IMPLEMENTATION AND VALIDATION OF THE GENERAL INTERNET SIGNALING TRANSPORT PROTOCOL

Bo Gao

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the degree of Master of Science (Computer Science) at

Concordia University

Montreal, Quebec, Canada

April 2008

©Bo Gao, 2008

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-40936-7
Our file Notre référence
ISBN: 978-0-494-40936-7

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Abstract

Implementation and Validation of the General Internet Signaling Transport Protocol

Bo Gao

The IETF NSIS Working Group was created in 2001 and the WG is trying to standardize an IP signaling protocol using a two-layer signaling protocol suite, which was first introduced by B. Braden. In 2002, Xingguo Song in the High Speed Protocols Laboratory in Concordia University has done a verification using the formal modeling language SDL.

The main objective of the new signaling protocol suite is to overcome all of the drawbacks (Complexity, Scalability, Mobility Support, Messaging Reliability, etc.) that the Resource Reservation Protocol has. It should be applicable not only for Quality of Service but also other signaling application protocols.

In 2003, the NSIS Working Group produced a two layer Framework, which consists of a Transport Layer Protocol and several Signaling Layer Protocols. It also provided a premature but concrete solution for the NSIS Transport Layer Protocol, which is known as the General Internet Signaling Transport Protocol (GIST) now. While we were implementing the GIST, the specification of GIST has evolved into a very well defined Internet-Draft.

The main objective of my thesis is to design and develop an implementation of GIST to verify the GIST protocol stack and at the same time provide our own comments which could be useful to the NSIS WG.

Acknowledgements

I would like to thank Dr. Atwood for his support. His expertise in Networking and Engineering helped me to finish the implementation. His valuable advice encouraged me to finish my thesis. Things I learned both from him and from the project have already started to benefit me and will continue into the future.

Also I would like to thank my parents, my wife and my son. Without their support, my thesis would not be accomplished.

Table of Contents

List of Figures viii

List of Tables x

List of Acronyms xi

Chapter 1 Introduction 1

1.1 Signaling, Quality of Service, RSVP and Its Signaling Model 1

1.2 The Constraints of RSVP 2

1.3 Next Step In Signaling 4

1.4 Motivation and Scope of the Thesis 5

Chapter 2 General Internet Signaling Transport Protocol 7

2.1 The Requirements of NSIS 7

2.2 The Framework Proposed by the NSIS WG 8

2.2.1 Two-Layer Model 8

2.3 The GIST (General Internet Signaling Transport Protocol) 9

2.4 Design Overview of GIST 9

2.4.1 Message Transportation Mode 10

2.4.2 The Message Routing Information 10

2.4.3 GIST Message 11

2.4.4 GIST Peer Relationship 11

2.4.5 Signaling Session 12

2.5 A Simple Example 13

2.6 GIST Message Processing Overview 15

2.6.1 The Interface between GIST and the Signaling Application 15

2.6.2 Message Routing State and Messaging Association State 16

2.6.3 Message Reception, Processing and Transmission 17

2.6.4 Message Routing State and Messaging Association State Maintenance............... 19

Chapter 3 Implementation.. 24

3.1 The Overall Structure of the Project 24

3.2 Timer.. 28

3.2.1 The Delta List of Timer Events 28

3.2.2 The Data Structure Used in GIST Timer... 28

3.2.3 The Timer Posix thread 31

3.2.4 Insert a Timer.. 34

3.2.5 Delete a Timer 36

3.2.6 Clear All the Timers for a State Machine... 38

3.3 State Machine... 39

3.3.1 Querying Node State Machine 41

3.3.2 The Responding Node State Machine 43

3.3.3 The Messaging Association State Machine.. 45

3.4 State Machine Implementation 47

3.4.1 The GIST Message.. 49

3.4.1.1 GIST Message Type 50

3.4.1.2 Common Header... 53

3.4.1.3 TLV Objects.. 54

3.4.2 The Pseudo Code of the Event Distributor.. 56

3.4.3 The Implementation of *gistdemux* 61

3.4.4 The Implementation of the Querying Node State Machine................................... 62

3.4.5 The Implementation of Responding Node State Machine...................................... 69

Chapter 4 Validation.. 74

4.1 The Test Tool Designed 74
4.2 A Brief Introduction of Our GIST Implementation 79
4.3 Testing in a Networking Environment 79
4.3.1 Testing With Four Hops and Two Flows without Mapping Table 80
4.3.2 Testing with Flow-Next Hop Mapping Table 81
4.4 Normal Message Delivery and State Refreshing 84
4.5 Messaging Association Multiplexing 85
4.6 Validation of Soft State Timers 88
4.6.1 MA_Connect Timer Validation 88
4.6.2 Refresh_QNode Timer Validation 89
4.6.3 No_Response Timer Validation 92
4.6.4 Inactive_QNode Timer Validation 95
4.6.5 Expire_RNode Timer Validation 97
4.6.6 No_Confirm Timer Validation 99
Chapter 5 Conclusion and Future Work 102
5.1 Conclusion 102
5.2 Future Work 102

List of Figures

Figure 1: NSIS Protocol Stack 9

Figure 2: A Simple GIST Example 13

Figure 3: Message Sequence at State Setup 21

Figure 4: Overview of the Project 25

Figure 5: Querying Node System Architecture 26

Figure 6: Responding Node System Architecture 27

Figure 7: A Delta List 28

Figure 8: Insert a Timer 36

Figure 9: Delete a Timer 38

Figure 10: Querying Node State Machine 43

Figure 11: Responding Node State Machine 45

Figure 12: Messaging Association State Machine 47

Figure 13: Revised Querying Node State Machine 49

Figure 14: Ping Objects contained in NSLP Object 77

Figure 15: GIST Test without Flow-Next Hop Mapping Table 80

Figure 16: GIST test with Flow-Next Hop Mapping Table 82

Figure 17: Normal Message Delivery and Soft State Refreshing 85

Figure 18: Messaging Association Multiplexing 87

Figure 19: MA_Connect Timer 90

Figure 20: Refresh_QNode Timer 91

Figure 21: No_Response Timer in the Awaiting Response State 92

Figure 22: No_Response Timer in the Awaiting Refresh State 94

Figure 23: Inactive_QNode Timer 96

Figure 24: Expire_RNode Timer Time-out at Established State 98

Figure 25: Expire_RNode Timer Time-out at Awaiting Refresh State 99

Figure 26: No_Confirm Timer Time-out at Awaiting Refresh State................................... 101

Figure 27: Revised Messaging Association State Machine 104

List of Tables

Table 1: List of Events for GIST..40

List of Acronyms

ATM	Asynchronous Transfer Mode
GIST	General Internet Signaling Transport
IETF	Internet Engineering Task Force
LSP	Label Switching Path
MA	Messaging Association
MPLS	Multi Protocol Label Switching
MRI	Message Routing Information
MRM	Message Routing Method
MTU	Maximum Transfer Unit
NAT	Network Address Translation
NLI	Network Layer Information
NSIS	Next Steps in Signaling
NSLP	NSIS Signaling Layer Protocol
NTLP	NSIS Transport Layer Protocol
POSIX	Portable Operating System Interface
QoS	Quality of Service
RAO	Router Alert Option
RSVP	Resource Reservation Protocol
SDL	Specification and Description Language
SIP	Session Initial Protocol
TLV	Type Length Value
WG	Working Group

Chapter 1

Introduction

1.1 Signaling, Quality of Service, RSVP and Its Signaling Model

In the telecommunication and networking area, signaling refers to the information exchange concerning the establishment and control of a connection and the management of the network, in contrast to the user information transfer. For example, Signaling System 7 is used to set up the telephone calls rather than for transferring the user voice data. SIP (Session Initiation Protocol) is used to set up Internet telephone calls, while the voice data is not transported by SIP itself.

Achieving Quality of Service (i.e., providing better services to source data flows) requires resource reservation control mechanisms. Quality of Service can provide different priority to different users or data flows, or guarantee a certain level of performance to a data flow in accordance with requests from the application program or the internet service provider policy. Quality of Service guarantees are important if the network capacity is limited, for example voice over IP and IP-TV, since these often require fixed bit rate and are delay sensitive.

RSVP (Resource Reservation Protocol) [RFC2205] is a protocol that has been used by hosts to request a specific QoS (Quality of Service) from the network and could be classified as an Internet signaling protocol. RSVP can make resource reservations for both unicast and multicast applications. The flow sender advertises the traffic characteristics along the unicast or multicast routes, which are provided by routing protocols, to the flow receiver using a "Path" message. These "Path" messages will set up "Path" state in each node along the way. The Path state includes at least the unicast IP address of the previous hop node, which is used to route the "Resv" message hop-by-hop in the reverse direction. The flow receiver will initiate the Reservation Request by sending out "Resv" messages toward the flow sender

once it receives the "Path" message. These "Resv" messages will install the "Resv" state to reserve the resources along the path.

Inside every RSPV aware node, RSVP communicates with two local decision modules: admission control and policy control. Admission control determines whether the node has sufficient available resource to supply the requested QoS. Policy control provides the authorization for QoS request. If both checks succeed, the RSVP module sets parameters in a packet classifier and packet scheduler to obtain the desired QoS.

1.2 The Constraints of RSVP

RSVP has been proved to be a very well designed protocol. It can support end-to-end resource reservation for both Unicast and Multicast optimally. The design of RSVP distinguished itself by a number of fundamental ways; particularly, soft state management, two-pass signaling message exchanges, receiver-based resource reservation, and separation of QoS signaling from routing [2]. The soft state approach can grantee that in the absence of the refresh messages, the Resv and Path states can time out automatically and be deleted. When the receiver issues the Resv request, the request will propagate to all of the routers in the reverse direction of the data path. In this way, the RSVP requests could be merged, resulting in a protocol that scales well when there are a large number of receivers [2]. RSVP is not a routing protocol. It is using current and future routing protocol to send packets and RSVP consults the local routing table to obtain routes.

However since it was published, many extensions have been added to it. The following are some examples of RSVP extensions.

- DiffServ Interface

RFC2996 introduces a DCLASS Object to carry Differentiated Services Code Points (DSCPs) in RSVP message objects [2].

- ATM interface

[RFC2379] and [RFC2380] define RSVP over ATM implementation guidelines and requirements to interwork with the ATM (Forum) UNI 3.x/4.0 [2].

- MPLS Traffic Engineering

RSVP-TE specifies the core extensions to RSVP for establishing constraint-based explicitly routed LSPs in MPLS networks using RSVP as a signaling protocol. RSVP-TE is intended for use by label switching routers (as well as hosts) to establish and maintain LSP-tunnels and to reserve network resources for such LSP-tunnels [2].

- Simple Tunneling

[RFC2746] describes an IP tunneling enhancement mechanism that allows RSVP to make reservations across all IP-in-IP tunnels, basically by recursively applying RSVP over the tunnel portion of the path [2].

- IPsec Interface

RSVP can support IPsec on a per-address, per-protocol basis instead of on a per-flow basis. [RFC2207] extends RSVP by using the IPsec Security Parameter Index (SPI) in place of the UDP/TCP-like ports [2].

- Refresh Reduction

[RFC2961] describes mechanisms to reduce processing overhead requirements of refresh messages, eliminate the state synchronization latency incurred when an RSVP message is lost, and refresh state without the transmission of whole refresh messages [2].

Some new deployments such as RSVP-TE are totally unrelated with what RSVP was originally designed for [2]. RSVP's transport mechanism, performance, security and mobile capability need to be reevaluated. With more and more extensions added to it, some constraints are emerging. The following are some specific constraints faced by RSVP.

- Complexity

RSVP was originally designed for multicast networks. In order to adapt to multicast network, reservation style, scope object, and blockade state are used in the basic protocol. RSVP not only needs to deal with QoS message transportation but also needs to deal with the entire QoS message processing (e.g., communication with Policy Control Unit, Admission Control Unit, Packet Classifier and Packet Scheduler).

- Scalability

RSVP suffers from the scalability problems as it is a per-flow based protocol and the number of states is proportional to the number of RSVP sessions. Path and Resv states have to be maintained in each RSVP router for each session.

- Mobility

In mobile networks, the movement of a mobile node may not properly trigger a reservation refresh for the new path. Therefore, a mobile node may be left without a reservation up to the length of the refresh timer [2].

- Message Reliability

RSVP does not have a good message delivery mechanism. If a message is lost on the wire, the next re-transmit cycle by the network would be one soft-state refresh interval later. By default, a soft state refresh interval is 30 seconds. Also, it does not separate the transport functions from protocol processing. So a lot of effort has to be spent on per-session timer maintenance, message retransmission and message sequencing.

- MTU Problems

An RSVP message must be fit entirely into a single non-fragmented IP datagram [2]. Routers simply can not detect and process RSVP message fragments as IP packet reassembly can only be done at the flow receiver.

1.3 Next Step In Signaling

The Next Steps in Signaling Working Group was formed in 2001 and is trying design an IP signaling protocol for signaling information about a data flow along its path in the network. The new IP signaling protocol could be used for Quality of Service and also for other signaling applications.

The NSIS working group is concentrating on a two-layer signaling paradigm. The idea of using two-layer architecture was first brought by B. Braden [6]. A NTLP (NSIS Transport Layer Protocol) will transport the NSLP (NSIS Signaling Layer Protocol). This two-layer model then can be used to separate the transport of a signaling protocol from application signaling. It will allow for a more general signaling protocol to be developed to support

signaling for different services. In this way, modularity can be achieved. Also the extension of signaling application protocol could be easily done without affecting the lower layer transport protocol.

The lower layer transport protocol will try to use the current internet transport layer protocol such as TCP. In this way, the NSIS protocol suite will be relieved from managing the congestion avoidance, message retransmission and security. Problems such as the MTU problem and the reliable transmission problem can be avoided then also.

The NSIS signaling suite will only consider the unicast flows now and thus will reduce the complexity that the RSVP has.

The NSIS also take the mobility environment into consideration. Specifically, not like in RSVP, where the flow is 1:1 mapping with session. NSIS could allow the update of the flow:session mapping. A new flow can be added to a session and an old flow can be deleted from it, effectively transferring the network control state between data flows to keep it associated with the same session.

The Framework of NSIS proposed by NSIS WG can be found at [1].

1.4 Motivation and Scope of the Thesis

In 2002, Xingguo Song in High Speed Protocols Laboratory in Concordia University has done a verification using formal modeling language SDL for the two-layer protocol architecture. After that, the NSIS WG group proposed the Internet-Draft for NSIS Transport Layer Protocol, which is known as GIST (General Internet Signaling Transport) protocol. A lot of concepts needed to be verified and clarified against the specification. Questions such as how good the GIST specification is and how difficult it will be if GIST is to be implemented are a particular interest of the NSIS WG.

The main objective of my thesis is to design and develop an implementation of GIST to verify the GIST protocol stack including the state machines proposed by NSIS WG and to propose suggestions on GIST development.

The thesis project consists of four parts:

- Explanation of the NSIS framework and the GIST Internet Draft

- Design of the software architecture for implementation of GIST including timer and state machine
- Development of an implementation of GIST using C/Linux
- Validation of various GIST scenarios including various timers based on our implementation

Chapter 2

General Internet Signaling Transport Protocol

2.1 The Requirements of NSIS

Bearing all of the problems of RSVP in mind, the NSIS WG analyzed the requirement of the next generation internet signaling protocol and proposed a framework with a two-layer protocol architecture. The following are the basic requirements for the Next Generation Internet Signaling Protocol Suite.

- NSIS must be designed modularly

NSIS should be able to work over any kind of network. NSIS should be extensible in the future with different add-ons for different environments or scenarios. It will not only support QoS but also it should be easily extended to support other signaling applications such as NAT and firewall control.

- NSIS should provide availability information on request

NSIS SHOULD provide a mechanism to check whether state to be set up is available without setting it up. In some scenarios, e.g., the mobile terminal scenario, it is required to query whether resources are available, without performing a reservation on the resource [3].

- The NSIS entities must be allowed to placed anywhere in the network.

The protocol must be able to work in any scenarios such as host-to-network-to-host, edge-to-edge, and network-to-network. In the contrast, the standard RSVP can only work end-to-end.

- Automatic Release of State after Failure must be possible [3]

RSVP is using a soft-state timer to automatic release useless states. NSIS has to do the same thing to prevent the stale states within the network while adding the robustness.

- NSIS states must be addressed independently of Flow Identification.

Various scenarios in the mobility area require this independence because flows resulting from handoff might have changed end-points, etc., but still have the same service requirements [3].

- NSIS must be scalable.

As NSIS needs to accommodate several signaling applications and also the use of signaling by hosts will become universal, the scalability of state per NSIS entity must be achieved.

2.2 The Framework Proposed by the NSIS WG

In order to achieve a modular solution for the NSIS requirements, the NSIS protocol suite consists of two layers.

2.2.1 Two-Layer Model

The NSIS protocol stack (Figure 1) consists of NTLP (NSIS Transport Layer Protocol) and a number of NSLP (NSIS Signaling Layer Protocol). The NTLP proposed by NSIS WG is known as GIST (General Internet Signaling Protocol).

The NSIS protocol stack works as follows:

When a message is ready to be sent out, it is passed to NTLP by one NSIS Entity along with what flow it is for, the NTLP then gets it to the next NSIS Entity, where it is received. It is the NTLP's responsibility to find its peer. If there is an appropriate signaling application locally, the receiving NSIS Entity will pass it upwards for further processing. The signaling application then can generate another message to be sent out via NTLP. In this way, the transport service provided by one NTLP instance can be used by many different signaling applications and the modularity will be achieved.

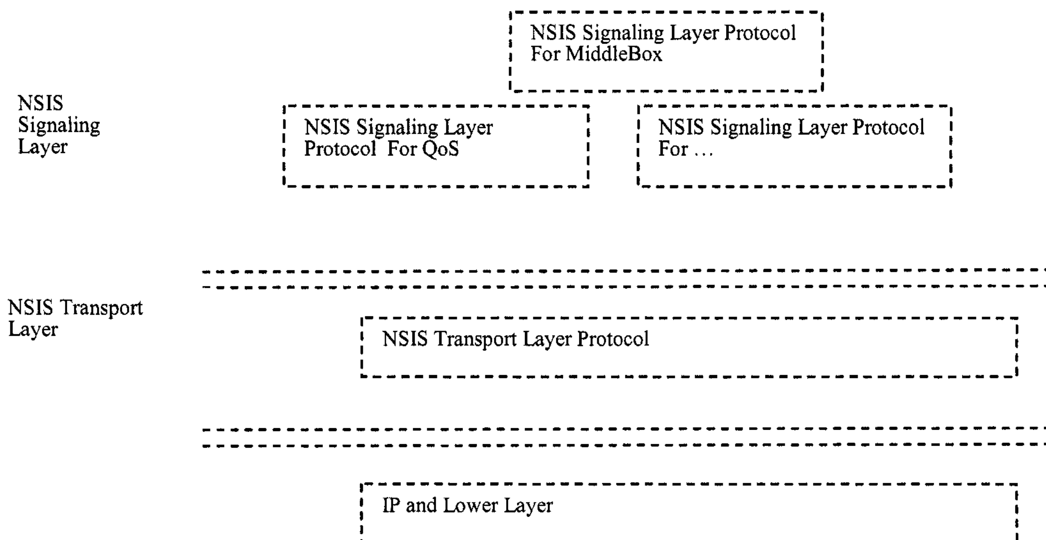


Figure 1: NSIS Protocol Stack

The NTLP on one NSIS Entity does not use any knowledge about addresses, capability or statues of any other NSIS Entity another than its direct peer [1].

2.3 The GIST (General Internet Signaling Transport Protocol)

The Internet Draft [4] gives a concrete solution for NTLP (NSIS Transport Layer Protocol, which is known as GIST). We first give out a design overview of GIST, then a message processing overview.

2.4 Design Overview of GIST

The GIST basically provides the Routing and Transport service to the Upper Layer.

2.4.1 Message Transportation Mode

GIST has two types of operation mode: Datagram Mode (D-Mode) and Connection Mode (C-Mode).

Datagram mode is mainly used for small and infrequent messages. With Datagram mode, the GIST provides an unreliable transportation service to the application layer signaling, which means that the reliability will be taken care of by application layer signaling itself. The Datagram mode is using UDP.

There is a special Datagram Mode, called Q-mode, which is used when no routing state exists. The Q-mode message is usually used for peer discovery (e.g., Messaging Association Setup). It is using an IP raw socket with a Router Alert Option so that the Routers in the data path can intercept the message.

Connection Mode is mainly used for larger messages and can only be used between two nodes with a peer relationship. The connection mode can provide a reliable transportation service for the signaling application layer. Fox example, if the signaling message is so large that it requires fragmentation, it must use C-mode. In order to transfer a signaling message to the peer, the node must have a peer relationship with that peer. The relationship between those two nodes is called a Messaging Association. The Messaging Association is totally internal to the GIST.

GIST usually uses the three-way handshake in D-mode to set up the Messaging Association and then transfers signaling messages with C-mode by the peer connection associated with the Messaging Association.

2.4.2 The Message Routing Information

The baseline message routing functionality in GIST is that signaling messages follow a route defined by an existing flow in the network, visiting a subset of the nodes through which it passes [4]. The GIST design encapsulates the routing-dependent details as the Message Routing Method.

The definition of Message Routing Method includes a MRI specification (Message Routing Information), which includes at least the flow's destination and source address. The MRI

always includes a flag to distinguish the directions the message flow can take, which is either "downstream" or "upstream". The definition of MRM includes a specification of the IP-level encapsulation of the messages, which probe the network to discover the adjacent peers. The signaling message is traveling in the same direction as the flow defined by the MRI. The MRI is explicitly passed between signaling applications and the GIST. The signaling applications must define which MRM they require. Signaling applications may use fields in the MRI in their packet classifiers; if they use additional information for packet classification, this would be carried at the NSLP level and so would be invisible to GIST. Any node hosting a particular signaling application needs to use a GIST implementation that supports the corresponding MRMs [4].

2.4.3 GIST Message

GIST defines six types of messages, which are Query, Response, Confirm, Data, Error and MA-Hello. All signaling application data are carried in the payload of those messages. The Query, Response and Confirm are used to set up the Routing State and Messaging Association by Handshake. The Query Node will initiate the handshake by sending out the Query with an IP raw socket, which is usually including a Router-Alert Option in the IP header. The correct peer will intercept this Query and becomes the Responding Node. A Query always triggers a Response in the reverse direction as the second message of the 3-way handshake. The Responding Nodes usually install the Routing State until the Querying Node returns a confirm message.

The Data message is used to carry and deliver the signaling application data. Error messages are used to report errors. The MA-Hello message can be used as a keep-alive for the Messaging Association protocols.

2.4.4 GIST Peer Relationship

Peering is the process whereby two GIST nodes create message routing states that point to each other [4].

The peer relationship is set up by handshake between the Query Node and the Responding Node. From the Querying Node's view, the identity of the Responding Node is not known at

the time the Query is sent. Only after the Query is intercepted by a node along the path, which also decides to send a Response to the Query Node, the peer relationship is then set up. Nodes not hosting the NSLP just forward the Query transparently.

2.4.5 Signaling Session

GIST requires signaling applications to associate each of their messages with a signaling session [4]. A signaling application provides the SID (Session Identifier) to GIST whenever it needs to send out a message and the GIST will deliver the message received to the corresponding signaling application when the received message has an SID that is associated with the signaling application. All messages for the same flow may have the same session ID. Messages for different flows could have the same session ID also. This could be useful in mobile networks. When the Mobile Node moves from one access point to another access point, its IP address may change, but the flow is still there and requires the same service regardless of the different access network. By associating a different flow with the same Session ID, the double resource reservation problem can be resolved

GIST does not perform any validation on how signaling applications map between flows and sessions as the SID is passed by the signaling application. From GIST's view, the SID is only an opaque 128-bit number.

