifetime of a given routing state. Signaling application can, for example, state that it is not interested of given routing state anymore.

**InvalidateRoutingState** is a primitive for signaling application to signal GIST that the next NSLP level hop may not be valid anymore. Thus, with this primitive, the signaling application can trigger peer rediscovery process.

### 3.3 Quality of Service Signaling

Quality-of-Service NSIS Signaling Layer Protocol (QoS NSLP) is the QoS solution of NSIS framework. It uses hop-by-hop communications provided by GIST, and is conceptually similar to Resource ReSerVation Protocol (RSVP)[MKM10]. The QoS NSLP is not bound to any specific QoS model, it is used just to transfer QoS information along the data flow path. QoS information is encapsulated into Quality-of-Service Specification objects (QSPEC)[A10a].

QoS NSLP uses GIST with path-coupled MRM to send messages from QNI (QoS NSLP Initiator) – possibly visiting several QNEs (QoS NSLP Entity, i.e. network node supporting QoS NSLP) – toward QNR (QoS NSLP Responder, the last QNE on the signaling path). These QoS NSLP components are shown in Figure 11.

**Message Types**

Four types of messages are defined for QoS NSLP, **RESERVE**, **QUERY**, **RESPONSE** and **NOTIFY**. A brief description of each message is given below.
Figure 11: QoS NSLP Components. Nodes unaware to GIST or QoS NSLP omitted (adapted from [MKM10]).

**RESERVE** is used to modify reservation state. Reservation may be created, altered, refreshed or removed.

**QUERY** may be used to ask details about reservation state or any appropriate parameters. Query does not modify reservation state.

**RESPONSE** is sent as a reply to an earlier reserve or query message. It may be a positive or negative confirmation of reservation state modification request, or an answer to a query message.

**NOTIFY** is similar to response, but sent asynchronously, without request. Error conditions are typically reported using notify messages.

Basic sequence of QoS NSLP messaging is shown in Figure 12. On left, a QNI sends *RESERVE* message toward QNR. The message is captured and resent at two QNEs before reaching the QNR. The receiving node (QNR) sends *RESPONSE* message using the same path reversed that the *RESERVE* was delivered.

The messages above may carry information encapsulated into QoS NSLP objects. Full description for each individual object type and messages they may be carried in, is given in [MKM10]. However, a brief description for some of the most interesting QoS NSLP objects is given below.

**RII** (Request Identification Information) object is used to map a response to a request. If a QoS NSLP node sends a message and needs a reply, it includes a RII object that carries a random 32 bit value as a payload. The responding QoS NSLP node includes the identical RII object into the response it sends, thus allowing the original sender to map the response to the original message.
Figure 12: Basic QoS NSLP reservations. Sender initiated reservation on left, receiver initiated reservation on right (adapted from [MKM10]).

**QSPEC** is a container for QoS model specific information[^10a]. It may be used to inform other QoS nodes needed or available resources, or any other information the given QoS model such as Y.1541[^10b] or RMD[^10] requires.

**INFO-SPEC** objects carry information within response messages relating to earlier query or reserve messages. It may be used to inform the sender of query or reserve about successful or unsuccessful operation. Also occurred error situations may be reported using INFO-SPEC objects.

### 3.4 Middlebox Signaling

Network Address Translators (NATs)[SH99] have been used in mapping addresses from one address space to another for years, and it seems that they are not going to vanish in the near future[WB11]. Address translation has eased network management by providing independence on public addresses. Also stretching the depleting IPv4 address space has been possible with NATs. On the other hand, NATs cause a lot of problems for communicating systems and protocols. A host on the public side of the network is unable to contact the host behind most NAT variants if the NAT does not have explicit support for the given service. Also outgoing connections from the private address space to a host on the public network may be impossible if the used protocol is not *NAT friendly*. Application level gateways (ALG)[SH99] may help for certain cases, but are not an all-inclusive solution.
It is common to use restrictive firewall policies to secure networks. For example, deny all incoming traffic and allow only specific outgoing services. The situation becomes even worse when firewalling policy mandates that firewalls must allow traffic only when the legitimate service is active, and deny traffic at another time.

Next Steps In Signaling working group of IETF has specified NSLP for NAT/Firewall signaling\[S^{+}10\] to make more dynamic middlebox signaling possible. A simple scenario with two middleboxes (adapted from \[S^{+}10\]) is shown in Figure 13. It contains two private networks that are connected to a public network via middleboxes. In this scenario direct communication between private networks could be impossible without third party (rendezvous service) or active middlebox management.

Figure 13: A simple NAT/FW NSLP Scenario with two middleboxes (adapted from \[S^{+}10\]).

NAT/Firewall NSLP design leads to a dynamic configuration of middleboxes. This configuration signaling is done on the data path to find middleboxes that may need reconfiguration. As with other NSLPs, NAT/Firewall NSLP uses GIST as a transport protocol. With the help of GIST, NAT/Firewall NSLP nodes are found at the network, and NAT or firewall rules may be set at each middlebox. Besides allowing traffic, NAT/Firewall NSLP may be also used to set restrictive rules to deny unwanted traffic.

NAT/Firewall NSLP uses four message types to create and manage signaling sessions, CREATE, EXTERNAL, NOTIFY and RESPONSE. A node (NI, NSIS Initiator) initiates the communication using CREATE or EXTERNAL messages. The network may have any number of forwarding nodes (NF, NSIS Forwarder) that may adjust their configuration based on the message payload, and forward the message toward the responding node (NR, NSIS Responder). When the NR receives CREATE or EXTERNAL message, it responds to the previous NF (or to NI if no NFs were on the path) by sending RESPONSE message. This flow of NAT/Firewall messages is shown in Figure 14.
A quite simple example of NAT/Firewall operations could be one, where an application on the NI node (shown earlier in Figure 13) wants to communicate with another application running on node NR. Now the NAT/Firewall NSLP at node NI sends a message toward NR, and it is first captured at MB1. Local configuration changes may be done as needed on the node MB1, and the MB1 forwards the message toward NR. Eventually the message has traveled through the network, and reaches MB2, where GIST and NAT/Firewall NSLP capture it. Message payload tells the NAT/Firewall NSLP at MB2 what kind of configuration is needed, configuration is altered and message forwarded to NR. Now NR sends the response back toward NI, and finally when the response is received, the application at NI may communicate directly with NR.

While NAT/Firewall NSLP may help in NAT traversal, it is not a silver bullet solving all the NAT-related problems. Should we have a middlebox scenario of enough complexity, such as the one shown in Figure 15, the NAT/FW NSLP alone may not be able to sort things out, and it may be a need help also from other systems such as Domain Name Service (DNS)[S+10].

Both the NI and NR reside in the same private network behind the MB1. However, they also happen to have a private networks of their own, created by MB2 and MB3. Now the NI has no means to communicate to NR directly without external help, because the NI can not know the real address of NR, and the inner private address spaces may also overlap. Even in this scenario adapted from [S+10], the NAT/Firewall NSLP may see some use, assuming the network hosting also firewalls needing configuration.
Figure 15: A scenario where NAT/FW NSLP alone cannot solve problems caused by multiple middleboxes. (adapted from [S+10]).

3.5 Summary

Next Steps In Signaling framework provides signaling applications transport service with peer discovery and two different transport modes. Lightweight D-mode is suitable for applications that do not require reliable or secure transport service, while more robust C-mode provides reliable signaling transport including optional protection for message integrity and confidentiality. The transport service also detects and informs NSLP of changes in the network.

Signaling applications use the transport service via NSLP API. The API as the whole system is somewhat complex, and thus suitable for demanding applications such as existing QoS and NAT/FW NSLPs, but complexity is a burden for simpler applications. It is however possible to implement an abstraction layer on NSLP level to provide a simple transport service for simple applications.
4 Messaging NSLP

This chapter presents Messaging NSLP, a new messaging-oriented signaling layer application. It provides peer locating, addressing and message transport service to messaging applications using GIST for its own transport needs.

4.1 Idea and Motivation

The Messaging NSLP itself is a new signaling application of NSIS framework. To be more specific, the new NSLP uses GIST to find Messaging NSLP neighbors and to transport messages to and from them. The NSIS stack with Messaging NSLP and Messaging Applications is shown in Figure 16, and a more formal specification of the Messaging NSLP is presented as Appendix A.

![Figure 16: NSIS Protocol Stack with Messaging NSLP and Messaging Applications](image_url)

Traditional internet signaling protocols are, in addition to routing protocols, mainly focused on Quality of Service [Bru04]. Also other signaling purposes exists, for example resource balancing (battery, bandwidth, CPU load) of routers. One specific case where it is important to spread state information between routers is routing in mesh networks. With information of nearby routers we can adjust routing and gain efficiency and reliability benefits when taking account the very limited resources of mesh routers.

A design of a new messaging protocol which is to be used to transfer generic data between signaling nodes is presented in this thesis. The protocol will use the existing NTLP (NSIS Transport Layer Protocol), GIST, as the transport mechanism. Also
other NSIS concepts such as authorization and peering information objects [M+11, M+08] are considered.

While the first goal of the Messaging NSLP is to transfer measurement/monitoring data, the new NSLP is to be generic enough to support also future needs in generic lightweight signaling needs. The protocol is built to be extensible, thus allowing flexibility over different scenarios, not limiting to router-to-router monitoring data distribution.

Besides monitoring data transport and resource availability signaling, also other possible use scenarios have been considered. One additional use could be distribution of data about network attacks or malicious traffic. For example, security related services such as attack detection and isolation could make use of the new NSLP [Mar05]. When attacks or anomalies are detected on the network, the hosts detecting the anomaly could signal routers and firewalls to block the malicious data already on the edge of the local network [SCB03]. On the other hand, it would also be possible to not to limit to the local network, and we could try to isolate the attacker by propagating the information nearer the attack source [SB99a, SB99b].

Background of Messaging NSLP is at delivering monitoring data. While it is now general purpose messaging solution, monitoring applications are still important. One simple example of monitoring application could be a distributed network traffic monitor application. Simple incarnation of network traffic application could be a system that consists of probe nodes that report traffic numbers to other traffic monitoring nodes. These sink nodes can have multiple roles, sink for one data flow direction and probe for another. This way monitoring nodes could gather information about network use, where are the bottlenecks and what nodes cause the most traffic. This kind of monitoring application was demonstrated during ABI (Algorithms for Broadband Infrastructure) project where work for this messaging system began.

While the protocol itself is kept very simple, it is meant to be flexible by independent applications. In terms of messaging NSLP, a new application is created by allocating new application identifier (see Section 4.2). Modifications to the NSLP itself are not needed.
4.2 Messaging Applications

While GIST is handling the actual sending and receiving of messages, it can not understand the content of Messaging NSLP messages. When GIST receives a message with NSLP identifier matching the Messaging NSLP, it is passed to the Messaging NSLP process. Further, the Messaging NSLP does not understand content of Messaging Application data. When Messaging NSLP receives a message from GIST, it just forwards the message to Messaging Application with corresponding Application identifier.

There can be several different applications that use the messaging service provided by the Messaging NSLP. Each application has a unique application identifier that is used by the Messaging NSLP to find proper recipient application for each message received.

Different data streams are identified by GIST session identifier (SID). Applications do not select the SID themselves, it is provided by Messaging NSLP. Session identifier use of Messaging NSLP is quite liberal, at least when considering GIST specification. While multiplexing different applications under single session identifier would be possible, it would also mean a significant complexity increase for Messaging NSLP.

