Customizable Services for Application-Layer Overlay Networks

by

Tony Yu Zhao

A thesis submitted in conformity with the requirements
for the degree of M.A.Sc
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2012 by Tony Yu Zhao

Contents

1 Introduction **1**

1.1 Application-layer Overlay Networks 1

1.1.1 Data Delivery in Application-layer Overlay Networks 4

1.2 Advantages of Data Delivery Services Realized at the Application-layer . 5

1.3 Towards a Flexible Overlay Architecture 7

1.3.1 Data Delivery Services as Overlay Middleware Services 8

1.3.2 Current Challenges 10

1.4 Contributions 11

1.5 Organization 13

2 Related Works **14**

2.1 Configurable Network Services 14

2.1.1 The x-kernel 14

2.1.2 Extensible Transport 15

2.1.3 Protocol Boosters 15

2.1.4 NetServ 16

2.2 Active Networks 17

2.2.1 Softnet 17

2.2.2 Programmable Switches - A Discrete Approach 18

2.2.3 ACTIVE IP 18

2.2.4 ANTS - An Integrated Approach 18

2.2.5 ESP 19

2.3 Automatic Generation of Network Protocols 19

2.3.1 Network Protocols Expressed as State Machines 20

2.3.2 Network Protocols Expressed as Grammar 20

2.4 Automatic Generation of Overlay Networks 21

2.4.1 MACEDON 21

2.4.2 Declarative Networking 21

3 Background **23**

3.1 HyperCast 23

3.1.1 Overlay Socket 24

3.1.1.1 Adapter 25

3.1.1.2 Overlay Node 27

3.1.1.3 CSA Processor 27

3.1.1.4 Forwarding Engine 28

3.1.1.5 MessageStore 31

3.1.2 HyperCast Services 35

3.1.2.1 Hop-to-Hop Acknowledgement 36

3.1.2.2 End-to-End Acknowledgement 41

3.1.2.3 Other Services 45

3.2 SCXML: State Chart XML 46

3.2.1 Apache Commons SCXML 46

3.2.1.1 SCXML Engine 48

3.2.1.2 Domain 48

3.2.1.3 Bridge 48

3.3 XQuery 49

3.3.1 Saxon XQuery and XSLT Processor 49

4 Design of Customizable Services 51

4.1 Finite-State Machine 53

4.2 Executable Specification 57

4.2.1 Inputs 58

4.2.1.1 Basic Events 58

4.2.1.2 Composite Events 61

4.2.1.3 Mapping of Events to Input Symbols 64

4.2.2 History of Basic Events 67

4.2.3 Outputs 68

4.2.3.1 Actions - Set of Overlay Network Primitives 69

4.2.3.2 Mapping of Output Symbols to Actions 72

4.2.3.3 Format of Executable Specification 73

4.2.4 Expressiveness of Executable Specification 74

4.2.4.1 Custom Control Messages 74

4.2.4.2 Expressiveness of Composite Events 74

4.2.4.3 Types of Expressible Services 76

4.3 Executable Specification Processor 83

4.3.1 Creation and Deployment of Services 83

4.3.2 Using Services 84

4.3.3 Finite-State Machine Execution 86

5 Implementation 90

5.1 MessageStore Extensions 90

5.1.1 Implementation Overview 93

5.1.2 A Service as an Executable Specification 97

5.1.2.1 Header and Meta Data 98

5.1.2.2 Events 100

5.1.2.3 States and State Transitions 104

5.1.2.4 Actions 106
5.1.3 New MessageStore Objects 109
5.1.3.1 ExecSpec Object 109
5.1.3.2 GenericFSM Object 112
5.1.4 Service Execution 116
5.1.4.1 Message Arrival Event 116
5.1.4.2 Timer Expiration Event 120
5.1.4.3 Handling Events 122
5.1.4.4 Executing Actions 124
5.1.5 List of Supported Actions - Network Primitives 131
5.1.6 Discussion of the Implementation 133
5.1.7 Mechanisms to Improve MessageStore Performance 134
5.1.7.1 Pre-allocation of Finite-State Machines 134
5.1.7.2 "Warm-starting" the Apache Commons SCXML Engine 137
5.2 Service Deployment 140
5.2.1 Services Server 140

6 Evaluation 144

6.1 Experiment Setup 146
6.1.1 Testbed Network 146
6.1.2 Network Topology 146
6.1.3 Overlay Network Topology using DT Protocol 148
6.1.4 Underlay Substrate 151
6.2 Overview of Experiments 152
6.2.1 Measurement Methodology 152
6.2.2 Configuration Parameters 152
6.2.3 Experiment Topologies 154
6.2.4 List of Experiments 155

6.3 Hop-to-Hop Acknowledgment in Single-Hop Network 157
6.3.1 Java-based Implementation and SCXML-Based Implementation Performance at send rate of 100 Mbps, 10 Mbps, and 1 Mbps 157
6.3.2 Bottleneck Analysis of Java-based Implementation 164
6.3.2.1 The effect of the number of stored FSMs in MessageStore on per message processing time 170
6.3.3 Bottleneck Analysis of SCXML-based Implementation 174
6.3.4 Performance Improvements for SCXML-Based Implementation . . 181
6.3.4.1 Performance improvements using preallocation of FSMs 181
6.3.4.2 Performance improvements using warm-start of SCXML engine 185
6.3.4.3 Performance at send rate of 100 Mbps, 10 Mbps, and 1 Mbps using warm-start 188
6.3.4.4 Overhead of performance improving mechanisms 192
6.3.5 Sustainable Throughput and Delay 193
6.4 Hop-to-Hop Acknowledgment in Multi-Hop Network 196
6.4.1 Two-Hop Transfer for Java-based Implementation (at send rate of 4.0 Mbps) and SCXML-Based Implementation (at send rate of 1.0 Mbps) Performance 196
6.4.1.1 Bottleneck Analysis of Java-based Implementation . . . 200
6.4.1.2 Bottleneck Analysis of SCXML-based Implementation . 201
6.4.1.3 Sustainable Throughput and Delay 205
6.4.2 Five-Hop Transfer for Java-based Implementation (at send rate of 1.8 Mbps) and SCXML-Based Implementation (at send rate of 0.7 Mbps) Performance 207
6.5 End-to-End Acknowledgment in Single-Hop Network 209

6.5.1 Java-based Implementation and SCXML-Based Implementation Performance at send rate of 100 Mbps, 10 Mbps, and 1 Mbps 209
6.5.2 Bottleneck Analysis of Java-based Implementation 214
6.5.3 Bottleneck Analysis of SCXML-based Implementation 220
6.5.4 Sustainable Throughput and Delay 224
6.6 End-to-End Acknowledgment in Multi-Hop Network 226
6.6.1 Two-Hop Transfer for Java-based Implementation (at send rate of 4.0 Mbps) and SCXML-Based Implementation (at send rate of 1.0 Mbps) Performance 226
6.6.2 Five-Hop Transfer for Java-based Implementation (at send rate of 0.8 Mbps) and SCXML-Based Implementation (at send rate of 0.7 Mbps) Performance 230
6.7 Memory Usage 232
6.7.1 Hop-to-Hop Acknowledgement Service in Single-Hop Network . . 232
6.7.2 End-to-End Acknowledgement Service in Single-Hop Network . . 237

7 Conclusions and Future Work **239**
7.1 Conclusions 239
7.2 Future Work 241

Bibliography **243**

List of Figures

1.1 A application-layer overlay network. Overlay nodes with logical addresses 1, 2, 3 corresponding to substrate elements with different IPv4 address. Network applications communicate to the overlay nodes using logical addresses are not aware of the substrate addresses or topology. Overlay nodes sends and receives messages using the IPv4 interfaceses and port numbers. 2

1.2 Flooding of messages of 3 nodes 5

1.3 Differences between data delivery over the Internet and over an application-layer overlay network 6

1.4 Evolution of application-layer overlay networks 8

1.5 Service layers in overlay middleware 9

1.6 Adding data delivery services for application-layer overlay middleware . . 10

3.1 An overlay socket and its components 26

3.2 (a) A HyperCast application message (b) A HyperCast application message marked with a service and a payload (c) A HyperCast application message marked with a service without a payload. 29

3.3 How a message is process by its finite-state machine (FSM) in MessageStore 32

3.4 Services in HyperCast 35

3.5 (a) The sender (S) delivers a message to its children nodes (node 1 and node 2). (b) When node 1 and node 2 get the message, they send H2H ACKs to parent node S. In the meantime, the data is forwarded to their children node (node 3, 4, 5, 6). (c) When node 3, 4, 5, 6 receive the message, they send H2H ACKs to their respective parents nodes 1 and 2. 37

3.6 State machine diagram for Hop-to-Hop Acknowledgement service 40

3.7 (a) The sender (S) delivers a message to its children node (node 1 and node 2), which forward the message to their children nodes (nodes 3, 4, 5, 6). (b) When leaf nodes 3, 4, 5, and 6 receive the message, they send a Full ACK to their parent nodes. (c) The ACKs are merged at node 1 and 2. (d) Node 1 and 2 will send Full ACKs to their parent node, the sender (S). 42

3.8 State machine diagram for End-to-End Acknowledgement service 44

3.9 Interacting components of the Apache Commons SCXML framework . . 47

4.1 Abstraction of a service 51

4.2 A service decomposed 52

4.3 A finite-state machine diagram 54

4.4 A Mealy machine and a Moore machine with the same behaviour 55

4.5 Using timers to specify a given sequence of actions 60

4.6 Finite-state machine for message selection service 78

4.7 Finite-state machine for connection establishment service between two nodes 79

4.8 Finite-state machine for a leaky bucket transmitter 80

4.9 Message processing in *MessageStore* 84

4.10 Execution models for finite-state machines 87

4.11 Finite-state machine execution engine and its interaction with the overlay node 88

5.1 Message handling in MessageStore 91
5.2 Overview of implementation in MessageStore to use executable specification for services 94
5.3 Mapping of a message arrival or timer expiration to an event in the executable specification 95
5.4 Mapping of an action in the executable specification to a Java method . . 96
5.5 The structure for an executable specification of a service defined by a finite-state machine 97
5.6 Relationships of the SCXML representation of the executable specification to the ExecSpec object 110
5.7 A GenericFSM object extends the AbstractStateMachine object and instantiate three actions 112
5.8 Component of GenericFSM object and interactions with ExecSpec object 114
5.9 Processing of a message arrival event in MessageStore 119
5.10 Processing of a timer expiration event in MessageStore 121
5.11 Event processing by the handleEvent method in GenericFSM Object . . 123
5.12 Creating an action in GenericFSM object using SCXML custom actions interface 126
5.13 Action processing by the dispatchAction method in GenericFSM Object . 127
5.14 Pre-allocation of GenericFSM objects during MessageStore initialization . 135
5.15 How preallocated FSMs are used in MessageStore 135
5.16 Warm-start mechanism during MessageStore initialization 138
5.17 Interactions of MessageStore with the Services Server 140
5.18 How MessageStore handles a message with a new serviceID 141

6.1 Network topology setup of experiments 147
6.2 Application-layer overlay topology setup of experiments 150
6.3 Sequence number versus time for Hop-to-Hop Acknowledgement 159

6.4 Per message delay versus time for Hop-to-Hop Acknowledgement 160

6.5 Throughput versus time for Hop-to-Hop Acknowledgement 163

6.6 MessageStore per message processing time versus time for Hop-to-Hop Acknowledgement of the Java-based implementation 166

6.7 Backlog of messages buffered in MessageStore for data and control message, and the number of stored FSMs in MessageStore versus time for Hop-to-Hop Acknowledgement for the Java-based implementation 168

6.8 MessageStore per message processing time versus time for Hop-to-Hop Acknowledgement of the Java-based implementation without removing any MessageStore stored FSMs 171

6.9 Backlog of messages buffered in MessageStore for data and control message, and the number of stored FSMs in MessageStore versus time for Hop-to-Hop Acknowledgement for the Java-based implementation without removing any MessageStore stored FSMs 172

6.10 Throughput versus time for Hop-to-Hop Acknowledgement for the Java-based implementation without removing any MessageStore stored FSMs . 173

6.11 MessageStore per message processing time versus time for Hop-to-Hop Acknowledgement for the SCXML-based implementation 175

6.12 Component breakdown of MessageStore per message processing time versus time for Hop-to-Hop Acknowledgement for the SCXML-based implementation 176

6.13 Backlog of messages buffered in MessageStore for data and control message, and the number of stored FSMs in MessageStore versus time for Hop-to-Hop Acknowledgement for the SCXML-based implementation . . 178

6.14 MessageStore per message processing time versus time for Hop-to-Hop Acknowledgement for the SCXML-based implementation with 0, 100, 200 preallocated FSMs 182

6.15 MessageStore per message processing time versus time for Hop-to-Hop Acknowledgement for the SCXML-based implementation with warm-start and 0, 100, 200 preallocated FSMs 186
6.16 Sequence Number versus time for Hop-to-Hop Acknowledgement with warm-start for the SCXML-based implementation 189
6.17 Per message delay versus time for Hop-to-Hop Acknowledgement with warm-start for the SCXML-based implementation 190
6.18 Throughput versus time for Hop-to-Hop Acknowledgement with warm-start for the SCXML-based implementation 191
6.19 Throughput versus time for Hop-to-Hop Acknowledgement 193
6.20 Per message delay versus time for Hop-to-Hop Acknowledgement 194
6.21 Average throughput versus time for Hop-to-Hop Acknowledgement . . . 195
6.22 Average per message delay versus time for Hop-to-Hop Acknowledgement 195
6.23 Sequence number versus time for Hop-to-Hop Acknowledgement for the Java-based implementation 197
6.24 Sequence number versus time for Hop-to-Hop Acknowledgement for the SCXML-based implementation 198
6.25 Throughput versus time for Hop-to-Hop Acknowledgement for the Java-based implementation 199
6.26 Throughput versus time for Hop-to-Hop Acknowledgement for the SCXML-based implementation 199
6.27 MessageStore per message processing time versus time for Hop-to-Hop Acknowledgement for the Java-based implementation 202
6.28 MessageStore per message processing time versus time for Hop-to-Hop acknowledgement for the SCXML-based implementation 204
6.29 Average throughput versus time for Hop-to-Hop Acknowledgement . . . 205
6.30 Average per message delay versus time for Hop-to-Hop Acknowledgement. 206

6.31 Average throughput and average delay for each receiver for Hop-to-Hop Acknowledgement using the Java-based implementation 207
6.32 Average throughput and average delay for each receiver for Hop-to-Hop Acknowledgement using the SCXML-based implementation 208
6.33 Sequence number versus time for End-to-End Acknowledgement 210
6.34 Per message delay versus time for End-to-End Acknowledgement 211
6.35 Throughput versus time for End-to-End Acknowledgement 213
6.36 MessageStore per message processing time versus time for End-to-End Acknowledgement of the Java-based implementation 215
6.37 Backlog of messages buffered in MessageStore for data and control message, and the number of stored FSMs in MessageStore versus time for End-to-End Acknowledgement for the Java-based implementation 218
6.38 MessageStore per message processing time versus time for End-to-End Acknowledgement for the SCXML-based implementation 221
6.39 Backlog of messages buffered in MessageStore for data and control message, and the number of stored FSMs in MessageStore versus time for End-to-End Acknowledgement for the SCXML-based implementation . . 222
6.40 Average throughput versus time for End-to-End Acknowledgement 224
6.41 Average per message delay versus time for End-to-End Acknowledgement 225
6.42 Average throughput versus time for End-to-End Acknowledgement 227
6.43 Average per message delay versus time for End-to-End Acknowledgement 227
6.44 Average throughput and average delay for each receiver for End-to-End Acknowledgement using the Java-based implementation 230
6.45 Average throughput and average delay for each receiver for End-to-End Acknowledgement using the SCXML-based implementation 231
6.46 Heap memory usage versus time for Hop-to-Hop Acknowledgement for the Java-based implementation 234

6.47 Heap memory usage versus time for Hop-to-Hop Acknowledgement for the SCXML-based implementation 234
6.48 Heap memory usage versus time for Hop-to-Hop Acknowledgement for the SCXML-based Implementation (Sender (S), with warm-start, with initial and max heap size of 512MB) 236
6.49 Heap memory usage versus time for Hop-to-Hop Acknowledgement for the Java-based implementation 237
6.50 Heap memory usage versus time for End-to-End Acknowledgement for the SCXML-based implementation 238

List of Tables

3.1 Messages for the Hop-to-Hop Acknowledgment service 38

3.2 Timers for the Hop-to-Hop Acknowledgment service 39

3.3 Messages for the End-to-End Acknowledgment service 43

3.4 Timers for the End-to-End Acknowledgment service 43

3.5 Other HyperCast Services 45

4.1 Example of one-to-one mapping between basic events and input symbols 64

4.2 Example of many-to-one mapping between basic events and input symbols 65

4.3 Example of one-to-many mapping between basic events and input symbols 65

4.4 Types of Actions 68

4.5 Format of an executable specification 73

4.6 Composite events expressible using first-order logic 75

4.7 State transitions for message selection service 78

4.8 State transitions for connection establishment service 81

4.9 State transitions for a leaky bucket transmitter 82

5.1 Events for Hop-to-Hop Acknowledgment Service 100

5.2 Supported Actions in MessageStore 132

6.1 Definition of Measured Metrics 153

6.2 List of Experiments for Hop-to-Hop Acknowledgement service 155

6.3 List of Experiments for End-to-End Acknowledgement service 156

6.4 MessageStore per message processing time of first 5 data messages at the Sender (S) and Receiver (R) without warm-start 183
6.5 MessageStore per message processing time of first 5 data messages at the Sender (S) and Receiver (R) with warm-start 185
6.6 Overhead of Performance Improving Mechanisms 192

Chapter 1

Introduction

Networking today serves increasingly complex distributed applications in increasingly heterogeneous networking environments (*e.g.* wireless ad-hoc networks, sensor networks and mesh networks). Applications such as distributed file-sharing or peer-to-peer multimedia streaming typically implement custom network services based on application requirements. As new network applications emerge, a more efficient way of deploying these services should be investigated.

1.1 Application-layer Overlay Networks

Application-layer overlay networks provide a feasible solution for the introduction of custom network services without changes to the existing network infrastructure.

An *overlay network* is a logical network built on top of one or more existing networks, called *underlay networks*. Communication endpoints, called *nodes*, in the overlay network can be thought of as being connected by virtual links, called *logical links*, each of which corresponds to a path through one or more *physical links* in the underlying network. For example, the Internet was built as an overlay network over the *telephone network* [23] and *peer-to-peer* file-sharing systems are overlay networks built on top of the Internet [21].

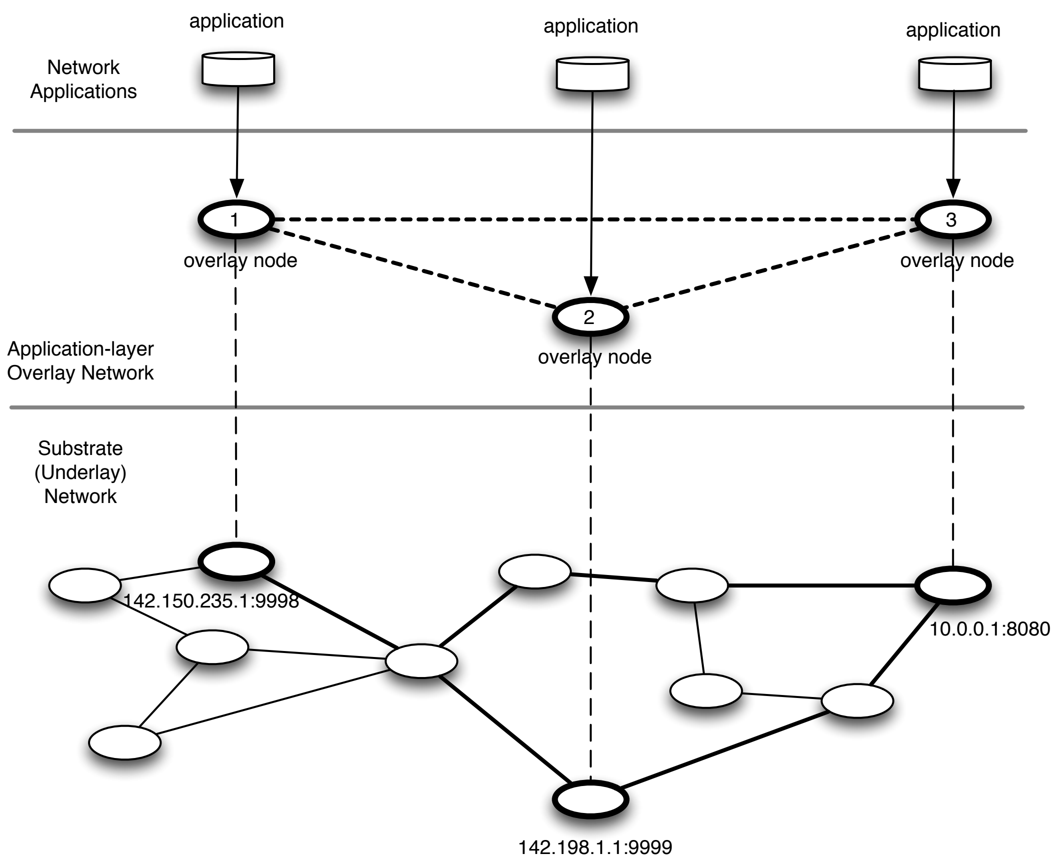


Figure 1.1: A application-layer overlay network. Overlay nodes with logical addresses 1, 2, 3 corresponding to substrate elements with different IPv4 address. Network applications communicate to the overlay nodes using logical addresses are not aware of the substrate addresses or topology. Overlay nodes sends and receives messages using the IPv4 interfaceses and port numbers.