If messages have the same SID and they also need to be delivered reliably, they will be delivered in order. In the routing state table, the triplet (MRI, NSLP, SID) is the key.

2.5 A Simple Example

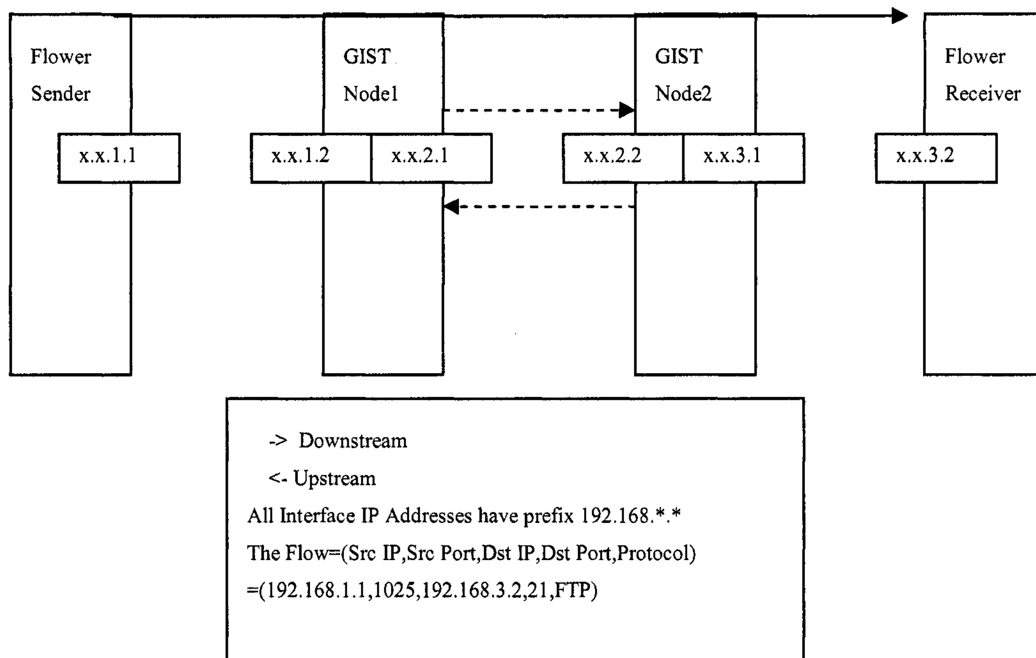


Figure 2: A Simple GIST Example

The following is an example of GIST usage in a relatively simple signaling scenario. The example is different from the one given in [4] as the message needs the reliable delivery service from GIST.

Suppose that the signaling application is a RSVP-like signaling protocol and flow sender 192.168.1.1 wants to reserve the resource for the flow to the receiver 192.168.3.2.Thus the flow is [192.168.1.1, 1025, 192.168.3.2, 21](Source IP, Source Port, Destination IP, Destination Port). The signaling can take place along the entire end-to-end path, while the role of GIST is only to transfer a signaling message over a single segment of the path between neighboring resource-capable nodes [4]. GIST is always triggered by the local signaling application. This example here only describes how the GIST delivers the signaling message between two adjacent nodes along the path. We suppose a signaling message is being processed above the GIST layer by the signaling application in GIST Node1.

1, The signaling application on GN1 determines that this message is a simple description of resources that would be appropriate for the flow. It also determines that the signaling message needs to be delivered to the next downstream signaling application peer on the path with reliability.

2, The message payload is then passed to the GIST layer in GN1, along with a definition of the flow [192.168.1.1, 1025, 192.168.3.2, 21] and description of the message transfer attributes (in this case, requesting reliable transmission). GIST determines for this particular message that a Messaging Association needs to be set up for the message as it needs to be delivered reliably. It then finds a free empty Messaging Association entry and queues the signaling application data at this entry so that the data can be delivered to the signaling application on GIST Node2 reliably once the Messaging Association is set up.

3, GN1 therefore constructs a GIST Query, which will be used to initiate a Messaging Association. The Query is encapsulated in a UDP datagram and injected into the network with a destination address 192.168.3.2. At the IP level, the destination address is the flow receiver, and an IP Router Alert Option (RAO) is also included.

4, The Query passes through the network towards the flow receiver, and is seen by each router in turn as each Router must process message with IP Router Alert Option. GIST-unaware routers will not recognize the RAO value and will forward the message unchanged; GIST-aware routers that do not support the NSLP in question will also forward the message basically unchanged, although they may need to process more of the message to decide this [4].

5, The Query message is going to be intercepted by GIST Node2 finally and is going to try to deliver the NSLP payload (in our case, it is empty) to the NSLP. NSLP will indicate to NTLP on GN2 that a Messaging Association is needed between GN1 and itself. Then, GN2 is going to issur a Response to GIST Node1 (it either can learn the GIST Node1 IP address from the Query as there is a GIST Node1's interface address included in the Query, or can learn it from a system call provided by the Operating System) and inside the Response, it includes the transport layer protocol it wants to use for the Messaging Association (for example TCP) and a listening point that it can accept the TCP request from GIST Node1(for example, its IP

Address and TCP listening port). Both the Query and Response are Datagram-Mode messages.

6, After GIST Node 1 receives the Response, it will take the initiative to set up the Messaging Association. In our case, the GIST Node1 will set up a TCP connection with GIST Node2. This TCP connection will be used for all data messages transferred after. Then GIST Node1 will send out a Confirm message to GIST Node2 to confirm the Messaging Association has been set up. Inside the Confirm message it can include the signaling application Data which is already queued on this Messaging Association. At this point, the three-way handshake has been finished.

7, From then on, all of signaling application messages either delivered from GIST Node1 or GIST Node2 can be transported reliably with this Messaging Association.

2.6 GIST Message Processing Overview

Although we have given a simple example in section 2.5 to see how GIST works, the details involved in this protocol are still worthy of more explanation.

2.6.1 The Interface between GIST and the Signaling Application

GIST is designed to provide a common transportation service to the different signaling application layers. The signaling application's data is transparent to GIST and the difference between Datagram-Mode and Connection-Mode should not be visible at the signaling application. In additional, message fragmentation and reassembly, small message bundling and congestion control should not be visible at the signaling application either.

GIST will take whatever actions are needed to provide the services, which are determined by the message transfer attributes provided by the signaling application. For example, if the signaling application asked for reliable and secure message transfer, GIST will set up a Messaging Association for it. Reliable message transfer means the message must be delivered to the signaling application in the peer in order. If there is a chance that the message was not delivered, an error must be indicated to the local signaling application

identifying the routing information for the message in question [4]. GIST implements reliability by using an appropriate transport protocol within a Messaging Association, so mechanisms for the detection of message loss depend on the protocol in question; for the current specification, the case of TCP is considered.

In contrast, if the signaling application asks for unreliable transfer service, the GIST may transfer the message one time, several times or not at all, with no error indication in any case. The signaling application itself has to have its own mechanism for message reliability.

2.6.2 Message Routing State and Messaging Association State

Message Routing State:

GIST uses a Message Routing State table to maintain message routing state for every flow to process the outgoing message. Each entry in the table tells how the message is to be routed, the session being signaled for, and the signaling application itself.

The Message Routing State table is indexed by a key of triplet:

- Message Routing Information:

It includes the Flow Identifier(for example, flow sender's Source IP and Source Port and flow receiver's Destination IP and Destination Port) , the direction in which to send the message.

- Session ID

The Session ID with which the message should be associated.

- NSLP ID (NSIS Signaling Layer Protocol ID)

This is an IANA-assigned identifier associated with the NSLP that is generating messages for this flow. The inclusion of this identifier allows the routing state to be different for different NSLPs [4].

The routing information associated with a given {MRI, Session ID, NSLP IP} triplet in every entry includes peer identity and a UDP port for a Datagram-mode message or a reference to a MA for a Connection-mode message. Those entries are created by the GIST handshake protocol. Each entry in the routing state table has an associated validity timer for how long it

can be considered accurate; when this timer expires, the entry must be purged if it has not been refreshed.

Messaging Association State:

The GIST Node needs to maintain a Messaging Association State table which keeps all the peer relationships between this node and other nodes. Every entry at least includes:

- A queue of messages that are queued for transmission while an MA is being established.
- A timer that tells how long the state can last before it is refreshed.

2.6.3 Message Reception, Processing and Transmission

The following describes the Message Processing part when a Messaging Association already exists.

Message Reception:

- Message received in Connection-mode:

The Messaging Association will provide the complete message to GIST. The GIST will use the received message to find the corresponding entry in the message routing table and let the state machine stored in the entry deal with the received message.

- Message received in Datagram-Mode:

Just as with the Connection-mode, the GIST will use the received message to find the corresponding entry in the Message Routing table and let the state machine deal with that message.

- Message received in Query-Mode (the special Datagram-mode):

As the Q-mode message is sent out with an IP Router Alert Option with the flow receiver's IP as the destination IP address, it can be seen by each signaling node and be intercepted by the appropriate peers. Unless there is a match between the NSLPID of received message and the local signaling applications, it must be forwarded transparently. The different significance between the RAO and the NSLPID values: the meaning of a message (which signaling application it refers to, whether it should be processed at a node) is determined only

from the NSLPID; the role of the RAO value is simply to allow nodes to pre-filter which IP datagram is a Q-mode GIST message [4].

Message Processing:

If a Query message is received, the GIST must ask the signaling application to see if the signaling application wishes to become a signaling peer with the Querying Node. If the receiving signaling application wishes to set up the Messaging Association with the Querying Node, GIST must continue with the hand-shake to set up message routing state. If the signaling application does not wish to set up routing state with the Querying Node, GIST must propagate the Query and no message is sent back to the Querying Node.

While the receiving signaling application is processing the received signaling message, the GIST on the receiving node can synchronously set up the Messaging Association by handshake with the Querying Node.

For all messages other than Query, if the message includes an NSLP payload, the message must be delivered locally to the signaling application identified by the NSLP ID [4].

Message Transmission:

Whenever a message is sent out, the GIST must make a decision whether the message must be sent in C-mode or D-mode.

If the signaling application has requested reliable delivery the signaling message must sent out in Connection-mode.

If the size of a GIST message (including the IP header, UDP header, GIST header, GIST objects and any NSLP payload) exceeds a fragmentation-related threshold, it must be sent over Connection-mode, using a Messaging Association that supports fragmentation and reassembly (e.g., TCP). In the current GIST specification, the Datagram-mode message size must not exceed the least of the following three quantities: the Path MTU to the next peer, the first-hop MTU and 576 bytes.

For the connection-mode, GIST must not deliver messages for the same session over multiple Messaging Associations in parallel. GIST will queue the message to the Messaging Association first if there is an appropriate Messaging Association found from the routing state table. If there is no appropriate Messaging Association, the message must be queued while one is created.

If GIST has decided to send messages in Datagram-Mode and also the routing state already exists, the message will be sent out with the normal D-mode encapsulation directly to the address from the routing state table. If the message is a Query, the message will be sent out using the raw IP socket with a Router Alert Option.

Node Not Hosting the NSLP:

If Nodes receive messages where they have no signaling application corresponding to the message NSLPID, the messages must be forwarded transparently.

2.6.4 Message Routing State and Messaging Association State Maintenance

The main responsibility of GIST is to manage the routing state and Messaging Associations. Routing state is installed and refreshed by GIST handshake messages [4]. Messaging Associations are set up by the normal procedures of the transport and security protocols that comprise them, using peer IP addresses from the routing state [4]. Once a Messaging Association has been created, its refresh and expiration can be managed independently from the routing state [4].

Routing State and Messaging Association Creation:

A complete sequence of message exchange for GIST Routing state and Messaging Association state setup is shown in Figure 3.

The initial message in any routing state maintenance operation is a Query sent from a Querying Node. There is a router alert option so that the message can be intercepted at the Responding Node. The MRI (Message Routing Information), SID (Session ID) and NSLPID are identifiers for the flow, the session and the NSLP protocol. This triplet can be used to index the message routing table. Query-Node Network Layer Information consists of the

Query Node's interface IP address and UDP port number, which could be used by the Responder to send the Response back.

Once a Query is received, the Responding Node must return a Response. The Response also includes the Network Layer Information of the Responding Node, which could be used by the Query Node to see if the Responding Node is already a known peer and to see if there is a Messaging Association that is already set up with this peer and if that Messaging association could be reused. If it is a new peer and also a Messaging Association is needed for this peer, the Response must include a protocol stack for the possible Messaging Association. Through the exchange of the stack-protocol and stack-config-data objects by Query and Response, the Querying Node and Responding Node can make an agreement which transportation protocol and transport layer secure protocol can be used in the Messaging Association setup. For the current specification, only TCP and TLS are considered. The Querying node will always take the initiative to set up the Messaging Association once the Querying node and the Responding node have made the agreement.

After a Messaging Association has been set up, a Confirm message must be sent out by this Messaging Association. At this point, we can say the Messaging Association for downstream has been set up. The association can also be used in the upstream direction for the MRI and NSLPID carried in the Confirm, after the Confirm has been received [4].

The Querying node must install the Responder address, derived from the R-Node Network Layer info, as routing state information after verifying the Query Cookie in the Response [4]. The Responding node may install the Querying Node's address as peer state information at two points in time:

1. After the receipt of the initial Query, or
2. After a Confirm containing the Responder Cookie.

The Responding node should derive the peer address from the Q-Node Network Layer Information if this was decoded successfully [4]. Otherwise, it may be derived from the IP source address of the message.

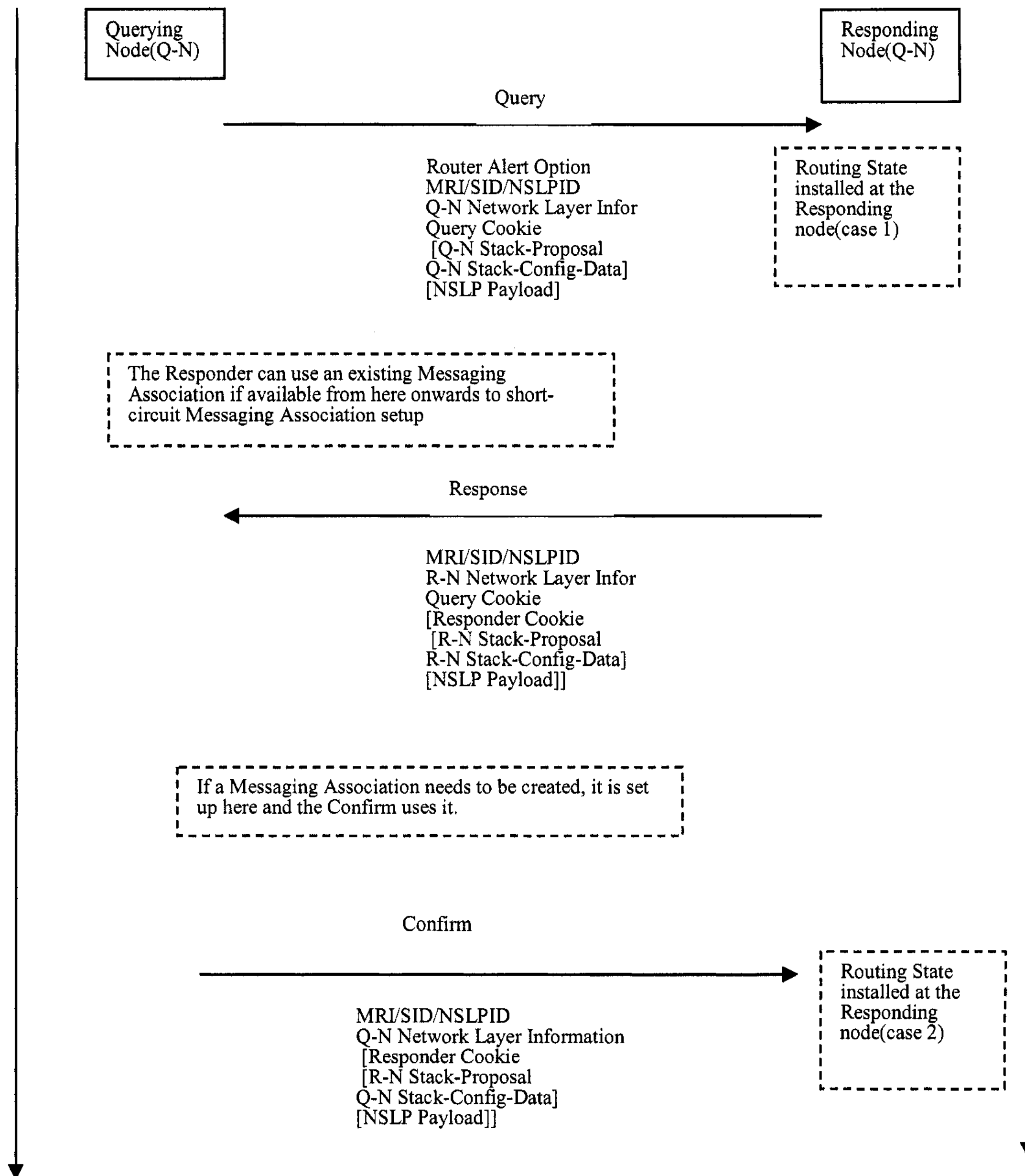


Figure 3: Message Sequence at State Setup

Messaging Association Multiplexing:

A Messaging Association between two peers could be reused by different flow and sessions. It is called Messaging Association multiplexing. Multiplexing ensures that the MA cost

scales only with the number of peers, and avoids the latency of new MA setup where possible [4].

Messaging Association multiplexing is achieved by the Network-Layer-Information (NLI) object, which is carried by Query, Response and Confirm messages.

The Network-Layer-Information consists of:

- Peer-identifier:

For a given node, this is an interface independent value with opaque syntax. It must be chosen so as to have a high probability of uniqueness across the set of all potential peers, and should be stable as least until the next node restarts [4]. For routers, the Router-ID, which is one of the router's IP addresses, may be used as one possible value for the Peer-identity.

- Interface Address:

This is an IP address through which the signaling node can be reached [4]. The Messaging Association is associated with the NLI object that was provided by the peer in the Query, Response and Confirm at the time when the Messaging Association was first set up.

Querying Node and Responding Node will check to see if there is a matching Peer-ID and Interface address in the Messaging Association table after they first time receive the Response, Query and Confirm. If they find that there is a matching one, a new Messaging Association would not be set up. Rather, the Querying Node and the Responding node will complete the rest of three-way handshake with this existing messaging-association.

Routing State Maintenance:

Each item of routing state expires after a lifetime that is negotiated during the Query/Response/Confirm handshake [4]. The Network Layer Info (NLI) object in the Query contains a proposal for the lifetime value, and the NLI in the Response contains the value the Responding node requires [4]. A default timer value of 30 seconds is recommended [4]. In our implementation the Inactive_QNode and the Expire_RNode are the timers for Querying Node and Responding Node respectively. The Querying Node must insure that a Query is received before the Expire_RNode timer expires. Otherwise the Responding node may delete

the Routing State. Receiving the Query and Confirm will refresh the routing state at the Responding Node for the corresponding flow, while receiving the Response will refresh the routing state at the Querying Node for the corresponding flow. There is no mechanism at the GIST level for explicit teardown of routing state [4]. However, GIST must not refresh routing state if a signaling session is known to be inactive, either because upstream state has expired, or because the signaling application has indicated via the GIST API that the state is no longer required.

In [4], it also recommends that the refreshing handshake happens when between ½ and ¾ of the routing state validity time has elapsed since last successful refresh. In our implementation, the refreshing timer is defined by the Refresh_QNode, which is 15 seconds.

Messaging Association Maintenance:

Messaging association state is another state kept by both Querying Node and Response Node. There is a timer associated with this state, which is called the MA-Hello timer. During the Messaging Association setup, the Querying Node and Responding node exchange their own MA-Hold-Time timer as part of the Stack-Configuration-Data. A node may tear down that Messaging Association if it has not received any traffic from its peer. A node that wishes to keep the Messaging Association can send the MA-Hello to indicate this. In [4], 30 seconds for MA-Hold-Time is recommended. Messaging association is not visible outside of the GIST layer, therefore, even if GIST tears down and later re-establishes a Messaging Association, signaling applications can not distinguish this from the case where the MA is kept permanently open [4].

Chapter 3

Implementation

3.1 The Overall Structure of the Project

Our implementation is based on Linux and coded in C language as Linux is a POSIX-compliant operating system. Our implementation is used for only validation right now and could be extended to a full version in the future (please see Chapter 5: Conclusion and Future Work) as the timers and state machine can be reused. We are using two threads for GIST. One thread (Event Distributor) is responsible for collecting all messages from timer pipe, from API pipe and from all TCP sockets in socket bank of the Messaging Association table, IP raw socket (UDP socket in our case) and distribute all messages to different Querying Node and Responding Node state machines.

As an IP raw socket is difficult to access in our environment, we are using a UDP socket instead. It wouldn't have any effect to our validation and also could be easily replaced with the IP raw socket in the future.

The event distributor is using the "*int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)*" system call with 500 ms as the timeout parameter to read the messages and event. Then, it calls function "*mrs_t * gistdemux(const gist_msg_s *)*" to find the corresponding Querying/Responding Node state machine and passes the received message to the corresponding state machine to deal with it according to the state stored in the state machine.

Another thread (Timer Manager) is responsible for the timer management, which always checks the most front element from a delta list and writes a message to the timer pipe if it finds that some events have timed out. It is using one Mutex(*pthread_mutex_t mutex*) to protect the link-list. Those two threads are using a pipe for message passing. The timer management thread is writing to the pipe and the Event Distributor is reading the time-out event from the pipe and the message is flowing only in one direction from timer management thread to the event distributor. Therefore, only one pipe is needed.

As both PIPE and FIFO are file descriptors in Linux system, there is no difference between a pipe and the socket. So the event distributor can use the *select* system call, which can do the I/O on multiple file descriptors simultaneously.

We also designed a very simple NSIS signaling application protocol for simulation and validation, which is called NSIS Ping. The NSIS Ping is communicating with the GIST by Posix FIFO. The NSIS Ping is only collecting the IP address of the nodes it passed by.