4.3 Messaging NSLP Messages and Messaging API

The Messaging NSLP provides a Messaging API for application developers. An application may communicate with remote applications via this API. Internally the Messaging NSLP uses three message types, request, notification and error. Use of Messaging API and these three messages are described below.

Main Message Types: Request and Notification

The basic message type Messaging NSLP uses is notification. If an application needs to send data without need of reply, it uses a notification message. If the application wants a reply, it should use a request message. If application receives a request message, it should send a notification message as a reply. Application may of course send notification messages even if it has received only notifications from the remote peer, but in this case these messages have no Messaging NSLP level relation. Two basic messaging scenarios are shown in Figure 17.
Figure 17: Two messaging scenarios: Notify with and without request.

Error

Most error conditions that may occur in Messaging NSLP are of such type that it is very hard or even impossible to recover. There still are some error conditions and exceptions that are recoverable. Also with some error types it may be beneficial to inform the other communicating party about the error condition. For these purposes there is message type error included into the Messaging NSLP. Information about the error condition is encoded into a error object included into a error message.

Error class that indicates error severity and error code that identifies the error within error class are always carried in error messages. Error source identifier that identifies the sender of error message and Error-specific information such as sender-provided error message are optional.

Quick Ack

It is easy to discover scenarios where fulfilling a request may take significant amount of time or other resources. For example, a monitoring application could ask network link load statistics of next 24 hours from another host. Naturally answering this kind of request takes a long time. The Messaging NSLP has a feature designed to inform the query sender about received request. This mechanism is called a quick ack, and it may be sent as a response to a request. The query sender may also explicitly request the quick ack within the request message.

Quick acks are notification messages, and they are formed by Messaging NSLP. Main payload of a quick ack is the copy of Request Identification Information object (RII)
carried in the request.

**Messaging NSLP API**

The interface toward applications is very simple with no more than four operations, `open()`, `read()`, `write()` and `close()`. All data transfer between application and Messaging NSLP is done with these operations.

**Open()** is used by application to acquire a SID to use with `write()` operations. Application needs to first call `open()` to be able to send data to new hosts. If application has received a message from network with `read()` operation, it already has an identifier for active session, and call for `open()` operation is not needed.

**Read()** is used by application to transfer data from the Messaging NSLP process. On `read()` operation the Messaging NSLP stores application data into a buffer given with the `read()` request. The Messaging NSLP also provides the session identifier and the type of received message. Request Identification Information (RII) object is also provided if the message received was a request or a response for a request sent earlier by this Messaging Application. If a RII is not included, the Messaging sets the RII value to 0. `Read()` operation returns the amount of bytes read and a negative value on error.

**Write()** is used by application to transfer data to the Messaging NSLP. In addition to the actual message data, session identifier and type of message are provided as a parameter. If the Messaging Application wishes to send a request message, it creates a 32bit, session-wide unique RII value. If the application has received a request and wishes to respond, it includes the the RII provided with the request. If the application is sending a notification not related to any request, it uses a zeroed RII value.

**Close()** is used by application to signal the Messaging NSLP that no more data is going to be written or read within the corresponding session. When application decides that it does not need to use an existing session anymore, it should call `close()` operation. A call to `close()` operation signals the Messaging NSLP that it is allowed to tear down corresponding session. Close operation allows Messaging NSLP also to invalidate GIST routing states that relate to the expired session.
4.4 Message Delivery

While Messaging NSLP provides Messaging API for messaging applications, GIST provides transport service for NSIS signaling applications, including Messaging NSLP, via GIST NSLP API. A brief description of GIST NSLP API is presented below, and a more detailed explanation is given as Appendix B of [SH10].

Outgoing signaling application data is passed to GIST via `SendMessage(..)` call of GIST NSLP API. Together with NSLP data, also other pieces of important information such as peering information data, NSLP identifier, and session identifier are passed to GIST. Messaging application does not need to worry anything else on GIST level than providing session id to the messaging API call when sending data out. Messaging NSLP takes care filling the rest of the parameters. Messaging NSLP encodes all the messaging application data as NSLP Data.

Incoming data is received from GIST with `RecvMessage(..)` call. Again, the Messaging NSLP parses messaging application data from the received NSLP Data object, and provides it to the given messaging application.

GIST signals NSLPs about routing state changes via NSLP API primitive `NetworkNotification(..)`. Messaging NSLP takes care of these changes, and terminates affected sessions. Messaging applications do not see these notifications, they just need to react correctly if a session is terminated.

Another NSLP API primitive that Messaging NSLP needs to handle, is `MessageStatus(..)`. Should the GIST be unable to send a given message, it informs the sending NSLP with MessageStatus primitive. This case is reported to the messaging application by terminating corresponding messaging session. Messaging NSLP hides other, non-fatal MessageStatus cases such as changed security parameters from the messaging applications.

Messaging NSLP also uses primitives `SetStateLifetime(..)` and `InvalidateRoutingState(..)` while communicating with GIST, but these are not directly visible to messaging applications. In other words, actions of messaging application may cause Messaging NSLP to send these primitives to GIST, but Messaging NSLP hides details from applications.
4.5 Peer Discovery

GIST level peer discovery is explained in Section 3.2. While GIST takes care of finding NSLP level peers on the network (see Figure 10 on page 23), Messaging NSLP is solely responsible for messaging application level peer discovery. This application level peer discovery is explained below.

When GIST has received a message from network and the message seems to be carrying NSLP Identifier of Messaging NSLP, it asks Messaging NSLP whether it wants to make a peering relationship with the sending Messaging NSLP node or not. In an easy case both the sending and receiving GIST nodes support Peering Information Objects (PIO), and the Messaging NSLP has the knowledge of sending messaging application (because the application identifier is given as a PIO payload). If, for some reason, the PIO data is not available, the Messaging NSLP needs to make the peering relationship, and be ready to forward the messages of the new session to the next suitable NSLP.

Thus, only when the PIO data is available, the receiving Messaging NSLP node is able to deny peering requests if it does not support the given messaging application. For all the other cases some sort of state creation is needed. Possible choices are:

1. Messaging application is locally supported: The data is passed to the application

2. Messaging application is not locally supported: A forwarding session is set up, and the message is forwarded toward the destination.

3. Messaging application is not locally supported, and this node is the destination.

A scenario of multiple network nodes that have different messaging application configurations is shown in Figure 18. All nodes are running NSIS Transport Layer Protocol GIST and Messaging NSLP. For comparison, a lower level GIST peering scenario is shown earlier in Figure 10. All nodes, excluding node $R3$ have support for peering information objects. The application $APP1$ has initiated messaging from node $R1$ toward the endpoint at node $Rn$. Detailed description of peering decisions and what kind of state changes have been made is shown below.

$R1$ is a node running messaging application $APP1$. The application initiated peer discovery on messaging application level by sending a message toward signaling endpoint, node $Rn$. 
**R2** runs only messaging application *APP2*. When NTLP level asked the Messaging NSLP whether it wants to create a peering relationship or not, the Messaging NSLP sees from the PIO data that it does not host the appropriate application. Thus, the Messaging NSLP denies peering, and the NTLP level query is forwarded to node *R3*.

**R3** does not have PIO support. Thus, the NTLP on node *R3* sends an error message back to the node *R2*, and eventually (maybe after some retries) the node *R2* moves to fallback mode, and sends the message without PIO. Now Node *R3* is able to handle the query, but the Messaging NSLP has no means to know whether the message has any value to it or not. Now, the only possible action for the Messaging NSLP on node *R3* is to create the peering relationship. After the NTLP level handshakes are ready, the Messaging NSLP on node *R3* receives the data payload, and sees that the data was addressed to unsupported application. Thus the Messaging NSLP modifies the session to be in a *forwarding* state, and sends the data toward node *R4*.

**R4** has PIO support, but because *R3* has not, also the *R4* needs to create a forwarding state for this messaging session. It forwards the data to node *R5*, now including the PIO information.

**R5** sees from the PIO information that it supports the given application, and creates a peering relationship with node *R4*. It also forwards the query toward the messaging endpoint *Rn*.

**Rn-1** is the second to last node before signaling endpoint. It happens to support the given application, creates the peering relationship with the previous node

---

**Figure 18:** Messaging NSLP Peering decisions done while having a heterogeneous set of messaging applications.
and forwards the message toward the endpoint $R_n$ just like any other node on the messaging path.

$R_n$ is the endpoint of messaging session initiated by $R_1$. However, the $R_n$ does not support the application $APP_1$. Thus, it sends an error message of class permanent failure with code *Endpoint missing application* to $R_{n-1}$. Now $R_{n-1}$ learns that it is actually the endpoint, and does not need to send any messages on this session to node $R_n$. Alternatively, should the node $R_n$ actually support $APP_1$, it would just create the peering relationship with $R_{n-1}$ and stop there because it sees that it is the messaging session endpoint.

### 4.6 Messaging NSLP Message and Object Formats

Each Messaging NSLP message begins with a header that carries message type and flags. Also an identifier for the messaging application is included. One detail the Messaging NSLP needs to take care of, is the Session Identifier (SID) carried within GIST headers. Session identifiers are used to identify GIST-level sessions, and they must be selected in such way that they can not be guessed by a hypothetical attacker.

The basic structure of our NSLP message can be seen in Figure 19. The Messaging NSLP header is followed by a set of objects: *RII, Authorization, Application Data, Error*. Error objects may be carried only within *error* messages, other objects have no such limitations.

Besides the objects of its own, also objects of other origins may be used with the Messaging NSLP. For example, many messaging applications may deal with sensitive data or there may be other needs to make sure that all participants are authorized to communicate. While GIST provides us some degree of security, there is also need for more fine-grained authorization within Messaging NSLP and messaging applications. Authorization extension for NSIS signaling layer protocols is presented in [M+11]. This authorization mechanism uses AUTH_SESSION object which may be used as specified in [M+11].

Disposition of objects in Messaging NSLP messages is not free. The order of objects and what object types may be carried within certain message types is shown in Table 2. For example, *request* messages need to have *RII* and *APP_DATA* objects, and *notification* messages may carry anything but *ERR* objects.
4.7 Message and Object Processing

The Messaging NSLP message and object processing is described in this section. First we assume that we have an extended implementation of GIST specification that supports Peering information objects (see Section 4.5 and Appendix A.4.4) to be able to make peering decisions based on Application Identifiers. The alternate mode of message delivery (hop-by-hop) if such information is not available, is also shown.
Receiving Messages

Message receiving state machine is shown in Figure 20. The process begins when GIST receives a message from network, and the Messaging NSLP calls `RecvMessage(..)` API call. Next the Messaging NSLP checks whether the message is meant for this Messaging NSLP host or not by extracting the Application Identifier information from the possible Peering information object carried within received message. Application information may not always be present in queries, if the node that sent the request does not support Peering information objects. It is also checked whether this is a GIST Query or not by checking the `Routing-State-Check` value got from `RecvMessage` API call. If the message was not a query, then we should have knowledge of application identifier (it is carried within Messaging NSLP message header) and we already know that we have a corresponding application.

![Figure 20: State machine: Receiving messages.](image)

After these bits of information have been extracted, the Messaging NSLP makes the decision what to do with the message. There are four alternatives:

1. If the received message was a GIST Query and this Messaging NSLP host runs the corresponding Messaging Application, then it is known that the message is creating a new connection and the peering process is continued. This Messaging NSLP host must continue the peering process also if the query did not
carry Peering information object, because sender of the received query may not support PIOs.