An *application-layer overlay network* is an overlay network where all the nodes in the network are end-systems running applications. In comparison to lower-layer network elements such as routers and switches, these nodes are user applications that can implement capabilities beyond basic operations of storing and forwarding packets. Application-layer overlay networks can construct and manipulate their own network topologies without modifying lower-level network infrastructures. Application-layer overlay networks are used as a means to provide custom network services using custom network topologies, custom addressing schemes, and custom routing semantics not available in the underlying networks. An application-layer overlay network is illustrate in Figure 1.1.

Many of these application-layer overlay networks exist in the literature. For example, Chord [42], Tapestry [51], Pastry [40] are application-layer overlay networks providing distributed directory services. Distributed directories are optimized data repositories supporting data search, data browsing, data storage and data retrieval operations. The aforementioned systems implement distributed hash tables (DHTs). Distributed hash tables create an overlay network topology that efficiently maps identifiers/keys to overlay nodes and stores data in (key, value) pairs much like a hash table. In particular, Chord [42] nodes organize themselves in a logical ring topology, known as the Chord ring. The Resilient Overlay Network (RON) is an application-layer overlay network on top of the existing Internet routing substrate that allows distributed Internet applications to detect and recover from path outages and periods of degraded performance [23]. RON nodes monitor the quality of Internet routing path and decide whether to route packets directly over the Internet or indirectly by way of other overlay nodes in RON's own logical topology. End System Multicast (ESM) [31] is an application-layer overlay network that supports a multicast service for streaming live, high quality video and audio to a large number of end-hosts. ESM maintains its network topology as a tree of overlay nodes and consistently manipulates the tree to minimize end-to-end latency.

Applications such as distributed file sharing requires reliable transmission of data from one sender to many receivers. Peer-to-peer multimedia streaming applications benefits from reliable, in-order data communication. We believe that application-layer overlay networks serve as a possible solution for large-scale, distributed applications with specialized data delivery requirements.

1.1.1 Data Delivery in Application-layer Overlay Networks

In any communication system, a specified set of *data delivery* capabilities are provided as a service. Data delivery is the process of moving data (in the form of messages) containing data from a sender to a receiver. Once the sender (S) and the receiver (R) are connected, data can be transferred from S to the R along the established path $(S, N_0, N_1, ..., N_n, R)$; where $N_0, N_1, ..., N_n$ denote intermediate nodes along the path.

The sender transmits a sequence of data units $(d_0, d_1, d_2, d_3, ..., d_{n-1}, d_n)$ containing information to the receiver. Under ideal conditions, the data received at the receiver is identical to the original sequence. However, in an unreliable communication network, links connecting network nodes are subject to data duplication, reordering, and loss. A data delivery service that allow data duplication, reordering, or loss is called a *best-effort* service. For example, User Datagram Protocol (UDP) provides a best-effort data delivery service. On the other hand, a *reliable* service guarantees that a data unit d_i transmitted from the sender to the receiver is eventually received. A *no-duplicates* service guarantees that no data unit d_i transmitted by the sender is received more than once by the receiver. An *in-order* service guarantees that the sequence of data units received at the receiver, at any time, is in the same order as they were sent by the sender. The Transmission Control Protocol (TCP) a reliable, in-order data delivery service.

General purpose transport-layer protocols like TCP cannot guarantee end-to-end data delivery beyond best-effort. In an application-layer overlay network, there is no notion of end-to-end connection establishment at the transport-layer between two overlay nodes with a path of more than one hop. TCP can transfer messages reliably and in-order between two directly connected overlay nodes, but an overlay node can still drop an message after it has been received but before it could be forwarded. A cause of such a drop can be a change in the network topology while a message is en route from the sender to the receiver. In application-layer overlay networks where nodes have high mobility, if a reliable data service is required, it must be provided by the application. As a result, many overlay network applications provide custom data delivery services, on top of UDP. These custom data delivery services are realized by the use of control messages (acknowledgements, negative-acknowledgements, etc.) at the application-layer.

1.2 Advantages of Data Delivery Services Realized at the Application-layer

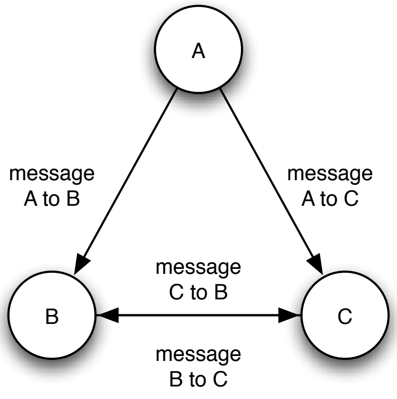
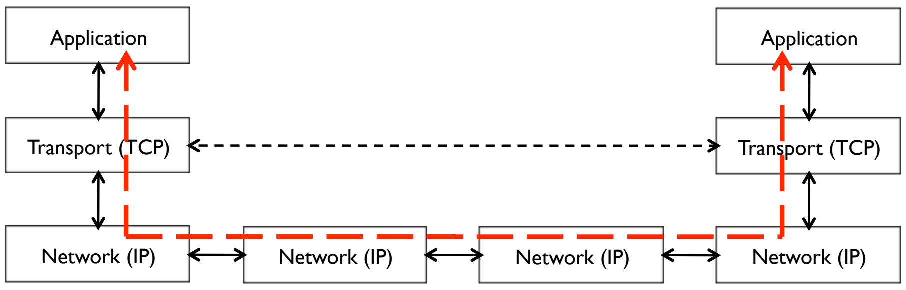


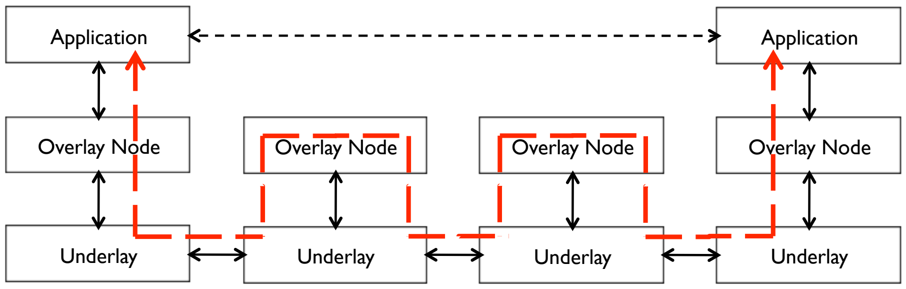
Figure 1.2: Flooding of messages of 3 nodes

Consider the problem of eliminating duplicate packets in an application-layer overlay network that uses *flooding* for distributing messages over an UDP underlay. In flooding, every overlay node tries to forward every message to each of its neighbours. Flooding is very wasteful in terms of the networks bandwidth consumption. Duplicate packet transmission occur, unless certain precautions are taken. For example in Figure 1.2, if node B and node C are neighbours with node A and neighbours with each other they will receive a message sent from source node A twice. For example, node B does not know that the message has already been received; the original message from source A and the duplicate message from source C are treated as separate messages. If nodes retain information on received messages and do not forward perviously received messages, duplicate messages can be eliminated.

The above scenario is an example of a *value-added* data delivery service in an application-layer overlay network. The duplicate-elimination service, at the application-layer, adds value beyond the *best-effort* service.



(a) Internet



(b) Application-layer overlay network

Figure 1.3: Differences between data delivery over the Internet and over an application-layer overlay network

In application-layer overlay networks, data delivery services are provided at the application-layer compared to data delivery over the Internet which are provide at the transport layer. As illustrated by Figure 1.3a, over the Internet, node-to-node connection is established at the network-layer. Messages are transmitted using IP in the network-layer hop by hop across IP nodes using best effort data delivery and the end-hosts use control messages to provide better than best-effort service at the transport-layer. In application-layer overlay networks, node-to-node connection is established at the same layer as data delivery. This allow data delivery to leverage the intermediate nodes ($N_0, N_1, ..., N_n$) on the path from the sender to the receiver.

For example, in a network sensitive to delay, rather than only having packet retransmission processing at the sender and receiver, intermediate nodes can work cooperatively to retransmit lost packets to improving latency. As another example of cooperative processing, each overlay node can aggregate data before sending it to its next-hop destination, thus reducing overall network bandwidth.

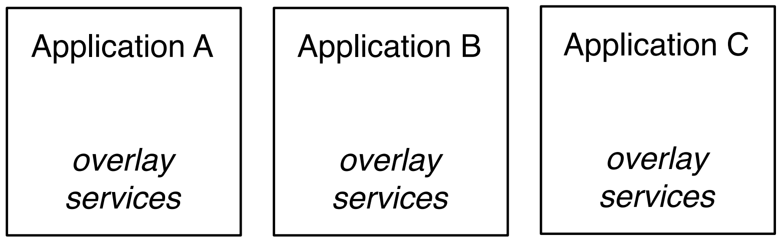
In order to leverage intermediate nodes over the Internet, the network-layer must be extended universally, which makes large-scale deployment unfeasible.

Moreover, data delivery services implemented at the application-layer overlay networks gives applications control over delivery semantics. Data delivery services can be optimized for specific network applications and network environments. Furthermore, many services can be operated simultaneously on the same set of overlay nodes for different applications.

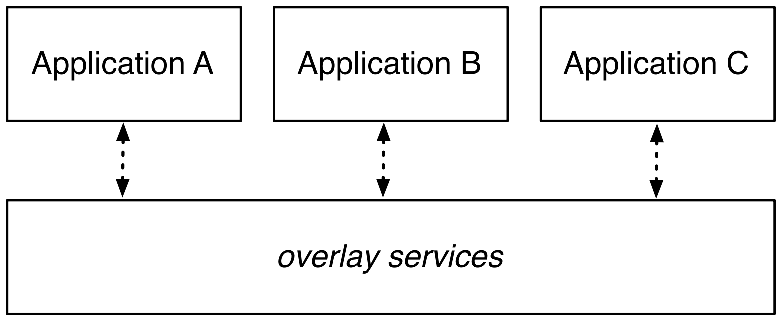
1.3 Towards a Flexible Overlay Architecture

Network applications implementing application-layer overlay networks are composed of (1) overlay services such as node discovery, topology construction and management, addressing, routing, and (2) application-specific services. Early application-layer overlay networks such as Chord [42], RON [23], and ESM [31] integrated application-specific services and overlay services. In this approach, every network application using the overlay network must provide an implementation of application-layer overlay network as illustrated in Figure 1.4a.

In the integrated approach, network applications in the same domain (peer-to-peer file-sharing, multimedia streaming, etc.) implement similar overlay functions. In a *middleware* approach, shown in Figure 1.4b, overlay functions are available as services to be used by multiple applications. The middleware exposes interfaces for user applications to access services provided by the application-layer overlay network. For example,



(a) Integrated approach



(b) Middleware approach

Figure 1.4: Evolution of application-layer overlay networks

openDHT [38] is middleware providing a distributed hash table (DHT) service to network applications through simple `put` and `get` operations.

Modern overlay middleware systems such as iOverlay [35] and HyperCast [6] implement more general middleware infrastructures supporting multiple overlay topologies, node discovery algorithms, routing schemes, multicast models, etc. These overlay middleware systems are designed with interfaces between the application and overlay services allowing developers to create network applications without detailed knowledge of internal details of application-layer overlay networks.

1.3.1 Data Delivery Services as Overlay Middleware Services

There are advantages for overlay middleware to support data delivery services where all network applications can take advantage of a unified implementation. Overlay middleware systems can provide a platform to integrate data delivery services into overlay middleware functionality. Overlay middleware can allow network applications to select appropriate data delivery semantics on-demand. In this way, data delivery services can be viewed as services built on top of overlay services such as addressing, node discovery, topology management, and routing as illustrated by Figure 1.5 since data delivery assumes that connectivity between overlay nodes already exist.

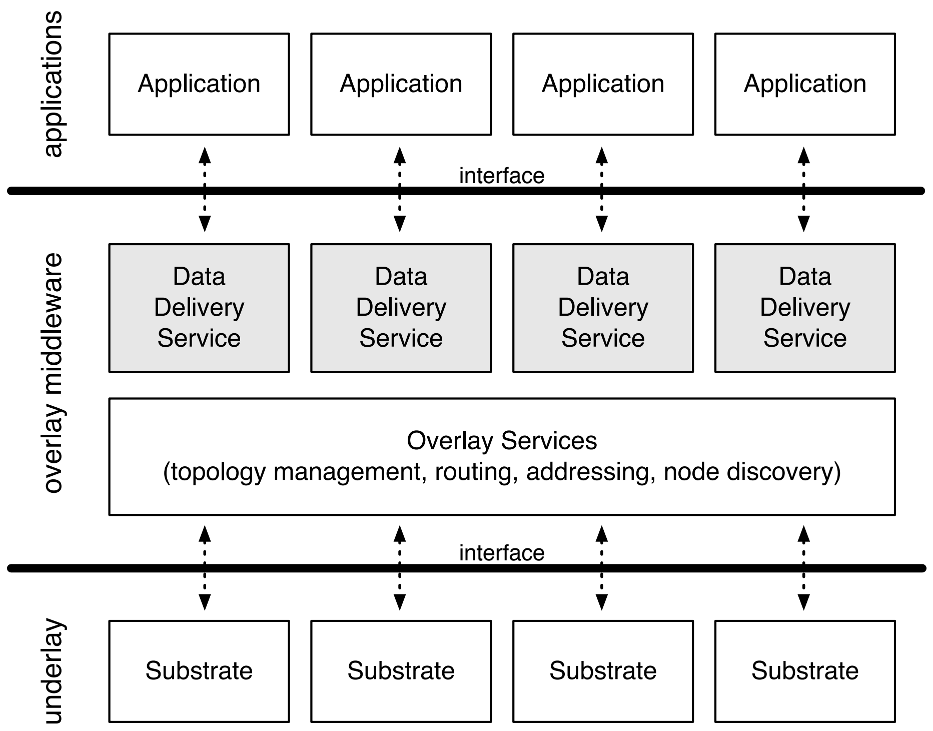


Figure 1.5: Service layers in overlay middleware

Some overlay middleware systems, such as HyperCast [6], implement a selected set of data delivery services and provide an interface for network applications to add new services whenever more functionality is needed. HyperCast [6] instantiates and executes data delivery services based on markings messages. New data delivery services can be loaded on-demand to the overlay middleware by applications. This flexible mechanism is illustrated in Figure 1.6. Here an application introduces to the overlay middleware a new service, which is dynamically loaded into the overlay middleware. Network applications determine the data delivery service to use based on marking messages. When an overlay node receives a message marked for a certain service, the appropriate data delivery service is instantiated. A state machine at the overlay nodes is created for each marked messages. Control messages can then update a message's state information when they are received at the overlay node.

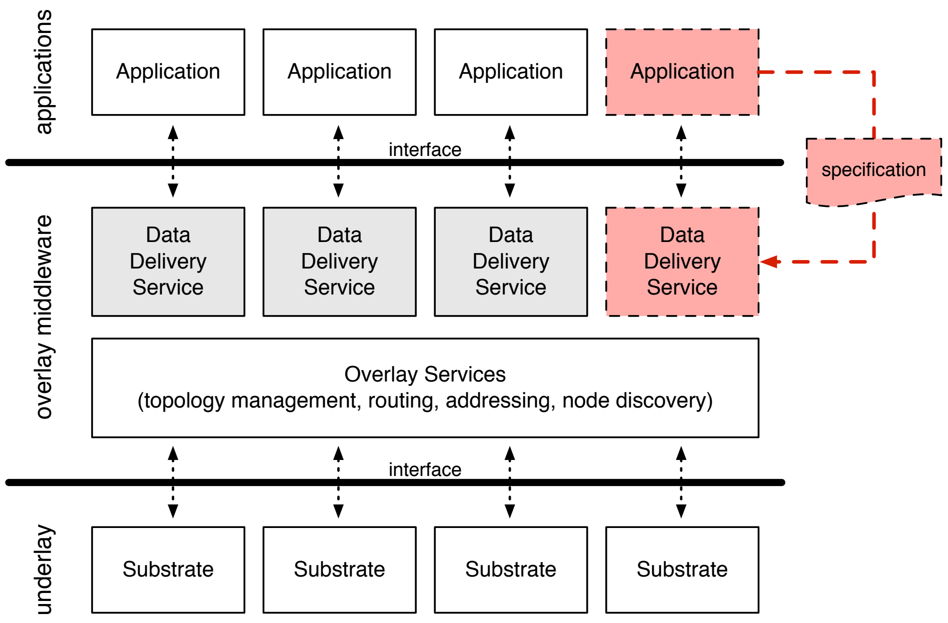


Figure 1.6: Adding data delivery services for application-layer overlay middleware

1.3.2 Current Challenges

A data delivery service is composed of (1) a portion unique to each service and (2) common tasks shared between many services. The common tasks include creating messages, sending messages, etc. The portion that is unique to each service is its *behavior*, *i.e.*, the processing flow. Existing overlay middleware provide a shared library for the common tasks but their is no unified set of common tasks between different overlay middleware. Hence a data delivery service may require a different implementation for each different overlay middleware.

Overlay middleware may have many data delivery service that do the same thing. There is no easy way to determine if any two implementation are of the same data delivery service. Hence, common data delivery services may be implemented multiple times. Also there is no way to determine if implementations of the same data delivery service on different overlay middleware perform the same tasks. This motivates having services expressed in a form other than code.

Currently, data delivery services are hard-coded, in the sense that it is not possible to pass executable code between overlay nodes. If overlay nodes can execute a service directly from a description of the service, then it becomes feasible to deploy services to overlay nodes without modifying the software. If overlay nodes receive descriptions of services and execute them directly, a mechanism to validate a service specification during run-time is needed. The specification needs to be checked against malicious behaviour and security breaches.

In this thesis, we present an overlay middleware design for deploying new data delivery services that address these challenges and potentially foster the rapid evolution of services for overlay middleware systems.

1.4 Contributions

In this thesis, we propose a novel method for network applications to declare and add new data delivery services into an overlay middleware system. We extend the current architecture for data delivery services employed by HyperCast [6]. We define each service in terms of a specification. A *specification* for a service is the set of requirements that the service must satisfy which includes its behaviour logic. These specification can be dynamically loaded by overlay nodes.

We define a format for service specifications and provide an execution environment to execute these specifications directly. Within the scope of our research, we define data delivery services that react to two types of asynchronous events 1) arrival of messages and 2) expiration of timers. The behaviour portion of a service is described using a finite-state automata. We create a small set of *overlay network primitives* representing actions that are the common tasks shared by all data delivery services. The set of overlay network primitives are actions supporting a wide variety of data delivery services, but not necessarily every possible service.

We use XML [17] to describe service specifications. XML is a W3C standard that provides a simple format that is widely understood making it interoperable in many application domains. We validate the service specifications using XML schema descriptions during runtime.

For our prototype system, we leverage the architecture of HyperCast [27]. A component is developed, which allows applications to dynamically define data delivery services and deploy it in the overlay network. We employ a SCXML execution engine [1] to execute these service specifications. Since all data structures and functions used by the data delivery service are defined in XML, we create a general and interoperable prototype that could be adapted to any overlay middleware system. Our methodology easily allows our system to be improved, as better XML parsers and XML schema validators become available in the future.