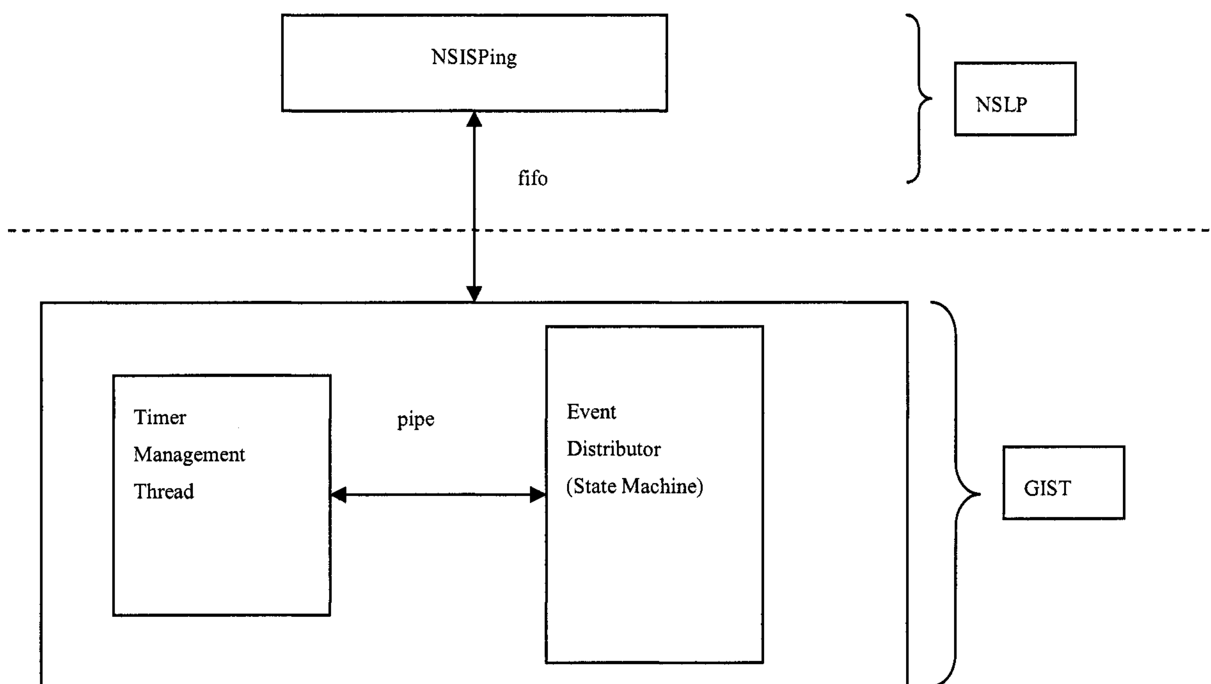


Figure 4: Overview of the Project

The GIST event distributor is not only monitoring the PIPE and the FIFO but also other UDP sockets and TCP sockets. Every Messaging Association has a TCP socket and the GIST also has one UDP socket for sending and receiving Query and Response. The Responding Node also has a TCP socket as the listening point. So the set of file descriptors that the *select()* is

monitoring includes FIFO, PIPE, TCP sockets for Messaging Association, UDP socket and a TCP socket as listening point if it is a Responding Node.

The overall software architecture for the Querying Node is in Figure 5.

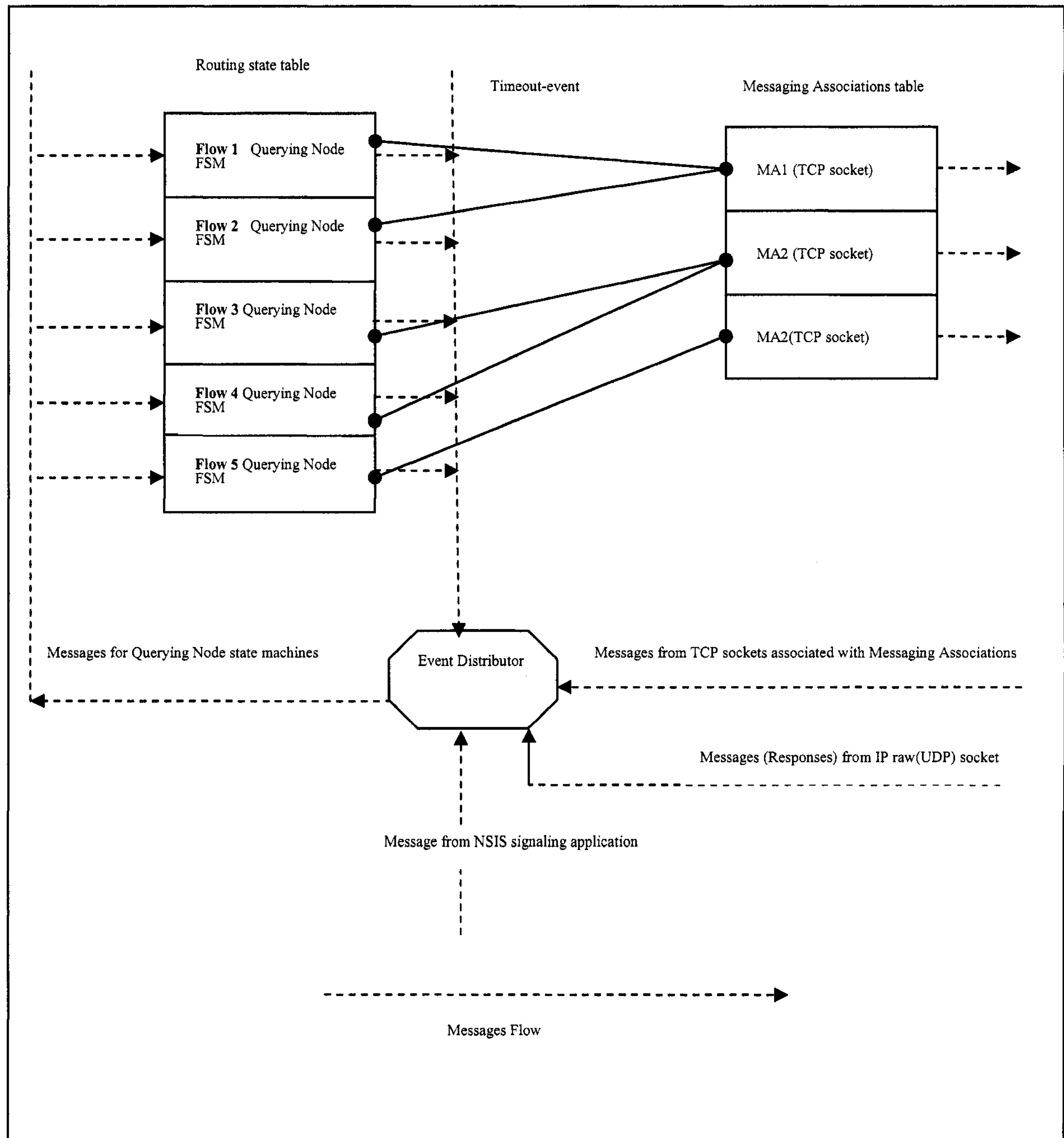


Figure 5: Querying Node System Architecture

From Figure 5, we also can see that one Messaging Association can be shared by different Querying Node state machines.

The System Architecture for Responding Node is shown in Figure 6.

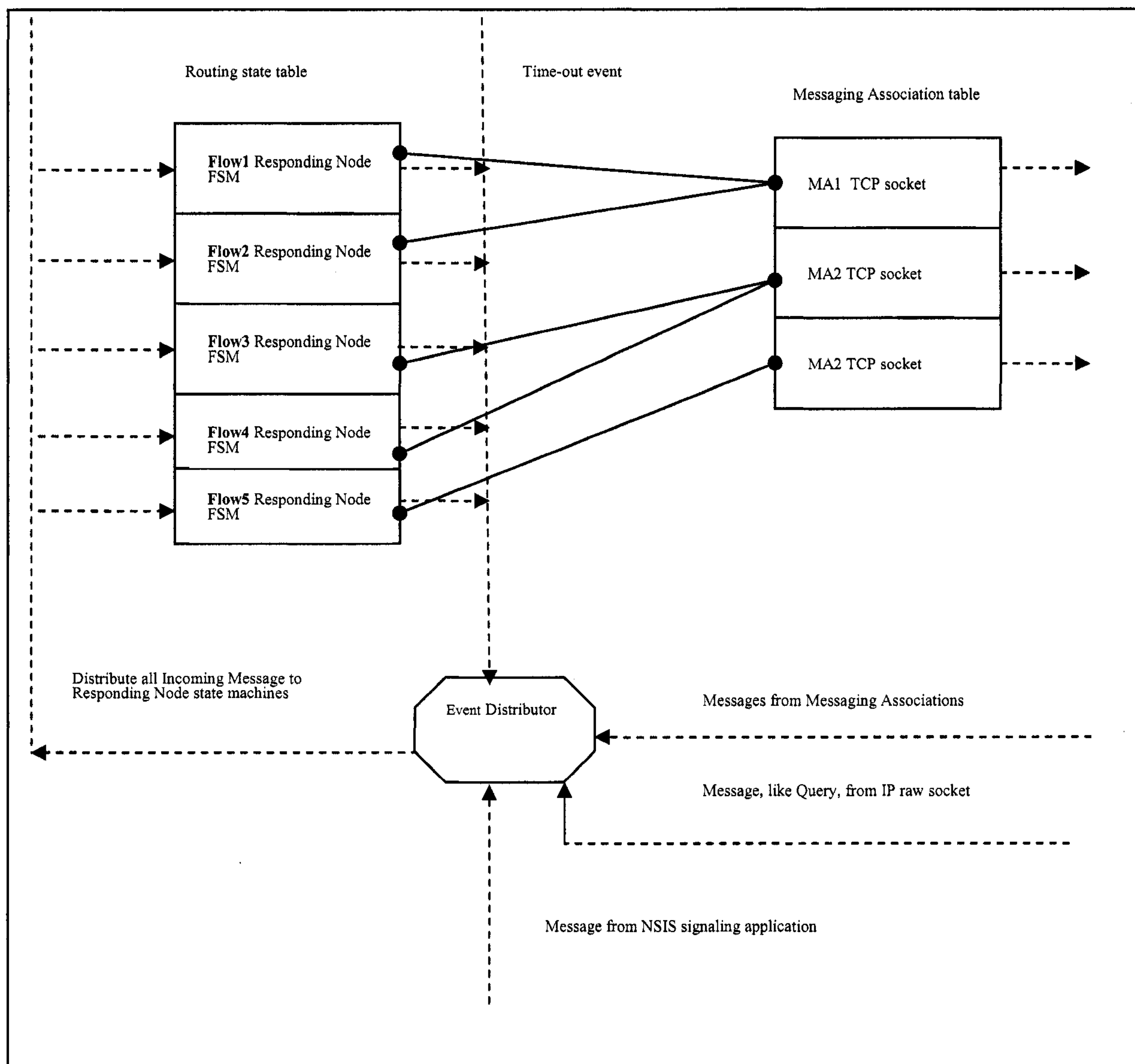


Figure 6: Responding Node System Architecture

As most routers deployed in the current Internet are using UNIX-based operating system, we think that our implementation could be easily extended to an industrial version. The timer management and state machine implementation would be reusable without much modification.

3.2 Timer

As GIST is a soft state protocol, it needs timers to purge the stale routing state entries. Also GIST needs several timers to handle Query and Response retransmission. The timer plays a very important role in the system design. This section shows how a single thread can manage all of the timers. The timer we are using in the project is similar to the one used by TCP. Please refer to the Chapter 14 of [9].

3.2.1 The Delta List of Timer Events

We are using a delta list, which is a linked list, to store all of timers. Each item on the delta list corresponds to a timer event that could expire in the future. In each delta item, there is the *t_timeleft* field, which gives the time at which the event should occur and all the events are ordered by the time at which they will occur. The time stored in *t_timeleft* is relative time not absolute time and is counted as milliseconds. For example, in Figure 7, the delta list contains the events that will occurs at 500 milliseconds, 600 (500+100) milliseconds, 900 (600+300) milliseconds, 1400 (900+500) milliseconds and 1600 (1400+200) milliseconds.

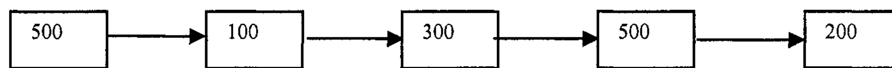


Figure 7: A Delta List

3.2.2 The Data Structure Used in GIST Timer

typedef enum{

TO_No_Response,

TO_Refresh_QNode,

TO_MA_Connect

```
        TO_Inactive_QNode,
        TO_Expire_RNode,
        TO_No_Confirm
}event_type;

#define No_Response 500  /*500 milliseconds*/
#define Refresh_QNode15000  /*15 seconds*/
#define MA_Connect 500  /*500 milliseconds*/
#define Inactive_QNode 30000 /*30 seconds */
#define Expire_RNode 30000 /*30 seconds*/
#define No_Confirm 500 /*500 millisecons*/
typedef struct event_messaging_asscociation
{
        event_type type;
        int mrs_num;
}evt;
typedef struct timer_node{
   long t_timeleft; /*time for it to expire*/
   long t_time;    /*the time when this entry to queued*/
   long t_pipe;    /*pipe to sent the event*/
   evt t_evt;      /*the message itself*/
   struct timer_node * next;
}tm_node;
```

```
typedef struct timer_list
{
   pthread_mutex_t mutex;
   int counter;
   tm_node * tm_head;
}tm_list ;
```

The *event_type* is the type of timers. We have 6 types of timer for both Querying Node and Responding Node state machine (No_Response, Refresh_QNode, Inactive_QNode, MA_Connect, Expire_RNode, No_Confirm).

The timer No_Response is used for Query retransmission in case there is no Response received. No_Response is defined as 500 milliseconds and it will be dynamically increased at the run time using a binary back-off exponential algorithm.

The timer Refresh_QNode is used to indicate that how much time left to send out a Query for refreshing the Responding Node.

The timer MA_Connect is used to wait for the Messaging Association to complete.

The timer Inactive_QNode indicates how long the state machine will wait after no traffic is currently being handled before being refreshed. This is reset whenever the state machine handles an NSLP Data message, in either direction. When it expires, the state machine may be deleted. The period of the timer can be set at any time via the API (SetStateLifetime), and if the period is reset in this way the timer itself must be restarted.

The timer Expire_RNode indicates when the routing state stored by this state machine needs to expire. It is reset whenever a Query or Confirm (depending on local policy) is received indicating that the routing state is still valid. Note that this state cannot be refreshed from the Responding Node.

The timer No_Confirm indicates that a Confirm has not been received in answer to a Response. This is started whenever a Response is sent and stopped when a Confirm is received.

There is a field *mrs_num (message routing state number)* in the *evt* type, it tells which routing state machine created this timer.

Inside the *timer_list* type, there is a Posix Mutex type variable *mutex*. It is used for protecting the delta list and for making certain that only one thread, which could be either event distributor thread or timer thread, can access the delta list at a time. The variable *mutex* is initialized to the constant PTHREAD_MUTEX_INITIALIZER. The variable *mutex* can be locked and unlocked by following functions:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

If we try to lock a Mutex that is already locked by some other threads, *pthread_mutex_lock* is going to block until the Mutex is unlocked *by pthread_mutex_unlock.*

3.2.3 The Timer Posix thread

```
int thread_timer()

{

        long now,lastrun;

        long delta;

        tm_node* tm_ptr;

        int old_counter;

        tm_ptr=t_list.tm_head;

        lastrun=get_time();

        for(;;)

        {
```

```
usleep(SLEEPTIME*1000);
Pthread_mutex_lock(&t_list.mutex);
if(t_list.tm_head==NULL)            {
        lastrun=get_time();
        Pthread_mutex_unlock(&t_list.mutex);
        continue;
}
now=get_time();
delta=now-lastrun;
lastrun=now;
old_counter=t_list.counter;
while((t_list.tm_head!=NULL)&&(t_list.tm_head->t_timeleft <=delta))
{
        delta=delta-t_list.tm_head->t_timeleft;
        /* a timer expires, we write that event to the timer pipe*/
        Write(timer_pipe[1],&t_list.tm_head->t_evt,sizeof(evt));
        tm_ptr=t_list.tm_head;
        t_list.tm_head=t_list.tm_head->next;
        free(tm_ptr);
        t_list.counter--;
}
if(t_list.tm_head)
        t_list.tm_head->t_timeleft-=delta;
Pthread_mutex_unlock(&t_list.mutex);
```

```
    }
}
```

The timer thread is an infinite loop. Inside each run of the loop, the thread is sleeping 50 milliseconds and then it will lock the variable *mutex* for exclusive access of the delta list. In each iteration, it will also compute the elapsed time from last run by subtracting the value of *lastrun* from the current time, which is gotten from function *get_time()*.

As long as the delta list is not empty, *thread_timer* will go to process the items on the delta list. First the *thread_timer* compares the time remaining for the first item to expire with the time that has elapsed. If it finds that the first item should have expired, it subtracts the time left of that item from the *delta*, writes the event to the timer pipe and frees that item. The second item will become the first item. Then it is going to process it. Variable *delta* always contains a relative time, so *timer_thread* can compare it directly to the time stored in the individual item.

When *timer_thread* finishes removing items that have expired, the delta list could be empty. If the delta list is empty, no further processing is needed. However, if the delta list is not empty, the time remaining for next item to expire must be greater than *delta*. In such cases, *timer_thread* reduces the time of the remaining item by *delta* before beginning the next cycle of the delay.

```
long get_time()
{
    struct timeval t;
    if(gettimeofday(&t,NULL)==0)
        return(t.tv_sec*1000+t.tv_usec/1000);
    else err_sys("gettimeofday error");
  return -1;
}
```

The *get_time* will return the current system time in milliseconds.

3.2.4 Insert a Timer

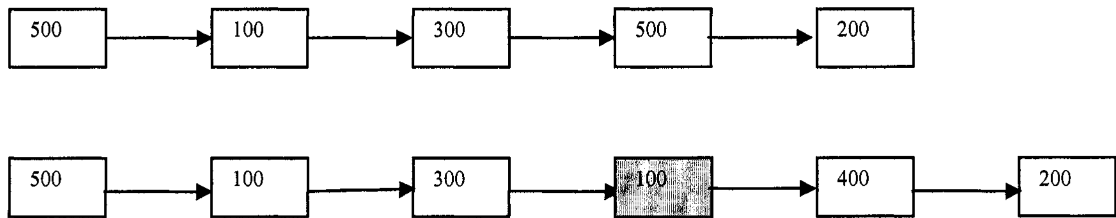
```
int timer_insert(const evt * e,long timeleft)
{
        /*node will be inserted into between prevp and p*/
        tm_node * nodep, * prevp,* p;
        nodep=prevp=p=NULL;
        nodep=(tm_node *)malloc(sizeof(tm_node));
        nodep->t_timeleft=timeleft;
        nodep->t_time=get_time();
        memcpy(&nodep->t_evt,e,sizeof(evt));
        nodep->next=NULL;
        timer_clear(e);
        Pthread_mutex_lock(&t_list.mutex);
        t_list.counter++;
        if(t_list.tm_head==NULL)
        {
                t_list.tm_head=nodep;
                Pthread_mutex_unlock(&t_list.mutex);
                return 1;
        }
        for(prevp=NULL,p=t_list.tm_head;p;p=p->next)
        {
        if(nodep->t_timeleft < p->t_timeleft)
```

```
                break;
            nodep->t_timeleft-= p->t_timeleft;
            prevp=p;
    }
    nodep->next=p;
    if(prevp!=NULL)
            prevp->next=nodep;
    else t_list.tm_head=nodep;
    if(p!=NULL)
            p->t_timeleft -= nodep->t_timeleft;
    Pthread_mutex_unlock(&t_list.mutex);
    return 1;
}
```

The *timer_insert* first allocates a new item and then fills all of the fields of that item. The parameter *timeleft* is the time remaining for this item to expire. Then it is going to clear the same type of timer for the same routing state machine. So when we are inserting a timer, we are actually restarting that timer. Then it will lock the *mutex* for exclusively access for the delta list.

Next *timer_insert* must find out where it can insert that item into the delta list. When it passes through the delta list, *timer_insert* subtracts the remaining times of the items on the delta list from the time remaining for the new item to expire. When it finds the time whose remaining time is greater the remaining time of the new item, the loop terminates. It then inserts the node between the *prevp* and *p*. It then subtracts the remaining time on the *p* node to keep the remaining time to be a relative time.

Please see the following example:



Insert a timer that expires 1000 milliseconds later

Figure 8: Insert a Timer

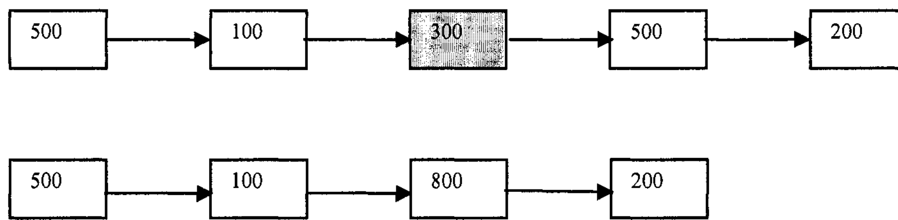
3.2.5 Delete a Timer

```
int timer_clear(const evt * e)
{
        tm_node * prev_tm_nodep, * tm_nodep;
        long timespent;
        Pthread_mutex_lock(&t_list.mutex);
        if(t_list.tm_head==NULL)/* it is empty*/
        {
                Pthread_mutex_unlock(&t_list.mutex);
                return -1;
        }
        prev_tm_nodep=NULL;
        for(tm_nodep= t_list.tm_head;tm_nodep!=NULL;tm_nodep=tm_nodep->next)
        {
```

```
        if((e->type==tm_nodep->t_evt.type)&&(e->mrs_num==tm_nodep-
>t_evt.mrs_num))
        {
                timespent=get_time()-tm_nodep->t_time;
                if(prev_tm_nodep!=NULL)
                        prev_tm_nodep->next=tm_nodep->next;
                else t_list.tm_head=tm_nodep->next;
                /*adjust the time value of next node in the link list*/
                if(tm_nodep->next!=NULL)
                        tm_nodep->next->t_timeleft+=tm_nodep->t_timeleft;
                Pthread_mutex_unlock(&t_list.mutex);
                t_list.counter--;
                free(tm_nodep);
                return timespent;
        }
        prev_tm_nodep=tm_nodep;
    }
    Pthread_mutex_unlock(&t_list.mutex);
    return -1;
}
```

First the *timer_clear* lock the *mutex* for exclusive access to the delta list. Then it will go through the delta list and try to find the desired item to delete. After it finds that desired item, it will update the delta list. It will need to adjust the next node's expire time.

Following is an example to delete a timer:



Delete a timer which expires at 900 milliseconds later

Figure 9: Delete a Timer

3.2.6 Clear All the Timers for a State Machine

```
int timer_delall(const int mrs_num)
{
        evt temp;
        event_type i;
        for(i=TO_No_Response;i<=SEND;i++)
        {
                temp.mrs_num=mrs_num;
                temp.type=i;
                timer_clear(&temp);
        }
        return 1;
}
```

The *timer_delall* is very straightforward and it just deletes all of timers of the desired state machine.

3.3 State Machine

In the GIST specification [4], the authors recommend four different state machines: Node level state machine, Querying Node state machine, Responding Node state machine, Messaging Association state machine.

Node level state machine:

It is responsible for the processing of events that can not be directed towards a more specific state machine, for example, inbound messages for which no routing state currently exists. This machine exists permanently, and is responsible for creating per-MRI state machines to manage the GIST handshake and routing state maintenance procedures [4].

Querying Node State Machine:

It is responsible for sending out Query and Confirm, receiving Response and initiating the state refresh. It is also responsible for delivering and receiving NSIS signaling application protocol data.

Responding Node State Machine:

It is responsible for receiving Query and Confirm, and sending out the Response to finish the three-way handshake state setup. It is also responsible for delivering and receiving NSIS signaling application protocol.

Messaging Association State Machine:

Managing the TCP connection and make sure the TCP connection is not shut down.

The specification of the state machine is using a lot of events which could have prefix rx_, tg_ , er_ or to_. The rx_ represents the incoming messages. The tg_ represents API/lower layer triggers. The er_ represents error conditions. The to_ represents the time out event.