2. If the received message was a GIST Query for some non-locally supported Messaging Application, the Query is just forwarded.

3. If the received message was not a GIST Query, and this Messaging NSLP host runs the corresponding Messaging Application, the message is passed to the application in question.

4. If the received message was not a GIST query, and the corresponding application is not available, this message relates to a flow being forwarded hop-by-hop. The message is just passed back to GIST level, toward the next hop.

Opening New Connection

Peering process is completely transparent to the sending Messaging Application supposing that the Messaging Application in question has called \textit{Open(\ldots)} operation of Messaging NSLP API.

A call to \textit{Open(\ldots)} operation by a Messaging Application is an order to the Messaging NSLP to allocate new Session Identifier for the new connection. When the new SID has been allocated, the Messaging NSLP calls \textit{SendMessage(\ldots)} operation of the GIST NSLP API. This is done to find the next application level hop at the network with help of GIST.

The application identifier is included into the \textit{SendMessage(\ldots)} call as Peering Information Data (see Section 4.4). The Messaging NSLP must be prepared for case where the next hop does not support PIO. This case is indicated by GIST level error message. If Messaging NSLP receives error message that indicates lack of PIO support, it needs to repeat \textit{SendMessage(\ldots)} call, but this time omitting PIO data.

Parallel to the \textit{SendMessage(\ldots)} call the Messaging NSLP returns the SID of the new connection to the Messaging Application. Now the Messaging Application can use \textit{write(\ldots)} operation with the new SID and start the actual messaging with a remote application.
Sending Notifications

Send process of notification messages is shown in Figure 21. Here state Idle means that the Messaging NSLP is not processing the message in question at the moment. We assume that the messaging application has already called operation `open()` and thus it has knowledge of a session the message it is sending will be related to.

The application calls `write()` operation and application data is eventually sent to the network using `SendMessage()` GIST API call. Send process ends when `write()` operation returns and return value is delivered to the application.

![Figure 21: Notification message processing (send)](image)

Receiving Notifications

Quite simple receive process of notification messages is shown in Figure 22. Notification message is received from the network by GIST, and delivered to the Messaging NSLP via `RecvMessage()` operation. Application can read the data from the Messaging NSLP buffers by calling `read()` operation. Together with application specific data application receives the identifier of session that can later be used with `write()` operations without first calling `open()`.

Note that Figure 22 is meant to be examined together with Figure 20. These state machines have a common state, namely `Incoming message to local application`.

Sending Requests

Again we assume that the messaging application has an open session to be used. The request sending process (see Figure 23) starts when application calls `write()` operation of Messaging NSLP API. A RII object has to be constructed and included with the application data into the newly created request message. After the request
message is constructed and sent to the network using SendMessage() operation of GIST API, the process moves to waiting state. There are two possible responses: If long request processing is predicted, the receiving Messaging NSLP may have sent a response as a quick ack. The second alternative is that the application has processed the request and actual response has arrived. Both responses are read from GIST by using the RecvMessage() GIST API call. Request processing ends when the sender application eventually gets the response (real response or quick ack).
Receiving Requests

Receive process of request messages is shown in Figure 24. This is quite a lot more complex compared to notification receiving process because of response handling.

The process starts when a request message is received via the GIST API. When the message is buffered, it is checked if the sender has requested a quick ack to be sent. If a quick ack was requested a notification message is constructed. The notification/quick ack message includes the RII object received within the request.

After the possible quick ack is sent using the GIST API, the message data of request is passed to the application via read() operation of Messaging API. Eventually the application has processed the request, and it can call write() operation to send the response. Response sending is done just like sending of normal notifications.

Figure 24: Request message processing (receive)

Error Message Processing

Most errors are network related, and tearing down local state and possible logging the error condition are the methods to recover. These conditions are interpreted as permanent errors that should not trigger sending of any error messages. In case of network errors, for example partitioned network, it is often impossible to reach the
peer with error messages.

Normally error conditions are transparent to messaging applications. Indirect information about error conditions is however often given to applications by tearing down sessions.

The case when signaling endpoint does not have required application may be the most common error type for Messaging NSLP. In this case the Messaging NSLP at endpoint replies the sender with an error message. When this *permanent error* with code *Endpoint missing application* is received, the session state is transparently set to be a non-forwarded one. Messaging application does not receive any indication about this error.

## 4.8 Summary

Several scenarios for simple signaling applications exist. It would be huge task to create signaling transport protocol for each of these applications. Even using GIST as transport mechanism is somewhat cumbersome, not least due to heavy standardization effort required to allocate NSLP identifier.

Messaging NSLP provides a simple transport service to signaling applications. It refines peer discovery service of GIST and provides application level peer discovery service to messaging applications. Messaging NSLP services are used via a simple API. Four operations are provided to application developers: *open()*, *close()*, *write()* and *read()*.

The Messaging NSLP does not take active role in maintaining connections if the network below is dynamic. This means that Messaging sessions may be torn down at any time, and Messaging NSLP does not attempt reconnecting these sessions. The Messaging Application receives a notification that a session it has been using is not anymore active, but the possible reconnection attempt is left to be responsibility of the Messaging Application. This restriction is done to simplify Messaging NSLP. Without this simplification, the Messaging NSLP would have been basically *GIST on GIST*. While transferring some session management responsibility from Messaging NSLP to Messaging Applications, this restriction also helps keeping Messaging API as simple as possible.
5 Implementation and Testing

Software implementations of GIST and Messaging NSLP are described in this chapter. Functional testing including scenarios such as one shown in Figure 18 was done during implementation work. Computers used in testing include virtualized PCs, laptop PCs and small form-factor Alix systems\textsuperscript{4}. Processing times and network latencies were not tested due to heavy dependency of specific implementation. Overhead analysis, test cases and respective results are however covered.

5.1 Software Implementations

University of Helsinki GIST (UHGIST) implementation used to test Messaging NSLP was created by Nuutti Varis. The UHGIST implementation consists of over 20,000 lines of C code (C99) and it is freely available\cite{Var08b}. The UHGIST conforms to GIST draft specification of September 2008 with some limitations. For example, peer authorization and NAT traversal were not implemented\cite{Var08a}. However, the core GIST was implemented and provided stable test platform for Messaging NSLP development. The UHGIST contains also support for path-coupled multicast message routing method and Peering Information Objects (PIO).

Application programming interface for NSLPs to be run on top of UHGIST was implemented by the author. It consists of about 4600 lines of C (C99) code. As with the core UHGIST implementation, the only external library used is the (optional) OpenSSL.

Messaging NSLP was implemented by the author, and it consists of about 7500 lines of C (C99) code. Only NSLP API library and standard C libraries were used. The implementation supports use of proposed extension to the GIST, Peering Information Objects. However, the use of PIOs is optional to support also GIST implementations that lack support of non-standard PIO-extension.

5.2 Overhead Analysis and Testing

At least some overhead, i.e. additional transferred bytes, additional use of memory and additional CPU processing cycles, is always present when processing and trans-\textsuperscript{4}Oracle VirtualBox was the chosen virtualization platform. For more information about Alix boards, see http://www.pcengines.ch/alix.htm
ferring application payloads. It is good to have at least some degree of knowledge of overhead even while Messaging NSLP is not meant to be used in scenarios needing real-time signaling or bulk data transfer.

It is very hard to get exact numbers for on-wire overhead on complex system such as combination of GIST and Messaging NSLP. Overhead is generated on multiple layers: IP, UDP/TCP, (D)TLS, GIST, Messaging NSLP and finally Messaging NSLP applications all add their part of overhead in form of different headers and padding bytes. Rough understanding of overhead figures may be acquired by examining most frequently used object formats of each protocol used.

**Messaging NSLPs Memory and CPU Consumption**

Inspecting memory and CPU use of implementations is even more difficult than analyzing on-wire overhead. Both the Messaging NSLP and the GIST utilize heavily threads which makes inspecting CPU and memory use even harder.

Different profiling tools exist to get details on CPU usage. One of easiest to use is `time`, a small simple application to examine resource use of a given software. By default, its output consists of three values:

**real** “wall clock” time the process was running.

**user** time the process spent in user-space.

**sys** time the process spent in kernel code.

If we take the sum of **user** and **sys**, we get a somewhat useful metric of total CPU consumption of this process, including its children. At least GNU time can also show other statistics such as maximum used memory and number of context switches. Also debugging tool Valgrind\(^5\) provides us some metrics of memory use, even on these multi-threaded applications.

Test cases shown below were also run using `time` tool, but it seems that these test cases were just too simple for the measurement precision of `time`. Significantly larger and more complex test cases would be needed to get meaningful results. With the test cases below, **user** and **sys** values were most times both 0.000s, with some random occurrences of 0.001s. No further CPU consumption tests were ran.

\(^5\)See http://www.valgrind.org for more information.
Test Cases

Memory use of Messing NSLP and on-wire overhead of NSIS suite was inspected with a custom simple messaging application. It is tailored to send test payloads of different sizes from one Messaging NSLP node to a user-defined receiver node.

The following test cases for memory analysis were successfully ran using the test application:

1. Start the Messaging NSLP, let it run one minute and shut it down.
2. Start the Messaging NSLP and the messaging application, shut them down.
3. Start the Messaging NSLP and then start and shut down the messaging application ten times, then shut down the NSLP.
4. Start the Messaging NSLP and the messaging application, send one small message using a D-Mode session.
5. Start the Messaging NSLP and the messaging application, send ten small messages using individual D-Mode sessions.

The following test cases for on-wire overhead were successfully ran using the test application:

1. Transfer a four-byte word of payload in D-Mode.
2. Transfer a four-byte word of payload in C-Mode.
3. Transfer a four-byte word of payload five times in D-Mode.
4. Transfer a four-byte word of payload five times in C-Mode.
5. Transfer just over 1000 bytes of payload, requesting D-Mode transport, which is automatically elevated to C-Mode by GIST.
6. Transfer just over 1000 bytes of payload five times using C-Mode transport.
Results of Memory Tests

Memory consumption was measured with Valgrind, by checking total number of heap allocations made. Use of stack was not inspected. However, it is worth mentioning that Messaging NSLP implementation uses several threads, and for example on Debian GNU/Linux operating system each thread has 8 megabyte of stack space by default, i.e. over 50 megabytes of memory is reserved for the NSLP.

Results for heap memory consumption tests are shown in Table 3. First rows specify memory allocations in static cases where no messages are sent. At the top is the case where no applications are present, i.e. the Messaging NSLP is started and left running for one minute and then gracefully shut down by sending it a SIGINT signal. The next rows show number of bytes allocated when one or ten applications registered and exited before shutting the NSLP down. The last two rows have separate columns for the node sending messages and for the node receiving them. Each message was sent using individual session identifier to get the maximized memory use. The test results show that about 20 kilobytes of heap allocations were made for each registered application at the Messaging NSLP. At the receiving end each message caused a bit over one kilobyte per message (i.e. per session) of heap allocations. The first message received took a bit larger share because the implementation initializes its receive buffers when needed. Heap memory use at sending node is not constant, and it fluctuates a bit. This is due to buffered reads to /dev/urandom which is used to generate session identifiers. A rough figure of bytes needed to create a session and to send a message using it is about 1400 bytes.