In the remaining chapters, we describe in detail our design and describe classes of data delivery problems that can be solved by our approach. We believe that our design provides an effective method which allows services to evolve rapidly, and change functionality dynamically based on network application requirements.

1.5 Organization

This thesis is organized as follows. Chapter 2 surveys related work. Chapter 3 gives background detail on the enabling technologies for our system. Architectural goals and details of our approach are provided in Chapter 4. Chapter 5 describes the implementation of our prototype. We evaluate our methodology in Chapter 6. We conclude with discussions and future work in Chapter 7.

Chapter 2

Related Works

This chapter describes literature that inspired our work. The majority of the research adds support for new network services in the network-layer, the transport-layer, or the application-layer using application-layer overlay networks.

2.1 Configurable Network Services

A number of efforts support network service configurability by user applications. Some of these are described below.

2.1.1 The x-kernel

The x-kernel is a framework for implementing network protocols [34]. X-kernel provides a library of protocol elements for manipulating messages, addresses, events, memory tables, threads, etc. There is operating system support for a generic mechanism for composing these protocol elements based on layering to create larger protocols. The x-kernel project demonstrated that large protocols can be expressed in terms of layered *micro-protocols* while still maintaining high performance [37]. While the main objective of x-kernel was to design, build, and test protocol implementation techniques, it provided a dynamic method to compose protocol elements on a per-packet basis allowing new transport layer services to be introduced.

2.1.2 Extensible Transport

There has been a steady stream of research over the years on configurable and extensible transport solutions to unify different transport protocols. Most notably Bridges *et al.* [28] proposes a general transport architecture where all possible transport functions are assembles from small modules called *micro-protocols*. *Micro-protocols* are small algorithms that implement individual mechanisms of reliable delivery, congestion control, in-order delivery, flow control, etc. Applications create a customized data transport solution by selecting a set of *micro-protocols* and assembling them together. The result is a flexible mechanism to address application defined transport requirements.

Since there is no reasonable way to determine the types of *micro-protocols*, new *micro-protocols* are created whenever more functionality is needed, which may results in increasingly complex *micro-protocol* implementations and integration problems over time [28]. Given the possibility of having a large number of *micro-protocols*, a non-trivial mechanism is needed to compose them together in the correct order to achieve the desired behaviour without creating harmful side-effects.

2.1.3 Protocol Boosters

Feldmeier *et al.* [32] presented a methodology for enhancing network layer and transport layer protocols using elements called *protocol boosters*. The approach taken by the researchers leverages the layered TCP/IP suite and allows sub-layers to be inserted between two end-hosts in a manner that is transparent to the rest of the protocol stack. *Protocol boosters* allow dynamic protocol customization by network applications. *Protocol boosters* are agents (software modules) that reside anywhere in the network and operate independently or cooperatively with other *protocol boosters* using their own control messages to “boost” (add new functionality) to existing transport protocols. The emphasis of *protocol boosters* is to improve performance by adapting network processing to both the application and network environment without altering existing network protocol behaviour. For example, acknowledgment messages (ACKs) can be compressed on a system using a *protocol booster* to reduce ACK latency over asymmetric transmission channels. Also, video transmitted over a lossy link may be augmented or boosted by a forward correction (FEC) module using this approach.

2.1.4 NetServ

NetServ [41] is an extensible architecture deploying new core network services. Resources and functions on a network node is broken up into small, reusable building blocks. A new core network service is composed from these building blocks and executed using Java which provides security, resource management, and portability. Network services, modules that are Java JAR files, are deployed dynamically using the component framework for Java.

2.2 Active Networks

In an *active network*, routers or switches perform customized computations on the messages flowing through them. These networks are "active" in the sense that nodes, routers and switches can perform computations on, and modify, the packet contents. Packets contain executable code. This processing can be customized on a per-user or per-application basis. Users use their packets to program the network. The concept of active network emerged from the Defence Advanced Research Projects Agency (DARPA) in the 1990s. Active networks emerged to address (1) the difficulty of integrating new protocols into a standardized network infrastructure, (2) poor network performance due to redundant network operations at several protocol layers, and (3) the difficulty of accommodating new services in the existing network architecture. Active networks gave rise to the idea of a programmable network. Since active networks operate at the network layer they are suited to implement network protocol extensions such as routing.

2.2.1 Softnet

Softnet [50] is an experimental, distributed packet radio network from the 1980s that allowed users to define their own high level-services (datagram, virtual calls, file transfer, mailboxes, etc.), as well as changes of link layer protocols. Softnet nodes were dual processor systems with one processor dedicated to the link and the other to user tasks. Packets in Softnet are considered to be programs and are interpreted at intermediate nodes. A standard set of functions allows the packets to control the node hardware, *e.g.*, allowing packets to retransmit themselves to other nods, store programs in remote nodes, and synchronize with other packets. Softnet is the first example of a programmable network and inspired interest in active networks.

2.2.2 Programmable Switches - A Discrete Approach

The *programmable switch* approach to active networking maintains the existing packet format and provides a discrete mechanism for downloading programs [22]. The SwitchWare project developed a programmable switch that allows digitally signed type-checked modules to be loaded into network nodes. Out-of-band program loading is used to support new services, called *value-added services*. For example, users can inject custom processing programs into routers and send packets through these routers to use the value-added services. Services are separated from the switching by moving their processing to special processors. This approach is "passive" since users directly extend the functionality of switches beforehand and network elements can only perform pre-loaded computations on user data.

2.2.3 ACTIVE IP

ACTIVE IP is an attempt to inject programs into the network to tailor node processing [49]. The ACTIVE IP system was a proof-of-concept active network designed for network probing. It allowed Tcl code fragments to be tagged with user IP packets that are custom processed using a set of available primitives. ACTIVE IP tried to show that network services can be composed from a relatively small set of primitives. It also showed that Tcl programs can be compactly expressed thus allowing them to be transferred as payload.

2.2.4 ANTS - An Integrated Approach

The *capsule* approach replaces passive packets with packets containing executable programs which encapsulated in transmission frames and executed at each network node [44, 48]. In this approach, every message is a program. ANTS is an integrated approach to active networks where user programs are integrated with packet data. When a capsule arrives at an active node, its content is executed. The capsule have the ability to invoke build-in primitives which provide access to node resources. The primitives build into each node include (1) packet manipulation, (2) node resource access, and (3) control of packet flow. The capsules approach allows application-specific processing to be injected into the network in order to build custom services such as multicast, data aggregation, etc. Tennenhouse and Wetherall [43] studied issues related to component specifications, active storage, multicast acknowledgement fusion, and network-based-traffic filtering. One key idea of the capsules approach is that the programs contained within each packet are dispatched to a *transient execution environment* where they are safely executed. The capsules consume short term storage which is eventually "garbage-collected" by the node. The execution environment restricts resource access outside of the environment to only built-in primitives.

2.2.5 ESP

To address the need for network support for end-to-end services, *ephemeral state processing* (ESP) [29] provides special "ESP" packets to create and manipulate small amount of temporary state at routers using predefined operations, called instructions. States are "soft" and reclaimed if they are not refreshed by new ESP packets. ESP is different from ANTS in that each ESP packet can invoke only a single instruction on a state making the per-packet processing time bounded. The researchers identified three applications of ESP: (1) controlling packet flow, *i.e.*, creating ephemeral state at router interfaces where packet forward/drop decisions will be made, (2) calculations on user data and data aggregation, and (3) discovering topology information.

2.3 Automatic Generation of Network Protocols

We next describe efforts to express network protocols as specifications.

2.3.1 Network Protocols Expressed as State Machines

There exist a body of work on automatic generation of network protocol implementations from specification of the OSI protocols expressed as finite-state machines. Examples include Esterel [26], Estelle [45], LOTOS [45], SDL [45], etc. These specifications were compiled into implementations [30, 47] using custom compilers. The primary focus of this research is on supporting formal specification and verification of protocols while increasing readability and reusability of code. Here, state machines transitions are triggered by arrival of messages and timer events from the network or from the local node.

2.3.2 Network Protocols Expressed as Grammar

An alternative to state machines is RTAG [24], where network protocols are expressed as a grammar. Incoming messages and events are expressed as tokens causing reductions in grammar rules. The behaviour of the protocols is held on the parser stack rather than encapsulated as a finite-state machine. An advantage of this approach is that specification forms can be restricted through syntax of tokens as in network programming languages. For example, PLAN [33], a Packet Language for Active Networks, is a programming language for active networks. It is designed to be compact so programs can be carried directly in packets.

2.4 Automatic Generation of Overlay Networks

The following body of literature explores the automatic generation of overlay networks from a specification.

2.4.1 MACEDON

MACEDON [39] adopts a finite-state machine approach to automatically generate overlay network code from a specifications. MACEDON can implement a number of overlay protocols (*e.g.*, Chord, Pastry, etc) with only a few hundred lines of MACEDON specification code. The performance of a MACEDON network is comparable to a custom C++ implementation.

2.4.2 Declarative Networking

Declarative Overlays [36] is a methodologies for automatically generating overlay networks using a declarative language. The prototype system, *P2*, enables applications to express application-layer overlay networks in a compact and reusable specification. Different from *protocol centric* overlay specification approaches such as *MACEDON* [39], which specify overlay execution via automata for event and message handling, *P2* uses a *structure-centric approach* by viewing the network as a relational database. *P2* uses a declarative language based on database query languages to describe relationships between network nodes. This methodology is similar to the approach taken for distributed hash tables (DHTs), which specifies overlays by focusing on a network graph structure (hypercube, torus, de Bruijn graph, small-world graph, etc.), whose invariant properties must be maintained via asynchronous messaging [36].

P2 maintains the overlay by running distributed queries over network graphs. The structure of the overlay network is specified as the relationship between entries in the tables representing overlay nodes, and control messages (represented as <key, value>

tuples) are queries on those tables. In *Declarative Routing* [36], the authors demonstrated that recursive queries can be used to express a variety of wired and wireless routing protocols in a concise manner.

Declarative approaches cannot express applications requirements for different combinations of reliable data delivery, in-order data delivery, congestion control, and ow control end-to-end services without severe modifications to the declarative language constructs [36]. Implementing these using declarative networking require support for timers (e.g., retransmission timers, state timers). The current declarative language does not support multi-rule atomicity. This leaves specifications susceptible to race conditions [36].

Chapter 3

Background

In this chapter, we describe three technologies that are relevant to our research. Our design and implementation extend and leverage these software packages.

3.1 HyperCast

HyperCast [6] is an overlay middleware implemented in Java that provides abstractions and primitives for network applications to build and maintain application-layer overlay networks. HyperCast provides a facility that enable development of new services. Our work extends HyperCast to enable customizable services.

Network applications use HyperCast to construct an application-layer overlay network on top of underlay networks, called *substrate networks* in HyperCast, without changing Internet routing or switching infrastructure. In HyperCast, a substrate network is any packet-based communication channel, which provides either a unicast or multicast packet delivery. TCP/IP, UDP, IP, Ethernet, 802.11 networks, and other overlay networks are all examples of substrate networks.

Our design and implementation is part of the HyperCast project. Below we describe portions of HyperCast middleware that are relevant relevant to our research.

3.1.1 Overlay Socket

HyperCast introduces the concept of *overlay sockets* as end-points of communication in an overlay network. Overlay sockets are equivalent to overlay nodes. A HyperCast overlay network is simply viewed as a collection of overlay sockets. Applications can use the APIs of the overlay sockets to send application messages to other overlay sockets. Overlay sockets are created and configured by applications with attributes that specify the name of the overlay network to join, the type of overlay network topology (Delaunay Triangulation, Hypercube, Spanning Tree, DHT, etc.), the communication protocol to use for overlay messages (UDP, TCP), etc. To maintain the overlay network, overlay sockets exchange control messages, called *protocol messages* among each other.

Each overlay socket defines two types of identifiers. The *logical address* uniquely identifies an overlay within a overlay network. Overlay network functions such as naming, routing, and topology management use logical addresses. For example, a node in HyperCast determines a next hop destination for each message using only the destination node's logical address of that message. The format of the logical address depends on the network topology of the overlay network. For example, a logical address can be a positive integer (Spanning Tree) or a multi-dimensional coordinate spaces (x, y) (Delaunay Triangulation). An overlay socket can have multiple heterogenous substrate networks. Each of these substrate networks is identified by an *substrate address*. A substrate address consists of an *underlay address* and information that uniquely identifies the substrate (*address realm* and *protocol type*). The underlay address identifies a network attachment point (unicast) or multiple network attachment points (multicast or anycast) and is used to send and receive messages. Examples of address realms are public IPv4 network, private IPv4 networks, IPv6 network, ethernet, etc. The protocol type identifies different instance of the same network. HyperCast support multiple formats of substrate addresses. For a simple example, if the substrate network is TCP in a public IPv4 address realm, then the substrate address is $< publicip, tcp, 128.100.100.128 : 8080 >$.

In HyperCast, the overlay node identifier, *i.e.*, the logical address, and its substrate network attachment points, *i.e.*, the substrate addresses, are decoupled. Applications only deal with logical addresses, and do not see the substrate addresses of an overlay socket. Each logical address can be associated with multiple substrate addresses and each node in the overlay network maintains a set of address binding, called the *address list*. An overlay node does not need to have the complete address list of another overlay node to be able to contact it. An overlay node that is connected to only one substrate can only use substrate addresses that belong to that particular substrate.

HyperCast overlay sockets provide application programming interfaces (API) for network application. The overlay socket API allows applications to create a new overlay network, to join and leave existing overlay networks, and to send and received data from other overlay nodes once the overlay network is configured. Before an overlay socket is created, the application configures the components of the overlay socket such as the the overlay network to join, the type of overlay topology to use, the size of buffers, etc. Once created, the overlay socket can attempt to join an overlay network specified by the overlay network identifier. Once the overlay socket successfully joins the overlay network, it can create, transmit, and receive application messages. Application messages can be send via unicast, multicast, or flood.

Each overlay socket isolates overlay functions such as message forwarding, topology management, within well-defined modules. Figure 3.1 shows the main components of an overlay socket and their relationships to each other.

3.1.1.1 Adapter

Each overlay socket provides an abstraction layer of the substrate networks using an adapter. The overlay socket uses the adapter to send and receive messages to substrate networks identified by substrate addresses. The adapter maintains two buffers for incoming messages, one for protocol messages and one for application messages. The adapter

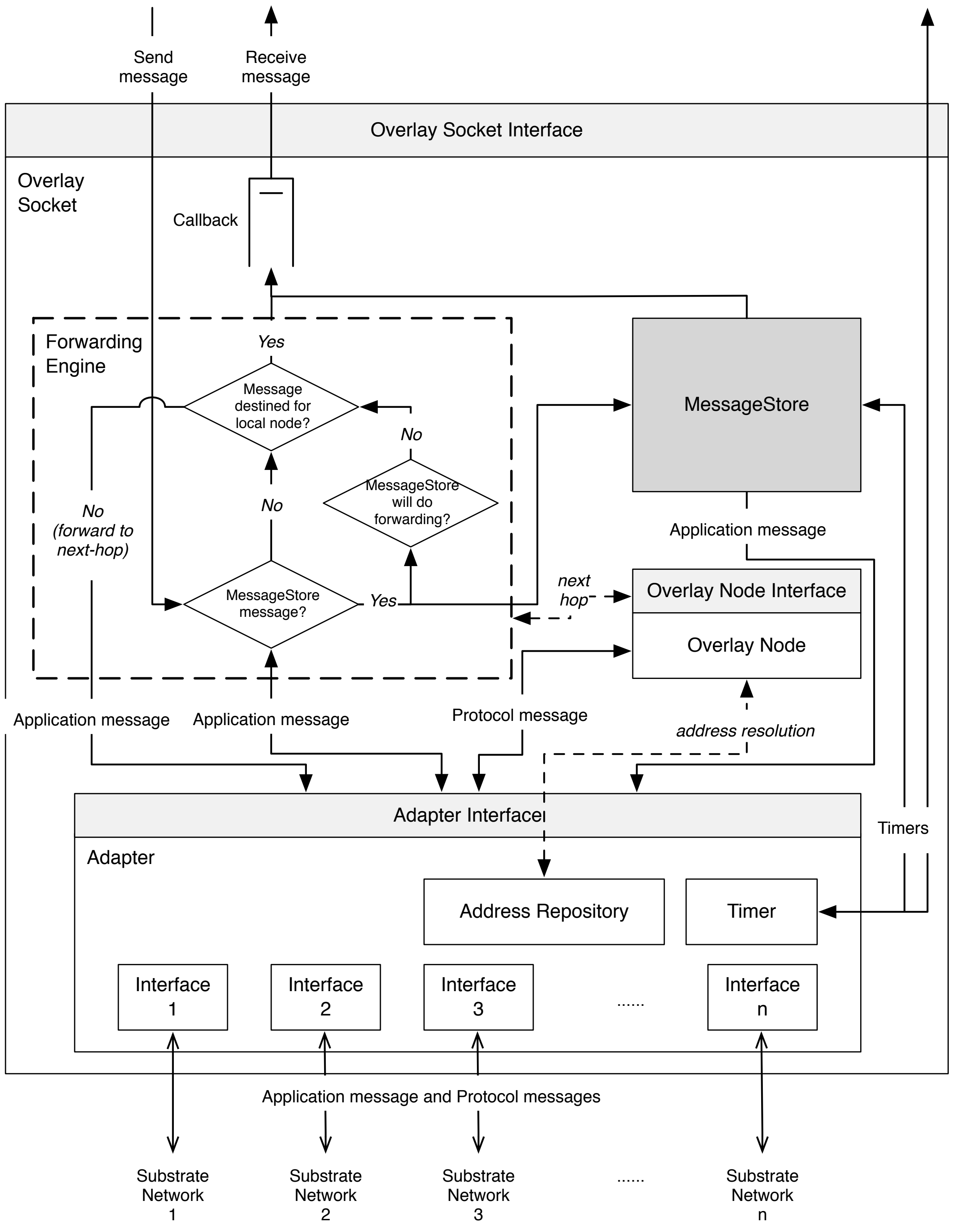


Figure 3.1: An overlay socket and its components

passes messages to other overlay socket components to be processed.

The adapter has one interface for each substrate network to which the overlay socket is connected. Each interface provides the send and receive functions for that substrate network. Some example interfaces are UDP Unicast interface, UDP Multicast interface, TCP interface, Ethernet interface, etc.

The adapter is responsible for the management of substrate addresses. The address repository is a subcomponent of the adapter that maintains the address bindings between the logical address and underlay addresses. The address repository is able to resolve a logical address to a substrate address. Moreover, the address repository allows other overlay socket components to add, remove, and update address bindings.

The adapter also contains a timer thread which provide timing service for all components of the overlay socket.

3.1.1.2 Overlay Node

The overlay node is responsible for maintaining the overlay network topology. It runs a protocol that establishes and maintains the overlay network topology using protocol messages. The overlay node is the only component in the overlay socket that is aware of the overlay topology. Each type of overlay topology has its own type of overlay node.

3.1.1.3 CSA Processor

The CSA processor is constructed as a layer between the adapter and overlay node and is transparent to both. All incoming and outgoing protocol messages are passed through the CSA processor. The CSA processor control the selection of the substrate addresses associated with a logical address.

3.1.1.4 Forwarding Engine

The forwarding engine implements the forwarding mechanics for application messages. It is responsible for sending, receiving, and forwarding application messages in the overlay network.

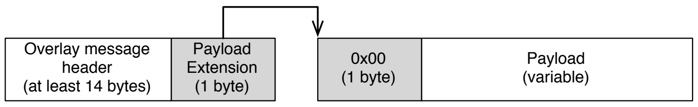
In HyperCast, all application messages are transmitted along trees that are embedded in the overlay network topology. Here we provide some definition regarding these embedded tree nodes that we will use later in this thesis. The topmost node in a tree is called the **root** node. Each node in a tree has zero or more **child** nodes, which are below it in the tree. A node that has a child is called that child node's **parent** node. A node has at most one parent. A **leaf** node is any node that does not have child nodes. A node p is an **ancestor** of a node q if it exists on the path from the root to node q. The node q is called a **descendant** of p.