Below is a list of those events, which are specified in [4]:

Table 1: List of Events for GIST

Name	Meaning
rx_Query	A Query received.
rx_Response	A Response received.
rx_confirm	A Confirm received.
rx_MA-Hello	A MA-Hello message has been received (not implemented).
tg_NSLPData	A signaling application has requested data transfer.
tg_Connected	The protocol stack for a Messaging Association has completed connection setup.
tg_RawData	GIST wishes to transfer data over a particular Messaging Association.
tg_MAIdle	GIST decides that it is no longer necessary to keep a Messaging Association open for itself.
er_NoRSM	A "No Routing State" error was received.
er_MAConnect	A Messaging Association protocol failed to complete a connection.
er_MAFailure	A Messaging Association failed.
to_No_Response	A Response has not been received.
to_Inactive_QNode	Time-out event for the Q-Node to be removed as there hasn't been any traffic for a certain time.
to_Refresh_QNode	Timeout event to indicate the routing state should be refreshed.
to_No_Confirm	Time-out to retransmit the Response as Confirm has not been received for a certain of time.
to_Expire_RNode	Timeout event for Responding Node to be removed as it has not been refreshed for a certain time.
to_Send_Hello	Timeout event to indicate that a MA-Hello message should be sent to the peer. (This timer is not implemented in our implementation.)

to_No_Hello	Timeout event to indicate how long it is since the last No_Hello is received from the peer. (This timer is not implemented in our implementation.)
to_MA_Connect	This is a new timer we added in our implementation. It indicates how long it is since the Querying Node has tried to set up the Messaging Association with the Responding Node.

3.3.1 Querying Node State Machine

There are three states in the Querying Node State Machines: Awaiting Response, Established and Awaiting Refresh.

Whenever a signaling application protocol sends a signaling message for a new flow, a Query Node State Machine is created for that flow. Then a Query is sent out and the No_Response timer is started and the Querying Node is transferred to *Awaiting Response* state. If a Response is received, the No_Response timer will be stopped and a Confirm will be sent out to finish the three-way handshake. The Querying Node state machine transfers to the *Established* state. Determined by the message attributes that the signaling application asked for, if there is a need for the Messaging Association, a Messaging Association also will be set up after the Response is received. Also if reliable transportation is need, the data from signaling application will not be sent out until the Messaging Association is set up and signaling application data will be queued for the pending Messaging Association before it is set up. If there is no Response received after the maximum retries of sending Query, the state machine will be deleted for that flow and an error will be reported back to the signaling application for that flow.

Once the Querying Node State Machine is in *Established* state, the Refresh_QNode and Inactive_QNode are started. In this state, data messages from signaling application and Responding Node can be received and sent out normally and also whenever such data messages are received, the Inactive_QNode will be restarted. Any Response received in this state will be ignored. If the Refresh_QNode expires, the state will transfer to *Awaiting Refresh* state and a Query is sent out for refreshing the state machine.

Once the Querying Node state machine is in the *Awaiting Refresh* state, the state machine is waiting for a Response. Same as in the *Established* state, data messages from signaling application and Responding Node can be received and sent out normally and also whenever such data messages are received, the Inactive_QNode will be restarted. If there is no Response received after maximum retries of sending Query, the Querying Node state machine will be deleted. If a Response is received, the state machine will transfer to the *Established* state.

There are three timers related to the Querying Node state machine:

- No_Response

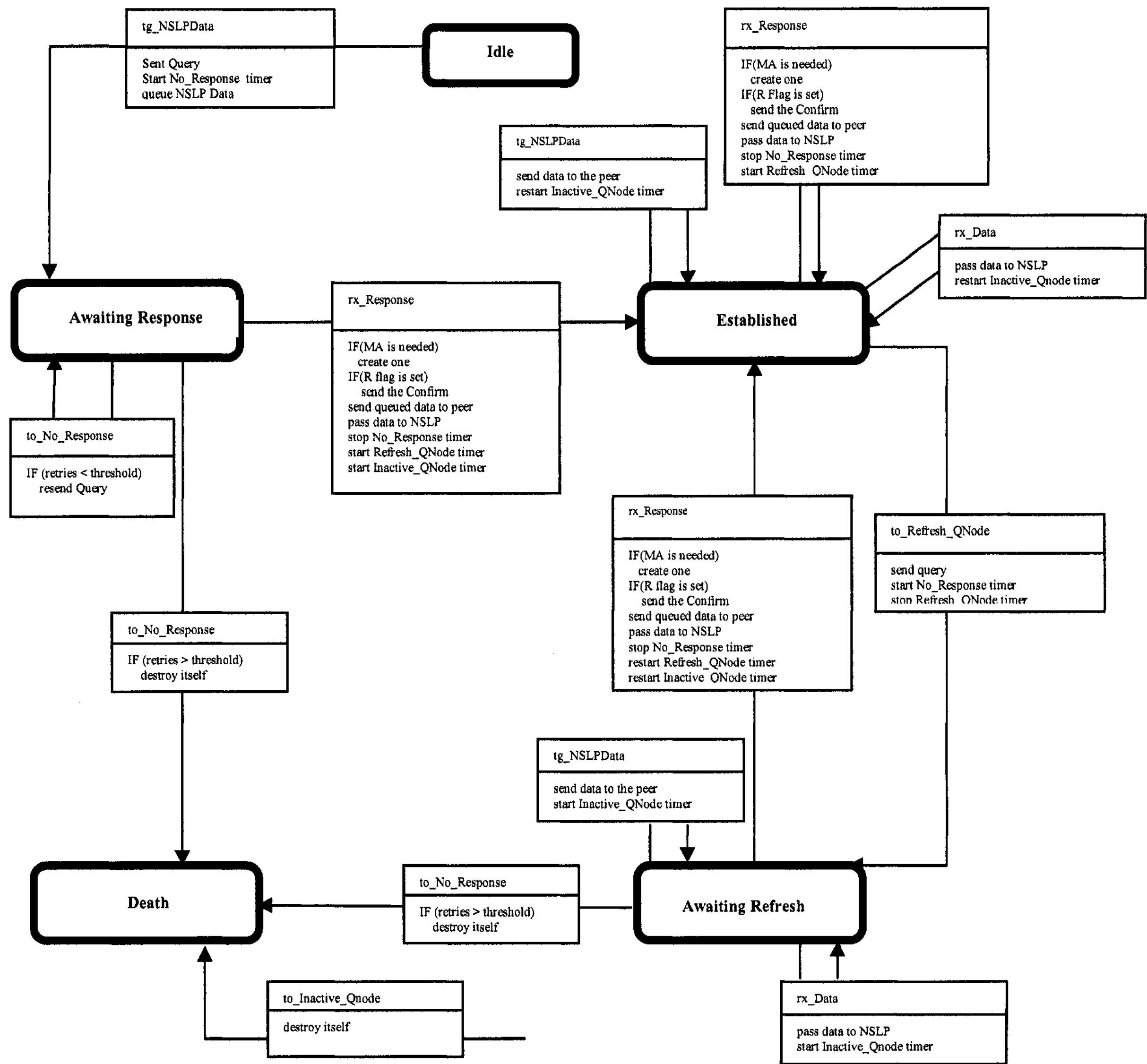
This timer is used to determine when to resend the Query when there is no Response. It is set to 500 milliseconds in our implementation. It is stopped whenever a Response is received.

- Refresh_QNode

This timer is used to determine when to send a Query for refreshing the state machine. It is set to 15 seconds in our implementation. It is restarted whenever a Response is received.

- Inactive_QNode

This timer indicates when the state machine needs to be deleted if there is no traffic being handled by the state machine. When this timer expires, the state machine must be deleted. It is set to 30 seconds in our implementation.



Querying Node State Machine

Figure 10: Querying Node State Machine

3.3.2 The Responding Node State Machine

There are also three states for the Responding Node state machine: *Awaiting Confirm, Established* and *Awaiting Refresh.*

When a Query is intercepted by the Responding Node, depending if a Confirm is needed (if the R: reply flag is set in the Response, that means a Confirm is needed for that Response) or

not, the state machine could transfer either to *Awaiting Confirm* or *Established*. If a Confirm is needed, the state machine transfers to *Awaiting Confirm* state. In the *Awaiting Confirm* state, if a message is received from NSLP, the data will be queued for a connecting MA. If a data message is received from the peer, a "No routing error" is sent back to the peer. If a Query is received from Querying Node, that means the last Response the Responding Node sent out might have been lost and a Response needs to be resent. If a Confirm is received, the state machine transfers to the *Established* state.

In the *Established* state, if a Confirm is received, the Confirm will be silently ignored. If a data message is received from the Querying Node, the data will be passed to NSLP for further processing. If a data message requested for transfer from NSLP, the data will be packed in a GIST Data type message and sent to the peer. It is possible that a Query is received in this state if a Confirm is not required for a Response. If a Query is received, a Response with R = 0 will be sent out and the Expire_RNode timer will be restarted. If a Query is received and a Confirm is also needed for a Response, the state machine will transfer to *Awaiting Refresh* state and the No_Confirm timer is started also.

In the *Awaiting Refresh* state, the data requested for transfer from NLSP will be packed in GIST data type message and sent to peer. The GIST Data message received from the peer will be passed to the NSLP for further processing. If a Confirm is received, the state machine transfers to *Established* state and the No_Confirm timer will be stopped and Expire_RNode timer will be started and any data in the Confirm will passed to the NSLP for further processing.

There are two timers used in the Responding Node State Machine:

- Expire_RNode

It indicates when the routing state should be purged out before it is going to be refreshed by its peer. This timer is set to 30 seconds in our implementation

- No_Confirm

It indicates that there is no Confirm so far being received for the Response. This timer is set to 500 milliseconds in our implementation.

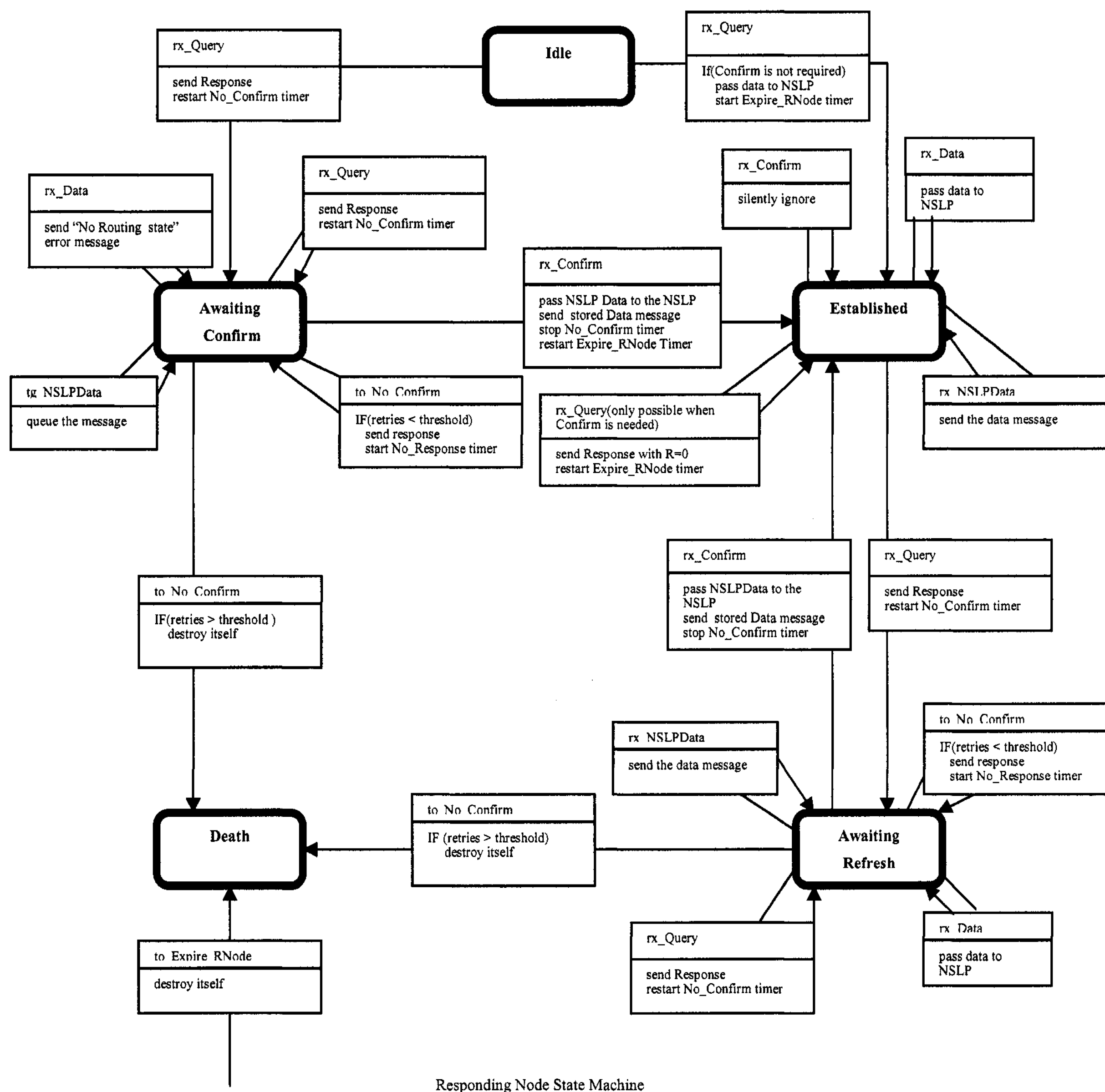


Figure 11: Responding Node State Machine

3.3.3 The Messaging Association State Machine

The Messaging Association state machine has three states: *Awaiting Connection, Connected* and *Idle*. Following is an example of the operations of the Messaging Association State Machine in the Querying node.

After the Messaging Association state machine is created and a connection will be initiated, the state machine transfers to the *Awaiting Connection* state. In this state, any message passed to the Messaging Association for transmission will be queued on it.

After the connection with the peer has been completed, the state machine will transfer to *Connected* state and a SendHello timer will be started.

The *Connected* state indicates that the messaging machine is ready to use. If the SendHello timer expires, a MA-Hello message will be sent out to ensure that the peer does not tear the Messaging Association down. If a MA-Hello is received from the peer, a MA-Hello reply will be sent back to the peer. In this state any data passed to the Messaging Association for transmission will be sent out to the peer and also the SendHello timer will be restarted as it indicates that the GIST on the node itself still has traffic for delivery. If all of the Routing State machines have indicated that they will not need this Messaging Association, a tg_MAIdle is triggered and the state machine will transfer to *Idle* state and a NoHello timer will be started.

In the *Idle* state, if the NoHello timer expires and that means the peer would not like to keep the Messaging Association open too. So the Messaging Association can be torn down and it will be destroyed. If there is any data requested for transmission or any data received from the peer, the state machine will transfer to the *Connected* state.

There are two timers used by the Messaging Association state machine:

- SendHello

When SendHello expires, it indicates that an MA-Hello message should be sent to the remote node.

- NoHello

When this expires, it indicates that the Messaging Association should destroy itself as no MA-Hello has been received from the remote node for a period of time.

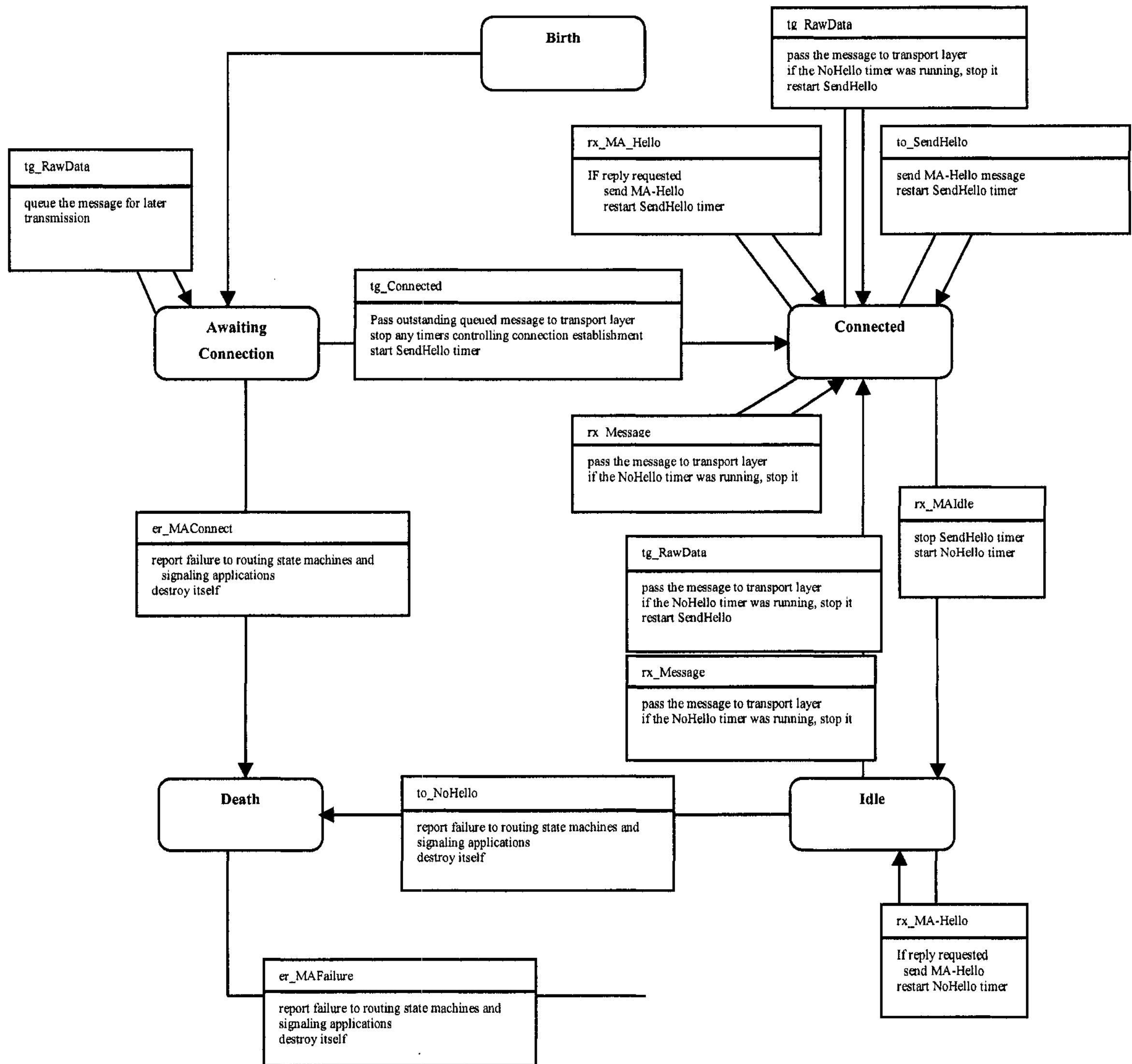


Figure 12: Messaging Association State Machine

3.4 State Machine Implementation

Our implementation is a little bit different from the one in [4]. In our implementation, the Messaging Association state machine is not implemented. Rather we added a MA_Connect at Querying Node for Messaging Association setup. The Messaging Association is only created at the three-way handshake phase. After that, if the TCP connection associated with the Messaging Association is broken, the TCP connection will not be reestablished rather

they will be report back to NSIS signaling application layer. It is up to the NSIS signaling application layer to decide how many messages have been transmitted and how many messages are left for transmission. In the implementation, we find the *connect()* system call is a blocking call by default. So we must change it to a non-blocking call and also we add another state: *Waiting MA* (Messaging Association) to the Querying Node state machine and add another timer: MA_Connect to the state machine. The *Waiting MA* state and MA_Connect together in our implementation replace the whole Messaging Routing state machine. Please refer to Figure 13.

If the Querying Node state machine, which is in *Awaiting Response,* receives a Response, the state machine will transfer to the *Waiting MA* and try to use the *connect()* system call to set up a TCP connection with the Responding Node. Before the *connect()* system call is called, the socket has to been created as a non-blocking socket. Whenever MA_Connect timer expires, the state machine will check to see if the number of times this timer has expired is greater than a threshold. If it is still less than a threshold, the *connect()* system call will be retried. If it returns "EISCONN", that means the socket has been connected and the state machine will transfer to "*Established*" state. Otherwise the MA_Connect timer will be started.

In this way, there are no blocking function calls in both the Querying Node and Responding Node while the Messaging Association still could be kept open by the periodic Confirm.

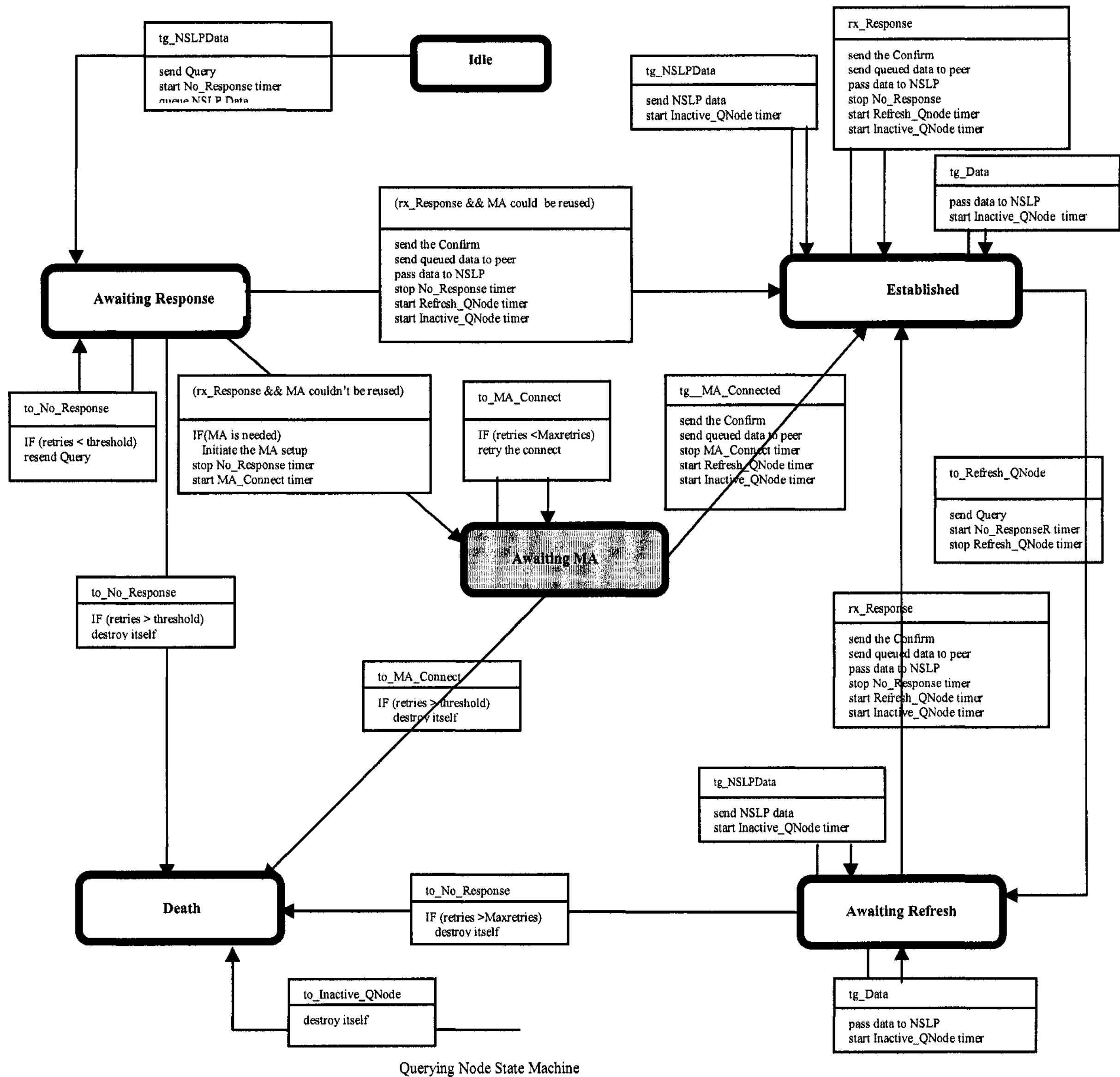


Figure 13: Revised Querying Node State Machine

3.4.1 The GIST Message

All of the GIST Messages begin with a common header, followed a sequence of TLV (Type Length Value) objects. There are six types of messages required for GIST.