<table>
<thead>
<tr>
<th>Task</th>
<th>Sending node</th>
<th>Receiving node</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSLP without application</td>
<td>1326 bytes</td>
<td></td>
</tr>
<tr>
<td>NSLP with application</td>
<td>22278 bytes</td>
<td></td>
</tr>
<tr>
<td>NSLP with 10 applications</td>
<td>210846 bytes</td>
<td></td>
</tr>
<tr>
<td>1 message</td>
<td>24002 bytes</td>
<td>23713 bytes</td>
</tr>
<tr>
<td>10 messages</td>
<td>42833 bytes</td>
<td>34881 bytes</td>
</tr>
</tbody>
</table>

Table 3: Messaging NSLP total heap memory use in different scenarios.
Results for On-wire Overhead Analysis and Tests

While use of different resources may be studied, CPU utilization and memory use are firmly tied to specific implementations. Transport overhead is, while not necessarily equal, at least similar across different implementations. Thus, the overhead analysis in this thesis focuses on on-wire overhead.

Overhead of GIST has already been studied \cite{FST+06, FST+09}. The following tables are based on these studies, and values were confirmed in UHGIST by analyzing captured test traffic. Results of these earlier studies are mostly compatible with UHGIST implementation, with the exception of UHGIST’s support for peering information objects. Also, MRI objects do not carry optional port information in UHGIST. Thus, UHGIST’s IPv4 MRI object is 4 bytes smaller than the one used in \cite{FST+06}.

Overhead below GIST layer is shown in Table 4. Minimal IP header size is 20 bytes for IPv4 and 40 bytes for IPv6. GIST Q-mode messages may optionally use IP Router Alert Option (RAO) that adds 4 bytes with IPv4 and 8 bytes with IPv6. Support for using RAO is implemented in UHGIST, but it is turned off by default. Thus, it is not included in the results below. On top of IP overhead comes transport protocol overhead, which is 8 bytes when using UDP and at least 20 bytes for TCP. These numbers are for single UDP datagram or TCP packet with minimal header and multiple transmissions are needed in GIST handshake process. Note that possible (D)TLS overhead on secured communications and overhead from protocols below IP are omitted.

<table>
<thead>
<tr>
<th></th>
<th>IPv4</th>
<th>IPv6</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td>20 bytes</td>
<td>40 bytes</td>
</tr>
<tr>
<td>RAO</td>
<td>4 bytes</td>
<td>8 bytes</td>
</tr>
<tr>
<td>UDP</td>
<td>8 bytes</td>
<td>8 bytes</td>
</tr>
<tr>
<td>TCP</td>
<td>20 bytes</td>
<td>20 bytes</td>
</tr>
</tbody>
</table>

Overhead of GIST Query Message is shown in Table 5. Query messages carry four-byte magic number before GIST common header. Object for message routing information is larger with IPv6 than its IPv4 counterpart due to larger IP addresses. UHGIST uses four bytes smaller MRI size (i.e. 20 bytes) than is shown in this and following tables. Sizes of session identifier and query cookie objects do not
differ depending on IP version. Network layer information object carries IP address information, and is again larger on IPv6. Stack proposal and stack configuration data objects are included when setting up C-mode communications, and they have no additional penalty for IPv6. Peering Information Objects are used to optimize peer discovery process, not to transfer actual application payloads. While all PIO-related transfer is considered to be overhead, it reduces number of false positives when making peering decisions. This overhead consists of 32bit PIO object header and \( n \cdot 32\text{bit} \) PIO data. With messaging NSLP, PIO data size is always 32bit and this means that the total PIO overhead is 8 bytes. Other NSLPs taking advantage of Peering Information Objects may use different PIO payload sizes.

Just as the other overhead figures, also number of overhead bytes for GIST Query was verified with UHGIST by capturing and analyzing data traffic sent by Messaging NSLP and test application.

Table 5: GIST Query message overhead. Stack proposal and stack configuration data objects are included only when setting up C-mode communications.

<table>
<thead>
<tr>
<th></th>
<th>IPv4</th>
<th>IPv6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magic Number</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>GIST Common Header</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>MRI</td>
<td>24</td>
<td>52</td>
</tr>
<tr>
<td>SID</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Query Cookie</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>Network Layer Information</td>
<td>24</td>
<td>36</td>
</tr>
<tr>
<td>Stack Proposal</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Stack Configuration Data</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Peering Information</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Total without stack objects</td>
<td>124</td>
<td>164</td>
</tr>
<tr>
<td>Total with stack objects</td>
<td>156</td>
<td>196</td>
</tr>
</tbody>
</table>

Overhead to send the first payload data bytes in D-Mode is shown in Table 6. IP and UDP layers add 28 or 32 bytes on IPv4 and 48 or 56 bytes on IPv6 depending on use of RAO. First segment of the UDP payload is the four-byte GIST magic number, followed by GIST common header. Message routing information object, session identifier, query cookie and network layer information objects are always present in GIST query messages. Peering information object is included in the calculation even while it is only an optional extension. Optional NAT Traversal Objects may also be
used, but this overhead is omitted here. Messaging NSLP objects, i.e. Messaging NSLP header and Application Data objects are carried within NSLP Data structure. This is not the complete overhead due to GIST’s handshake being also completed.

Table 6: Overhead to send the first payload data unit in D-Mode.

<table>
<thead>
<tr>
<th></th>
<th>IPv4</th>
<th>IPv6</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP Header(+RAO)</td>
<td>20(+4)</td>
<td>40(+8)</td>
</tr>
<tr>
<td>UDP Header</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Query without stack objects</td>
<td>124</td>
<td>164</td>
</tr>
<tr>
<td>NSLP Data TLV</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Messaging NSLP Header</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Messaging NSLP Application Data TLV</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Total with RAO</td>
<td>172</td>
<td>236</td>
</tr>
</tbody>
</table>

Overhead to send the first payload data bytes in C-Mode is shown in Table 7. First a Query message with peering information, stack proposal and stack configuration data objects is sent. Response is equal to Query with the exception of added response cookie and omitting PIO. Confirm is the first message that may carry NSLP Data. It lacks magic number and peering information object because it is sent over previously-negotiated TCP connection. The NSLP Data in Confirm message generates an overhead of 4 bytes due to NSLP Data header. With Messaging NSLP the application data carried in TLV structure follows the Messaging NSLP header. With UHGIST, NSLP Data is not carried in Confirm messages and a separate GIST Data message is used instead.

Overhead of GIST NSLP Data message is shown in Table 8. As other GIST messages, also NSLP Data message contain GIST Common header. Message Routing Information, Session Identifier and NSLP Data objects are always present. Network Layer Information object is carried only when using D-Mode transport. Noteworthy in this table is, that single GIST Data message takes significant overhead penalty when using D-Mode. C-Mode session setup has its own overhead, but when C-Mode session is ready, it is cheaper to use, at least when TCP acknowledgments are not considered.

Messaging NSLP object sizes are shown in Table 9. All Messaging NSLP messages start with Messaging NSLP header. Other Messaging NSLP objects are included depending on message type. For example, a notification message has Messaging NSLP header and its optional components are RII and Application Data objects.
Table 7: Overhead to send first payload data unit in C-Mode. The data is sent within Confirm message. Subsequent data units may be sent with GIST Data messages. Optional peering information object is included only into the Query message.

<table>
<thead>
<tr>
<th></th>
<th>IPv4</th>
<th>IPv6</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query:</td>
<td>116</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>PIO</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Stack Proposal</td>
<td>12</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Stack Configuration Data</td>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Response:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Query+response cookie-PIO</td>
<td>184</td>
<td>224</td>
<td>←</td>
</tr>
<tr>
<td>Confirm:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Same as Query</td>
<td>144</td>
<td>212</td>
<td></td>
</tr>
<tr>
<td>NSLP Data TLV</td>
<td>4</td>
<td>4</td>
<td>←</td>
</tr>
<tr>
<td>Messaging NSLP Header</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Messaging NSLP Application Data TLV</td>
<td>4</td>
<td>4</td>
<td>←</td>
</tr>
<tr>
<td>total</td>
<td>520</td>
<td>696</td>
<td>←</td>
</tr>
</tbody>
</table>

Total number of transmitted bytes of a single C-Mode session used to transfer 4 bytes of application data is shown in Table 10. The transmitted data was captured using tcpdump and analyzed with Wireshark. Total overhead including IP layer and transport layer headers was 938 bytes. This is quite close to being the worst case scenario, i.e. C-Mode session is used to transfer single really small payload.

5.3 Summary

Results of overhead analysis and testing were presented in this chapter. Tests and analysis mainly focused on on-wire overhead, i.e. additional transferred bytes on top of application payloads. Test results confirm results of analysis done on specification level. Also limited sanity checks for memory consumption of Messaging NSLP were done. Simple test cases used were unable to show any meaningful results on use of CPU resources. Thus, additional considerably broader testing would be needed to get significant results of CPU consumption.

6Dissector for NSIS family of protocols was used to analyze the captured data. It supports standardized NSIS protocols and features, but for example Messaging NSLP and Peering Information Objects needed manual analyzing. It is available at http://nsis.srmr.co.uk/xis/wireshark.html.
Table 8: Overhead of GIST Data Messages sent in D-Mode or in C-Mode. Network Layer Information object is carried only in D-Mode. Optional NAT traversal object is not included.

<table>
<thead>
<tr>
<th>Object</th>
<th>IPv4</th>
<th>IPv6</th>
</tr>
</thead>
<tbody>
<tr>
<td>GIST Common Header</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Message Routing Information</td>
<td>24</td>
<td>52</td>
</tr>
<tr>
<td>Session Identifier</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Network Layer Information</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>NSLP Data</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>80</td>
<td>56</td>
</tr>
</tbody>
</table>

Table 9: Sizes of Messaging NSLP objects. Optional error-specific information for error object and optional authentication objects not included. Application data consists of four-byte object header and $4n$ bytes of application data.

<table>
<thead>
<tr>
<th>Object</th>
<th>Size (with IPv6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messaging NSLP Header</td>
<td>8</td>
</tr>
<tr>
<td>RII</td>
<td>8</td>
</tr>
<tr>
<td>Error</td>
<td>12 (24)</td>
</tr>
<tr>
<td>Application Data</td>
<td>$4 + 4n$</td>
</tr>
</tbody>
</table>

On-wire overhead tests were ran using UHGIST and Messaging NSLP implementations. Results differed a bit different with existing, published overhead analysis results. Rational explanations were found for these differences, and UHGIST seems to be compliant to GIST specification.

Levels of on-wire overhead were found to be very high on some cases. However, GIST and Messaging NSLP have never been designed to be low-overhead protocols. Large overhead is due to extensive peer discovery and peer relationship maintenance processes which leads to robust, predictable system. It is also noteworthy to mention that while number of transferred overhead bytes is somewhat high, the number of transport protocol messages (i.e. UDP datagrams or TCP messages) is quite optimized.
Table 10: UHGIST sending NSLP data using C-Mode transport.