Multicast messages are forwarded downstream with the sender as the root of the tree. Unicast messages are sent upstream in an embedded tree with the receiver at the root. When flooding is used, an overlay socket forwards application messages to all its neighbours in the overlay topology and all neighbours that receive the message will forward the message to all of its neighbours. The forwarding mechanisms are realized based solely on the logical addresses of application messages. Forwarding decision are made based on the destination logical address for unicast messages and the source logical address for multicast messages.

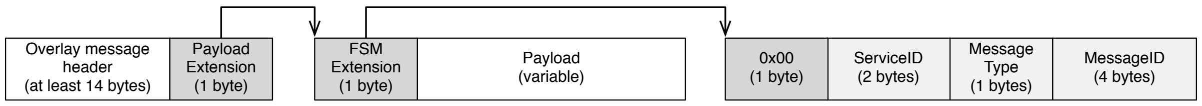
When an overlay socket receives an application messages it is passed to the forwarding engine. An application can receive application messages through a blocking read or through a non-blocking callback. If the message the local overlay socket is the destination of the message, it is passed to the application. If the message has to be forwarded, the forwarding engine communicates with the overlay node to determine the next-hop(s) and forwards the message.

The forwarding engine is also responsible for delivering application messages marked with services to the MessageStore. By default, HyperCast provides a best-effort (unreliable, unordered, possible duplication) delivery service for application messages. Application messages can be marked as a MessageStore message to realize an enhanced delivery service. The description of the MessageStore is described in the next section. Here, we describe how the Forwarding Engine processes these special messages.

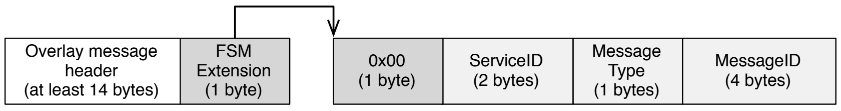
A default HyperCast application message is composed of a header and a sequence of extensions as seen in Figure 3.2(a). The header of an application message specifies parameters such as version, delivery mode, source logical address, previous-hop logical address, hop limit, etc. Following the overlay message header is a set of extensions. The extensions are linked by the extension field as shown in dark grey. The sequence of extensions are terminated with the "0x00" in the extension field. The data for the application message is linked as an extension called "Payload Extension".


Overlay message header (at least 14 bytes)
Payload Extension (1 byte)
0x00 (1 byte)
Payload (variable)

(a)



(b)



(c)

Figure 3.2: (a) A HyperCast application message (b) A HyperCast application message marked with a service and a payload (c) A HyperCast application message marked with a service without a payload.

For HyperCast application messages containing payload marked with an enhanced delivery service, the application message contains an additional extension called the *FSM*

Extension as shown in Figure 3.2(b). This extension identifies the type of service in the `ServiceID` field. The `MessageType` field identifies a message type that is depended on the service. The `MessageID` field specifies a message identifier that is created at the source of the message when the application payload is sent for the first time. This message identifier should uniquely identify a message among all messages currently processed by all overlay sockets of the overlay network. However, uniqueness is not enforced by HyperCast. Payload messages marked with an enhanced delivery service is called a MessageStore *data message*. Additionally, an application message can be marked with an enhanced delivery service without containing payload data as shown in Figure 3.2c. This type of message is called a MessageStore *control message*.

A comprehensive description of the message formats in HyperCast can be found in the HyperCast document [8].

When sending or receiving application messages marked with an enhanced delivery service, the forwarding engine first checks if the application message has an *FSM Extension*. If the message has this extension, the message is cloned and the cloned message is passed to the MessageStore to be processed. A thread in the MessageStore processes the cloned message currently with the forwarding engine from this point onward. The original message resumes processing in the forwarding engine. If the message is to be received by the application, it is the responsibility of the MessageStore to forward the cloned message to the application. This is because MessageStore determines when and how a message should be forwarded to the application yet messages destined for other nodes in the overlay network should be forwarded as soon as they are received by the forwarding engine.

In the next section, we describe in detailed the MessageStore component.

3.1.1.5 MessageStore

MessageStore is a component of the the overlay socket responsible for realizing services beyond the default best-effort delivery service. It is an optional component inside an overlay socket. MessageStore is initialized with the HyperCast configuration object when the overlay socket is created. There is a thread responsible for processing application messages delivered to the MessageStore. When the overlay socket joins the overlay network, this thread is started. When the overlay socket leaves the overlay network, this thread is terminated.

In HyperCast, the MessageStore provides a set of services that can be invoked when application messages are sent marked with an enhanced delivery service. For each service, a finite-state machine (FSM) specifies how messages are handled by the MessageStore. The service also specifies the supported delivery mode.

When MessageStore receives a new message marked with a service's identifier (`ServiceID`), a finite-state machine instance for that service is created for that message. MessageStore creates a new finite-state machine instance for every data message received with a new message identifier ("MessageID"). The finite-state machine instance stores the received data message corresponding to the message identifier, and additional control information. Using the finite-state machine, the MessageStore is determines the state of the data message. It then can send or receive control messages to or from other overlay nodes in the overlay network.

Figure 3.3 illustrates the steps involved in processing a MessageStore message. The following steps occur:

1. The message is received at the MessageStore by the method `receiveMessage`. The message is checked to determine if its a data message containing payload or a control message. The message is then written to the MessageStore FIFO queues `DataBuffer` (for data messages) or `ControlBuffer` (for control message)

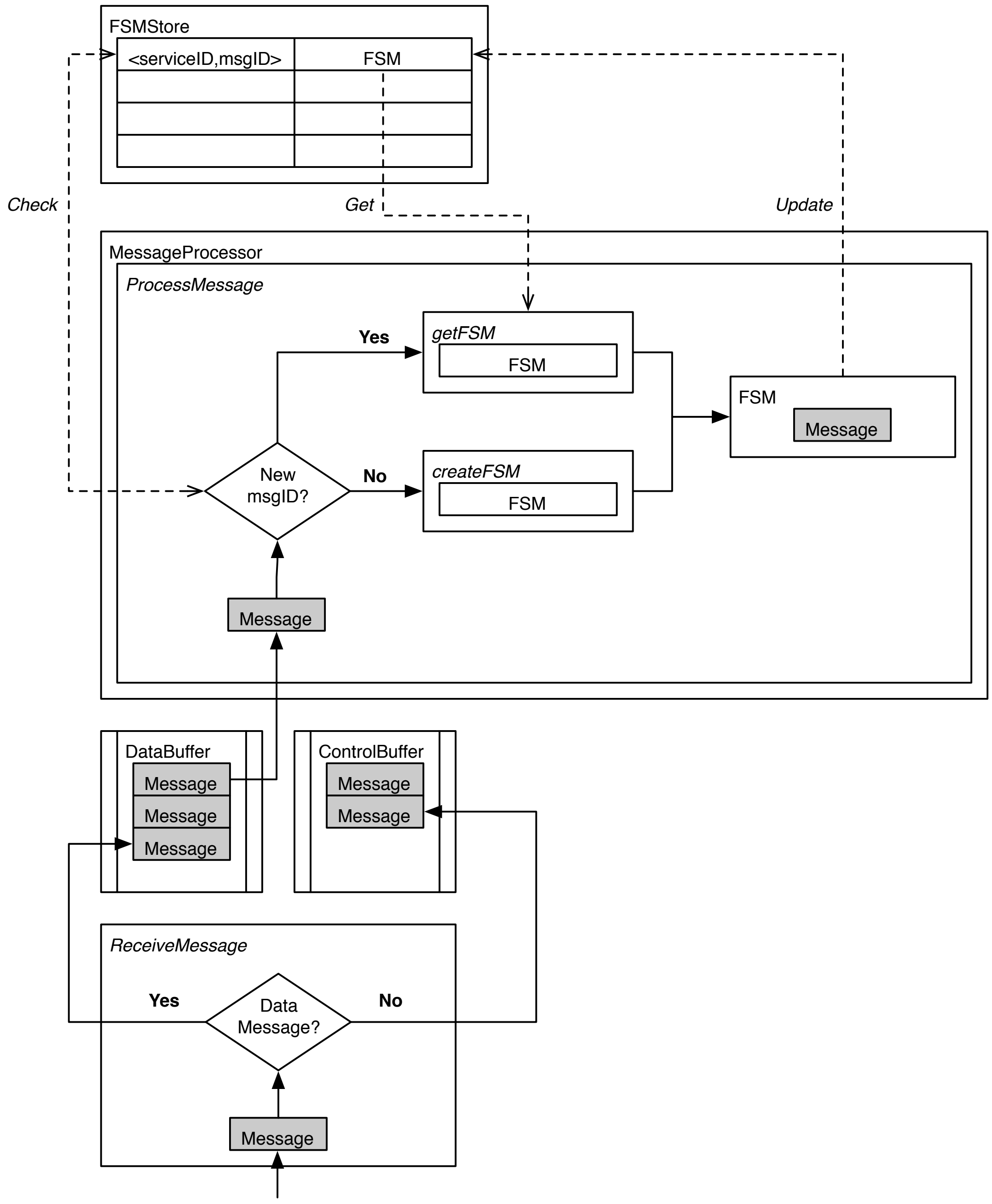


Figure 3.3: How a message is process by its finite-state machine (FSM) in MessageStore

correspondingly.

2. The MessageStore has a MessageProcessor thread that constantly checks if the queues `DataBuffer` and `ControlBuffer` have messages available, and, if so, dequeues them to be processed.

3. The `processMessage` method processes each message. The service identifier of message is extracted and checked against the list of supported services in MessageStore.

   If the service is found, the message identifier `MessageID` of the message is extracted and checked along with the service identifier `ServiceID` (the unique identifier of a finite-state machine instance for a data message consists of a service identifier and a message identifier) against the MessageStore repository of stored finite-state machine instances. If the repository does not contain a finite-state machine instance with this unique identifier, it means the message is new. A new finite-state machine instance is created using the `createFSM` method. If the repository already contains the finite-state machine (FSM) instance, the reference to it is obtained using the `getFSM` method from the MessageStore repository.

   Data messages contain payload data and control messages contain control information (i.e. acknowledgments, no-acknowledgment, etc.). Both message types are recognized as part of one finite-state machine instance for a service if they have the same unique identifier. In the case of data messages, if two messages have the same unique identifier, they are treated as two copies of the same message, even if they have different payloads or different source logical addresses. The unique identifier of a control message associates the control message with a data message.

4. If a message is a data message, the message is stored with the finite-state machine instance if it is new. If the message is a control message, the corresponding finite-state machine instance is updated. Depending on the service, this may involve sending new control messages to other overlay nodes, transmitting the data message, or setting timers.

5. The finite-state machine instance for message is stored (new message) or updated (message already received) in the finite-state machine repository of MessageStore.

For a detailed implementation of the HyperCast MessageStore, please refer to the design document [7].

3.1.2 HyperCast Services

HyperCast provides six services within the MessageStore component. Each of these services is identified by a unique service identifier and implemented as a finite-state machine. The finite-state machines for the services are implemented by a common MessageStore finite-state machine interface as shown in Figure 3.4. In the next sections we describe in detail two of the reliable message delivery services: Hop-to-Hop acknowledgment and End-to-End acknowledgment. The other four services will be described concisely at the end of this section.

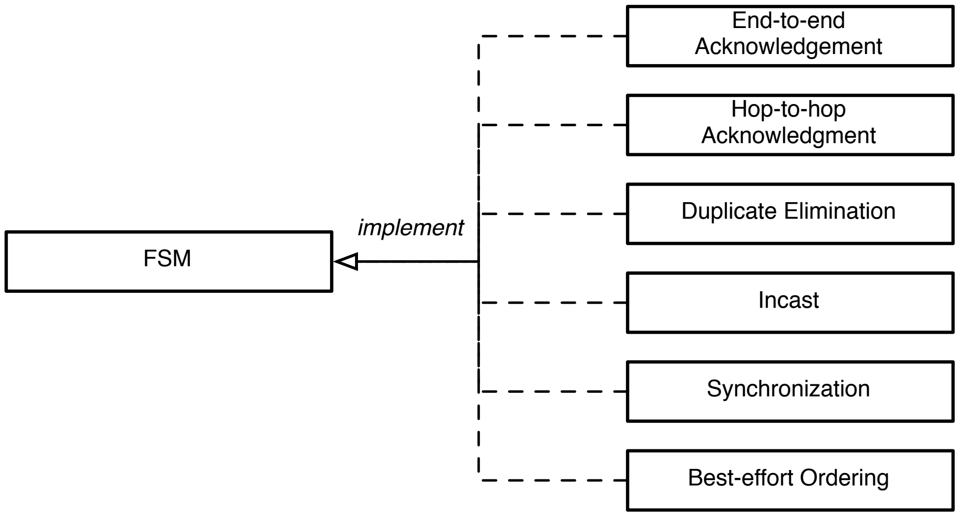


Figure 3.4: Services in HyperCast

3.1.2.1 Hop-to-Hop Acknowledgement

Hop-to-Hop Acknowledgement is a service for reliable transfer of messages across a single-hop in an overlay network topology. Hop-to-Hop acknowledgement enhances single-hop reliability over unreliable communication links (*e.g.*, UDP). In this service, an acknowledgment from a child node indicates to the sender that an immediate child node has received the message successfully. However, the sender does not know if all nodes have successfully received the message.

A sender that has transmitted a data message expects to receive an acknowledgement from children nodes to which the message was sent. When an overlay node receives a message, it immediately sends an acknowledgment, called the H2H ACK, to its parent node. If the node is not a leaf in the tree, it also forwards the message to its immediate children. Figure 3.5 shows an example of this service.

After a time limit, if a sender has not received an acknowledgment from a child, it sends to the child a message to request an acknowledgment (ACK Request) in case the acknowledgment was lost. If a child node does not have the message, then it sends a negative-acknowledgment (NACK) in reply.

A retransmission is performed only when the sending node receives a NACK. The intended receiver of a data message (the child node) is responsible for recovering the data message if it is not received. If the data message cannot be recovered, then the node will give up. Additionally, the sender (the parent node) is responsible for requesting acknowledgment from its child nodes using the ACK Request message. If it still does not receive an acknowledgment after a reasonable amount of time, then it will give up. Thus, each node in the overlay network makes a best-effort to enhance single-hop reliability.

The Hop-to-Hop Acknowledgement service uses five message types which are described in Table 3.1.

Four types of timeouts are used for the Hop-to-Hop Acknowledgment service. The timers are described in Table 3.2.

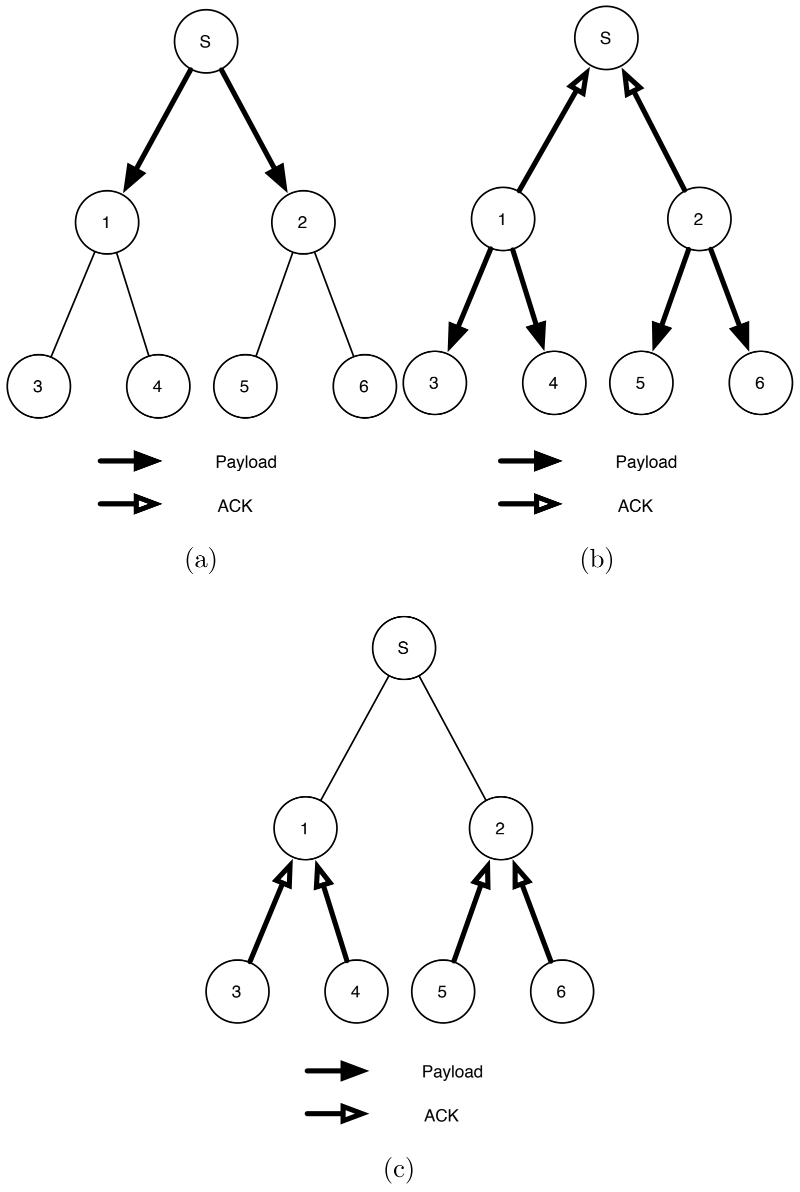


Figure 3.5: (a) The sender (S) delivers a message to its children nodes (node 1 and node 2). (b) When node 1 and node 2 get the message, they send H2H ACKs to parent node S. In the meantime, the data is forwarded to their children node (node 3, 4, 5, 6). (c) When node 3, 4, 5, 6 receive the message, they send H2H ACKs to their respective parents nodes 1 and 2.

In Figure 3.6, the finite-state machine for the Hop-to-Hop Acknowledgment service is shown. States are indicated by circles. The initial state is the grey single circle

Table 3.1: Messages for the Hop-to-Hop Acknowledgment service

Message	Description
Payload	Payload message contains the payload data. The payload message is forwarded along the edges of a spanning tree from a source to the destination, where the source is the root of the spanning tree. Each node that receives a payload message forwards it to its children in the tree as well as to the local application. If a node has the payload message stored in the MessageStore and has already forwarded the payload message, then it will not forward the payload message again. A node that receives a NACK from a child node and that holds the payload message retransmits the payload message to the requesting child node.
H2H ACK	A H2H ACK acknowledges the receipt of a single payload message. A H2H ACK is sent immediately after a payload is received for the first time or immediately after an ACK Request message is received.
ACK Request	ACK Request message is used to request the transmission of a H2H ACK from a child node. The ACK Request message is sent from the parent node to its immediate children.
NACK	A NACK is sent from a child node to its parent in the tree to request for the retransmission of a payload message.
Reset	A node sends a reset message when it gives up or is forced to give up on a particular message. A node that receives a reset message sends the reset message to all of its children.

Table 3.2: Timers for the Hop-to-Hop Acknowledgment service

Timer	Description
Timeout_{ACK}, Timeout_{maxACK}	If a payload message is transmitted for which H2H ACK messages are expected, then the Timeout_{ACK} is set. If not all H2H ACK messages are received when the Timeout_{ACK} timer goes off then an ACK Request message is transmitted to the children which have not yet sent H2H ACK messages. Also, a new Timeout_{ACK} is scheduled. The state machine is reset and Reset messages are transmitted to all children nodes if not all H2H ACK messages are received after Timeout_{maxACK}.
Timeout_{NACK}, $\text{Timeout}_{maxNACK}$	A Timeout_{NACK} is scheduled immediately after a NACK is transmitted. If the message has not arrived after Timeout_{NACK}, then another NACK message is requested and a new Timeout_{NACK} is scheduled. The state machine is reset and Reset messages are transmitted to all children nodes if not all H2H ACK messages are received after $\text{Timeout}_{maxNACK}$ times out.
Timeout_{RST} and Timeout_{maxRST}	A Timeout_{RST} is scheduled when the state machine is reset and Reset messages are transmitted to all children after Timeout_{RST}. All state information is deleted after Timeout_{maxRST}.
Timeout_{Msg}	Once all required operations for a payload message have been performed, the service enters the "Done" state where the Timeout_{Msg} is set. All of the state information is deleted after Timeout_{Msg}.

labelled “Init”. The grey double circle labelled “Done” is the final state for normal service operation. Arcs represent state transitions; each arc is labelled with an input event (E) which triggers the transition and the actions (A) to be performed as result.