GIST-MESSAGE = Query / Response / Confirm / Data / Error / MA-Hello

3.4.1.1 GIST Message Type

For convenience, in our implementation we add two more types of message: one for Data passed by the NSLP and another one for the time-out event. Those two new types of messages can make our functions for state machines more consistent. So the messages type definition in our implementation is as follows:

```
typedef enum{
        GIST_QUERY=0,
        GIST_RESP=1,
        GIST_CONFIRM=2,
        GIST_DATA=3,
        GIST_ERROR=4,
        GIST_MAHELLO=5,
        GIST_NSLP_DATA=6, /*we create a new message type for Data from GIST*/
        GIST_TIMEOUT=7 /* we create a new message type for time-out event*/
} gist_type;
```

As it is too long to give the definition of every message type in C code, all other messages are specified in a high level ABNF format

Query: A Query is always sent in D-mode, with a Router Alert Option. In addition to the common header, it contains certain mandatory control objects

```
  Query = Common-Header
        [ NAT-Traversal-Object ]
        Message-Routing-Information
        Session-Identification
        Network-Layer-Information
```

Query-Cookie

[Stack-Proposal Stack-Configuration-Data]

[NSLP-Data]

The R flag of common header in the Query message must be set as a Query always elicits a Response. In out implementation, we don't have the NAT-Traversal-Object now as it wouldn't affect our validation.

Response:

Response = Common-Header

[NAT-Traversal-Object]

Message-Routing-Information

Session-Identification

Network-Layer-Information

Query-Cookie

[Responder-Cookie

[Stack-Proposal Stack-Configuration-Data]]

[NSLP-Data]

The Response must include the Message-Routing-Information of the Responding Node, which would be used for setting up the message routing state in the Querying Node.

Confirm:

A Confirm must be sent out in the C-mode and must also be sent before other messages for this routing state.

Confirm = Common-Header

Message-Routing-Information

Session-Identification

Network-Layer-Information

[Responder-Cookie

[Stack-Proposal

[Stack-Configuration-Data]]]

[NSLP-Data]

GIST Data Message: The GIST Data message is used to transport the NSLP data message without modifying the GIST state.

Data = Common-Header

[NAT-Traversal-Object]

Message-Routing-Information

Session-Identification

[Network-Layer-Information]

NSLP-Data

Error: An Error message reports the error determined at the GIST level.

Error = Common-Header

[NAT-Traversal-Object]

[Network-Layer-Information]

GIST-Error-Data

MA-Hello:

This message is always sent in C-Mode and on the Messaging Association. It only contains a Common Header and a Hello-ID.

MA-Hello = Common-Header

Hello-ID

The following NLSP Data message and timer-out event message are added by us in our implementation.

NSLPData:

The NSLP Data message is used to pass data from the NSLP to the GIST for transportation.

NSLP Data = Common Header

NSLP ID

Session-Identification

Message Routing Information

Life Time for Querying Node or Responding

NSLP Object

GIST Time-Out event:

The GIST Time-Out event message is used to wrap the time-out event so that the implementation of the functions for the state machines can be consistent.

GIST Time-Out Event = Common Header

Time Out Event

3.4.1.2 Common Header

The first object in all types of GIST messages is the common header. The common head includes a type field and the length of the message beside the common header itself. It also has a hop count to prevent infinite message looping. The common header is defined in C code as follows:

```
typedef struct
{
        unsigned int version:8;
        unsigned int hops:8;
        unsigned int length:16; /*length after the common header*/
        unsigned int nslpID:16;
        unsigned int type:8; /* GIST type*/
        unsigned int s_f:1; /* IP source address is the same as the signaling source address*/
        unsigned int r_f:1; /* a reply of this message is explicitly requested*/
        unsigned int e_f:1; /* if the message was explicitly routed*/
        unsigned int:5 ;
}cmn_hdr_t; /*common-header type*/
```

The *type* in the common header defines the type of the message.

3.4.1.3 TLV Objects

All of the data following the common header are encoded as a sequence of type-length-value objects. Each object can occur at most once. Which object can be included in the message is determined by the message type and the encapsulation mode (C-Mode, D-Mode and Q-mode). All of the following TLV object definitions can be found in [4].

Message-Routing-Information (MRI): Information sufficient to define how the signaling message should be routed through the network [4].

Message-Routing-Information = message-routing-method

method-specific-information

Session-Identification (SID): The GIST session identifier is a 128 bits, cryptographically random identifier chosen by the node that originates the signaling exchange [4].

Network-Layer-Information (NLI): This object carries information about the network layer attributes of the node sending the message, including data related to the management of routing state [4]. This includes a peer identity and IP address for the sending node. This peer identity can be used for Messaging Association reuse. When the Querying Node receives a Response from Responding Node, it will check the peer-identity in the Response to see if there is already a Messaging Association for this peer. When the Responding Node receives a Confirm from the Querying Node, it will check the peer-identity of the Confirm to see if there is already a Messaging Association for this peer. It also includes IP-TTL information to allow the IP hop count between GIST peers to be measured and reported, and a validity time (RS-validity-time) for the routing state [4].

Network-Layer-Information = peer-identity

interface-address

RS-validity-time

IP-TTL

Stack-Proposal: This field contains information about which combinations of transport and security protocols are available for use in messaging associations. In our implementation, only TCP is considered.

*Stack-Proposal = 1*stack-profile*

*stack-profile = 1*protocol-layer*

Each protocol-layer field identifies a protocol with a unique tag; any additional data, such as higher-layer addressing or other options data associated with the protocol, will be carried in a MA-protocol-options field in the Stack-Configuration-Data TLV [4].

Stack-Configuration-Data (SCD): This object carries information about the overall configuration of a messaging association.

Stack-Configuration-Data = MA-Hold-Time

*0*MA-protocol-options*

The MA-Hold-Time field indicates how long a node will hold open an inactive association; MA-protocol-options fields give the configuration of the protocols (e.g., TCP, TLS) to be used for new messaging associations [4].

Query-Cookie/Responder-Cookie: A Query-Cookie is contained in a Query and MUST be echoed in a Response; a Responder-Cookie may be sent in a Response, and if present MUST be echoed in the following Confirm. Cookies are variable length bit strings, chosen by the cookie generator [4].

Hello-ID: The Hello-ID is a 32-bit quantity that is used to correlate messages in an MA-Hello request/reply exchange. A non-zero value MUST be used in a request (messages sent with R=1) and the same value must be returned in the reply (which has R=0) [4].

NSLP-Data: The NSLP payload to be delivered to the signaling application. GIST does not interpret the payload content [4].

GIST-Error-Data: This contains all the information to report the cause and context of an error [4].

GIST-Error-Data = error-class error-code error-subcode

common-error-header

[Message-Routing-Information-content]

[Session-Identification-content]

*0*additional-information*

[comment]

3.4.2 The Pseudo Code of the Event Distributor

All of GIST messages including timer event, NSLP data message, GIST data message, Query, Confirm, Response are distributed to either Querying Node or Responding Node state machine.

The pseudo code of event distributor of the Querying Node:

```
for(;;)
{
   Add all of the file descriptors into the reading file descriptor set (readingfdset)
  if((result=Select(maxfd+1,&readingfdset,NULL,NULL,&timv))>0)
  {
    if(the timer pipe is ready for reading)
    {
       reading the timer event from timer pipe
       wrap the timer event into a gist_msg_s type variable rmsg

      call gistdemux to find the corresponding message routing state entry
      /* mrsp=gistdemux((gist_msg_s *)rmsg); */

       call the corresponding function according the state stored in the state machine
       /* gistqqswitch[mrsp->qstate](mrsp,(gist_msg_s *)rmsg);*/
    } else if (the NSLP fifo is ready)
    {
        reading the data message from the NSIS signaling application layer
       call gistdemux to find the corresponding message routing state entry
       call the corresponding function according the state stored in the state machine

    } else if( UDP socket is ready )
    {
       reading the GIST message from the UDP socket
```

```
call gistdemux to find the corresponding message routing state entry
call the corresponding function according the state stored in the state machine
} else
{
    for(TCP socket in TCP sockets bank associated with Messaging Association)
    {
        if(any one is ready for reading)
        {
            reading the GIST message from the Messaging Association
            call gistdemux to find the corresponding message routing state entry
            call the corresponding function according the state stored in the state
            machine
        }else if (anyone is ready for close)
        {
            close the socket, and find the corresponding MA
            notify all of state machines stored in message routing state table using
              this MA that it has been closed.
        }
    } /*end of inner for */
} /*end of if */
} /*end of the select*/

} //end of for loop
```

The "*mrsp=gistdemux((gist_msg_s *)rmsg);*" is used to find the corresponding message routing state entry. The "*gistqqswitch[mrsp->qstate](mrsp,(gist_msg_s *)rmsg)*" is used to call the corresponding functions according to the state stored in the state machine. Also the *select()* can be used to detect if the peer has closed a TCP socket. If the peer has closed the TCP socket, the MA associated with the TCP socket is set to free to use and all of the message routing state machines associated with the MA must be notified.

The pseudo code of event distributor of the Responding Node:

```
for(;;)
{
    Add all of the file descriptors into the reading file descriptor set(readingfdset)
   if((result=Select(maxfd+1,&readingfdset,NULL,NULL,&timv))>0)
  {
     if(the timer pipe is ready for reading)
     {
         read the timer event from timer pipe
         wrap the timer event into a gist_msg_s type variable rmsg

        call gistdemux to find the corresponding message routing state entry
        /* mrsp=gistdemux((gist_msg_s *)rmsg); */

         call the corresponding function according the state stored in the state machine
         /* gistqqswitch[mrsp->qstate](mrsp,(gist_msg_s *)rmsg);*/
     } else if (the NSLP  fifo is ready)
```

```
{
    reading the data message from the NSIS signaling application layer
    call gistdemux to find the corresponding message routing state entry
    call the corresponding function according the state stored in the state machine

} else if( UDP socket is ready )
{
    reading the GIST message from the UDP socket
    call gistdemux to find the corresponding message routing state entry
    call the corresponding function according the state stored in the state machine
} else if( the TCP listening socket is ready)
{
    accept the incoming connection request
    put the new socket and peer address into a Messaging Association
}
else {
        for(TCP socket in TCP sockets bank associated with Messaging Association)
        {
            if(any one is ready for reading)
            {
                reading the GIST message from the Messaging Association
                call gistdemux to find the corresponding message routing state entry
                call the corresponding function according the state stored in the state
                machine
```

```
            }else if(anyone is ready for close)
            {
                close the socket, and find the corresponding MA
                notify all of state machines stored in the message routing state table using
                this MA that it has been closed.
            }
        } /*end of inner for */
    } /*end of if */
} /*end of the select*/

} //end of for loop
```

The Responding Node event distributor is very similar to the Querying Node event distributor. One difference is that Responding Node needs to detect when a new TCP connection request is coming.

3.4.3 The Implementation of *gistdemux*

The *gistdemux* finds the correct message routing entry for the incoming GIST events.

The *gistdemux* for the Querying Node:

Because we use a static array to store the message routing entries, the *gistdemux* just sequentially searches the table by comparing the MRI in the incoming message and the MRI stored in the message routing entry. Whenever it finds a match, it just returns the message routing entry in the table.

The only exception is when a NSLP Data Message is received, it is first to search the message routing table to try to find a matching entry. If it could not find it, it must find a new and free message routing entry to store the corresponding state machine.

The *gistdemux* for the Responding Node:

The *gistdemux* is very similar with the one on the Querying Node. It just sequentially searches the table by comparing the MRI in the incoming message and the MRI stored in the message routing entry. Whenever it finds a match, it just returns the message routing entry in the table.

When a Query is received by the Responding Node, the Responding Node needs to use the Message Routing Information to search if there is a matching message routing entry corresponding to this MRI. If it could not find one, it will create the Responding Node state machine and pass the Query to the Responding Node state machine.

When a Confirm is received, the Responding Node will use the peer identity (the IP address of the Querying Node) to search the Messaging Association table to see if there is already a Messaging Association for this peer so that the Messaging Association can be reused.

3.4.4 The Implementation of the Querying Node State Machine

The Querying Node state machine is very straightforward, for every state there is a corresponding function and we define an array of function pointers and the array is indexed by the states. So the event distributor can call the corresponding function according to the state stored in the message routing state entry.

```
typedef enum
{
        GISTQS_BIRTH=0,
        GISTQS_WRESP=1,/* waiting response */
        GISTQS_WMACO=2,/* wait Messaging Association to finish */
        GISTQS_ESTAB=3,
        GISTQS_WREFR=4, /* waiting refresh */
        GISTQS_DEATH=5
}gistqstate; /* GIST Querying Node state */
/*the Querying Node state machine functions*/
```

```
int qbirth(mrs_t * ,gist_msg_s * );

int qwresp(mrs_t * ,gist_msg_s * );

int qwmaco(mrs_t * ,gist_msg_s * );

int qestab(mrs_t * ,gist_msg_s * );

int qwrefr(mrs_t * ,gist_msg_s * );

int qdeath(mrs_t * ,gist_msg_s * );

int (* gistqqswitch[5])(mrs_t * ,gist_msg_s *)=

{

        qbirth,qwresp,qestab,qwrefr,qdeath

};
```

Below we give out the pseudo code of every function:

qbirth:

The *qbirth* is responsible to send out the query whenever it receives some data from the NSIS signaling application layer.

```
int qbirth(mrs_t * mrsp,gist_msg_s * msgp )

{

   IF( msgp is a NSLP Data type message )

   {

        queue the message to the message routing entry

        send out the query

        start No_Response timer

        transfer state machine to Awaiting Response state

   }

}
```

qwresp: In the *Awaiting Response* state, the state machine is waiting for the Response. If a Response is received the function is going to check to see if there is a Messaging Association that can be reused. If there is not a Messaging Association that can be reused, the function will start a new TCP with the Responding Node. If the TCP connection can not be finished right away, the state machine will transfer to the *Waiting MA* state.

```
int qwresp(mrs_t * mrsp,gist_msg_s * msgp)
{
    IF (msgp is a Response)
    {
        IF(there is no Messaging Association associated with this message routing entry)
            check to see if there is a Messaging Association that we can reuse by comparing
            the peer identity contained in the Network Layer Information in the Response
            IF(there is no Messaging Association that we can reuse)
            {
                find a free Messaging Association entry
                associate this Messaging Association with the message routing state entry
                create the socket for the new Messaging Association
                associate the socket with this Messaging Association.
                set the socket to non-block socket
                initiate the TCP connection to the Responding Node
                IF(TCP socket connection is not finished right away)
                {
                    delete the No_Response timer
                    start the MA_CONNECT timer
                    transfer the state machine to Waiting MA state
```

```
            return
          }
        }ELSE
      reuse the Messaging Association and associate it with the message routing state entry
      set the non-blocking socket to a  blocking socket
      send out the Confirm and all of the queued data
  } ELSE IF( msgp is a No_Response time-out event)
  {
    IF(times we resent the Query has been greater than the threshold)
    {
      remove all of timers associated with this message routing state entry
      destroy the state machine
    } ELSE
    {
      resent the Query
      restart the No_Response timer
    }
  }
return 0;
}
```

qwmaco: In the *Waiting MA* state, the state machine is waiting for the Messaging Association to finish. If a MA_Connect timer expires in this state, the function will retry to connect the Responding Node to establish the Messaging Association. If the times we have retired are greater than the maximum times allowed, the state machine will be destroyed.

```
int qwmaco(mrs_t * mrsp,gist_msg_s * msgp)
{
   IF((msgp is a MA_Connect time-out event)&&(retries < threshold))
  {
     try to call the connect() system call to establish the TCP connection
    if(connection is successfully established)
    {
        set the socket to blocking socket
        send out the confirm
        send out all of the queued data
        start Refresh_QNode timer
       start Inactive_QNode timer
    } ELSE
    {
       set  retries to zero
       delete all of timers for this routing state
      destroy the state machine stored by this routing state entry
    }
  }
}
```

qestab: In the *Established* state, the state machine can send data to and receive data from NSIS signaling application layer protocol normally. It also can send data to and receive data from Responding Node normally.

```
int qestab(mrs_t * mrsp,gist_msg_s * msgp)
{
  IF(msgp is a Data type message received from the Responding Node)
  {
      pass the message to the NSIS signaling application layer
      restart Inactive_QNode timer
  } ELSE IF(msgp is a Refresh_QNode time-out event)
  {
     send Query to the Responding Node
     start No_Response timer
     state machine transfer to Awaiting Refresh state
  } ELSE IF( msgp is a Response)
  {
     send out the Confirm to the Responding Node
    stop No_Response timer
    start Refresh_QNode timer
    start Inactive_QNode timer
  } ELSE IF(msgp is a NSLP Data)
  {
    pack data into a GIST Data message and send it to the Responding Node
    restart the Inactive_QNode timer
  }
}
```

qwrefr: In the *Awaiting Refresh* state, the state machine can send data to the NSIS signaling application layer protocol and Responding Node normally just as in the *Established* state. The state machine can also receive data from NSIS signaling application layer protocol and Responding Node normally just as in the *Established* state.

```
int qwrefr(mrs_t * mrsp,gist_msg_s * msgp)
{
  IF(msgp is a Response)
  {
     send out the Confirm
     delete the No_Response timer
     restart Refresh_QNode timer
    restart Inactive_QNode  timer
  } ELSE IF(msgp is a NSLP Data)
  {
    pack data into a GIST Data message and send it to the Responding Node
   restart the Inactive_QNode timer
  }ELSE  IF(msgp is a Data type message received from the Responding Node)
  {
       pass the message to the NSIS signaling application layer
       restart Inactive_QNode timer
  }
}
```

3.4.5 The Implementation of Responding Node State Machine

The Responding Node state machine is also very straightforward, for every state there is a corresponding function and we define an array of function pointers and the array is indexed by the states. So the event distributor can call the corresponding function according to the state stored in the message routing state entry.

```
typedef enum
{
        GISTRS_BIRTH=0,
        GISTRS_WCONF=1,
        GISTRS_ESTAB=2, /* Established */
        GISTRS_WREFR=3, /* Awaiting Refresh */
        GISTRS_DEATH=4
} gistrstate; /* gist Responding Node state */
int rbirth(mrs_t * ,gist_msg_s * );
int rwconf(mrs_t * ,gist_msg_s * );
int restab(mrs_t * ,gist_msg_s * );
int rwrefr(mrs_t * ,gist_msg_s * );
int rdeath(mrs_t * ,gist_msg_s * );
int ( *gistrrswitch[5])(mrs_t *,gist_msg_s *)=
{
        rbirth,rwconf,restab,rwrefr,rdeath
};
```

Below we give out the pseudo code of every function:

rbirth: In the *Birth* state, the state machine is just created and is passed with a Query message from the Querying Node. The state machine will reply with Response.

```
int rbirth(mrs_t * mrsp,gist_msg_s * msgp)
{
   IF(msgp is a Query)
   {
        send the Response to the Querying Node
        start the No_Confirm timer
        state machine transfer to Awaiting Confirm state
   }
}
```

wconf: In the *Awaiting Confirm* state, the state machine is waiting for a Confirm. Once a Confirm is received, the Data contained in the Confirm would be passed to signaling application data layer.

```
int rwconf(mrs_t * mrsp,gist_msg_s * msgp)
{
   IF(msgp is a Confirm)
   {
      send the data contained in the Confirm to the NSIS signaling application layer
        protocol
      state machine transfers to Eestablished state
      delete No_Confirm timer
     start Expire_RNode timer
   } ELSE IF(msgp is a time-out event)
```

```
{
    IF(retries  < threshold)
    {
        resend the Response
        restart No_Confirm timer
    } ELSE
    {
        set retires to 0
        delete all timers of this routing state entry
        destroy the routing state machine
    }
}
```

restab: In the *Established* State, the state machine can send and receive both GIST Data message and NSLP Data normally.

```
int restab(mrs_t * mrsp,gist_msg_s * msgp)
{
   IF(msgp is a GIST Data type message received from the peer)
   {
      pass the message to the NSIS signaling application layer
   } ELSE IF(msgp is a NSLP Data message received from the signaling application Layer)
   {
      send the data to the Querying Node
   } ELSE IF(msgp is a Query)
```

```
{
   send out the Response
   start No_Confirm timer
  transfer to the Awaiting Refresh state
} ELSE IF(msgp is a Confirm)
{
  silently ignore the message
}
}
```

rwrefr: In the *Awaiting Refresh* state, just like in the *Established* state, the state machine can receive and send the GIST Data message and NSLP Data message normally.

```
int rwrefr(mrs_t * mrsp,gist_msg_s * msgp)
{
   IF(msgp is a Confirm)
  {
     transfer the data contained in the Confirm to the NSIS signaling application layer.
     stop No_Confirm timer
     start Expire_RNode timer
    transfer state to Established state
  } ELSE IF(msgp is a GIST Data received from the peer)
 {
    pass the data to the NSIS signaling application layer
 } ELSE IF(msgp is a NSLP Data type message)
 {
```

```
    send the Data in a GIST Data message to the peer
 } ELSE IF( msgp is a No_Confirm timer event)
 {
    IF(retries < threshold)
    {
       resend the Response
       start No_Confirm timer
    } ELSE
    {
       set retries to zero
       delete  all of the timers of this state machine
       destroy the state machine itself
    }
  }
}
```

Chapter 4

Validation

By designing several testing scenarios we are able to confirm the correct operation of the GIST protocol stack. Firstly, our program is tested in our lab network, which consists of multiple machines, as our implementation can accommodate any number of machines in a network. The environment is not the same as the real internet network but we have made it as close as possible by simulating a Flow-Next Hop mapping table. Secondly, for the state machines, we designed the scenarios based on different type of timers. For every timer, there is a scenario corresponding to what will happen if the timer expires and there is also a scenario corresponding to what will happen if the timer never expires. Finally, we also designed a scenario to test how well the implementation is performing with the successful message delivery and a scenario to see if our implementation can achieve the Messaging Association multiplexing. All scenarios are supplemented with a diagram, which is drawn based on the specification of GIST. The testing would reveal if all of timer are defined properly or not and also reveal how well the specification of GIST is defined.

In order to simulate the Signaling application protocol, we designed a small NSLP Ping client to validate the GIST specification. The output of the state machine and timer can either be printed on the screen or to a log file, which could be analyzed later. Validation is done by matching the diagram with the messages written to the log file.