<table>
<thead>
<tr>
<th></th>
<th>Bytes</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Query:</strong> IPv4 + UDP</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>Magic number</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>GIST Common Header</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>MRI</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Session ID</td>
<td>20</td>
<td>→</td>
</tr>
<tr>
<td>NLI</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Query Cookie</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>Stack Proposal</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Stack Configuration Data</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Peering Information Object</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td><strong>Response:</strong> IPv4 + UDP</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>Magic number</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>GIST Common Header</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>MRI</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Session ID</td>
<td>20</td>
<td>←</td>
</tr>
<tr>
<td>NLI</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Query Cookie</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>Responder Cookie</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>Stack Proposal</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Stack Configuration Data</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>IPv4 + TCP SYN</td>
<td>60</td>
<td>→</td>
</tr>
<tr>
<td>IPv4 + TCP SYN,ACK</td>
<td>60</td>
<td>←</td>
</tr>
<tr>
<td>IPv4 + TCP ACK</td>
<td>52</td>
<td>→</td>
</tr>
<tr>
<td><strong>Confirm:</strong> IPv4 + TCP</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>GIST Common Header</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>MRI</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Session ID</td>
<td>20</td>
<td>→</td>
</tr>
<tr>
<td>NLI</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Responder Cookie</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Stack Proposal</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Stack Configuration Data</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>IPv4 + TCP ACK</td>
<td>52</td>
<td>←</td>
</tr>
<tr>
<td><strong>Data:</strong> IPv4 + TCP</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>GIST Common Header</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>MRI</td>
<td>20</td>
<td>→</td>
</tr>
<tr>
<td>Session ID</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>NSLP Data</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>IPv4 + TCP ACK</td>
<td>52</td>
<td>←</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>942</td>
<td></td>
</tr>
</tbody>
</table>
6 Summary and Future Work

Wheel has been reinvented several times on signaling protocols. Most signaling protocols have their own signaling transport methods and have solved the same problems over and over again. Network address translators and firewalls, i.e. middleboxes have created havoc in protocol design, not sparing signaling domain. Also mobility issues such as end-host and network mobility have added their flavors to signaling transport problems.

Next Steps In Signaling (NSIS) framework was created to address these problems. A common signaling transport layer, namely General Internet Signaling Transport (GIST) was specified in the IETF to provide transport service for signaling applications. The NSIS suite also includes two signaling applications, one for Quality of Service issues and one to configure middleboxes. These applications, i.e. NSIS Signaling Layer Protocols were also specified in the IETF. Future NSIS development is governed by Internet Assigned Numbers Authority (IANA) and IETF. For example, to create a new NSLP one needs to get the specification accepted in IETF to get NSLP identifier from IANA.

Services GIST provides to NSLPs include peer discovery and unreliable or optionally secured reliable transport. Peer discovery service means that GIST finds the next corresponding NSLP node on path toward signaling endpoint. Unreliable transport is a bit lighter regarding session setup than its reliable counterpart. Security properties of GIST transport service include protection against eavesdropping, ensuring message integrity and preventing replay attacks. Signaling applications use GIST services through an API that consists of several operations. On top of operations for sending and receiving data, the API also covers network events, errors and session state management. The API covers all GIST aspects, and allows application developer to have adequate knowledge of network state. However, the API is also somewhat cumbersome to use, and application developer needs to take care of non-trivial amount of details.

Messaging NSLP is a new signaling application that uses GIST services to fulfill its transport needs. The new NSLP is an abstraction layer that hides complex GIST features from Messaging Applications. Developers of Messaging Applications can use a simple Messaging API to open and close sessions and to transfer application data from one Messaging Application node to another.
While making signaling application development easier and more agile with the simpler API, the Messaging NSLP also lightens bureaucracy. A somewhat heavy, and at least time-consuming specification process in IETF is needed to get a new NSLP accepted. While Messaging NSLP has a reserved address space for messaging application identifiers that is intended for IETF/IANA coordinated applications, it also has quite a large pool of identifiers for local use.

Messaging NSLP takes advantage of new Peering Information Objects, an extension to GIST, to refine GIST peer discovery process. GIST is not always able to carry NSLP data within session setup messages due to transport options selected by the given NSLP. If the NSLP requests secure or reliable transport, no NSLP data may be sent on handshake phase. Peering information data is specified to be always non-secure and non-reliable, thus allowing transport even when the actual payload can not be carried before reliable, possibly secure transport channel is negotiated.

Application programming interface for NSIS Signaling Layer Applications and Messaging NSLP were implemented. Implementations are pure C code and the only external library used is OpenSSL, which is optionally used to encrypt NSLP API communications between GIST core and NSLPs.

Overhead analysis of GIST implementation and Messaging NSLP were performed, and results are compatible with earlier, third-party analysis. On top of on-wire overhead analysis done on specification level, also real GIST messages were captured and analyzed. No surprises were found when these results were compared. Bytewise overhead was found to be quite large as expected, but the number of network packets transmitted was reasonably low. This keeps number of round-trips low, and helps to maintain acceptable latencies.

Finalization and possibly extending Messaging NSLP specification are still left to be done. One possible future experiment could be implementing a publish-subscribe system over Messaging NSLP. Application identifiers could be used as pub/sub identifiers. Some sort of special identifier would be needed for a directory that would be used to advertise and query new data streams.

A signaling system consisting of NSIS and the Messaging NSLP resembles a software defined network in the sense that both have the philosophy of enabling researchers to test new features incrementally in a live production network. While Openflow is based on a centralized component having all the processing logic, NSIS and the Messaging NSLP distribute the processing to the network nodes. Some sort of SDN testbed could be implemented with the Messaging NSLP, but this would need
additional research and most likely implementation improvements.

If the work with Messaging NSLP is to be continued, then one possible work item would be creating an Internet Draft of the specification, and provide it to IETF community.
References


<table>
<thead>
<tr>
<th>ID</th>
<th>Author(s)</th>
<th>Title</th>
<th>Reference</th>
<th>Year</th>
<th>URL</th>
<th>Date</th>
</tr>
</thead>
</table>


Ros10 Rosenberg, J., Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer


SCB03 Smith, R., Chen, Y. and Bhattacharya, S., Cascade of Distributed and Cooperating Firewalls in a Secure Data Network. IEEE Transactions on Knowledge and Data Engineering, 15,5(2003), pages 1307–1315. DOI:10.1109/TKDE.2003.1232280.

Sch02 Schoenwaelder, J., Simple Network Management Protocol (SNMP) over Transmission Control Protocol (TCP) Transport Mapping. RFC 3430,


A Messaging NSLP Specification

Abstract

Messaging NSLP is a simple extensible messaging platform for signaling applications. It is using GIST from NSIS signaling suite for peer discovery, security and message transport. NSIS stack with Messaging NSLP and Messaging Applications is shown in Figure A.1. The NSIS transport layer protocol GIST is built on top of old, proven transport and security mechanisms such as TCP, UDP and TLS. Messaging NSLP uses GIST as a transport method, and provides messaging service to Messaging Applications. This specification provides low-level information of Messaging NSLP. Also an API for messaging applications is described.

The messaging service provides location, addressing and message transport service to applications. While providing messaging service to messaging applications, the Messaging NSLP itself is a signaling application on NSIS Framework. The NSIS Framework, or more specifically the transport service protocol GIST was chosen because it provides flexible transport service. The transport service includes reliable and unreliable message transport and security features such as per node authentication and message protection against modification, injection, replay and eavesdropping. These services are implemented using familiar, well-known protocols and methods. NSIS framework may also ease deployment in the future networks.

Figure A.1: NSIS Protocol Stack with Messaging NSLP and Messaging Applications
A.1 Introduction

Traditional internet signaling protocols are, in addition to routing protocols, mainly focused on Quality of Service\textsuperscript{A.1}. Also other signaling purposes exists, for example resource (battery, bandwidth, CPU load) balancing of routers. It is necessary to transfer information between routers, or in some cases between router and end host of communication. One specific case where it is important to spread state information of routers is routing in mesh network. With resource information of nearby routers it is possible to adjust routing tables and gain efficiency and reliability benefits when taking account the very limited resources of mesh routers.

While the first goal of the Messaging NSLP was to transfer measurement and monitoring data, it will form a generic messaging protocol to support also future needs. The main focus is on router-to-router communications, but the protocol is meant to be extensible so it will be suitable also for other scenarios such as router-to-host or host-to-router communications. Other use scenarios of Messaging NSLP include for example active firewall configuration, where Messaging NSLP is used to distribute information of ongoing network attacks. This way attacks may be stopped at the network perimeter or even closer to the attack source.

The Messaging NSLP does not take active role in maintaining connections if the network below is dynamic. This means that Messaging sessions may be torn down at any time, and Messaging NSLP does not try to reconnect these session. The Messaging Application receives a notification that a session it has been using is not active anymore, but it is left responsible of the possible reconnection attempt. This restriction is done to simplify Messaging NSLP. Without this simplification, the Messaging NSLP would have been basically \textit{GIST on GIST}. While transferring some session management responsibility from Messaging NSLP to Messaging Applications, this restriction also helps keeping Messaging API as simple as possible.

Messaging applications use Messaging API to access Messaging NSLP functionality. The API is kept simple to allow easy and fast application development and to avoid steep learning curve. There are two functions for session managements, \texttt{open()} and \texttt{close()}. Data is sent using \texttt{write()} function and received by using \texttt{read()} function.

\textsuperscript{A.1}Brunner, M., Requirements for Signaling Protocols. RFC 3726. Available online at https://tools.ietf.org/rfc/rfc3726.txt
A.1.1 General Internet Signalling Transport API

GIST provides transport service for signaling applications. Signaling application data is passed to GIST via *SendMessage(..)* call of GIST API. Together with NSLP data, also other important information is passed to GIST.

The whole structure of *SendMessage(..)* GIST API call is as follows:

```
SendMessage ( NSLP-Data, NSLP-Data-Size, Peering-Information-Data, 
              Peering-Information-Data-Size, NSLP-Message-Handle, 
              NSLPID, Session-ID, MRI, SII-Handle,  
              Transfer-Attributes, Timeout, IP-TTL, GIST-Hop-Count )
```

Mandatory parameters are briefly explained here.\(^\text{A.2}\)

**NSLP-Data** Messaging NSLP message

**NSLP-DATA-Size** Length of data

**Peering-Information-Data** Data to support conditional peering (See Section A.3) decisions.

**Peering-Information-Data-Size** Length of Peering Information Data.

**NSLP-Message-Handle** Handle that can be used by Messaging NSLP as a reference with subsequent MessageStatus notifications.

**NSLPID** GIST Identifier for Messaging NSLP

**Session-ID** Session Identifier. The Messaging NSLP creates and supplies this SID to application when application calls open().

**MRI** Message Routing Information object. Contains, for example, the flow destination address.

The rest of the parameters are optional, but still important. For example, Transfer-Attributes contain transfer mode specification, especially security and reliability attributes.

A Counterpart of SendMessage(..) in GIST API is RecvMessage(..). It is used to fetch a message that GIST layer has received from network.

RecvMessage(..) GIST API call has the following structure:

```
RecvMessage ( NSLP-Data, NSLP-Data-Size, Peering-Information-Data, 
              Peering-Information-Data-Size, NSLPID, Session-ID, MRI, 
              Routing-State-Check, SII-Handle, Transfer-Attributes, 
              IP-TTL, IP-Distance, GIST-Hop-Count, Inbound-Interface )
```

**NSLP-Data**  Messaging NSLP message

**NSLP-Data-Size**  Length of data

**Peering-Information-Data**  Data to support conditional peering (See Section A.3) decisions.

**Peering-Information-Data-Size**  Length of Peering Information Data.

**NSLPID**  GIST Identifier for Messaging NSLP

**Session-ID**  Session Identifier. The Messaging NSLP creates and supplies SID to application when sender application calls open().