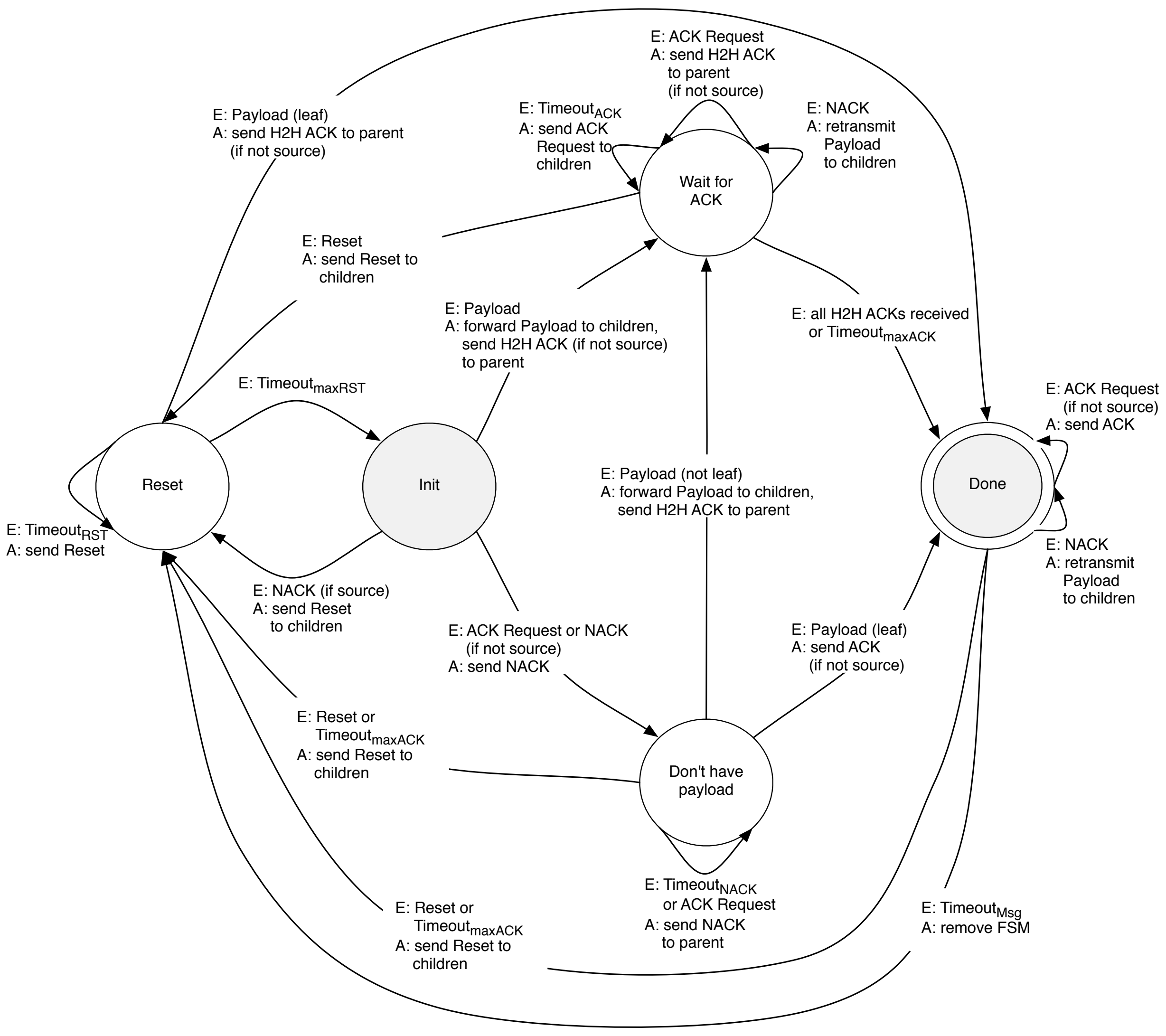

E: ACK Request
A: send H2H ACK to parent (if not source)
E: Timeout$_{ACK}$
A: send ACK Request to children
E: NACK
A: retransmit Payload to children
Wait for ACK
E: Payload (leaf)
A: send H2H ACK to parent (if not source)
E: Reset
A: send Reset to children
E: Payload
A: forward Payload to children, send H2H ACK (if not source) to parent
E: all H2H ACKs received or Timeout$_{maxACK}$
E: Timeout$_{maxRST}$
E: ACK Request (if not source)
A: send ACK
Reset
Init
E: Payload (not leaf)
A: forward Payload to children, send H2H ACK to parent
Done
E: Timeout$_{RST}$
A: send Reset
E: NACK
A: retransmit Payload to children
E: NACK (if source)
A: send Reset to children
E: ACK Request or NACK (if not source)
A: send NACK
E: Payload (leaf)
A: send ACK (if not source)
E: Reset or Timeout$_{maxACK}$
A: send Reset to children
Don't have payload
E: Timeout$_{NACK}$ or ACK Request
A: send NACK to parent
E: Reset or Timeout$_{maxACK}$
A: send Reset to children
E: Timeout$_{Msg}$
A: remove FSM

Figure 3.6: State machine diagram for Hop-to-Hop Acknowledgement service

3.1.2.2 End-to-End Acknowledgement

End-to-End Acknowledgement is a service for reliable end-to-end transfer of messages to all nodes in an overlay network. In this service, an acknowledgment (Full ACK) indicates that all nodes that are the sender's descendants have received the message.

Message transmission in this service proceeds as follows. A node that is a leaf in the tree sends an full acknowledgment (Full ACK) to its parent node upon receipt of a message. A node that is not a leaf in the tree will transmit a Full ACK to its parent node once it has received Full ACKs from all of its children. When the source of a payload message receives a Full ACK, then it knows all of the nodes that are its descendants in the tree have received the message. Figure 3.7 shows an example of this service in a tree topology.

If a non-leaf node received some acknowledgments but not all acknowledgments from its children nodes, then it will send a partial acknowledgment (Partial ACK) to its parent. This ensures that some kind of acknowledgement is transmitted upstream, even though not all nodes have acknowledged the receipt of the message. The transmission of Partial ACKs is enforced by timers.

After a time limit, if a sender has not received an acknowledgment from its child, a sender sends to a child node a message to request an acknowledgment (ACK Request) to query if the child has received the message. If a child node does not have the message, then it sends a negative-acknowlegdment (NACK) in reply. If the sender requires that all receivers must receive the message, a retransmission is performed when the sender receives NACKs.

There are a total of six message types that are used by the End-to-End Acknowledgement service. Payload, ACK Request, NACK and Reset messages have the same meaning as described in Hop-to-Hop acknowledgement service (Section 3.1.2.1). Two types of additional messages are listed (Table 3.3):

Timeouts for the End-to-End acknowledgement service include all the timeout types

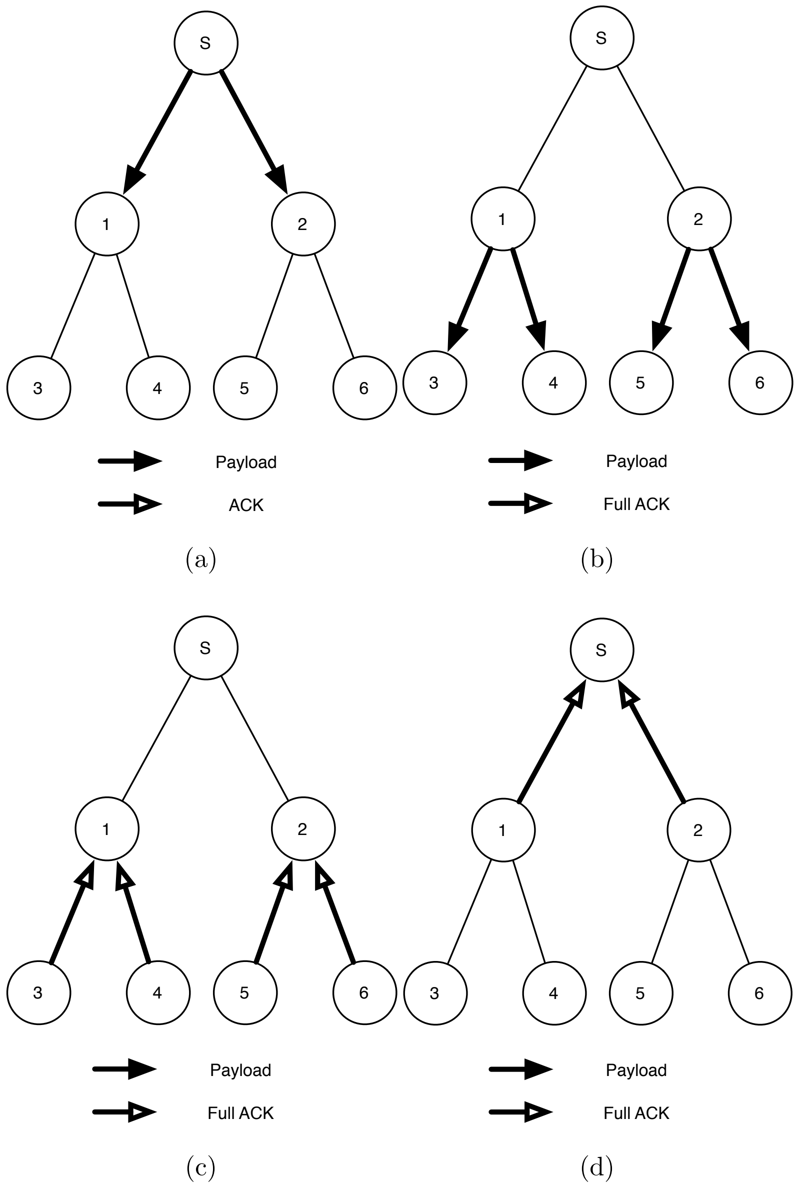


Figure 3.7: (a) The sender (S) delivers a message to its children node (node 1 and node 2), which forward the message to their children nodes (nodes 3, 4, 5, 6). (b) When leaf nodes 3, 4, 5, and 6 receive the message, they send a Full ACK to their parent nodes. (c) The ACKs are merged at node 1 and 2. (d) Node 1 and 2 will send Full ACKs to their parent node, the sender (S).

of Hop-to-Hop Acknowledgement service described in Section 3.1.2.1. Here the acknowledgment timers represent the full acknowledgment timers. One additional timeout is

Table 3.3: Messages for the End-to-End Acknowledgment service

Message	Description
Full ACK	A Full ACK is sent by a leaf node to its parent node after it receives a payload message. A non-leaf node sends a Full ACK to its parent node after it receives Full ACKs from all of its child nodes.
Partial ACK	A Partial ACK acknowledges a the node downstream is waiting for Full ACKs from its children. Non-leaf nodes send Partial ACKs periodically to their parent nodes.

described in Table 3.4.

Table 3.4: Timers for the End-to-End Acknowledgment service

Timer	Description
Timeout$_{PACK}$	Timeout$_{PACK}$ is scheduled to send Partial ACK periodically by non-leaf nodes in a tree to their respective parent nodes.

In Figure 3.8, the finite-state machine for the End-to-eop acknowledgment service is shown. States are indicated by circles. The initial state is labelled as “Init”. The final state is labelled as “Done”. Arcs represent state transitions; each arc is labelled with an input event (E) which triggers the transition and the actions (A) to be performed as result.

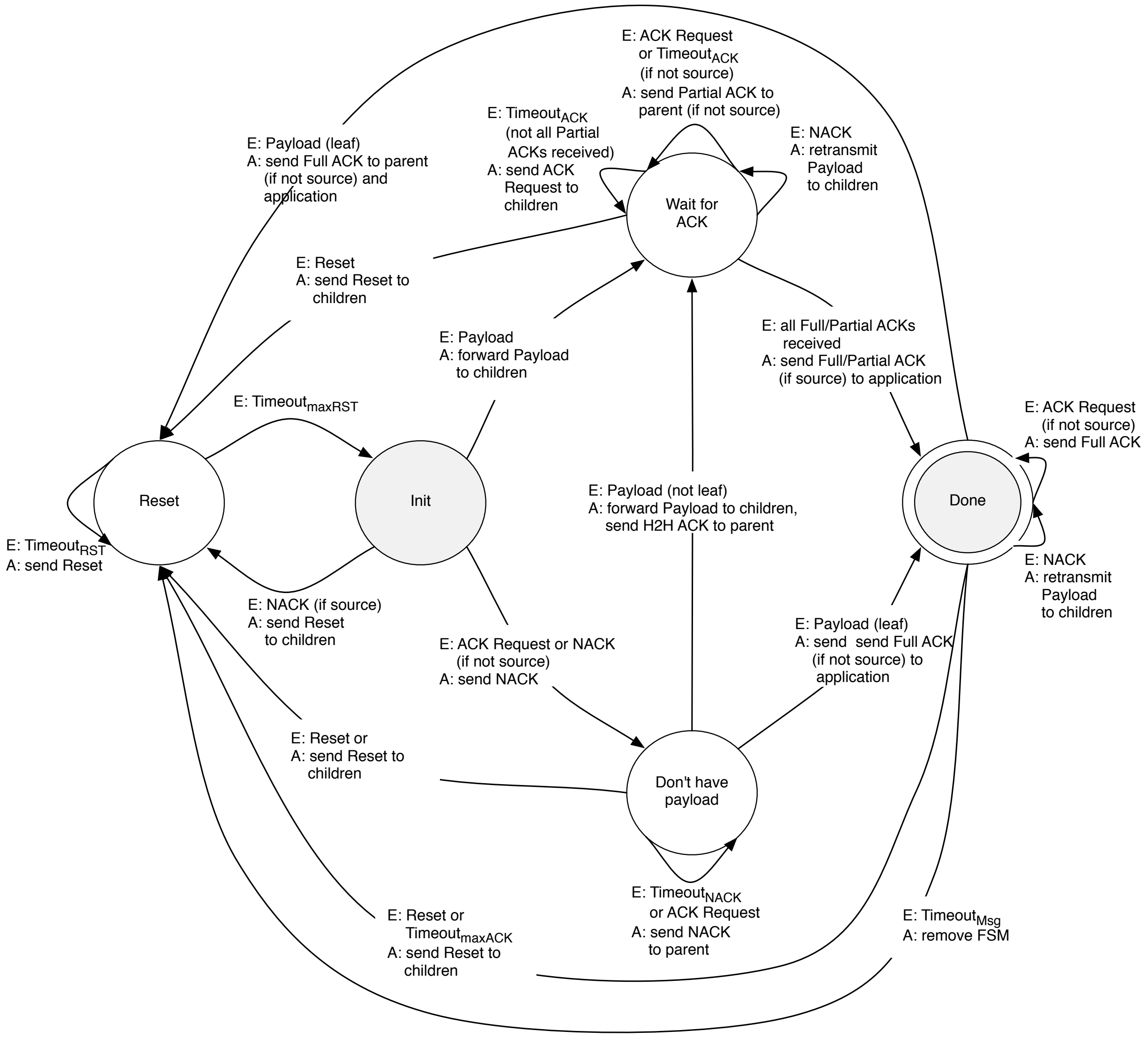

Reset
Init
Wait for ACK
Done
Don't have payload
E: Payload (leaf)
A: send Full ACK to parent (if not source) and application
E: ACK Request or Timeout$_{ACK}$ (if not source)
A: send Partial ACK to parent (if not source)
E: Timeout$_{ACK}$ (not all Partial ACKs received)
A: send ACK Request to children
E: NACK
A: retransmit Payload to children
E: Reset
A: send Reset to children
E: Payload
A: forward Payload to children
E: all Full/Partial ACKs received
A: send Full/Partial ACK (if source) to application
E: ACK Request (if not source)
A: send Full ACK
E: Timeout$_{maxRST}$
E: Payload (not leaf)
A: forward Payload to children, send H2H ACK to parent
E: Timeout$_{RST}$
A: send Reset
E: NACK
A: retransmit Payload to children
E: NACK (if source)
A: send Reset to children
E: ACK Request or NACK (if not source)
A: send NACK
E: Payload (leaf)
A: send send Full ACK (if not source) to application
E: Reset or
A: send Reset to children
E: Timeout$_{NACK}$ or ACK Request
A: send NACK to parent
E: Reset or Timeout$_{maxACK}$
A: send Reset to children
E: Timeout$_{Msg}$
A: remove FSM

Figure 3.8: State machine diagram for End-to-End Acknowledgement service

3.1.2.3 Other Services

HyperCast also provides several other services. They are described in Table 3.5. For a comprehensive document on all HyperCast service, we refer to [7].

Table 3.5: Other HyperCast Services

Network Service	Description
Duplicate Elimination	This service discards a message if it is a duplicate of an earlier received message.
Synchronization	Each overlay node stores each transmitted and received message and periodically synchronizes the stored message with its neighbors in the overlay network.
InCast	This service merges the payload of unicast messages with identical destination addresses and message identifiers.
Best Effort Ordering	In this service, received messages are passed to the application program in the order of sequence numbers. However, there can be "holes" in the middle of the sequence due to the loss of messages.

3.2 SCXML: State Chart XML

State Chart XML(SCXML), State Machine Notation for Control Abstraction is a markup language to express generic state machines in XML [17]. SCXML provides the XML syntax for specifying any deterministic FSM using XML. SCXML is an ongoing W3C standard. The last working draft of the specification was released by the W3C in February 2012 [16].

Our implementation uses SCXML to specify the finite-state machines for the services as SCXML documents. The details of SCXML documents can be found at the W3C working draft [16]. These SCXML document can then be executed by an SCXML based execution environment such as Apache Commons SCXML [1].

Here we provide a brief overview of the SCXML components. In Chapter 5 we will discuss in detail the relevant portions that our implementation uses.

3.2.1 Apache Commons SCXML

Apache Commons SCXML is a Java library that provides an execution framework for state machines specified in SCXML. Apache Commons SCXML provides classes to parse the SCXML specification of a finite-state machine to create a finite-state machine instance. The finite-state machine instance contains the software realization of the SCXML document. Since HyperCast MessageStore services are finite-state machines, they can be specified in SCXML.

The SCXML parser creates two additional interacting components from the SCXML Document. The first is the SCXML Datamodel, which contains application defined data elements. The data elements are in the form of XML DOM (Data Object Model) trees [15]. The SCXML Datamodel can be referenced in SCXML and other software components can also interact with the data through the Apache Commons SCXML libraries.

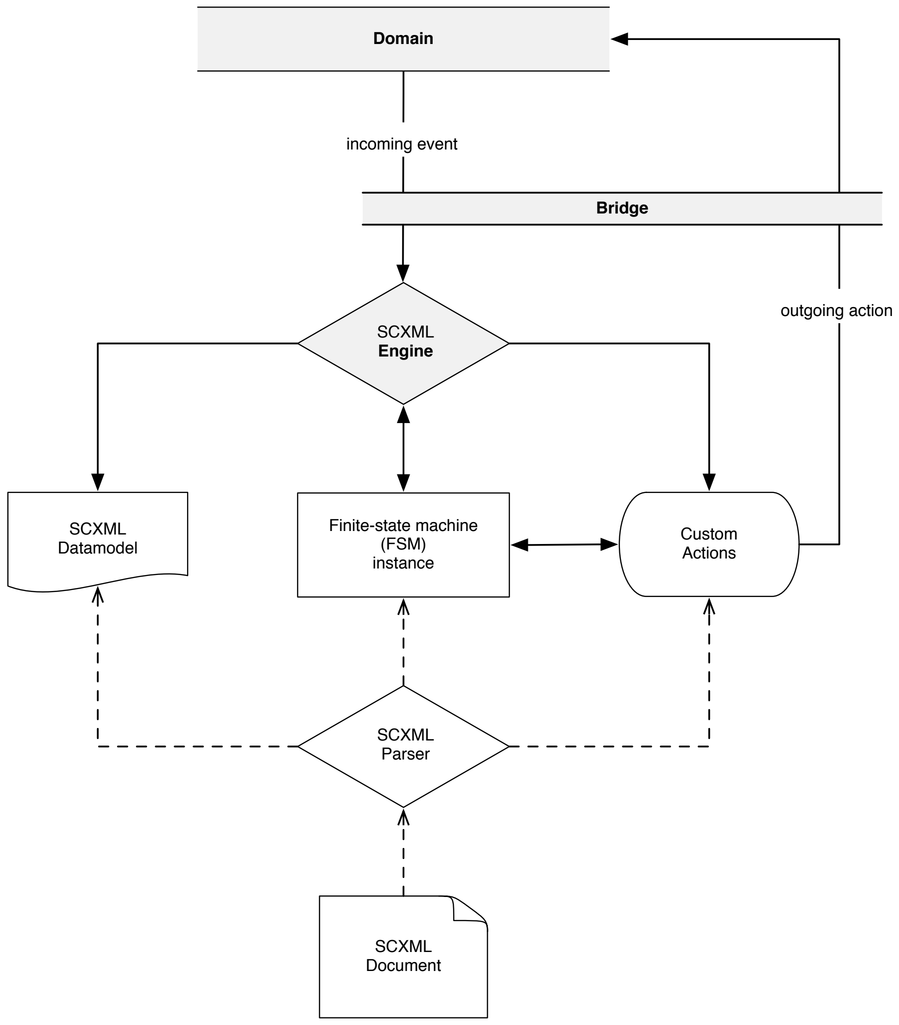


Figure 3.9: Interacting components of the Apache Commons SCXML framework

The Custom Actions interface allows applications to define custom SCXML tags that specify executable content within SCXML documents. Processing of the content will be delegated to components outside of the Apache Commons SCXML. The Custom Actions interface allow applications to define how and where the processing of these executable content occurs.