4.1 The Test Tool Designed

In order to verify the GIST specification defined by [4] with our implementation, we implemented a simple NSLP Ping protocol, which can be treated as a NSIS signaling application layer protocol. The NSLP Ping will send out a ping message along the flow path and every node within the path will add its own IP address and the time at which the message is received.

typedef struct

```
{
        unsigned int ipadd[1];
        unsigned int times[2];
} ping_obj_t;

typedef struct
{
        unsigned int version:8; //=1
        unsigned int hops:8;
        unsigned int length:8;
        unsigned int objs:8;/*2 number of objects every hop should add*/
        unsigned int iptype:16; /*=1 for IPv4 address*/
        unsigned int iplen:16;/*=16 for IPv4*/
        unsigned int tstype:16;/*2*/
        unsigned int tslen:16;/*=8*/
        char  data[1];
} ping_t;

typedef struct
{
        unsigned int ab_f:2;
        unsigned int:2 ;
        unsigned int type:12;
        unsigned int:4 ;
```

```
        unsigned int length:12; /* length after the gen_obj_hdr_t */
} gen_obj_hdr_t;

typedef struct
{
        gen_obj_hdr_t ;
        char  data[1];
}nslp_obj_t;
```

The ping_t will be packed into the NSLP Object. Please see Figure 14.

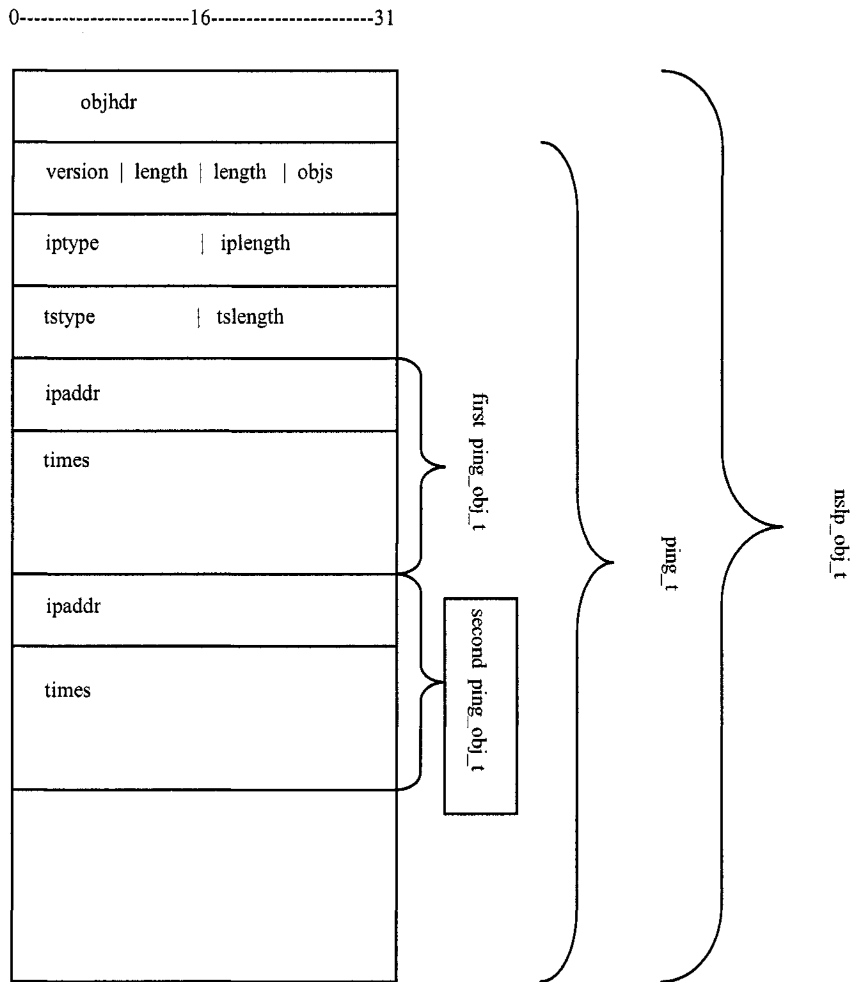


Figure 14: Ping Objects contained in NSLP Object

Suppose the GIST Querying Node's IP address is 132.205.96.51 and the GIST Responding Node's IP address is 132.205.96.49. The output on the Ping client in a successful message delivery scenario will be as follows

Hop 0 :inet_ntoa(((struct in_addr *)&pobjp->ipadd[0]))=132.205.96.51*

Fri Nov 2 02:04:51 2007

usec: 97766

Hop 1 :inet_ntoa(((struct in_addr *)&pobjp->ipadd[0]))=132.205.96.49*

Fri Nov 2 02:04:53 2007

usec: 744594

As there are only two nodes along the path, so we only have Hop0 and Hop1 here. Inside the Ping message we can see the first hop's IP is 132.205.96.51 and the time it receives the Ping message is "*Fri Nov 2 02:04:51 2007 usec: 97766*" and the Hop 1's IP address is 132.205.96.49 and the time it receives the Ping message is "*Fri Nov 2 02:04:53 2007 usec: 744594*".

In the following scenarios, we are using the Ping client to simulate the signaling application protocol, which asks for signaling message transport service from GIST.

The NSLP Ping has two parameters which are as follows:

--type=initiator/forwarder/receiver

Through this parameter you can specify which kind of node of the Ping application is. For example if you start Ping as "*./nslpping --type=initiator --flow=192.168.9.170r*", the NSLP ping application will start to send a message to the GIST on the same machine using a Pipe and then wait for the reply from the GIST. If you start NSLP Ping as "*./nslpping --type=forwarder*", the Ping application will wait for an incoming Ping message and adds its own IP address to it once it receives one and then passes the message back to the GIST for the next hop. If you start the Ping as "*./nslpping --type=receiver*", the NSLP Ping application will wait for an incoming Ping message from GIST and adds its own IP address to it once it receives one and then changes the Direction Flag in the Message Routing Information from Downstream to Upstream and passes the message to GIST. A more specific example how to start those two applications will be given in section 4.3.

--flow=xxx.xxx.xxx.xxx

The xxx.xxx.xxx.xxx stands for the destination IP address of the flow for which the signaling is about.

4.2 A Brief Introduction of Our GIST Implementation

The GIST can accept three parameters, which are as follows:

--prev=previous hop: This parameter is not used by current implementation and will be determined later.

--next=next hop: The parameter *next hop* is the default next hop for flows if a next hop could not be located in the Flow-Next Hop mapping table.

--file=filename: Filename is the name of the file which contains the Flow-Next Hop mapping table.

After the GIST receives a message from the signaling application, it will check the mapping table to see if it can find a destination IP address of a matching flow ID. If one is found, then it will use the next hop from the mapping table. If one couldn't be found, the next hop that is entered through the parameter will be used.

For example, the GIST is running on a machine whose name is *dikjstra.cs.concordia.ca*. It has a mapping table with content as below:

192.168.9.170 tux.encs.concordia.ca

192.168.9.171 knuth.cs.concordia.ca.

The *192.168.9.170* is the destination IP address of the first flow and *tux.encs.concordia.ca* is the next hop for it. While *192.168.9.171* is a destination IP address of another flow and *knuth.cs.concordia.ca* is the next hop for it. For all of other flows whose destination IP addresses couldn't be found in the Flow-Next Hop mapping table, GIST will use *knuth.cs.concordia.ca* as their next hop.

So, the command to start GIST in *diskstra.cs* is:

"*./gist --next=knuth.cs --file=dijkstra.txt*"

4.3 Testing in a Networking Environment

As our program is in user-space, we are using UDP to simulate the IP and using a Flow-Next Hop mapping table instead of accessing the Router-Alert-Option directly. The mapping table

can let each GIST choose the next hop for a specific flow dynamically. The previous hop is determined dynamically after the Responding Node receives a Query. So our implementation can accommodate any number of machines in the network.

All of tests are performed in our networking lab on Linux machines. We will illustrate the scenarios of both simple test and complex test and every test case has a topology.

4.3.1 Testing With Four Hops and Two Flows without Mapping Table

The following test case is performed with 4 hops for 2 different flows.

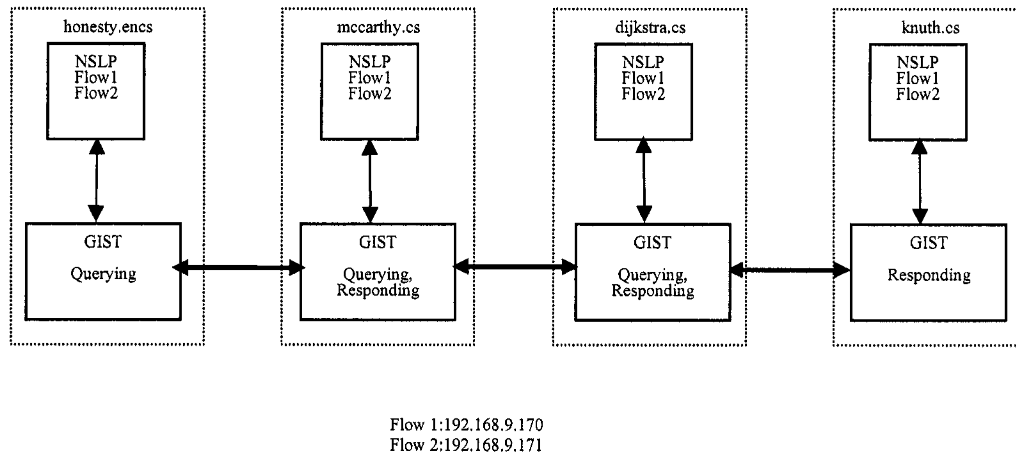


Figure 15: GIST Test without Flow-Next Hop Mapping Table

Commands to start the NSLP Ping and GIST on each machine are list below:

[honesty]>./gist --next=mccarthy.cs

[honesty]>./nslpping --type=initiator --flow=192.168.9.170

[honesty]>./nslpping --type=initiator --flow=192.168.9.171

[mccarthy.cs]>./gist --next=dijkstra.cs

[mccarthy.cs]>./nslpping --type=forwarder

[dijkstra.cs]>./gist --next=knuth.cs

[dijkstra.cs]>./nslpping --type=forwarder

[knuth.cs]>./gist

[knuth.cs>./nslpping --type=receiver

In this test case the NSLP Ping is used by two flows with destinations 192.168.9.170 and 192.168.9.171. The NSLP Ping messages for both flows travel from *honesty.encs* through *mccarthy.cs* and *dijkstra.cs* to *knuth.cs* and then travel back through *dijkstra.cs*, *mccarthy.cs* finally to *honesty.encs*. The NSLP Ping message finally is sent back to NSLP Ping. The Messaging Association multiplexing between every two adjacent hops is achieved.

4.3.2 Testing with Flow-Next Hop Mapping Table

The following test case is performed on a more complicated topology with 6 machines and every machine is using a Flow-Next Hop mapping table. The content of Flow-Next Hop mapping table of every machine is shown in the following diagram.

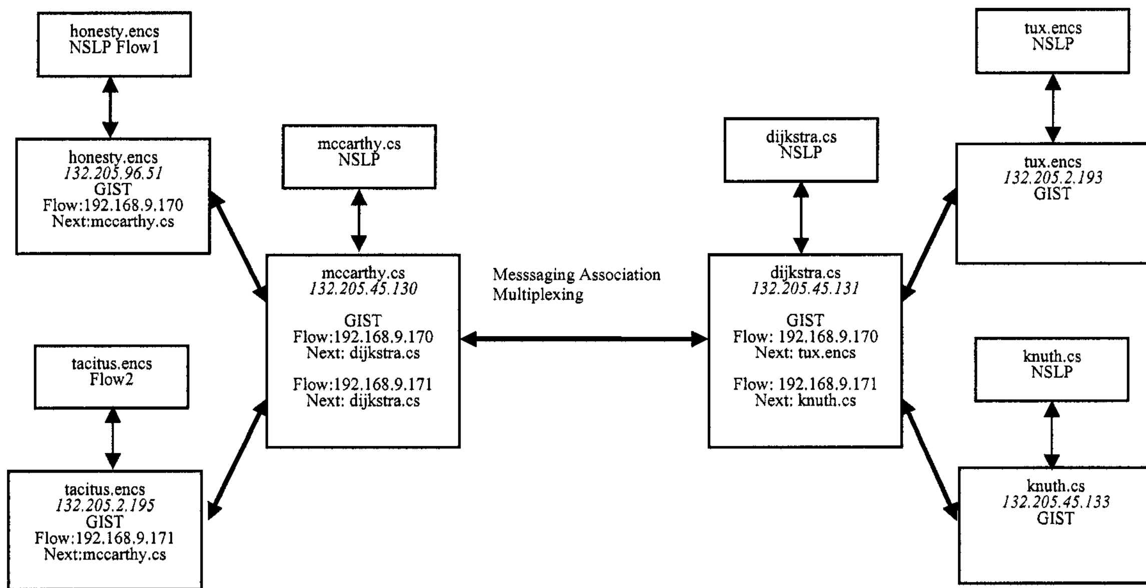


Figure 16: GIST test with Flow-Next Hop Mapping Table

The command to start GIST and NSLP Ping on every machine is as follows:

[honesty]>./gist --file=honesty.txt

[honesty]>./nslpping --type=initiator --flow=192.168.9.170

[tacitus]>./gist --file=tacitus.txt

[tacitus]./nslpping --type=initiator --flow=192.168.9.171

[mccarthy]>./gist --file=mccarthy.txt

[mccarthy]>./nslpping --type=forwarder

[dijkstra]>./gist --file=dijkstra.txt

[dijkstra]./nslpping --type=forwarder

[tux]>./gist --file=tux.txt

[tux]>./nslpping --type=receiver

[knuth]>./gist --file=tux.txt

[knuth]>./nslpping --type=receiver

The output seen from honesty.encs for the flow 192.168.9.170 is as follows:

Hop 0 :132.205.96.51 Sat Mar 15 20:47:29 2008 usec: 654911

Hop 1 :132.205.45.130 Sat Mar 15 20:47:29 2008 usec: 654992

Hop 2 :132.205.45.131 Sat Mar 15 20:47:29 2008 usec: 684275

Hop 3 :132.205.2.193 Sat Mar 15 20:47:29 2008 usec: 685243

Hop 4 :132.205.45.131 Sat Mar 15 20:47:29 2008 usec: 693926

Hop 5 :132.205.45.130 Sat Mar 15 20:47:29 2008 usec: 691443

We can see the NSLP Ping message is traveling from *honesty.encs* through *mccarthy.cs* and *dijkstra.cs* to *tux.encs* and then traveling back through *dijkstra.cs* and *mccarthy.cs* to *honesty.encs.*

The output seen from *tacitus.encs* for the flow 192.168.9.171 is as follows:

Hop 0 :132.205.2.195 Sat Mar 15 20:47:42 2008 usec: 283037

Hop 1 :132.205.45.130 Sat Mar 15 20:47:42 2008 usec: 287241

Hop 2 :132.205.45.131 Sat Mar 15 20:47:42 2008 usec: 295044

Hop 3 :132.205.45.133 Sat Mar 15 20:47:42 2008 usec: 298213

Hop 4 :132.205.45.131 Sat Mar 15 20:47:42 2008 usec: 306855

Hop 5 :132.205.45.130 Sat Mar 15 20:47:42 2008 usec: 303946

We can see the NSLP Ping message is traveling from *tacitus.encs* through *mccarthy.cs* and *dijkstra.cs* to *knuth.cs* and then traveling back through *dijkstra.cs* and *mccarthy.cs* to *tacitus.encs.*

4.4 Normal Message Delivery and State Refreshing

In this scenario, we are testing if the state machine is performing well with successful message delivery and state refreshing as we are expecting. The following output would be expected to be observed in the log file after we started Querying Node, Responding Node and the Ping Client.

NSLP Ping client in the Querying Node sends out Ping message to the GIST and waits for a Ping message to be sent back. The Querying Node will first set up a Messaging Association with the Responding Node. Then it will send the data within the Confirm message, which will be sent over the newly established Messaging Association. After the Responding Node receives the Confirm and it will pass the data in the Confirm to Ping client for further processing. The Ping client will add the IP address of itself and the time it received the Confirm and pass the message to the GIST. The GIST will construct a GIST Data message and send that Data message over the Messaging Association to the Querying Node. The Querying Node then passes the data contained in the GIST Data message to the Ping client. The Ping client will then print out the message. This is the first data exchange between Querying Node and Responding Node. After that both Querying Node and Responding Node will be periodically refreshed. The refresh is initiated by the Refresh_QNode timer expiration. Whenever this timer expires, a Query will be sent out. The Query will elicit a Response, which will refresh the Querying Node. A Confirm will be sent out to Responding Node after receiving the Response. The Responding Node will be refreshed then. The Querying Node could be in the *Established* or *Awaiting Refresh* state. The Responding also could be in the *Established* or *Awaiting Refresh* state. A normal GIST Data message can always be delivered in both ways. Please see Figure 17 for a more detailed description.

The output observed on the log file including all of timer events and message received for both Querying Node and Responding Node are the same as we expected from Figure 17 and the Ping message is successfully coming back with two IP addresses and two time stamps added in it. Please note that if a timer has a cross on it, it means the timer never expires, because it is already either restarted or removed after some new events.

Using the TCP as the transport protocol, the MTU problem that the RSVP has can be solved.

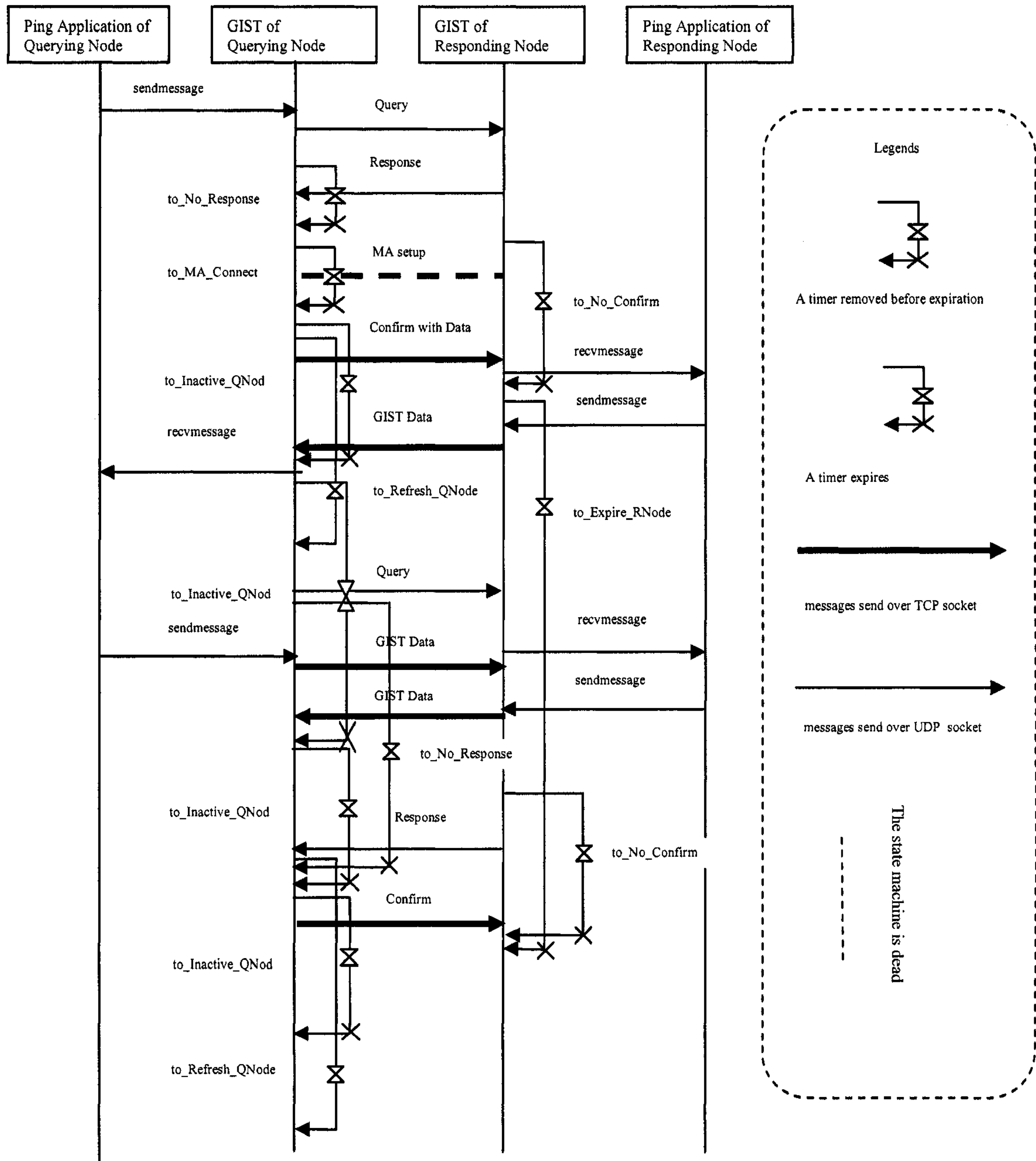


Figure 17: Normal Message Delivery and Soft State Refreshing

4.5 Messaging Association Multiplexing

In this scenario, we are testing the Messaging Association Multiplexing where a single Messaging Association could be used by multiple flows. Multiplexing insures that the MA cost scales only with the number of peers and avoids the latency of new MA setup.

When the Querying Node receives a Response that contains the Network Interface Information (Interface Address and Peer Identity) of the Responding Node, the Querying Node will check the Messaging Association table to see if there is a Messaging Association with that Responding Node. If there is already a Messaging Association with the peer, the Confirm will be sent out of along the Messaging Association. When the Confirm is received by the Responding Node, it also used the peer address identity in the NLI object of the Confirm message to see if there is already a message association established with the peer. Actually, when the Messaging Association is supposed to have been established at the time Querying Node initiated the TCP connection. So Messaging Association Multiplexing is achieved. Please see the Figure 18 for a more detailed description.

The approach is given as follows for testing Messaging Association Multiplexing:

First we start both the Querying Node and Responding Node. Then we started the Ping client twice for two different flows (192.168.9.171 and 192.168.9.170 respectively).

The output observed in the log file is the same as we expected from the Figure 18. The output we observed is that the first entry is allocated for the flow 0 and the second entry is allocated to flow 1. Both the first entry and second entry of message routing table are pointing to the same Messaging Association number, which is the first entry in the Messaging Association table.