**MRI**  Message Routing Information.

**Routing-State-Check**  Boolean value that is used to check if routing state creation is needed. See RFC 5971 and Section A.3.

**SII-Handle**  Pointer to a specific peer. May be used with explicit routing.

**Transfer-Attributes**  As with SendMessage but contains also information about addresses in MRI.

**IP-TTL**  The value of IP layer TTL.

**IP-Distance**  The number of IP hops from the sending peer.

**GIST-Hop-Count**  GIST hop count after decrement by GIST.

**Inbound-Interface**  Information of inbound network interface specific to GIST implementation.
A.2 Protocol Specification

The key words "MUST", "MUST NOT", "SHOULD", "SHOULD NOT", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.\textsuperscript{A.3} Non-capitalized key words do not carry special meaning.

A.2.1 Messaging Applications

Messaging NSLP provides transport service for simple Messaging Applications. These applications are identified by a 32bit application identifier. Each network node running Messaging NSLP may have its own selection of messaging applications.

A.2.2 Messages

The Messaging NSLP has three types of messages. The purpose of these messages is described below.

Requests and Notifications

There are three different types of messages in the Messaging NSLP. The basic message type is notification, that can be sent by application with or without an explicit request message. Use of these two messages is shown in Figure A.2. Regardless of the message type, payload of every received message is passed to a suitable application that is identified by application id.

![Diagram of messaging scenarios](image)

Figure A.2: Two messaging scenarios: Notify with and without request.

\textsuperscript{A.3}Key words for use in RFCs to Indicate Requirement Levels. RFC 2119. Available online at https://tools.ietf.org/rfc/rfc2119.txt.
Error

Most error conditions that might occur are of such type that it is very hard or even impossible to recover. It may be beneficial to inform the other communicating party about the error condition with some error types. For these purposes there is a message type \textit{error} included into the Messing NSLP. Information about the error condition is included within \textit{error} message by adding \textit{error object}, see Section A.4.5.

Quick ack

It is easy to discover scenarios where fulfilling a request may take significant amount of time or other resources. For example, a monitoring application could ask network link load statistics of next 24 hours from another host. Naturally answering this kind of request takes a long time. The Messing NSLP has a feature designed to inform the querier about received request. This mechanism is called a \textit{quick ack}, and it MAY be sent as a response to a request. The querier MAY also explicitly request the quick ack within the request message. A quick ack SHOULD be sent if explicit request for a quick ack is present in the received request message.

A Quick ack is a notification message. Its only payload MUST be a copy of Request Identification Information object (RII) of the \textit{request}.

A.3 Message Delivery

Normal case for messaging message delivery is when two or more Messing NSLP nodes send messages to each other within \textit{messaging sessions}. After session setup the message payload delivery is provided by GIST. The session setup is simple if all the nodes run same applications – each node on the signaling path joins the session. If the network has heterogeneous messaging application distribution, i.e. all nodes do not run same applications, some additional housekeeping is needed by Messing NSLP. This affects the session setup phase.

In session setup phase GIST finds the next Messing NSLP node by sending a GIST query message toward signaling endpoint. The query is processed on every Messing NSLP node. Each receiving Messing NSLP needs to decide whether this communication is meant to it or not. This decision requires knowledge of application identifier. Messing NSLP can not force application data sending on GIST query messages. Additionally, GIST query messages can not carry NSLP data (in context
of Messaging NSLP, messaging application data) if the session parameters require reliable or secure transport. This means that the Messaging NSLP cannot rely purely on GIST peer discovery.

Optimal case is when the Messaging NSLP receives a GIST query and it contains application identifier information. In this case the receiving node can do correct decision whether to peer with the sending node or not. If the information is not present, the Messaging NSLP needs to peer, and just pass the data forward toward signaling endpoint if it later learns that the application in question is not locally supported.

The Messaging NSLP peering problem is shown in Figure A.3. The sending Messaging NSLP node $S$ starts session setup toward ultimate receiver at node $R$. Both the $S$ and $R$ are running messaging application $A$, but while nodes $N_1$ and $N_2$ are running Messaging NSLP, they lack support for application $A$.

![Figure A.3: Peering Problem: What to do if a message is delivered to a node that does not support the application?](image)

For optimal peering decisions receiving Messaging NSLP nodes need to know the identifier of the sending application. Because GIST queries can not always carry data supporting peering decisions, a new extension for peering information data is proposed.\textsuperscript{A.4} This PIO (Peering Information Object) mechanism allows NSLPs to include data supporting peering decisions into GIST queries. Peering information objects are briefly covered also in Section A.4.4.

Because PIO mechanism is not necessarily supported on every GIST node, needs Messaging NSLP work also without it. To support working without peering information, the Messaging NSLP MUST accept peering requests if the PIO is not available. In later stage of messaging session Messaging NSLP can confirm whether the traffic is meant to it or not, i.e. is the sending application locally supported. If the application is not supported, the Messaging NSLP SHOULD forward the data toward signaling endpoint.

If PIOs are supported by local GIST implementation, the Messaging NSLP SHOULD include application identifier of the communicating messaging application into every `SendMessage()` GIST API call.

### A.3.1 Messaging API

While GIST is handling the actual sending and receiving of messages, it can not understand the content of Messaging NSLP messages. When GIST receives a message with NSLP identifier matching the Messaging NSLP, it is passed to the Messaging NSLP process.

There can be several different applications that use the messaging service provided by the Messaging NSLP. Each application has a unique application identifier that is used by the Messaging NSLP to find proper recipient application for each message received.

The interface toward applications is very simple with no more than four operations, `open()`, `read()`, `write()` and `close()`. Application interface after calling `open()` is shown in Figure A.4. All data transfer between application and Messaging NSLP is done by the application with these operations.

![Message NSLP application interface](image.png)

**Figure A.4**: Messaging NSLP application interface without `open()` and `close()`

#### A.3.2 open()

`open()` operation is used by application to acquire a SID to use with `write()` operations. Application MUST first call `open()` to be able to send data to new hosts. If application has received a message from network with `read()` operation, it already has an identifier for active session, and call for `open()` operation is not needed.
Open() is called with the following structure:

\[
\text{SID open( addr_type, addr_length, endpoint_address, secure, reliable)}
\]

- **addr_type**: type of address (IPv4, IPv6, SIP...)
- **addr_length**: address length
- **endpoint_address**: address
- **secure**: boolean value: to use secure data transfer mode or not
- **reliable**: boolean value: to use reliable data transfer mode or not
- **returns**: SID of new session (or negative value in case of error)

### A.3.3 read()

Read() operation is used by application to transfer data from the Messaging NSLP process. On read() operation the Messaging NSLP stores application data into a buffer given with the read() request. The Messaging NSLP also provides the session identifier and the type of received message. Request Identification Information (RII) object is also provided if the message received was a *request* or a response for a request sent earlier by this Messaging Application. If a RII is not included, the Messaging NSLP MUST set the RII value to 0. Read() operation returns the amount of bytes read and a negative value on error. In case of insufficient data buffer size the read() operation returns value *insufficient data buffer size*, and the Messaging Application SHOULD call close() for the session or call read() again with larger buffer size. The buffer is in unspecified state after *insufficient data buffer size* error, and the Messaging Application MUST NOT use its content.

The read() operation has following structure:

\[
\text{int read(*SID, void *data, int data_len, int *msg_type, int *rii)}
\]

- ***SID**: Pointer to space reserved for session identifier. Set by the Messaging NSLP.
- **void *data**: Storage for data to be received
- **int data_len**: Data storage size
- **int *msg_type**: Storage for the type of the message the data arrived within
int *rii Storage for the content of the possible RII object

returns Amount of bytes read, negative on error

Message type values:

notification: 10
    request: 20

Error codes:

insufficient data buffer size: -1

A.3.4 write()

Write() operation is used by application to transfer data to the Messaging NSLP. In addition to the actual message data, session identifier and type of message are provided as a parameter. If the Messaging Application wishes to send a request message, it creates a 32bit RII value that MUST be probabilistically unique within the context of a session (SID). If the application has received a request and wishes to respond, it MUST use the RII provided with the request. If the application is sending a notification not related to any request, it MUST set the RII value to 0. Write() operation returns non-zero in case of failure.

int write(SID, data_length, data, int msg_type, int rii)

SID Session Identifier, got as a return value of an earlier open() call

data_length Length of data to be written

data Application data object

int msg_type Type of message to be sent

int rii Content for Response Identification Information

returns non-zero value indicates failure
Message type values:

- notification: 10
- request (quick acks may or may not be sent): 20
- request (with explicit denial of quick ack): 22
- request (with explicit request of quick ack): 24

Error codes:

- invalid session: -1

A.3.5 close()

With close() operation application can signal the Messaging NSLP that no more data is going to be written or read within the corresponding session. When application decides that it does not need to use an existing session anymore, it SHOULD call close() operation. Call to close() operation signals the Messaging NSLP that it SHOULD tear down session corresponding session. Messaging NSLP SHOULD also invalidate routing states that relate to this session.

```c
int close(SID)
```

**SID** Session Identifier of the session that application does not want use anymore

**returns** non-zero value indicates failure

Error codes:

- invalid session: -1

A.4 Messaging NSLP Message and Object Formats

While the Messaging NSLP itself has only a small header, GIST message header contains mandatory parts defined in RFC 5971. One detail the Messaging NSLP needs to take care of is the Session Identifier (SID) carried within GIST headers. Session identifiers MUST be selected in such way that they can not be guessed by hypothetical attacker.

The basic structure of NSLP message can be seen in Figure A.5. The Messaging NSLP header is followed by a set of objects: *RII, Authorization, Application Data,*
**Error.** Error objects MUST be carried only within *error* messages, other objects have no such limitations.

![Messing NSLP message structure](image)

**Figure A.5:** Coarse-grained message structure of Messaging NSLP

Besides the objects of its own, also objects of other origins may be used with the Messaging NSLP. For example, many messaging applications may deal with sensitive data or there may be other needs to make sure that all participants are authorized to communicate. While GIST provides us some degree of security, there is also need for more fine-grained authorization within Messaging NSLP and messaging applications. Authorization for NSIS signaling layer protocols is specified in RFC 5981. This authorization mechanism uses AUTH_SESSION object, and it MAY be also used with the Messaging NSLP.

Disposition of objects in Messaging NSLP messages is not free. The order of objects and what object types may be carried within certain message types is shown in Table A.1.

---

Table A.1: Object disposition within different message types

<table>
<thead>
<tr>
<th>Object type</th>
<th>Request</th>
<th>Notification</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>RII</td>
<td>yes</td>
<td>optional</td>
<td>optional</td>
</tr>
<tr>
<td>ERR</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>AUTH</td>
<td>optional</td>
<td>optional</td>
<td>optional</td>
</tr>
<tr>
<td>APP_DATA</td>
<td>yes</td>
<td>optional</td>
<td>optional</td>
</tr>
</tbody>
</table>

### A.4.1 Messaging NSLP Header

Messaging NSLP header is present in every Messaging NSLP message. It contains following fields:

**Message Type** Type of message (8 bits).

**Message Flags** Message type specific flags (8 bits).

**reserved** Reserved

**Application ID** Application identifier (32 bits)

```
+---------------------------------------------+
| 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 |
| +---------------------------------------------+
| | Message Type | Message Flags | Reserved (16bit) |
| +---------------------------------------------+
| | +---------------------------------------------+
| | | Application ID (32bit) |
| +---------------------------------------------+
```

Message type may have value *request*, *notification* or *error*. This version of Messaging NSLP specification does not support other message types, but they may be added at later revisions.