The other interacting components of this framework, as defined by Apache Commons SCXML, are shown in Figure 3.9. Below we describe these components.

3.2.1.1 SCXML Engine

The SCXML engine is a generic event-driven state machine based execution environment. The finite-state machine instance can is executed by the SCXML engine. The SCXML engine does not store any information related to a particular finite-state machine instances. Therefore, it is possible for one SCXML engine to execute multiple finite-state machine instances. The SCXML engine implementation is single-threaded.

3.2.1.2 Domain

The Domain is the application that communicates with the SCXML engine. The domain is a software program that sends events to the SCXML engine and processes executable content from the Custom Actions interface. In our implementation, the Domain is a network application running a HyperCast overlay socket.

3.2.1.3 Bridge

The Bridge is the two-way communication link that 1) processes the events from the Domain and triggers those events in the SCXML engine updating the finite-state machine instance, and 2) processes the executable content from the Custom Actions interface and trigger them in the Domain.

3.3 XQuery

XQuery is a language for querying data represented in XML. Given an XML document represented as a DOM tree, XQuery uses XPath expression syntax to query specific parts of an XML document.

An XPath expression specifies a path. For example, given a XML document called `nodes.xml`:

```
<NodeA>
    <NodeB>
        <NodeC>100</NodeC>
    </NodeB>
</NodeA>
```

A XPath expression that queries the value of `<NodeC>` is: `/NodeA/NodeB/NodeC`. XPath syntax can be a complex expressions that specifies the direction to navigate from a node, and filters on a node's name and value. For a complete reference on XPath we refer to [19].

XQuery supplements the path expression provided by XPath by adding conditional expressions and comparisons. An example XQuery expression using conditional expression and comparisons that returns true if the value of `<NodeC>` is equal to `100` is:

```
for $x in doc("nodes.xml")/NodeA/NodeB/NodeC
return if $x=100
then <out>true</out>
else <out>false</out>
```

XQuery is a powerful query language that have many additional features. We have only showed a simple example of XQuery syntax. For a complete reference on XQuery we refer to [20].

3.3.1 Saxon XQuery and XSLT Processor

The Saxon XQuery and XSLT Processor [14] is a Java-based XQuery processor that is XML schema aware. An XML schema is a description of a XML document, expressed in terms of constraints on the structure and content of documents. The schema is used to validate the correctness of XML documents. For a complete reference on XML schema, we refer to [18].

Chapter 4

Design of Customizable Services

In this chapter, we describe a software architecture that enables custom data delivery services for application-layer overlay networks. These services are defined by applications and dynamically deployed to an application-layer overlay middleware system. We view a service as a distributed software application, on a non-empty set of cooperating overlay nodes $N_0, N_1, ..., N_n$, that receives inputs, from which the service produces outputs as shown in Figure 4.1.

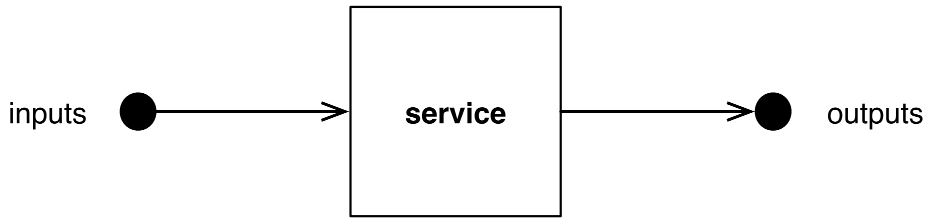


Figure 4.1: Abstraction of a service

Some services can sufficiently determine their outputs solely based on current inputs while other services need additional information about the changes in the inputs over time to determine its outputs. The concept of *states* is used to represent information about a service's history. In reference to automata theory, a state is a unique stage of processing in a service. A service can have one or more states. A service determines its next state based on its current state and on its current inputs. Hence, the current state contains the information on how the service reached the present situation.

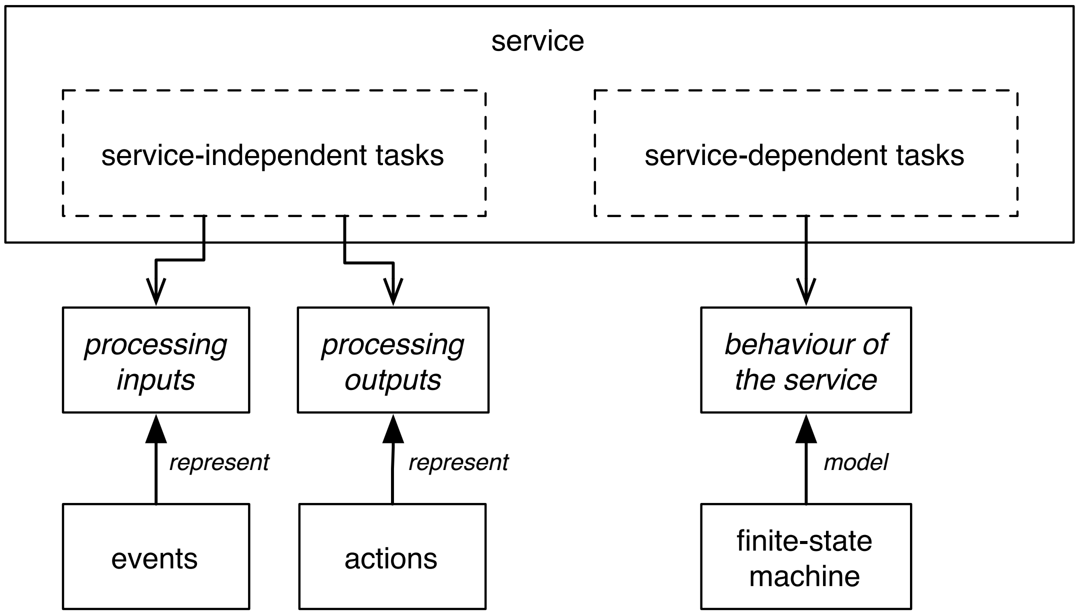


Figure 4.2: A service decomposed

We model a service with two components as illustrated by Figure 4.2. The first component involves processing of *service-independent* tasks. Different services share common tasks, *e.g.*, sending messages, setting timers, etc. While different overlay networks networks may specify different message formats, timer formats, and method signatures for common tasks, the semantics involved in processing these data structures and functions are common to all services. We decompose these common tasks into two sets: 1) the processing of inputs to the service, and 2) the processing of outputs produced by the service during execution. The inputs to be processed (*e.g.*, the arrival of messages, expiration of timers, etc.) are called *events* and the outputs produced (*e.g.*, overlay network function invocations such as sending messages, querying local node statistics, etc.) are called *actions*.

The second component are *service-dependent* tasks, *i.e.*, tasks that are unique to each service. For example, when and what outputs are produced when a service process certain inputs. In services that behave differently, this component characterizes the behaviour of each service. The behaviour logic is independent of the different formats for inputs and outputs. We model the behaviour of a service by a *deterministic finite-state machine*. In the next section, we formally define deterministic finite-state machines.

4.1 Finite-State Machine

A deterministic finite-state machine can be presented as a directed graph (Figure 4.3) with the following elements (E, S, δ, F, A), where:

- $E = (E_1, E_2, ..., E_n)$ is a finite, non-empty set of input symbols. These represents *events.*
- $A = (A_1, A_2, ..., A_n)$ is a finite, non-empty set of output symbols. These represents *actions.*
- $S = (S_0, S_1, S_2, ..., S_n)$ is a finite, non-empty set of states. In Figure 4.3, states are represented by circles with unique designator symbols, $S_0, S_1, or S_2$, written inside them.
- $S_0 \in S$ is the initial state. In Figure 4.3, the initial state is S_0.
- $F \subset S$ is the set of final states. In Figure 4.3, the final state is S_1, is drawn with a double circle.
- δ is the state-transition function: $(\delta : S \times E \rightarrow S \times A)$. In Figure 4.3, transitions are represented as edges between states. An edge is drawn as an arrow directed from the present-state to the next-state. The mapping E_i/A_i on an edge describes that the state transition occurs on an input symbol E_i and produces an output symbol A_i. For example, state S_0 transitions to S_1 on the input symbol E_1 and produces the output symbol A_1.

A finite-state machine can be *deterministic* and *non-deterministic.* In deterministic finite-state machines, every state has exactly one transition for each possible input $(\delta : S \times E \rightarrow S)$.

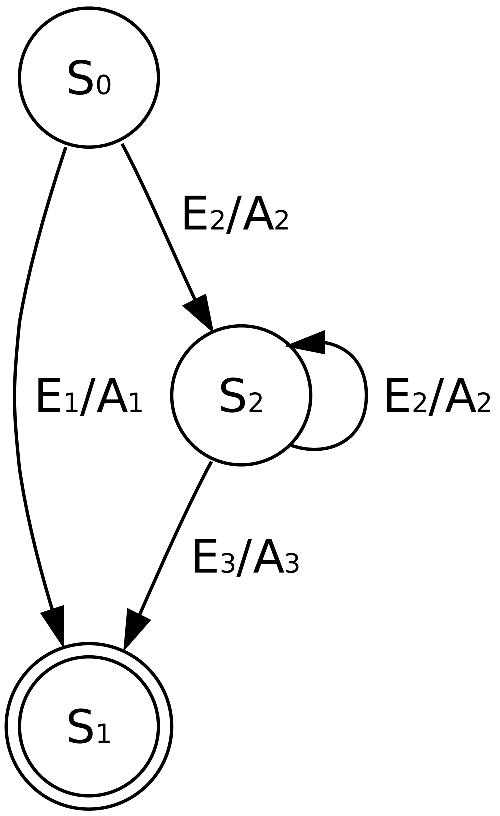
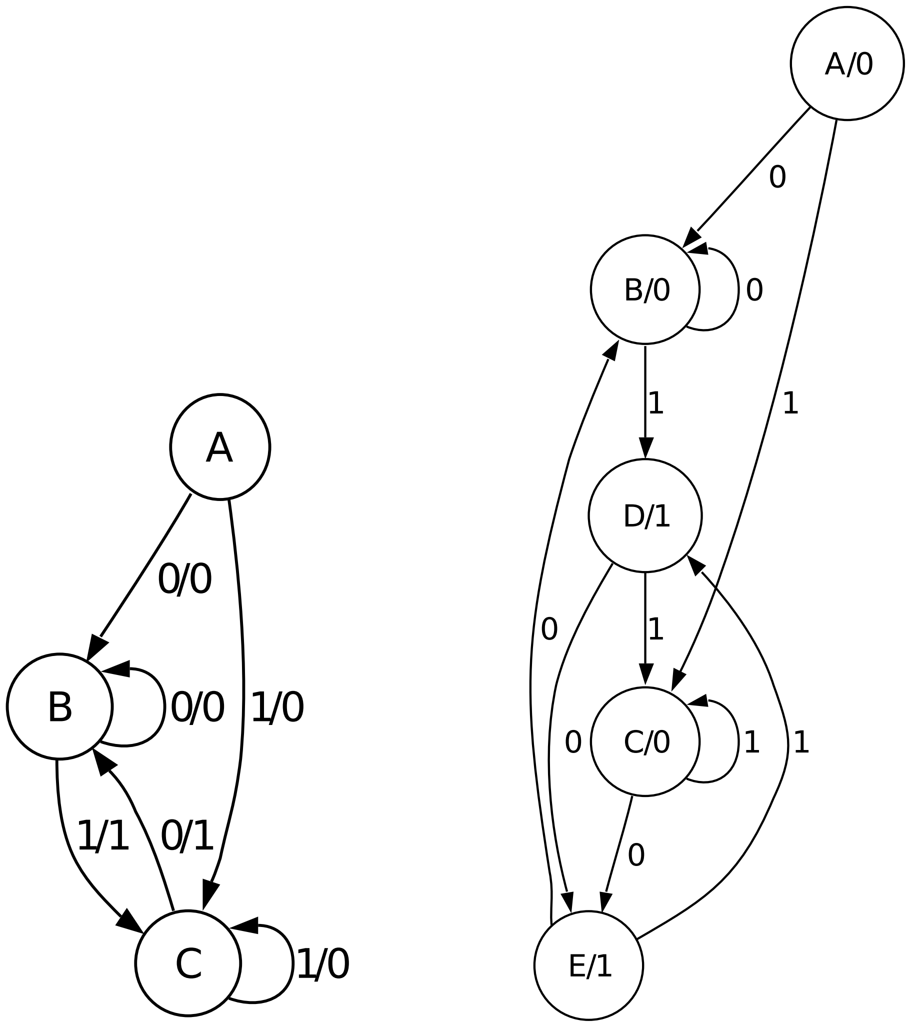


Figure 4.3: A finite-state machine diagram

The finite-state machine definition given above describes a *Mealy machine*, finite-state machines whose output symbols are determined by both their current states and by the value of their inputs ($\delta : S \times E \rightarrow S \times A$). This is in contrast to a *Moore machine*, whose output symbols are determines solely by the states ($\delta : S \rightarrow S \times A$). Moore machines are typically used in sequential digital logic hardware systems where states change only when the global clock changes, resulting in constant state transition rates. Mealy machines are widely used in software systems where the input symbols E are *asynchronous events* that occur at irregular intervals. An asynchronous event is initiated outside the scope of the finite-state machine software that is to be handled by the finite-state machine. For example, sensor outputs, user actions (mouse clicks, key presses), messages, and hardware and software timers. A state machine that uses asynchronous events as inputs is called *event-driven*.



(a) Mealy machine (b) Moore machine

Figure 4.4: A Mealy machine and a Moore machine with the same behaviour

In theory, any Mealy machine can be converted to a Moore machine. However, Mealy machines have two advantages over Moore machines. First, Mealy machines tend to have fewer states and state transitions due to the ability to have outputs on transitions rather than states. For example, in Figure 4.4, the Mealy machine in Figure 4.4(a) has 2 fewer states and 4 less state transitions compare to the Moore machine in Figure 4.4(b) with the same behaviour. Secondly, Mealy machine react faster to inputs since they do not have to wait for a clock. However, Mealy machines must take into consideration asynchronous feedback as input changes immediately cause output changes. This is typically not an issue for services as our inputs, described in Section 4.2.1, operate on a network time scale (subject to network delays) which is more than several order of magnitudes slower compared to processor clocks.

We limit the type of the finite-state machine in our design to be a deterministic, event-driven Mealy finite-state machine. This is because for overlay services, the service state on an overlay node changes when it receives information from, or transmits information to other overlay nodes. Since overlay nodes are typically independent end-hosts with different resource capabilities at different geographical locations, there may not be a notion of globally synchronized time when they communicate with each other. Hence, service inputs are asynchronous events coming from other systems.

4.2 Executable Specification

Event-driven deterministic finite-state machines can be used as a high-level design guideline for developing software. Such finite-state machines can be informal and translated into executable code. Our objective is to specify a service as a finite-state machine which can be executed directly, *i.e.*, an *executable specification*. An executable specification contains information on (1) accepted inputs to the service (*e.g.*, message arrivals, timer expirations, etc.), (2) outputs of the service that are invocations to tasks (*e.g.*, send data messages, send control messages, setting timers, etc.), and (3) the finite-state machine representing the behaviour of the service.

Executable specifications are created by network application developers. Application-layer overlay middleware running on overlay network nodes can interpret these executable specifications and execute them.

Our first task is to express the finite-state machines of services in a computer-usable form. We can use a graph description language to describe a finite-state machine in a machine readable form. Many graph description languages exists for finite-state machines, such as: DOT Language [3], Graph Modeling Language (GML) [10], GraphML [2], GXL [11], Directed Graph Markup Language [5], State-chart XML (SCXML) [16], etc.

In order to express our services as an executable specification we first define a template for executable specifications. First we define valid inputs that our services will accept. Next we define how these inputs are mapped to events E for the finite-state machine. Then we define a list of network tasks that the finite-state machines can invoke using output symbols A. lastly, we define how these network tasks are invoked by the service using output symbols A.

Our objective is to provide a design for the executable specification that can be used to define a wide range of useful services. Finite-state machines for services should be concise with a small number of input symbol set (E), output symbol set (A), states S, and state transitions (δ). The goal is to express the executable specification of a service by a small markup file which can be quickly parsed and validated.

In the following sections, we describe in detail our design of the executable specification.

4.2.1 Inputs

We define the inputs of a finite-state machine for a service to be asynchronous events. We define two categories of events: (1) *basic* events and (2) *composite* events.

4.2.1.1 Basic Events

Basic events are events that cannot be further decomposed with respect to being inputs to a service. They are identified by the symbol E_{basic} with the name of the events as subscript in the finite-state machine. For example, $E_{basic_{ACK}}$ is represents an acknowledgement message event. We restrict basic events to two types: (1) arrival of a message, and (2) expiation of a timer.

In an application-layer overlay network, overlay nodes communicate with each other with messages in order to establish connections, perform synchronization, transfer and request data, etc. There are two groups of messages: (1) data messages, and (2) control messages. Data messages are messages containing user payload and control messages contain control information between overlay nodes.

Services have the ability to set a timer, that will expire after a certain amount of time. Timers can be used to set the maximum waiting time until the occurrence of a specific event (*e.g.*, wait 10s for the arrival of a message). In this scenario, timers ensure that the finite-state machine does not wait indefinitely for an event. Timers can also be used to set the maximum waiting time for a task to complete (*e.g.*, wait 10s after performing a periodic synchronization of global clocks before reading the clock). In this scenario, timers are used to specify a temporal order.

For example, if a finite-state machine in state S_1 wants to create a sequence of output symbols representing actions that are order sensitive $A : (A_1, A_2, ..., A_n)$ based on a single input E_1, the transition $(\delta : S_1 \times E_1 \rightarrow S_2 \times A_1, A_2, ..., A_n)$ cannot guarantee the order that any A_i will finish before $A_{i+1}, ..., A_n$ because the execution of these actions A are are asynchronous to the exception of the finite-state machine.

What we want is a sequence of transitions $(\delta_1 : S_1 \times E_1 \rightarrow S_2 \times A_1)$, $(\delta_2 : S_2 \times$ "A_1" $\rightarrow S_3 \times A_2)$, ..., $(\delta_{n-1} : S_{n-1} \times$ "A_{n-1}" $\rightarrow S_n \times A_n)$. We put some symbols A_i in quotations because these are actually actions and not events. Hence, we add a new action called "create timer" that sets a waiting period for an action to finish and when the timer expires the event $E_{Time-out}$ is inputted to the finite-state machine, such that $(\delta_1 : S_1 \times E_1 \rightarrow S_2 \times A_1, create\ timer)$, $(\delta_2 : S_2 \times E_{Time-out} \rightarrow S_3 \times A_2, create\ timer)$, ..., $(\delta_{n-1} : S_{n-1} \times E_{Time-out} \rightarrow S_n \times A_n)$ as illustrated in Figure 4.5. This preserves the ordering of actions.