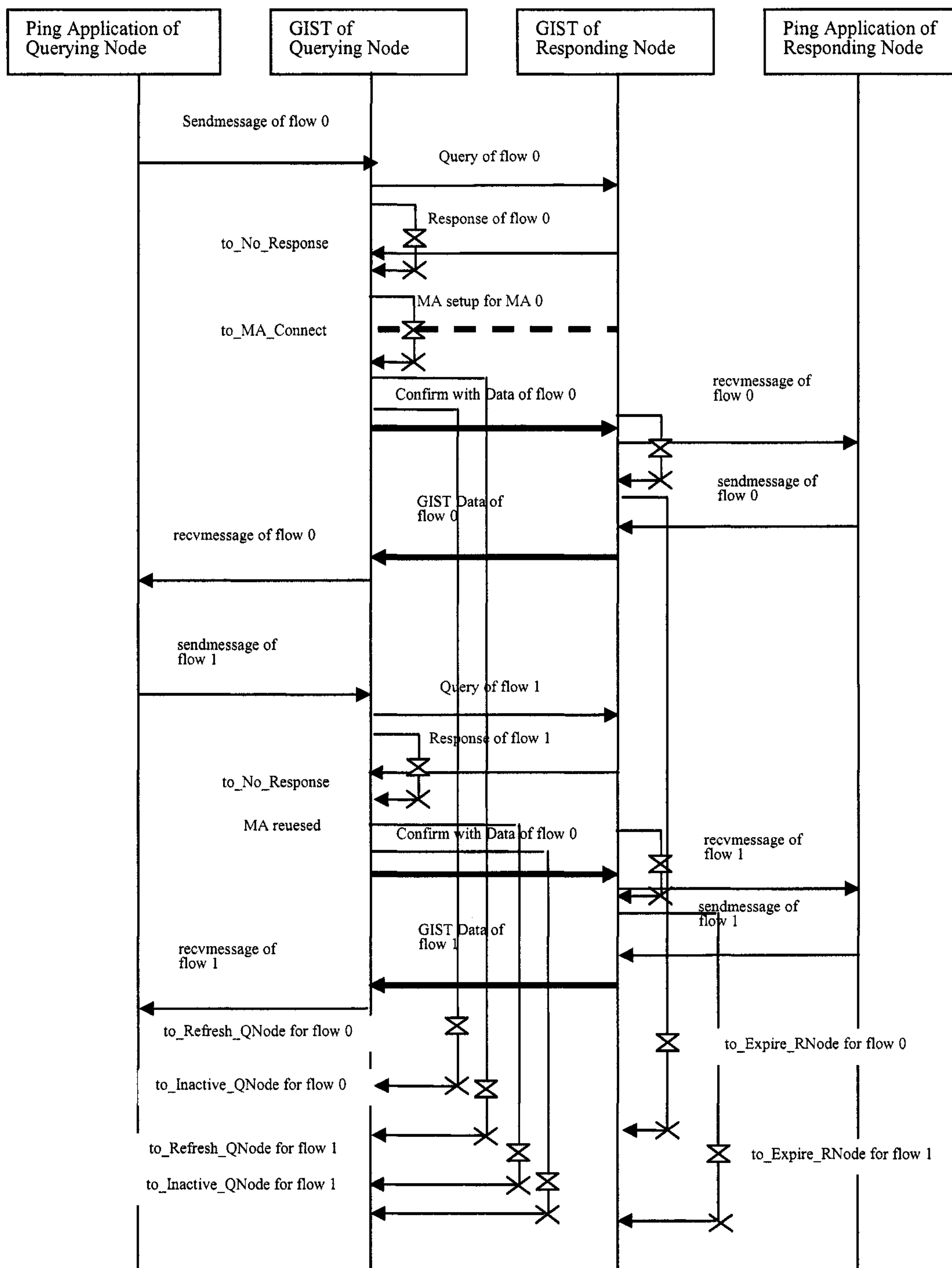


Figure 18: Messaging Association Multiplexing

4.6 Validation of Soft State Timers

As the timers are playing an important role in a soft state protocol like GIST, the following scenarios are to validate all the timers we implemented for Querying Node and Responding Node. If a Messaging Association could not be set up due to timer expiration or a state machine is torn down due to the timer expiration, an error message will be indicated back to the signaling application.

4.6.1 MA_Connect Timer Validation

In this scenario, we are testing the MA_Connect timer. The MA_Connect timer is used to set up the TCP connection with a non-blocking socket. It is used on the Querying Node after receiving the Response. The Querying Node will try to initiate the TCP connection with the Responding Node. If the listening point on the Querying Node is not ready, after six expirations of the MA_Connect timer, the state machine stored at that routing state entry will be removed. Please see Figure 19 for a detailed description. The dashed line in Figure 19 represents that the state machine is already dead and removed.

Meanwhile, if the Responding Node is brought up, the TCP connection will still be successfully established. The MA_Connect is used with a binary back-off exponential algorithm. The initial time value is set 500 ms and then the timer will be set to 1s, 2s, 4s, 8s, 16s and 32s respectively after each expiration. If the connection still could not be established after about 63.5 seconds the state machine will be removed. An error message will be returned to NSIS Ping application to state the error.

The approach of this scenario is provided in the following.

- We first started the Querying Node, while the Responding Node is not started.

  We observed that the state machine is removed because of the MA_Connect timer expiration.

- We first started the Querying Node, while the Responding Node is not started yet. After 30 seconds, we then started the Responding Node.

  We observed that the Messaging Association still can be established successfully.

4.6.2 Refresh_QNode Timer Validation

In this scenario, we are testing the Refresh_QNode timer. Whenever the Refresh_QNode timer expires, a Query is supposed to be sent out. The state machine is supposed to transfer to *Awaiting Refresh*. The state machine is then waiting for a Response from the Responding Node. Please refer to Figure 20.

The approach is explained as following:

We both started the Querying Node and Responding Node at the same time, after the Messaging Association is established, we observed the both Querying Node and Responding Node periodically refreshed due to the Refresh_QNode timer expiration.

The output of this scenario observed in the log file is the same as what we are expecting from Figure 20.

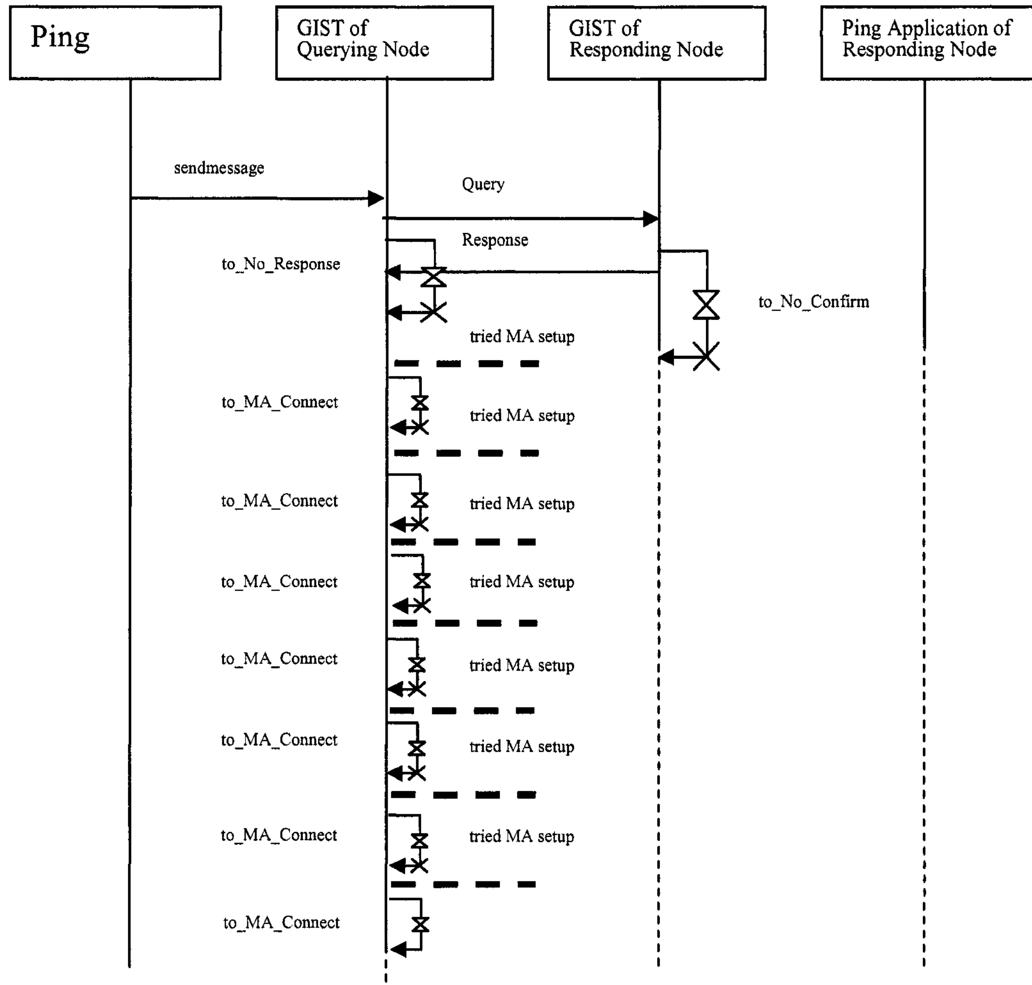


Figure 19: MA_Connect Timer

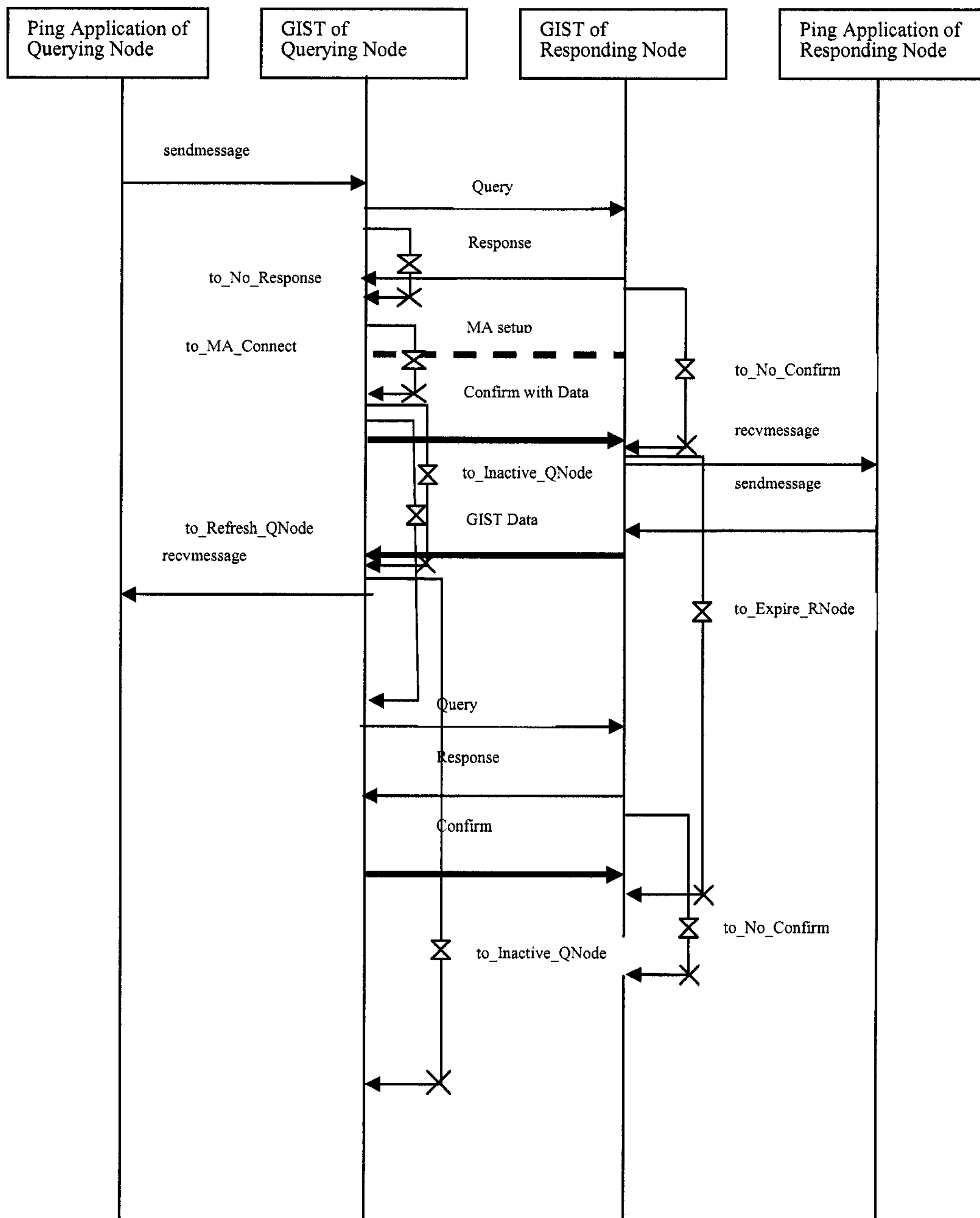


Figure 20: Refresh_QNode Timer

4.6.3 No_Response Timer Validation

In this scenario, we are testing the No_Response timer. The Querying Node state machine is waiting for a Response when it is in *Awaiting Response* or *Awaiting Refresh.* We validated both of them.

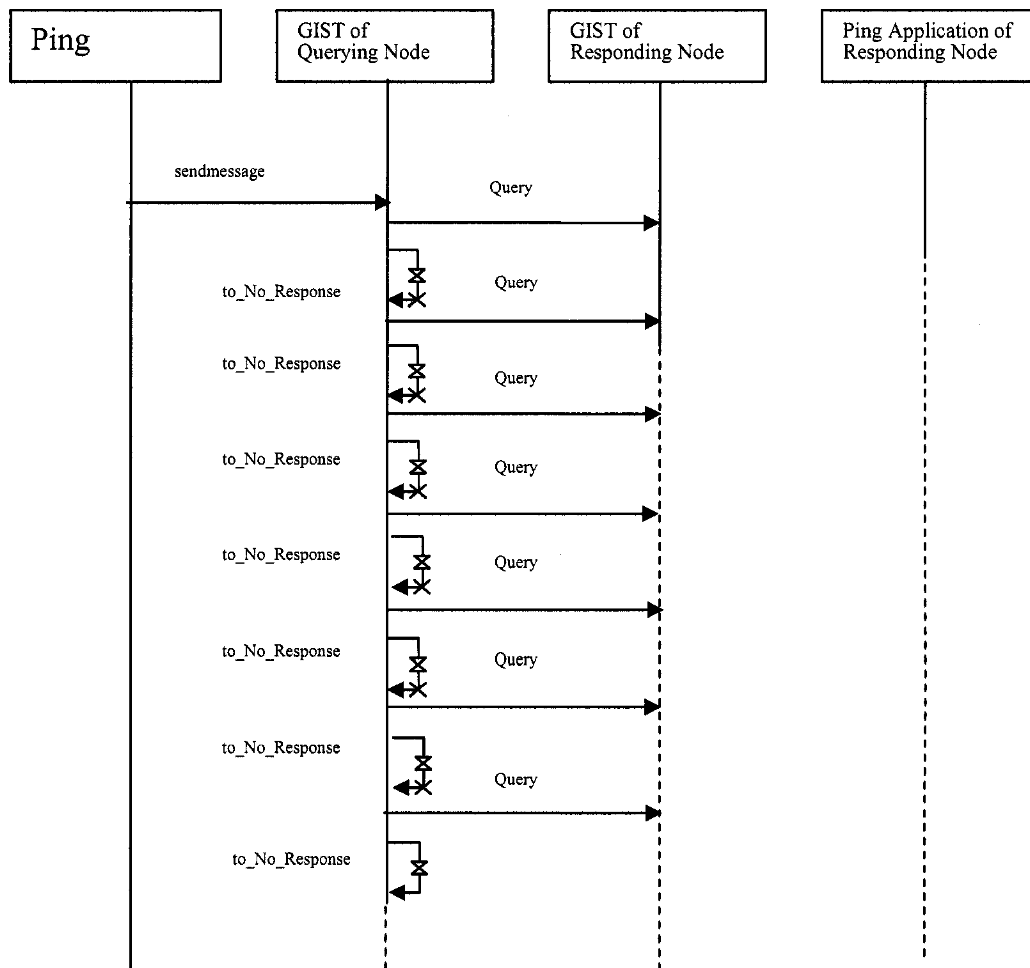


Figure 21: No_Response Timer in the Awaiting Response State

When the state machine is in *Awaiting Response*, the binary exponential back-off will be used for No_Response timer. The Query will be resent at most six times by default.

When Querying Node state machine is in *Awaiting Refresh*, if we killed the Responding Node, normally the Inactive_QNode will expire earlier than the No_Response timer. This is because whenever a Query is retransmitted, a binary exponential back-off will be used. The initial timer value is T1=500ms, which the back off process can increase up to a maximum value of T2 seconds. The T2 in our implementation is 64 * T1. So if the state machine is removed due to the expiration of No_Response timer, the time that has elapsed at least is equal 0.5s + 1s + 2s + 4s + 8s + 16s +32s = 63.5 seconds. While the Inactive_QNode timer is only 30 seconds in our implementation. So in order to let the No_Response timer expire before the Inactive_QNode, we change the T2 = 16 * T1. So the No_Response timer will expire at 15.5 (0.5+1+2+4+8 = 15.5 seconds) seconds. Please see Figure 22 below.

The approaches are provided below for both test cases of testing the No_Response timer:

- No_Response timer expiration when it is in *Awaiting_Response* state:

We modified the Responding Node and let it terminate automatically after receiving the first Querying. Then we started both Querying Node and Responding Node.

We observed the state machine is removed due to the No_Response timer expiration at the Quering Node.

- No_Response timer expiration when it is in *Awaiting_Refresh* state:

We modified the Responding Node to let it terminate after it sent the GIST_Data message back. We also changed the T2 for No_Response timer to 16 seconds.

We observed that the state machine in Querying Node is removed due to No_Response timer expiration.

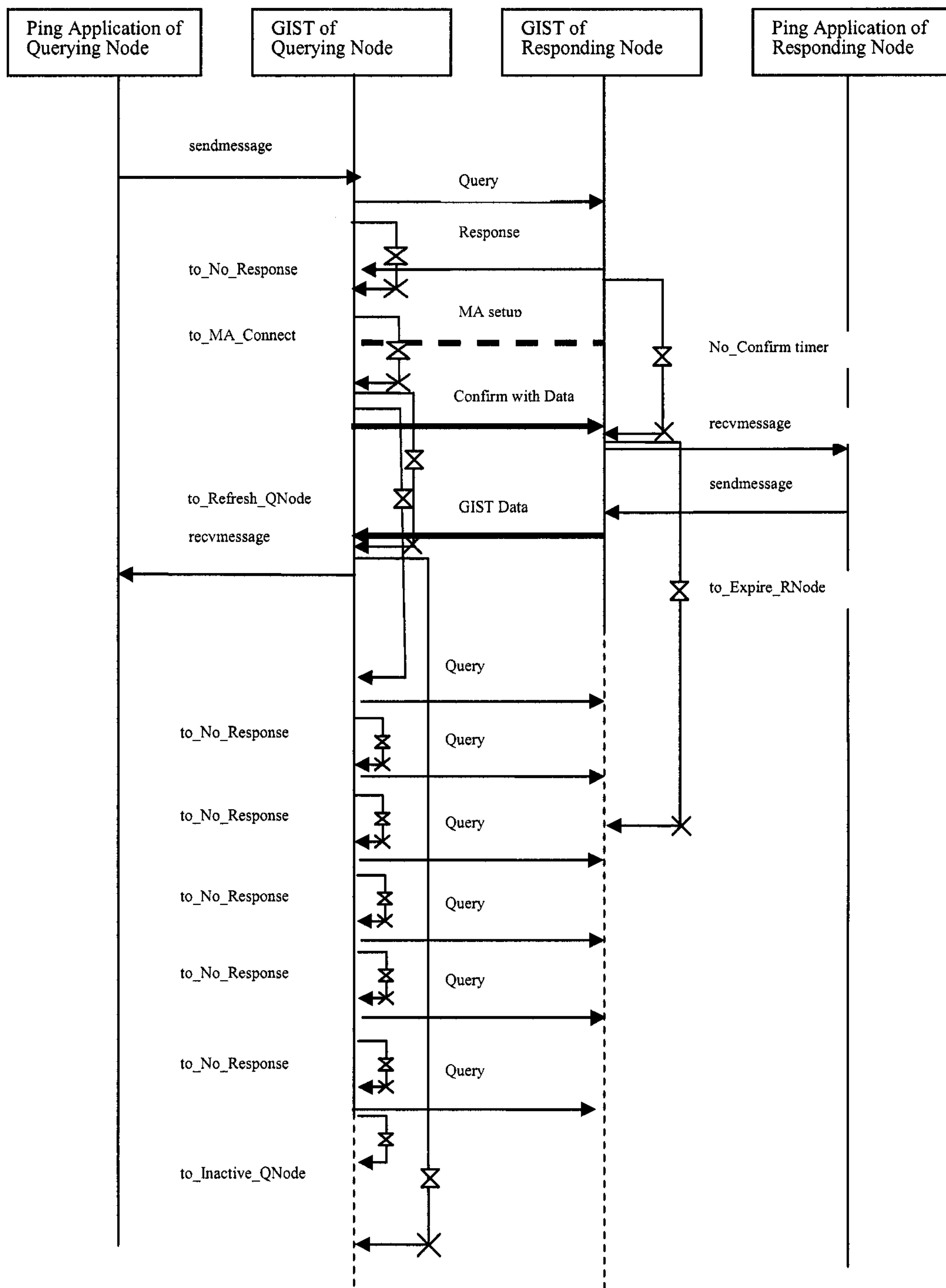


Figure 22: No_Response Timer in the Awaiting Refresh State

4.6.4 Inactive_QNode Timer Validation

In this scenario, we are testing the Inactive_QNode timer. The expected behavior is as follows.

In our implementation, the Inactive_QNode timer is set to 30 seconds, just as we mentioned in section 4.4.3, normally in the *Awaiting Refresh* state, the Inactive_QNode timer always expires earlier than the No_Response timer if the Responding Node is down already. Theoretically, if Responding Node is down, when Querying Node state machine is in *Established* state, the state machine will transfer to the *Awaiting Refresh* state. The Inactive_QNode will expire at the time when the Querying Node state machine is in *Awaiting Refresh* state.

The approach is given as follows for testing Inactive_QNode timer:

We started both the Querying Node and Responding Node at the same time. After the Responding Node replied the first GIST Data message, it would terminate itself.

We observed that the Querying Node state machine is removed when the *Inactive_QNode* timer expires at the *Awaiting Refresh* state

The results we observed from this scenario are the same as we expected.

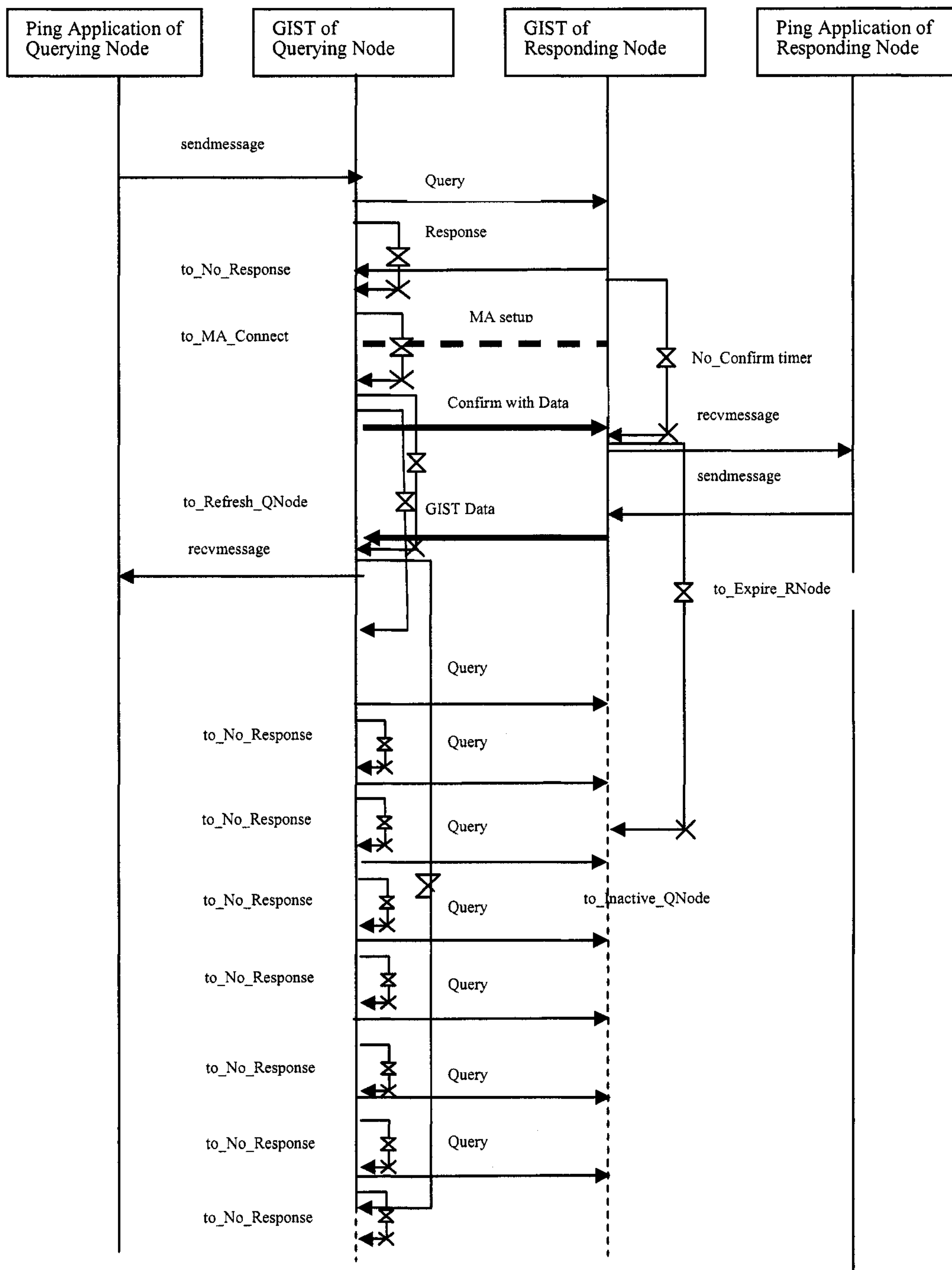


Figure 23: Inactive_QNode Timer

4.6.5 Expire_RNode Timer Validation

In this scenario, we are testing the Expire_RNode timer.