Numerical values of each message type are:

```
reserved: 0
notification: 1
request: 2
```
If a message of another type is received the message MUST be silently discarded.

Message Flags are message type specific. Two message flags, namely \( N \) and \( Q \) are specified for request messages with following structure:

\[
\begin{array}{c}
1 \\
8 9 0 1 2 3 4 5 \\
+---------------------+
| reserved |N|Q| \\
+---------------------+
\end{array}
\]

If \( Q \) flag is set in request, it means that sender wishes a quick acknowledgment (see A.2.2) to be sent. However, quick acks MAY be sent also without explicit request. Quick ack MUST NOT be sent if \( N \) flag is set in request. Combination where both the \( N \) and \( Q \) flags are set MUST NOT be used.

- \( NQ=00 \) Quick ack MAY be sent (receiver decision)
- \( NQ=01 \) Quick ack SHOULD be sent (explicit request by querier)
- \( NQ=10 \) Quick ack MUST NOT be sent (explicit request by querier)
- \( NQ=11 \) Not used (invalid value)

Application Identifier is a 32bit value carried within every Messaging NSLP message. It is used to associate a given message to the corresponding Messaging Application. The following ranges of application identifier values are specified:

- reserved: \( 0x0 \)
- for private/experimental use: \( 0x1 \) to \( 0xFFFFFFFF \)
- free to use: \( 0x1000000 \) to \( 0x3FFFFFFF \)
- reserved for assignments: \( 0x4000000 \) to \( 0xFFFFFFFF \)
A.4.2 Application Data Object

When messaging application sends message to another NSIS host the messaging NSLP encapsulates the message data into an application data object. It has a header that contains following fields:

A  Extensibility flag
B  Extensibility flag
r  Reserved

Type  Object type (APP_DATA)
Length  Object data length

Application Data  Application specific data

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|A|B|r|r|  Type  |r|r|r|r|  Length  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
// Application Data  //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Extensibility flags A and B may be used to carry information how to act if object type is unknown. Application data object is fundamental to every messaging application. It is error condition if application data object is not understood.

Within this specification, only allowed flag combination is A=0 and B=0. If object is not understood, the entire message containing it MUST be rejected and a protocol error with type illegal object type sent back. If any other extensibility flag combination seen, a protocol error with type invalid flags MUST be sent as a response and the message MUST be discarded.

AB=00  Mandatory: If the object is not understood, the entire message containing it MUST be rejected, and a protocol error message of type illegal object type sent back.
AB=01 Ignore: If the object is not understood, it MUST be deleted and the rest of the message processed as usual.

AB=10 Forward: If the object is not understood, it MUST be retained unchanged in any message forwarded as a result of message processing, but not stored locally.

AB=11 Illegal: The combination AB=11 MUST NOT be used.

A.4.3 Request Identification Information Object

Request Identification Information object is similar to its namesake in RFC 5974. It limits the propagation of notification message sent as response to a request. It ensures that the notification is not forwarded along the path further than the node that sent the original request. Request Identification Information objects are also used to bind notifications and quick acks to the corresponding requests.

Messaging Application can choose the value of RII, but it should be chosen in a way that it is statistically unique within context of the SID.

A Extensibility flag

B Extensibility flag

r Reserved

Type Object type (RII)

Length Object data length: Fixed 32bit word

RII data Identifier that MUST be unique within session.

As with other objects within the Messaging NSLP, the only allowed combination of extensibility flags is \( AB = 00 \).

### A.4.4 Peering Information Object

Peering Information Object is not a Messaging NSLP object. It is included within Query messages at GIST level and described briefly below.\(^{A,7}\)

The peering problem (see A.3) may create some trouble when creating Messaging Associations, if the network participating to the Messaging NSLP conversation has a heterogeneous application selection. While unmodified GIST can not carry NSLP payload with query messages, application identifier can be delivered to responding Messaging NSLP node by including Peering Information object within GIST query messages. To optimize peering process, PIO SHOULD be provided by Messaging NSLP in every SendMessage() GIST API call if the local GIST implementation supports it.

The PIO has the following structure:

<table>
<thead>
<tr>
<th>A</th>
<th>Extensibility flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Extensibility flag</td>
</tr>
<tr>
<td>r</td>
<td>Reserved</td>
</tr>
<tr>
<td>Type</td>
<td>Object type (PIO)</td>
</tr>
<tr>
<td>Length</td>
<td>Object data length (fixed 32bit)</td>
</tr>
</tbody>
</table>

**Peering Information** Application Identifier (32bit)

```
  +---------------+---------------+---------------+---------------+
  | A | B | r | r | Type | r | r | r | r | Length |
  +---------------+---------------+---------------+---------------+
  | Peering Information (Application Identifier) |
  +---------------+---------------+---------------+---------------+
```

Because unmodified GIST implementations do not know how to handle PIOs we need to set extensibility flags to give a processing advice to implementations unaware to PIOs. Strong recommendation of flag combination AB=00 is given in PIO specification.

A.4.5 Error Object

Error object carries information about error and one Error Object MUST be carried within each error message. Error Object MUST NOT be included within any other message type.