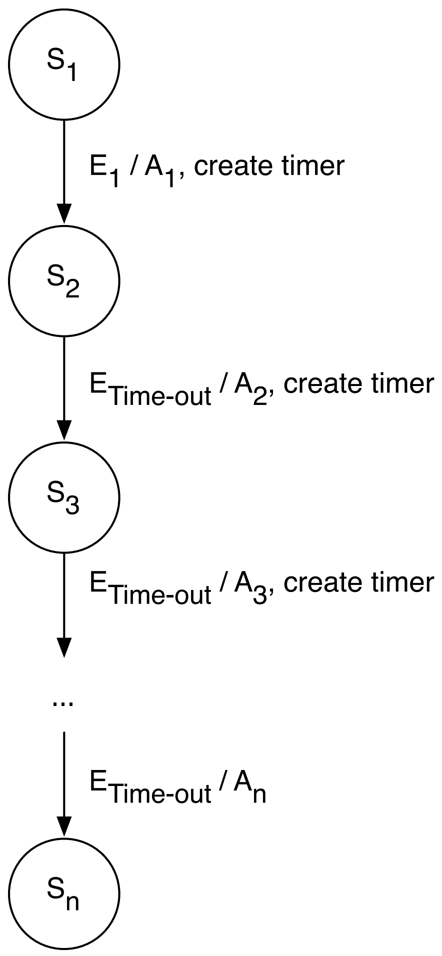

S_1
E_1 / A_1, create timer
S_2
$E_{Time\text{-}out}$ / A_2, create timer
S_3
$E_{Time\text{-}out}$ / A_3, create timer
...
$E_{Time\text{-}out}$ / A_n
S_n

Figure 4.5: Using timers to specify a given sequence of actions

4.2.1.2 Composite Events

Basic events specify individual message arrivals or individual timer expirations. However, some input conditions are based on several basic events, *i.e.*, a combination message of arrivals and timer expirations. For example, consider an overlay node thats sends messages to multiple neighbour nodes and waits for acknowledgement messages from each neighbour. After the message is sent, depending on the acknowledgement messages received from the neighbour nodes within a specific time period, the following outcomes are possible:

1. Acknowledgement messages from all neighbour nodes are received within a specified time interval.
2. Acknowledgement messages from some but not all neighbour nodes are received within a specified time interval.
3. No acknowledgement message is received from any neighbour nodes within a specified time interval.

Each outcomes results in different actions taken by the service. Evidently, these input conditions are based on a set of multiple message arrival events and one timer expiration event.

We define input conditions based on a set of basic events as *composite events.* Composite events are identified by the symbol $E_{composite}$ with the name of the event as the subscript in the finite-state machine. When given a collection of basic events, composite events describe (1) constraints on a set of basic events, (2) a combination of sets of basic events, and (3) quantification on a set basic events.

From an automata theory point of view, any composite events can be expressed with basic events and states/state transitions. For example, if we assume above that we only have three neighbours A, B, and C, and $E_{basic_{ACK_i}}$ is the basic event that an acknowledgement message arrived from neighbour i. The scenario that all messages arrived from neighbours A, B, and C is the set of all permutation of the set ($E_{basic_{ACK_A}}$, $E_{basic_{ACK_B}}$, $E_{basic_{ACK_C}}$). However, listing permutations of basic events would cause an explosion in states and state transitions for the finite-state machine. Also, using neighbour identifiers in states, the state machine must change if the neighbour set changes during runtime.

We use *first-order logic* to express composite events from a set of basic events. First-order logic is a formal reasoning system that defines a set of rules over a set of individual elements. First-order logic defines *predicates* $P(x)$, that denote a statement P concerning an element x, over a range of individual elements. A predicate can take the role as either a property on a element $P(x)$ or a relation between elements $P(x, y)$. The set defined by $\{x|P(x)\}$ or $\{x|P(x, y)\}$, is a set of elements for which P is true. For example, if an overlay node sends a message to all its neighbour nodes with identifiers A, B, C then $\{x|x \text{ is a neighbour}\}$ is the set $\{A, B, C\}$.

Using the acknowledgment message example mentioned previously, we can transform the natural language description of the three input conditions described above as first-order logic statements. The relationship that an acknowledgement message $E_{basic_{ACK}}$ has arrived from a specific neighbour d is defined by the expression $from(d, E_{basic_{ACK}})$. The relationship that a timer expired $E_{basic_{timeout}}$ for a specific neighbour d is defined by the expression for $timeout(d, E_{basic_{timeout}})$.

The statements at the beginning of this section are expressed in first-order logic as follows:

1. "For each neighbour d, there exists an acknowledgement message $E_{basic_{ACK}}$, such that $E_{basic_{ACK}}$ comes from d and there does not exist a timeout $E_{basic_{timeout}}$ for d indicating the maximum waiting time for the acknowledgment has expired for d."
   $\forall d : (\exists E_{basic_{ACK}} : from(d, E_{basic_{ACK}}) \cap \nexists E_{basic_{timeout}} : timeout(d, E_{basic_{timeout}}))$.

2. "For at least one neighbour d but not all neighbour d, there exists an acknowledgement message $E_{basic_{ACK}}$, such that $E_{basic_{ACK}}$ comes from d and there does not exist

a timeout $E_{basic_{timeout}}$ for d indicating the maximum waiting time for the acknowledgment has expired for d."

$\exists d : (\exists E_{basic_{ACK}} : from(d, E_{basic_{ACK}}) \cap \nexists E_{basic_{timeout}} : timeout(d, E_{basic_{timeout}})) \cap$
$\not\forall d : (\exists E_{basic_{ACK}} : from(d, E_{basic_{ACK}}) \cap \nexists E_{basic_{timeout}} : timeout(d, E_{basic_{timeout}}))$.

3. "There does not exist neighbour d, for which there exists an acknowledgement message $E_{basic_{ACK}}$, such that $E_{basic_{ACK}}$ comes from d and there does not exist a timeout $E_{basic_{timeout}}$ for d indicating the maximum waiting time for the acknowledgment has expired for d."
$\nexists d : (\exists E_{basic_{ACK}} : from(d, E_{basic_{ACK}}) \cap \nexists E_{basic_{timeout}} : timeout(d, E_{basic_{ACK}}))$.

Using these first-order logic expressions, we can concisely express these three complex composite events to generate input symbols $E_{composite_{all\ acks\ received}}$, $E_{composite_{some\ acks\ received}}$, and $E_{composite_{no\ acks\ received}}$ respectively.

4.2.1.3 Mapping of Events to Input Symbols

In this section, we describe our mechanism to translate generic message arrivals and timer expirations from the overlay node to specific events identified by unique input symbols that the finite-state machine can understand.

Basic events (messages or timers) have attributes such as message identifier, source address, destination address, message type, sequence number, timer identifier, etc. The executable specification defines a set of filters on these attributes that discriminate between different categories of messages and timers that the service accepts. Each category of messages and timer maps to their associated input symbol(s) in the finite-state machine. The mapping of basic events to input symbols can be one-to-one, one-to-many, or many-to-one. The filters are applied in the order they are define, hence, more discriminate filters need to appear ahead of less discriminant filters in the definitions.

Table 4.1: Example of one-to-one mapping between basic events and input symbols

#	Filter	Basic Event	Input Symbol
1	message type=ACK	Acknowledgement (ACK) message	$E_{basic_{ACK}}$
2	message type=Payload	Payload message	$E_{basic_{Payload}}$
3	message type=NACK	Negative-Acknowledgement (NACK) message	$E_{basic_{NACK}}$

Table 4.1 provides an example of one-to-one mapping of basic events to input symbols. Assuming a message has arrived, the executable specification applies the filters on the message attributes to characterize the message as an acknowledgement (ACK) message, a payload message, or a negative-acknowledgement (NACK) message each represented by a unique input symbol representing the associated basic event. Note that in this case we only applied filters on the message type, but filter can be applied to one or more

message attributes. For example, a filter of "message type=ACK, source address=1" will characterize messages that are acknowledgement messages coming from source address 1.

Table 4.2: Example of many-to-one mapping between basic events and input symbols

#	Filter	Basic Event	Input Symbol
1	-	Any message	$E_{basic_{message}}$

In certain scenarios it is desired to have many basic events that map to a single input symbol. For example, different messages types can map to a single $E_{basic_{message}}$ input symbol for a service that wishes to treat them indiscriminately as shown in Table 4.2. In this case, the filter does not check any message attributes.

Table 4.3: Example of one-to-many mapping between basic events and input symbols

#	Filter	Basic Event	Input Symbol
1	message type=ACK, source address = local node	Acknowledgment (ACK) message that is not expected	$E_{basic_{ACK}}$ $E_{basic_{error}}$

Conversely, a single basic event can map to more than one input symbol. Suppose an acknowledgement arrives ($E_{basic_{ACK}}$), and the source address is the local node, it indicates that an error occurred as the local node should not of received its own acknowledgment message. We can filter the acknowledgement message based on the source address and generate two basic events: 1) an acknowledgement message arrived, 2) an unexpected error. This is illustrated in Table 4.3.

Composite events are defined in the executable specification in the following format:

$$(E_{basic_{name1}}, E_{basic_{name2}}, ...), \text{first-order logic expression}, E_{composite_{name}}$$

Where $(E_{basic_{name1}}, E_{basic_{name2}}, ...)$ is the list of input symbols representing basic events that compose the composite event, and $E_{composite_{name}}$ is the symbol representing the composite event with a specific name. When an input symbol E_{basic} is produced in the process aforementioned, the first-order logic expressions for all composite events that are have the symbol in its list $(E_{basic_{name1}}, E_{basic_{name2}}, ...)$ are evaluated. If the evaluation results in a true statement, the corresponding composite event symbol $E_{composite_{name}}$ is generated.

4.2.2 History of Basic Events

In our design, a finite-state machine records all basic events in a *history of basic events*, and maintains them in a database. This database can be queried to build composite events as detailed in Section 4.2.1.2. The database stores only basic events information (messages and timer attributes), and not input symbols (E_{basic}) or state transitions. The reason is that the database of basic events is a separate component from the finite-state machine and input symbols and state transitions are only understood by the finite-state machine.

The history of basic events is needed for several reasons. Due to the mapping between basic events and symbols (one-to-one, one-to-many, many-to-one) the use of input symbols and states alone does not preserve all information about an event. A finite-state machine state will know the list of input symbols that has lead it to the current point of execution, but it cannot go back and retrieve the information regarding a specific event. As an example, if in the future, a message must be retransmitted, the entire message must be preserved; not just the knowledge that the message has arrived in the past.

By keeping a history of all events, our design does not need local variables to store information not inherent to the finite-state machine. Since all input symbols are based on events, the history permits the composition of any local variables anytime.

The history of basic events is a database that can be used to restore or rollback the execution of a finite-state machine.

Lastly, the history of basic events has the ability to provide the concept of time to the system. While the finite-state machine stores information about arrival order of events, the history of basic events also log the time of events. This is useful if the database needs to be queried for events that occurred in a specific time interval, for example, whether a message arrived within the last 15 seconds.

4.2.3 Outputs

Finite-state machine execution produces output symbols that map to a set of common tasks to be performed. We call these tasks *actions*. Actions are identified by the symbol A with the name of the action as the suffix. The actions are executed by the overlay node and are asynchronous to the execution of the finite-state machine. We define a small set of common actions that the finite-state machines can invoke to realize a wide spectrum services. We call this set of common actions *overlay network primitives*.

Actions can be invoked by the finite-state machine in three different ways as illustrated in Table 4.4. The table describes when an action can be performed and provides examples of actions.

Table 4.4: Types of Actions

Category	Description
Entry action	Action is performed when entering a state. For example, logging state information.
Exit action	Action is performed when exiting a state. For example, cleaning up data structures that are only within the scope of one state.
Transition action	Action is performed depending on the state and the input symbol ($\delta : S \times E \rightarrow S \times A$). For example, retransmitting a message in the event of a negative-acknowledgement message. Note that this is different from entry and exit actions because entry and exit actions are only performed when there is an external-transition (change of state), but transition actions are performed when in both external-transition and self-transitions (no change of state).

4.2.3.1 Actions - Set of Overlay Network Primitives

Our object is to define a small set of actions common to most overlay network services. The choice of the set of actions and their complexity represents a tradeoff. The number of required actions and the complexity of the finite-state machine is inversely proportional to the complexity of individual actions. If actions are too simple (store a variable, load a variable, set a variable, write to a register, etc.), then common tasks (*e.g.* sending a message) require a large number of these overlay network primitives. The finite-state machine require a large number of states and state transitions to define the large number of actions in their correct order. The executable specification, consequently, may become too large. Since executable specifications are create by network application developers, the actions should be linked to tasks from a human developer's perspective (*e.g.*, sending a message).

On the other hand, if actions are complex tasks, the resulting number of states of the finite-state machine would be small. However, as actions become more complex, they also become increasingly specialized. This means that the set of actions may grow large. This specialization may reduce the opportunity for sharing the set of available actions between finite-state machines of different services. For example, a complex action like"perform traffic shaping using token bucket algorithm" involves a subset of common tasks (setting timers to control the rate, buffering messages, etc.) that can be shared with other actions like "TCP congestion control" and "perform traffic shaping using leaky bucket algorithm". Hence in this case, instead of a large number of states when the actions are too fine-grained, we have an explosion of actions when overlay network primitives are overly complex and specialized.

Our objective is to define a set of actions that creates a small set of *overlay network primitives* which are common to most network services and with a moderate degree of complexity. These network primitives should be easily understandable by developers and useful in a variety of applications. We define these overlay network primitives below.

There are two categories of overlay network primitives: 1) primitives that manipulate messages, 2) primitives that perform network functions. First we provide the two primitives for manipulating messages.

1. $createMessage(*message, list < attribute, value >)$

   Creates a new message using *list*. The *list* is collection of required message attributes and their associated values. The attributes include:

   ```
   DeliveryMode
   DestinationAddress
   HopLimit
   MessageID
   MessageType
   Payload
   ```

   The `MessageID` identifies a message for a service. The `MessageType` identifies the message type (acknowledgement, negative-acknowledgement, payload, etc.). The `DeliveryMode` specifies if the message is to be sent using unicast, multicast, or flood. The `DestinationAddress` is the logical address of the destination node in the overlay network. This value can be left null in the case of multicast or flood. The `HopLimit` is the number of logical links traversed before the message is dropped. If this is left null, then it will default to a value of "255". The `Payload` is the data to be carried by the message and can be null in the case of control messages. When a message is created the `ServiceID` of the service is automatically set for the message. Each Message has a unique `<MessageID, ServiceID>` tuple. Note that we do not specify a `SourceAddress` and `PreviousHopAddress` for the message, this is set automatically to the logical address of the local node when the message is sent.

2. $setMessage(*message, list < attribute, value >)$

   Takes a message and modifies one or more attributes contained in *list*. The *list* is collection of message attributes and their associated values. The attributes that can be modified are the same as in the primitive $createMessage$.

Tasks such as copying of messages, concatenating of message payloads, etc. are not defined as primitives since they can be performed using $createMessage$ and $setMessage$. For example, copying a message is performed by creating a new message with all attributes of the message to be copied specified in the *list* parameter. The attributes for the message to be copied can obtained from stored message in the *history of basic events*.

Next we present primitives that perform network tasks.

1. *toApplication*$(*message)$

   Delivers a message to the application running the local overlay node.

2. $sendDataMessage(*message)$

   Sends a data message (messages with a payload) to the `DestinationAddress` with the specified *DeliveryMode*. The local node invoking $sendDataMessage$ sets its own logical address as the `SourceAddress` and `PreviousHopAddress`. The `MessageType` is set to "Payload".

3. $sendControlMessage(*message)$

   Sends a control message (messages without a payload) to the `DestinationAddress` with the specified *DeliveryMode*. The local node invoking $sendControlMessage$ sets its own logical address as the `SourceAddress` and `PreviousHopAddress`. The message must not contain a payload and `MessageID` must correspond to a data message.

4. $forwardMessage(*message)$

   Forwards a message not intended for the local node. The `HopLimit` is decremented and the `PreviousHopAddress` is set to the local node's logical address.

5. $setTimer(timerIdentifier, duration)$

   Sets a timer for the service with the `TimerID` given by the parameter *timerIdentifier*. The timer expires after the specified *duration* in milliseconds. When a timer is set the `ServiceID` of the service is automatically set for the timer. Each timer has a unique `<TimerID, ServiceID>` tuple.

6. $terminate(serviceIdentifier, messageIdentifier)$

   Terminate the finite-state machine instance for with a `ServiceID` given by the parameter *serviceIdentifier* and a `MessageIdentifier` given by the parameter *messageIdentifier*. Each finite-state machine instance is identified by a unique `<MessageID, ServiceID>` tuple.

7. $updateNodeInfo(*nodeInfo)$

   Updates the local node information, the event stores node information in the History of Basic Events that can be accessed by a service. Node information includes the local node's logical address and other topology information (parents, children, neighbours, etc.). This primitive is useful for updating neighbourhood information in networks with mobility.

4.2.3.2 Mapping of Output Symbols to Actions

An executable specification defines tasks it performs using a finite set of symbols A that correspond to actions. An actions consists of the symbol A_{name} which identifies the name of the action and a set of symbols $(p_1, p_2, ..., p_n)$ representing parameters associated with that action. These symbols are outputted by the finite-state machine asynchronously to the overlay node where these symbols are interpreted.

The overlay node maps the the symbol A_{name} to an invocation of a method with the *name*. The set of symbols $(p_1, p_2, ..., p_n)$ are then sent to that method as input data parameters for the method.

4.2.3.3 Format of Executable Specification

A service's executable specification have the format shown in Table 4.5.

Table 4.5: Format of an executable specification

Inputs	List of filtered input message events and their corresponding input symbols $E_{basic_{message}}$ List of filtered input timer events and their corresponding input symbols $E_{basic_{timer}}$ List of composite events composed from the defined basic events evaluated using first-order logic expressions and their corresponding input symbols $E_{composite}$
Behavior/Execution Flow	Finite-state machine (states S, initial state S_0, final states F, state transitions on inputs and producing outputs $\delta : S \times E \rightarrow S \times A$)
Outputs	List of actions and their corresponding output symbols A and parameters $(p_1, p_2, ...p_n)$ corresponding to an overlay network primitive

4.2.4 Expressiveness of Executable Specification

4.2.4.1 Custom Control Messages

Applications can define any number of custom control messages for their services. Using the *createMessage* primitive, custom `MessageType` attributes can be defined for a service. As long as the executable specification provide the correct mapping of these control message to input symbols based on the "MessageType=" filters, these control messages can be processed by the service as specified by the finite-state machine defined by the application. It is up to the onus of the created of the executable specification to ensure that the control messages that they create are also correctly processed as events. We do not place any restrictions on the number and usage of control messages.

4.2.4.2 Expressiveness of Composite Events

Composite events are evaluated using first-order logic expressions on a set of basic events. Table 4.6 shows the types of composite events that first-order logic can express.

There are limitations with using first-order logic. First, first-order logic predicates are associated with sets of individuals; in higher-order logic theories they may be also associated with sets of sets. Hence, some complicated features of natural language cannot be express in first-order logic. However, higher-order logic systems do not have complete formal deductive/inference systems. This makes it difficult to implement deductive/inference systems for high-order logic in software. We argue that events requiring higher-order logic are rare in communication systems where every message usually belong to only one set based on message type (acknowledgment, negative-acknowledgment, payload, etc.), and higher-order logic expressions like "there is at least one thing in common" between messages are not typically useful.

Second, first-order logic cannot express quantification over predicates [25]. An example of quantification over predicates is the statement "if the message A has the correct

Table 4.6: Composite events expressible using first-order logic

Type	Syntax	Example Usage
Property and Relationship	$P(x)$ $P(x, y)$	(1) arrived message has the expected sequence number: $expected(E_{basic_{message}})$ (2) message arrived from a source: $from(address, E_{basic_{message}})$
Equality	$x = y$, $x \neq y$	(1) two message arrival are the same: $E_{basic_{message1}} = E_{basic_{message2}}$
Combination	$P(x) \cap P(y)$, $P(x) \cup P(y)$, $P(x) \rightarrow P(y)$	(1) message arrived from destination A and destination B: $from(A, E_{basic_{message}}) \cap from(B, E_{basic_{message}})$ (2) message arrived from destination A or destination B: $from(A, E_{basic_{message}}) \cup from(B, E_{basic_{message}})$ (3) if message arrived from destination A, then message also arrived from destination B: $from(A, E_{basic_{message}}) \rightarrow from(B, E_{basic_{message}})$
Quantification	$\forall x$, $\exists x$, $\nexists x$ $\not\forall x$	(1) messages have arrived from all neighbours d: $\forall d : (\exists E_{basic_{message}} : from(d, E_{basicmessage}))$ (2) at least one messages have arrived from neighbours d $\exists d : (\exists E_{basic_{message}} : from(d, E_{basicmessage}))$ (3) no messages have arrived from any neighbours d $\nexists d : (\exists E_{basic_{message}} : from(d, E_{basicmessage}))$ (4) some (not all) messages have arrived from neighbours d: $\exists d : (\exists E_{basic_{message}} : from(d, E_{basicmessage})) \cap$ $\not\forall d : (\exists E_{basic_{message}} : from(d, E_{basicmessage}))$

sequence number, then there is at least one thing message A has in common with message B". Also the statement "for every set S of messages and every message x, either x is in S or it is not" cannot be expressed in first-order logic. Again, these expressions are not typically useful in a network setting.