The Expire_RNode timer is set to 30 seconds by default in our implementation. It could expire when the state machine is either in *Established* or *Awaiting Refresh* state. The Expire_RNode is first started when the state machine is transferring from *Awaiting Confirm* to the *Established* state. If the Querying Node is out of service after the Responding Node receives the Confirm, the Expire_RNode will finally expire and the Responding Node state machine will be removed finally.

The approach is given as follows.

- Expire_RNode expiration at *Established* state:

We modified the Querying Node so that the Querying Node terminates itself after it receives the GIST_Data from Responding Node. Then we started both the Querying Node and Responding Node.

The output observed on the log file for this scenario shows that the Responding Node state machine is removed because of the expiration of the Expire_RNode timer, which is the same as what we are expecting from Figure 24.

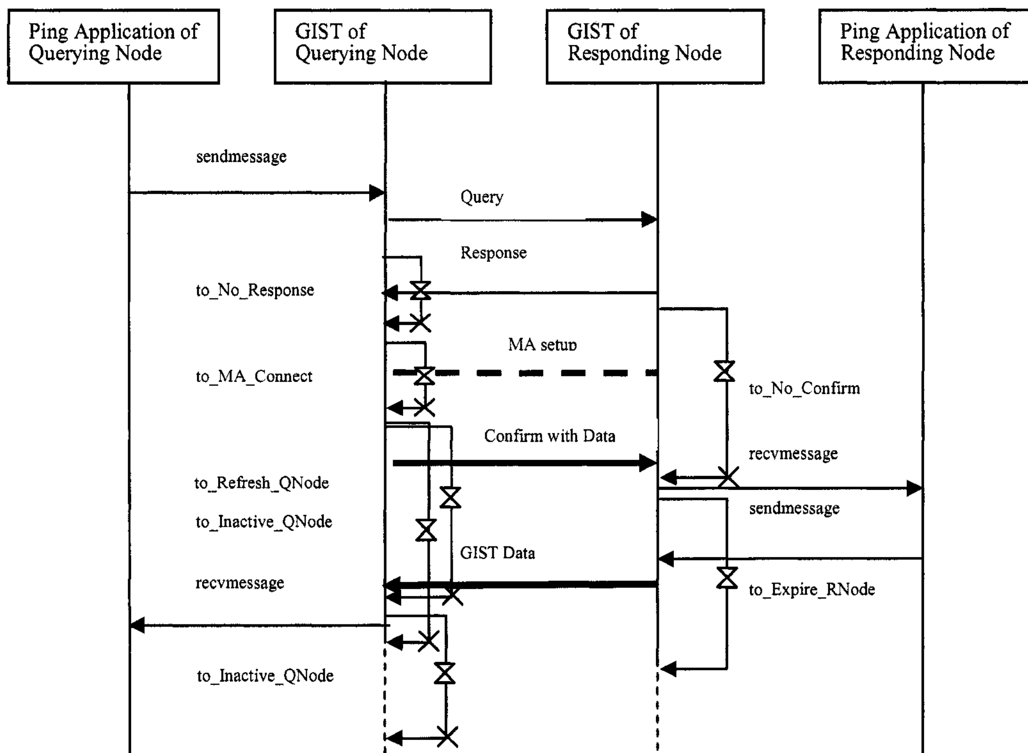


Figure 24: Expire_RNode Timer Time-out at Established State

When the Responding Node state machine is in the *Awaiting Refresh* state after receiving the Query, if the Querying Node is out of service, the Expire_RNode will finally expire and the Responding Node state machine will be removed finally.

The approach is given as follows.

- Expire_RNode expires at *Awaiting Refresh* state

We modified the Querying Node so that it would terminate itself after its Refresh_QNode expires and the first Querying for refreshing is sent out. Then we started both the Querying Node and Responding Node.

The messages observed in the log file are the same as we are expecting from Figure 25.

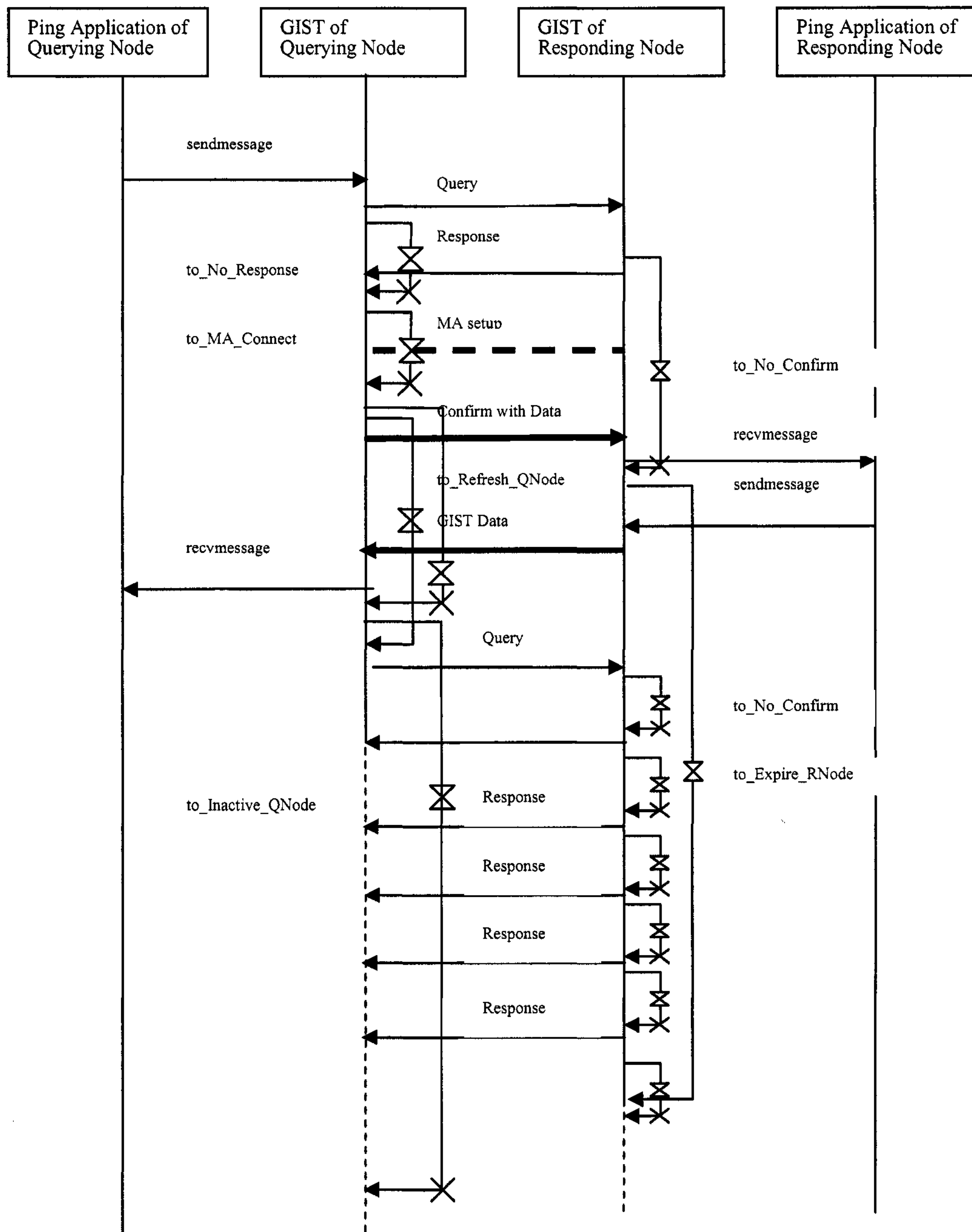


Figure 25: Expire_RNode Timer Time-out at Awaiting Refresh State

4.6.6 No_Confirm Timer Validation

In the scenario, we are testing the No_Confirm timer.

The No_Confirm time can either expire at *Awaiting Confirm* state or at *Awaiting Refresh* state. We validated both cases and the output is the same as what we are expecting. However, we only give a diagram for the case when the state machine is in the *Awaiting Refresh* state. The Response retransmission is using a binary exponential back-off algorithm. The initial timer value is T1=500 ms, which the back off process can increase up to a maximum value of T2 seconds. The T2 in our implementation is 64 * T1=32 seconds. Just as we have done with the timer No_Response, we modified the T2 = 16*T1 = 8 seconds. Therefore, the Responding Node state machine will be removed when No_Confirm timer expires five times (0.5+1+2+4+8=15.5 seconds, which is shorter than the Expire_RNode timer 30 seconds).

The approach is given as follows.

- No_Confirm timer expiration at *Awaiting Refresh* state:

We first modified the T2 to 16 seconds for No_Confirm timer and also modified the code so that it would terminate itself after it sends out the first Query for refreshing after expiration of Refresh_QNode timer. Then we started both the Querying Node and Responding Node.

The output observed in the log file is what we are expecting from Figure 26.

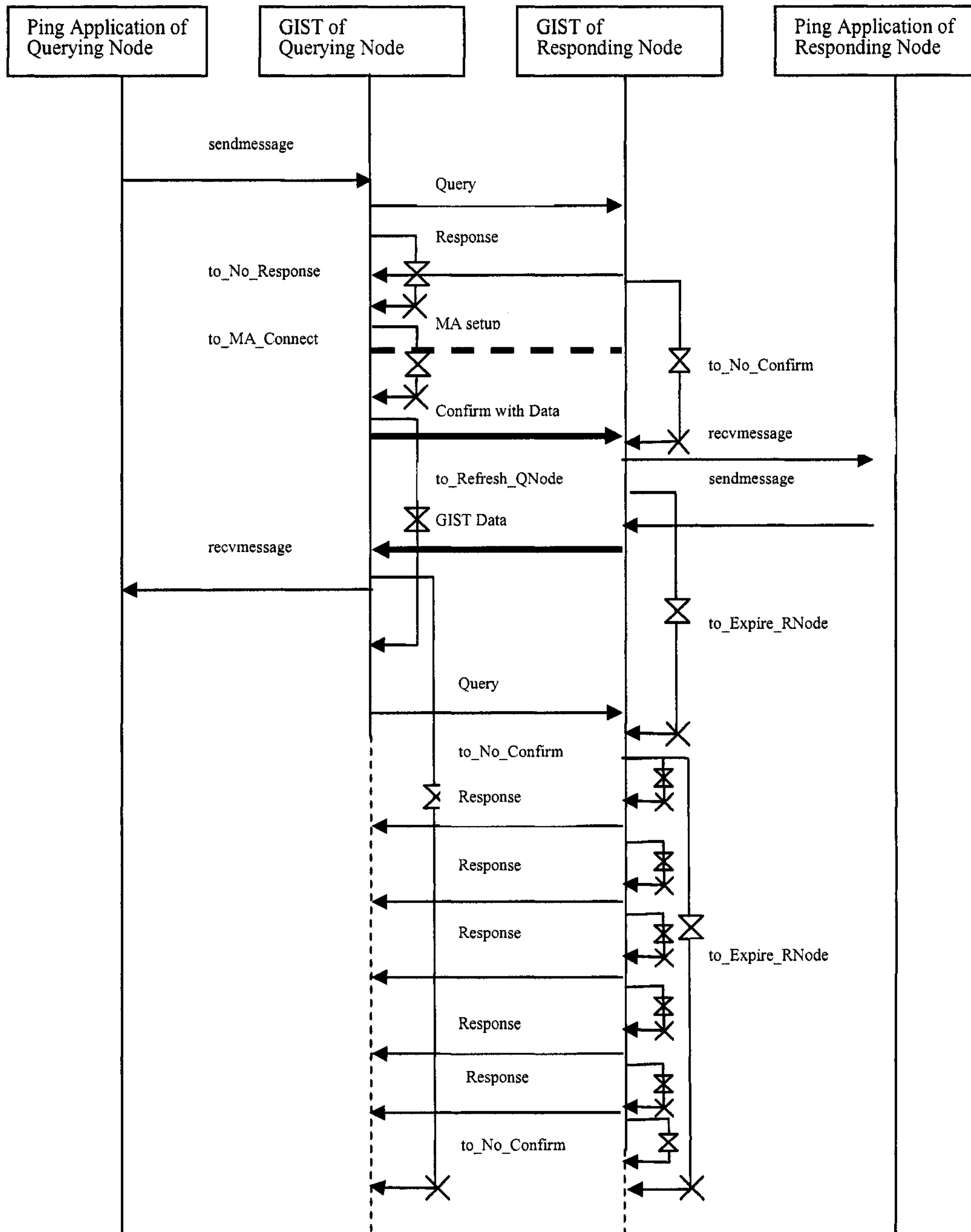


Figure 26: No_Confirm Timer Time-out at Awaiting Refresh State

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Our main purpose of the implementation is to validate the specification of GIST Internet Draft [4]. We tested our implementation in a scaled network by simulating a Flow-Next Hop mapping table.We validated the Messaging Association multiplexing and all of the soft state timers. All of the results are what we are expecting. From this point, we can say the specification of GIST is already well defined. Through the implementation, some minor issues are clarified by the authors of the Internet Draft and we also made some recommendations.

The two layer approach taken by GIST makes adding other signaling application very easy as it is designed with modularity in mind. Leaving the reliable message transmission security and congestion control to the existing transport layer like TCP relieves the GIST from managing message transmission load. It also greatly reduces the GIST complexity. The Messaging Association multiplexing used by GIST can minimize the number of peer relationships and it is very resource efficient.

Through the implementation, we learned how to put timer management and state machine together to construct a soft-state protocol like GIST. We also give out the specific timer management and state machine implementation. We believe that our implementation will be useful for the implementers in the future.

5.2 Future Work

In our implementation, we did not implement the Messaging Association state machine recommended in [4] as we found that the Confirm message is normally sent out every 16 seconds for refreshing and it is barely enough to keep the TCP connection of the Messaging Association active. Rather we added a new timer MA_Connect to the Querying Node state

machine in order to detect the connection failure. So we left implementation of the Messaging Association state machine for future work.

In the specification of Messaging Association of section 6.4 of [4], the author said that "Timers may also be necessary to detect connection failure (e.g., no incoming connection within a certain period), but these are not modeled explicitly". Considering the TCP case, as it is the only one explicitly defined in the Messaging Association, it is better to add such a timer to detect the connection failure especially in the Querying Node when the connect() system call is a blocking call by default. If we change it to non-blocking, we must need a way to detect when the connection could be established. So we explicitly added a new timer MA_Connect timer to the Messaging Association state machine. In the future, it would be useful to the implementer. Please see Figure 27.

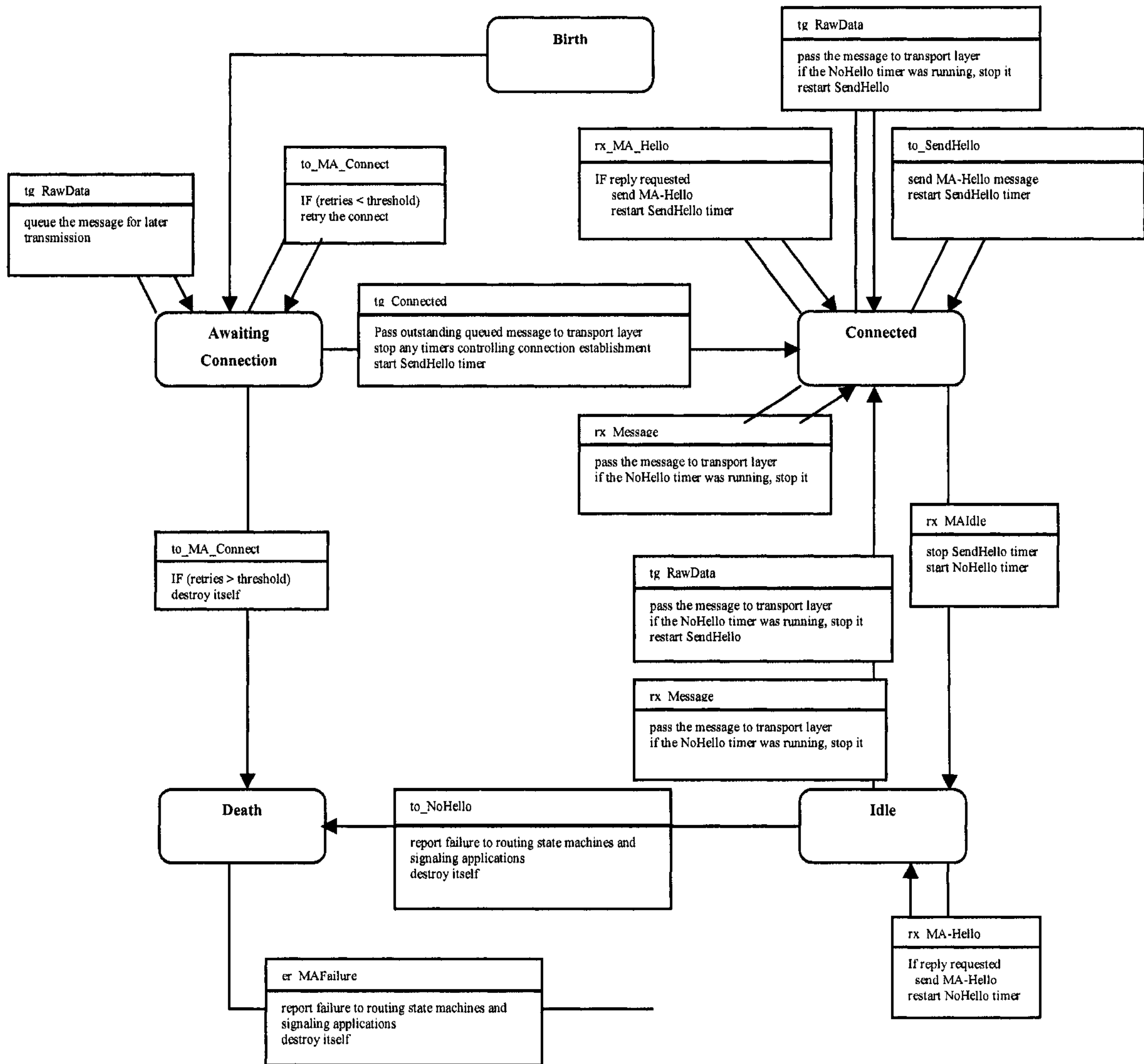


Figure 27: Revised Messaging Association State Machine

Bibliography

[1] R. Hancock, G. Karagiannis, J. Loughney, S. van den Bosch, "Next Steps in Signaling (NSIS): Framework", RFC 4080, June 2005

[2] J. Manner, X. Fu, "Analysis of Existing Quality of Service Signaling Protocols", RFC 4094, December, 2004

[3] M. Brunner, Ed, "Requirements for Signaling Protocols", RFC 3726, April 2004

[4] H. Schulzrinne, R. Hancock, "GIST: General Internet Signaling Transport", draft-ietf-nsis-ntlp-11, August 2006

[5] C. Dickmann, I. Juchem, S. Willert , X. Fu, "A stateless Ping tool for simple tests of GIMPS implementations", draft-juchem-nsis-ping-tool-02

[6] B. Braden, "A Two-Level Architecture for Internet Signaling", <draft-braden-2level-signaling-01>, IETF, November 2002

[7] Braden., R. Ed., et. al., "Resource ReSerVation Protocol (RSVP) -- Version 1 Functional Specification", RFC 2205, September 1997.

[8] H. Schulzrinne, R. Hancock, "GIST: General Internet Signaling Transport", draft-ietf-nsis-ntlp-00, Octobor 2003

[9] Douglas E. Comer, David L. Stevens, Internetworking with TCP/IP Volume II: Design Implementation and Internals, Prentice Hall, 1998

[10] Douglas E. Comer, David L. Stevens, Internetworking with TCP/IP Volume I: Principles Protocols and Architecture, Prentice Hall, 2000

[11] W. Richard Stevens, UNIX Network Programming Volume 2: Interprocess Communications, Prentice Hall, 1998

[12] W. Richard Stevens, UNIX Network Programming Volume 1: Networking APIs, Sockets and XTI , Addison-Wesley, 1998

[13] Michael J. Donahoo, Kenneth L. Calvert, TCP/IP Sockets in C, Practical Guide for Programmers, Morgan Kaufmann, 2000

[14] W. Richard Stevens, TCP/IP Illustrated, Volume 1: The Protocols, Addison-Wesley, 1994

[15] W. Richard Stevens, TCP/IP Illustrated, Volume 2: The Implementation, Addison-Wesley, 1995

[16] T. Sanda, X. Fu, S. Jeong, J. Manner, H. Tschofenig, "Applicability Statement of NSIS Protocols in Mobile Environments", draft-ietf-nsis-applicability-mobility-signaling-07.txt, July 9,2007

[17] T. Tsenov, H. Tschofenig, X. Fu, C. Aoun, E. Davies, "GIST State Machine", draft-ietf-nsis-ntlp-statemachine-04.txt, July 2007

[18] J. Manner, G. Karagiannis, A. McDonald, "NSLP for Quality-of-Service Signaling", draft-ietf-nsis-qos-nslp-15.txt, July 2007

[19] M. Stiemerling, H. Tschofenig, C. Aoun, E. Davis, "NAT/Firewall NSIS Signaling Layer Protocol (NSLP)", draft-ietf-nsis-nslp-natfw-15.txt, July 2007

[20] H. Chaskar, Ed. "Requirements of a Quality of Service (QoS) Solution for Mobile IP", RFC 3583, September 2003

[21] H. Tschofenig, D. Kroeselberg, "Security Threats for Next Steps in Signaling (NSIS)", RFC 4081, June 2005

[22] H. Tschofenig, R. Graveman, "RSVP Security Properties", RFC 4230, December 2005

[23] C. Shen, H. Schuzrinne, S. Lee, J. Bang, "NSIS Operation Over IP Tunnels", draft-ietf-nsis-tunnel-03.txt, September 2007