Error object structure is similar to INFO_SPEC object of QoS NSLP.\(^{A.8}\)

```
|A|B|r|r| Type |r|r|r|r| Length |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Error Code |E-Class|ESI Typ| ESI-Length |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
// Error Source Identifier //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
// Optional error-specific information //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fields in error object have the following meanings:

A  Extensibility flag
B  Extensibility flag
r  Reserved
Type Object type (Error Object)
Length Object data length

Error Code Error codes within specific error class (16bit)

**E-Class** Indicates the error severity class. The two currently specified classes are: *Protocol Error* and *Permanent Failure* (4bit)

**ESI Type** Type of Error Source Identifier. *IPv4* or *IPv6* (4bit)

**ESI-Length** Length of ESI address in 32bit words (8bit)

**Error Source Identifier** OPTIONAL Error Source Identifier address for diagnostic purposes

**Optional error-specific information** If object type is erroneous, the object type information MAY be included at the end of Error object (16bit Object type, 16bit reserved)

As with other objects within the Messaging NSLP, the only allowed combination of extensibility flags is AB=00.

Error classes and codes are summed up in Table A.2.

<table>
<thead>
<tr>
<th>Error class</th>
<th>Error code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>Illegal object type</td>
</tr>
<tr>
<td></td>
<td>0x02</td>
<td>Illegal flags</td>
</tr>
<tr>
<td>0x02</td>
<td>0x01</td>
<td>Internal or system error</td>
</tr>
<tr>
<td></td>
<td>0x02</td>
<td>Authorization failed</td>
</tr>
<tr>
<td></td>
<td>0x03</td>
<td>Endpoint missing application</td>
</tr>
</tbody>
</table>

**A.5 Message and Object Processing**

The Messaging NSLP message and object processing is described below. This description covers peering with and without peering information objects.
A.5.1 Receiving Messages

The beginning of message receiving process is shown in Figure A.6. It begins when GIST receives a message from network, and the Messaging NSLP calls `RecvMessage(..)` API call. Next the Messaging NSLP checks whether the message is meant for this Messaging NSLP host or not by extracting the Application Identifier information from the possible Peering information object carried within received message. Application information may not always be present in queries, if the node that sent the request does not support Peering information objects. It is also checked whether this is a GIST Query or not by checking the `Routing-State-Check` value got from `RecvMessage` API call. If the message was not a query, then we should have knowledge of application identifier (it is carried within Messaging NSLP message header) and we already know that we have a corresponding application.

After these bits of information have been extracted the Messaging NSLP makes the decision what to do with the message. There are four alternatives:

1. If the received message was a GIST Query and this Messaging NSLP host runs the corresponding Messaging Application, then we know that we are creating a new connection and continue the peering process. This Messaging NSLP host MUST continue the peering process also if the query did not carry Peering information object, because sender of the received query may not support PIOs.

2. If the received message was a GIST Query for some unknown Messaging Application we just forward the Query.

3. If the received message was not a GIST Query, and this Messaging NSLP host runs the corresponding Messaging Application, then we pass the Message to the Application in question.

4. If the received message was not a GIST query, and we do not have the application in question, we know this message relates to a flow we are just forwarding hop-by-hop. We just pass pass the Message back to GIST level toward the next hop.

Additional processing may be needed regarding signaling endpoints. If the node processing the message is the signaling endpoint, it knows that there is no next hop the message should be forwarded in case of non-existing local application. In
this case, the Messaging NSLP SHOULD send an error message of class *Permanent failure* of type *Endpoint missing application* to the node it received the message from. This is done to inform the previous node on signaling path about direction that it SHOULD NOT send messages belonging to this application anymore. The newly-found signaling endpoint may actually itself act as a forwarder due to missing local messaging application. In this case it also SHOULD send the identical error message toward signaling source. This way the knowledge of signaling endpoint finally reaches the actual signaling endpoint, and normal communications may continue.

### A.5.2 Opening New Connection

Peering process is completely transparent to the sending Messaging Application supposing that the Messaging Application in question has called `Open(..)` operation of Messaging NSLP API.

A call to `Open(..)` operation by a Messaging Application is an order to the Messaging NSLP to allocate a new Session Identifier for a new connection. When the new SID has been allocated, the Messaging NSLP calls `SendMessagge(..)` operation of the GIST NSLP API. This is done to find the next application level hop at the network with help of GIST.

---

![State machine: Receiving messages](image_url)
The application identifier SHOULD be included into the SendMessage(..) call as Peering Information Data (see Section A.1.1).

Parallel to the SendMessage(..) call the Messaging NSLP returns the SID of the new connection to the Messaging Application. Now the Messaging Application can use write(..) operation with the new SID and start the actual messaging with a remote application.

A.5.3 Sending Notifications

Send process of notification messages is shown in Figure A.7. Here state Idle means that the Messaging NSLP is not processing the message in question at the moment. We assume that the messaging application as already called operation open() and thus it has knowledge of a session the message it is sending will be related to.

The application calls write() operation and application data is eventually sent to the network using SendMessage() GIST API call. Send process ends when write() operation returns and return value is delivered to the application.

![Figure A.7: Notification message processing (send)](image)

A.5.4 Receiving Notifications

Quite simple receive process of notification messages is shown in Figure A.8. A notification message is received from the network by GIST, and delivered to the Messaging NSLP via RecvMessage() operation. Application can read the data from the Messaging NSLP buffers by calling read() operation. Together with application specific data, the application receives a session identifier that can later be used with write() operations without first calling open().

Note that Figure A.8 is meant to be examined together with Figure A.6. These state machines have a common state, namely Incoming message to local application.
A.5.5 Sending Requests

Again it is assumed that the messaging application has an open session to use. The request sending process (see Figure A.9) starts when application calls `write()` operation of Messaging NSLP API. A RII object has to be constructed and included with the application data into the newly created request message. After the request message is constructed and sent to the network using `SendMessage()` operation of GIST API, the system moves to waiting state. There are two possible responses: If long request processing is predicted, the receiving Messaging NSLP may have sent a response as a quick ack. The second alternative is that the application has processed the request and actual response has arrived. Both responses are read from GIST by using the `RecvMessage()` GIST API call. Request processing ends when the sender application eventually gets the response (a real response or a quick ack).

A.5.6 Receiving Requests

Receive process of request messages is shown in Figure A.10. This is quite a lot more complex compared to notification receiving process because of response handling.

The process starts when a request message is received via the GIST API. When the message is buffered, it is checked whether the sender has requested a quick ack to be sent. If a quick ack was requested, a `notification` message is constructed. The quick ack message includes the RII object (see Section A.4.3) received within the `request`. After the possible quick ack is sent using the GIST API, the request message data is passed to the application via `read()` operation of Messaging NSLP application API. Eventually the application has processed the request, and it can call `write()` operation to send the response. Response sending is done just like sending of normal notifications.
Figure A.9: Request message processing (send)

Figure A.10: Request message processing (receive)
A.5.7 Error Message Processing

When receiving a message of type \textit{error}, the messaging application MAY be informed of the error condition.

Error message processing is to be heavily modified within later revisions of the Messaging NSLP specification. Within this version, most error conditions are interpreted as permanent errors that should not trigger sending of any error messages. Most errors are network related, and local tearing down local state and possible logging the error condition are the methods to recover. In case of network errors, for example partitioned network, it is often impossible to reach the peer with error messages.
B  Peering Data for NSIS Signaling Layer Protocols

Network Working Group  J. Manner
Internet-Draft  L. Liuhto
Intended status: Standards Track  N. Varis
Expires: August 28, 2008  T. Huovila
University of Helsinki
February 25, 2008

Peering Data for NSIS Signaling Layer Protocols
draft-manner-nsis-peering-data-01.txt

Status of this Memo

By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she becomes aware will be disclosed, in accordance with Section 6 of BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on August 28, 2008.

Copyright Notice

Copyright (C) The IETF Trust (2008).

Abstract

When an NSLP protocol initiates a signaling session and requests either reliable or secure transport (or both), NSLP data can not be carried within the GIST Query. Thus the NSLP at the responding node can not have NSLP specific information for peering decisions. Next generation NSLP protocols may need more information to be able to make right peering decisions. This draft presents a new Peering
Information Object (PIO) for GIST intended to carry NSLP-specific peering data.

Table of Contents

1. Introduction ................................................. 3
2. Terminology and Abbreviations .............................. 3
3. Peering Information Object .................................. 3
   3.1. Fallback Method ....................................... 5
4. GIST API Issues ............................................. 6
5. Security Considerations ..................................... 7
6. IANA Considerations ......................................... 7
7. Normative References ........................................ 8
Appendix A. Changes since version -00 ........................... 8
Authors’ Addresses ............................................. 8
Intellectual Property and Copyright Statements ................ 10
1. Introduction

The General Internet Signaling Transport (GIST) ([I-D.ietf-nsis-ntlp]) provides a signaling transport service to NSIS Signaling Layer Protocols (NSLP). When an NSLP application wants to send a message to its next peer, GIST starts setting up a Routing State (RS) by sending a GIST Query message. This message carries the NSLP identifier (NSLP ID) and Message Routing Information (MRI) among others. The receiving GIST node running the same NSLP provides the MRI to the NSLP application and requests it to make a decision on whether to peer with the querying node.

The MRI carries very little information about the session that is to be set up, about the querying node, or its real intentions towards the signaling set up. It would be most beneficial to be able to include additional peering information to the receiving node. This would allow an NSLP application to make a better decision on whether the session should actually be set up with the querying node, or perhaps another one.

This specification presents a Peering Information Object (PIO) for GIST that can be used by NSLP applications to give more information for the NSLP at the responding node about the session being set up. The content of the PIO is opaque to GIST and only carried in GIST Query messages when setting up or refreshing Routing State. Since a Query is not protected in any way, the content of the PIO is not protected either. Since the content is NSLP-specific, it is possible to use various hashes and shared encryption keys between NSLP nodes to protect this data. Any such mechanisms are out of scope of this specification, and do not affect GIST.

2. Terminology and Abbreviations

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14, RFC 2119 ([RFC2119]).

All other terminology is taken from the GIST specification ([I-D.ietf-nsis-ntlp]).

3. Peering Information Object

The Peering Information Object (PIO) carries NSLP-specific data to help conditional peering decisions at the NSLP application in the responding node. The PIO object is carried in GIST Query messages.
The value for the Type field comes from shared GIST object type space. The Length field is given in units of 32 bit words and measures the length of the Value component of the TLV object (i.e. it does not include the standard header).

Type: 0x0b (TBD by IANA)

Length: Variable

The leading two bits of the TLV header are used to signal the desired treatment for objects whose Type field is unknown at the receiver [I-D.ietf-nsis-ntlp]. The following three categories of object have been identified, and are described here.

AB=00 ("Mandatory"): If the object is not understood, the entire message containing it MUST be rejected with an "Object Type Error" message (Appendix A.4.4.9) with subcode 1 ("Unrecognised Object").

AB=01 ("Ignore"): If the object is not understood, it MUST be deleted and the rest of the message processed as usual.

AB=10 ("Forward"): If the object is not understood, it MUST be retained unchanged in any message forwarded as a result of message processing, but not stored locally.

The AB-flags SHOULD have a value of "00" when used with the Peering Information Object. Any other value would result in an undesirable result, specifically:

1. AB=01 ("Ignore"): The RS is set up but the peer NSLP will not know that the Peering Information was not honored. Thus, the peering decision was made with less information than originally intended. Subsequent peering decisions will also be made with limited information. No indication is given to subsequent NSLP nodes on the path that peering data was originally given by the signaling initiator.

2. AB=10 ("Forward"): Same as above, but subsequent peering decisions may or may not be based on the peering data. The signaling initiator has no control of how the peering decisions
are done downstream.

With the value of "00", a peer node that does not support the Peering Information object will return an "Object Type Error" to the sender. This can then be used by the querying node to inform the NSLP that peering data can not be used. Currently, the GIST specification leaves it somewhat open as to which errors are propagated to the NSLP. The error in understanding the PIO object SHOULD be provided by GIST to the NSLP. Otherwise the querying NSLP node will not know why the session was not set up, and can not, e.g., try a fallback mechanism and set up a session without additional peering data.

GIST implementations SHOULD include a Peering Information Object within GIST Query messages just after the possible NSLP Data object, if such data was provided by application via NSLP API. GIST SHOULD store the PIO for Query retransmissions. Stored PIO may also be needed after the peering process completes. GIST implementations SHOULD enable replacing the NSLP provided PIO. The NSLP can give a new PIO to replace the old PIO for a given routing state. This new PIO SHOULD be used when sending GIST Query messages, but change of PIO does not mean any change in routing state validity. Application SHOULD also be allowed to remove PIO by providing an empty PIO via the NSLP API.

At this stage this specification does not support stacking of PIO objects. Thus, if an NSLP needs to include complex peering data, it can do so by encoding the structure within the PIO object data. The content of the PIO is opaque to GIST, same as with the NSLP Data.

Note that GIST fragmentation rules apply. Thus, the peering data must be limited in size to keep the size of GIST Query messages under the MTU derived by GIST. If the size of a GIST Query message exceeds the MTU, GIST SHOULD notify the NSLP about the issue. This allows NSLP to take appropriate action, e.g., it may reduce PIO size.

When using D-Mode, it is possible that both a PIO and an NSLP Data object would be included into a GIST Query. This may cause size of the Query to exceed MTU. When avoiding this type of MTU-related issue, GIST SHOULD prioritize PIO over NSLP Data. It is left for GIST implementations to decide when to switch to using C-Mode.

3.1. Fallback Method

This section is meant to give NSLP authors an idea how to work in a mixed environment where PIOs are not always available. This section is not normative.

NSLP Applications should be aware, that PIOs may not always be
available. When an NSLP uses the new Peering Information Object and runs into a GIST implementation that does not support PIO, it can use hop-by-hop NSLP layer forwarding to deliver NSLP Data to the correct recipient. When using hop-by-hop as a fall-back method, also replies are delivered hop-by-hop. In Figure 2 we see how handshake with PIO fails, and how NSLP retries without PIO.

![Diagram of handshake with PIO failure and retry without PIO]

**Figure 2**

1. NSLP Sends NSLP Data towards the flow destination with PIO.
2. GIST Sends a Query message to the network with the NSLP supplied PIO.
3. The Responding GIST Node is unable to process the PIO and returns an "Object Type Error" message to the Querying Node.
4. After maximum number of Query retransmissions, GIST sends an error message to NSLP indicating the error in routing state establishment.
5. NSLP Falls back to Non-PIO behavior, sending the NSLP Data towards the flow destination without the PIO.
6. GIST Sends a Query to the network without a PIO.
7. GIST at Responding Node sends a message to the NSLP requesting a peering decision.

4. GIST API Issues

GIST specifies several abstract API calls between the NSLP applications. The SendMessage and RecvMessage calls need
modifications to support PIO for GIST Query messages. Support for
passing peering information data to GIST is added to SendMessage.
RecvMessage is modified to give the peering data to the local NSLP at
the responding node.

SendMessage ( NSLP-Data, NSLP-Data-Size, Peering-Information-Data,
Peering-Information-Data-Size, NSLP-Message-Handle, NSLPID,
Session-ID, MRI, SII-Handle, Transfer-Attributes, Timeout, IP-TTL,
GIST-Hop-Count )

- Peering-Information-Data: Data to support conditional peering
decisions. NSLP should provide this data every time it calls
SendMessage primitive. Non-existent Peering-Information-Data
means removal of any existing Peering Information Data from GIST
data structures.
- Peering-Information-Data-Size: Length of Peering Information Data.

RecvMessage ( NSLP-Data, NSLP-Data-Size, Peering-Information-Data,
Peering-Information-Data-Size, NSLPID, Session-ID, MRI, Routing-
State-Check, SII-Handle, Transfer-Attributes, IP-TTL, IP-Distance,
GIST-Hop-Count, Inbound-Interface )

- Peering-Information-Data: Data to support conditional peering
decisions.
- Peering-Information-Data-Size: Length of Peering Information Data.

5. Security Considerations

The peering data is sent in a GIST Query and is unprotected.
Therefore, NSLP nodes that want to include some additional peering
data for the receiver must understand that GIST is unable to hide the
content from third parties. Since the content of a PIO is NSLP-
specific, it is possible to use various encryption keys between NSLP
nodes to protect the content of the PIO from eavesdropping. The
details of any such mechanisms are out of scope of this
specification, and do not affect GIST.

6. IANA Considerations

This specification makes the following request to IANA:

Assign a new object value for the Peering Information object (PIO)
from the GIST object value space.
7. Normative References

[I-D.ietf-nsis-ntlp]


Appendix A. Changes since version -00

Changes from version -00 include, but do not limit to:
  o Several editorial adjustments
  o More accurate terminology in the whole draft, for example "NSLP at the responding node" vs. "receiving peer"
  o More text about when GIST should store or remove stored PIOs
  o More text about MTU issues
  o Added a short section about fallback mechanism

Authors' Addresses

Jukka Manner
University of Helsinki
P.O. Box 68
University of Helsinki FIN-00014 University of Helsinki
Finland

Email: jmanner@cs.helsinki.fi
URI: http://www.cs.helsinki.fi/u/jmanner/

Lauri Liuhto
University of Helsinki
P.O. Box 68
University of Helsinki FIN-00014 University of Helsinki
Finland

Email: lliuhto@cs.helsinki.fi
URI: http://www.cs.helsinki.fi/u/lliuhto/
Full Copyright Statement

Copyright (C) The IETF Trust (2008).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY, THE IETF TRUST AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Acknowledgment

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).