Third, first-order logic also cannot express statements such as "message A arrived very slowly" or"message A's sequence number is very small" [25]. For an executable specification of services, we do not encounter these type of statements since all our inputs are independent events need that are not compared relatively to each other.

Lastly, first-order logic cannot express statements regarding the order of input events [25]. For example, "message A arrived before message B". First-order logic can only state "message A arrived $\cap$ message B arrived". However, this ordering of events in expressed by using timers in the finite-state machine as previously described in Section 4.2.1.1.

Given these arguments, we believe that composite events expressed in first order logic are sufficient for a wide range of services, despite some limitations the expressive power. First-order logic systems based on deductive reasoning and inferencing are readily available and can be leveraged by our design. For example, all relational databases are based on first-order logic.

4.2.4.3 Types of Expressible Services

Our network primitives allow a large set of network services to be realized. Simple tasks can be performed using a single action from the list. For example, the $createMessage$ primitive allows the creation of new messages of any identifier, message type, sequence number, and payload specified by the services. The created message can be transmitted to a destination using unicast, multicast, or broadcast using the $sendMessage$ primitive. Any message can be modified using the $setMessage$ primitive. Since all past events are persistently stored (Section 4.2.2), they can be queried any time by a service to obtain any previous message's attributes.

Using this set of network primitive, we are able to express all actions performed by data delivery services present in HyperCast [6] described in Section 3.1.2. Next we describe examples of other types of services that can be expressed.

Example 1: A service selectively accepts or discards messages based on message's source addresses; *i.e.*, a simple firewall allowing messages coming from certain logical addresses to pass through to the application. This service is realized by using filters defined in the executable specification for basic events (Section 4.2.1.3). Here, we define a set of filters on arriving messages based on messages' source address attribute. One filter is defined for every logic address that the firewall allows messages through. Hence, only accepted messages can trigger input symbols for the finite-state machine. Then the finite-state machine, upon receipt of the symbols, performs the action of delivering the message to the application. A timer is set to terminate the service after a specified duration. A finite-state machine diagram for this service is shown in Figure 4.6. The list of events and corresponding actions for this service is described in Table 4.7.

Example 2: A service that provides TCP-like connection establishment between two nodes. The service is realized through of a three way handshake between 2 nodes (A and B) as follows: Node B, after a setup period specified by the timeout, sends a SYN message to node A. Node A sends a SYN+ACK to node A when it receives the SYN message from A. Node B, upon recipe of the SYN+ACK message from node A, sends an ACK message to node A. The connection is established when node A receives the ACK.

A finite-state machine diagram for this service is shown in Figure 4.7. The list of events and actions for this handshaking service is described in Table 4.8.

Example 3: A traffic shaping service can be created by producing tokens (using timers) as inputs to the finite-state machine to specify when a message should be sent. For example, a leaky bucket traffic shaper can be specified by list of events and actions described in Table 4.9, and a finite-state machine diagram for this service is shown in

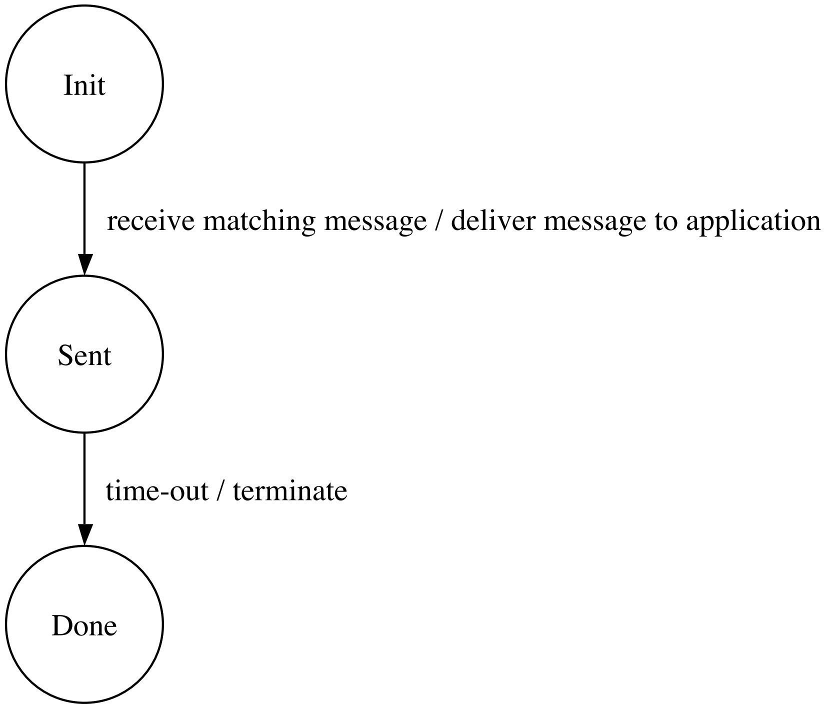


Figure 4.6: Finite-state machine for message selection service

Table 4.7: State transitions for message selection service

State	Event	Condition	Action
Init	Message arrived	Message's source address maps to a symbol	Go to state Sent and deliver the message to application
Init	Message arrived	Message's source address does not map to a symbol	Do nothing
Sent	Timeout to terminate the service	In all cases	Go to state Done
Done	-	-	Terminate the finite-state machine upon entry

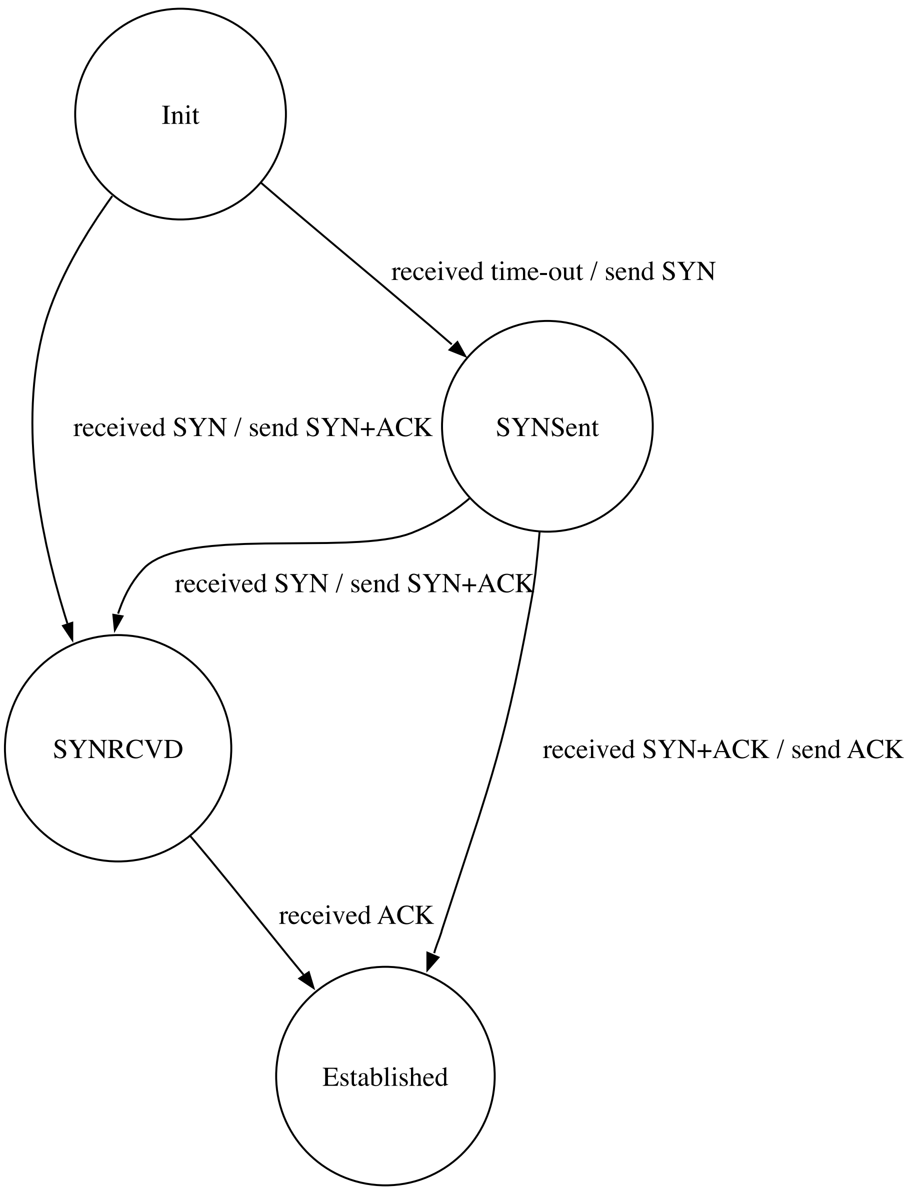

Init
received time-out / send SYN
received SYN / send SYN+ACK
SYNSent
received SYN / send SYN+ACK
SYNRCVD
received SYN+ACK / send ACK
received ACK
Established

Figure 4.7: Finite-state machine for connection establishment service between two nodes

Figure 4.8.

We believe our design is expressive enough to be extended to overlay routing, topology management, network address translation, and network discovery services with minor changes. All these overlay services are based on the transfer of special overlay control messages (beacon messages, control messages, bootstrap messages, etc.) that an executable specification can accommodates as described in Section 4.2.4.1. The history of

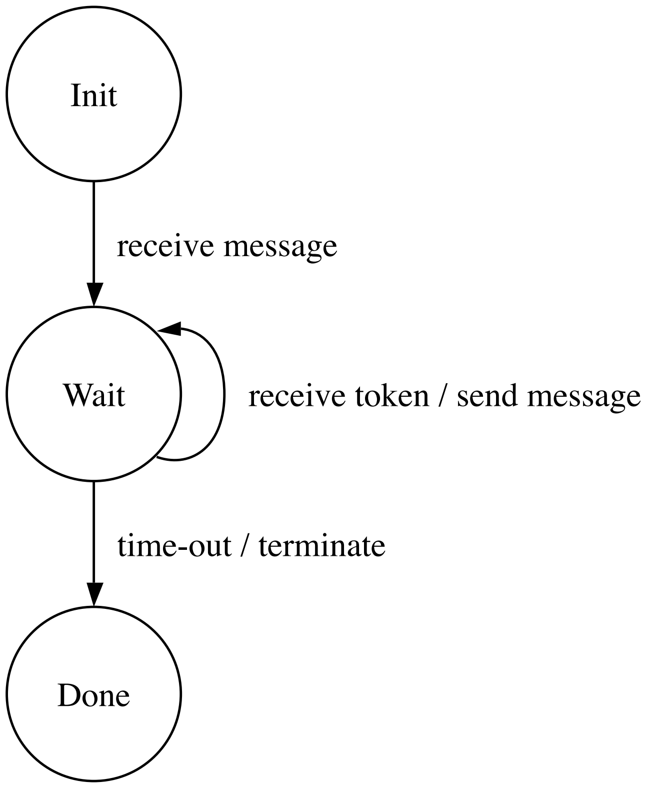


Figure 4.8: Finite-state machine for a leaky bucket transmitter

Table 4.8: State transitions for connection establishment service

State	Event	Condition	Action
Init	SYN message arrived	In all cases	Go to state SYNRCVD and send SYN+ACK message
Init	Timeout	In all cases	Go to state SYNSent
SYNRCVD	ACK message arrived	In all cases	Go to state Established
SYNSent	SYN message arrived	In all cases	Go to state SYNRCVD and send SYN+ACK message
SYNSent	SYN+ACK message arrived	In all cases	Go to state Established and send ACK message

basic events is able to store the complete message history of these control messages and it can be used by composite events to query network information. However, currently our design does not allow a finite-state machine defined by the application in an executable specification to be modified during runtime. The executable specifications are currently immutable. Some routing, network address translation, network discovery, and topology management services may require new states and state transitions to be created dynamically as the overlay network changes.

Services that our current design cannot express are services that require operation on a synchronized clock. For example, packet scheduling, queue management, measuring the network, etc. Our input events and output actions operate asynchronously, and on a network scale which is inadequate for time-sensitive tasks.

Table 4.9: State transitions for a leaky bucket transmitter

State	Event	Condition	Action
Init	Message arrived	In all cases	Go to state Wait
Wait	Receives a token	In all cases	Self-transition and send message
Wait	Timeout to terminate the service	In all cases	Go to state Done
Done	-	-	Terminate the finite-state machine upon entry

4.3 Executable Specification Processor

In this section we describe a software system on overlay nodes that can execute services from executable specifications. At a high level, our design extends the concept of MessageStore introduced in HyperCast [6]. The MessageStore is a component transparent to network applications that process service messages. The original MessageStore have a fixed set of services, our design extends this by allowing new services to be added to MessageStore during run-time.

4.3.1 Creation and Deployment of Services

New services are created by specifying new executable specifications with the format described in Section 4.2.3.3. Each service is assigned an unique identifier `ServiceID`. The local application that creates a new service directly load its executable specification to its overlay node's MessageStore and also publishes the executable specification on a remote server responsible for storing the executable specifications for all services. Overlay nodes without the new service can download its executable specification from the remote server.

Once an executable specification is downloaded successfully, it is validated and stored in MessageStore in a repository with the `ServiceID` as the search key. The validation process checks the executable specification to ensue the structure of the executable specification is complete, correct, and any data types are of the correct form (byte strings, characters, integers, floats, etc.). Currently there is no facility to check against malicious behaviour or security flaws in executable specifications.

Finite-state machines of the services can be instantiated given an executable specification of the service. These finite-state machines, when executed, realize the services.

4.3.2 Using Services

An application invoke a service by transmitting overlay data messages marked with a service (with `ServiceID` as one of its message attributes). The message can be sent using unicast, multicast, or flood in the overlay network. Any overlay node that sends or receives a message with marked with a `ServiceID` will process the message in its local *MessageStore*. The processing of this message in MessageStore is illustrated in Figure 4.9.

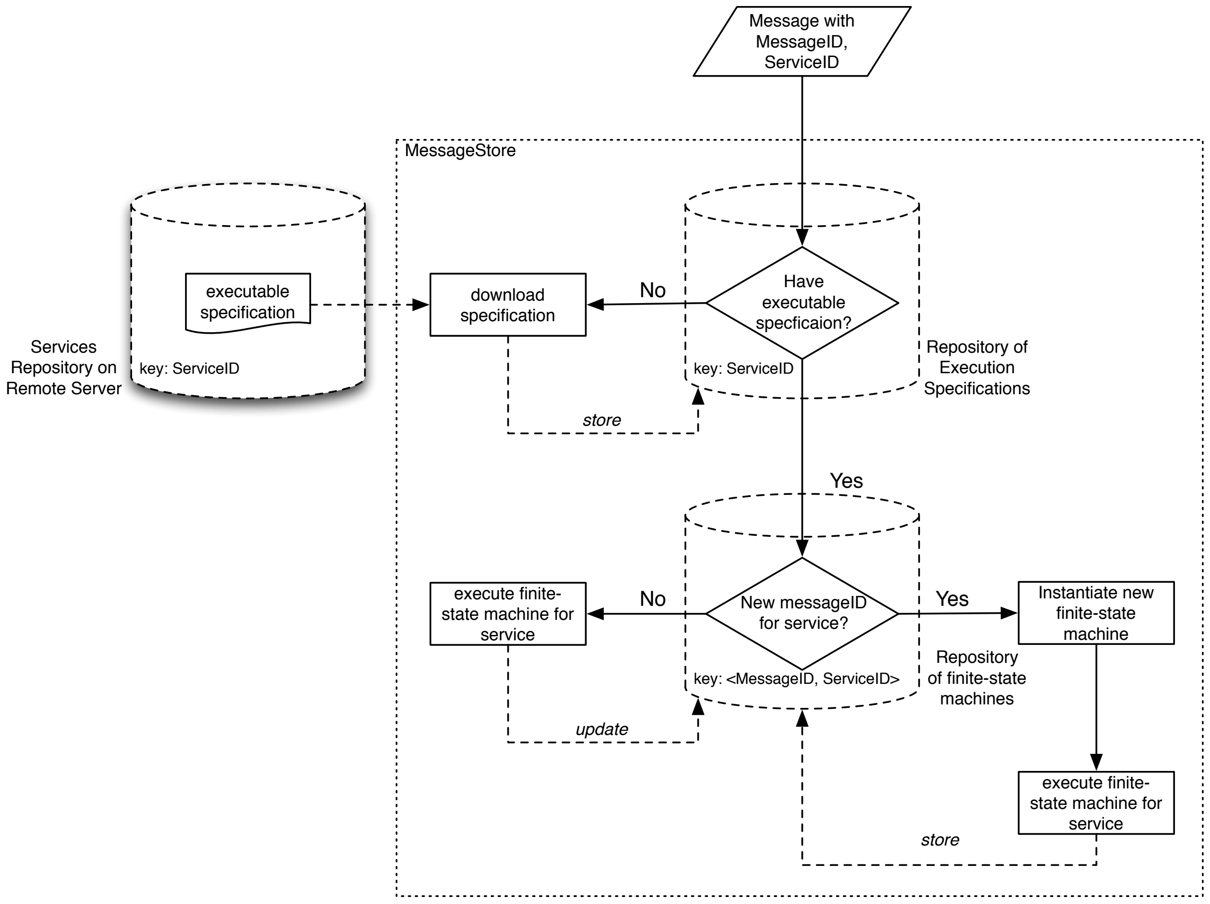

Message with MessageID, ServiceID
MessageStore
Services Repository on Remote Server
executable specification
key: ServiceID
download specification
No
Have executable specficaion?
key: ServiceID
Repository of Execution Specifications
store
Yes
execute finite-state machine for service
No
New messageID for service?
Yes
Instantiate new finite-state machine
key: <MessageID, ServiceID>
Repository of finite-state machines
update
execute finite-state machine for service
store

Figure 4.9: Message processing in *MessageStore*

When an overlay receives a message from the application or from another overlay node, the message is first examined to see if the message contains a `ServiceID` attribute. If this attribute is present, the message is associated with a service and the processing is delegated to the local MessageStore. MessageStore check its repository of executable specifications for the `ServiceID` (*i.e.*, if it knows how to process the service). The downloaded executable specification is validated and parsed, and stored in MessageStore's repository of services. If the service is not found in the MessageStore, the node attempts to download the service's executable specification from a remote server that contains the executable specifications of all known services. In the scenario where the executable specification is not found on the remote server, an overlay node sends a *service request message* to all its neighbours asking any neighbours with the executable specification for the service to publish the executable specification to the remote server. The overlay node then waits and attempts to download the executable specification from the remote server. This is repeated for a application specified number of tries, after which the message with with the unknown `ServiceID` is dropped.

The received message is then examined for its `MessageID` attribute. Service messages are uniquely identified by their `MessageID`s. MessageStore instantiates a new instance of a service as a finite-state machine (defined by the service's executable specification) when it receives a message with a new `<MessageID, ServiceID>` tuple. In other words, a finite-state machine is instantiated for every message with a unique `<MessageID, ServiceID>` tuple. The finite-state machine is then executed. A finite-state machine of a `<MessageID, ServiceID>` tuple is stored locally in MessageStore in a repository until the service for that tuple completes. In the case that the received message has a preexisting `<MessageID, ServiceID>` tuple in the MessageStore repository, the corresponding finite-state machine is obtained, executed, and updated.

Messages are categorized as *data messages* containing payload and *control messages* containing control information (i.e. ACK, NACK, etc.). Both message types are recognized as part of one instance for a service, *i.e.*, one finite-sate machine, if they have the same `<MessageID, ServiceID>` tuple. For data messages, if two messages have the same `MessageID` and `ServiceID` they are treated as two copies of the same message, even if they have different payloads or different source addresses. Control messages that have the same `MessageID` and `serviceID` as a data message belong to the same service instances (*i.e.*, they are control messages for the data message).

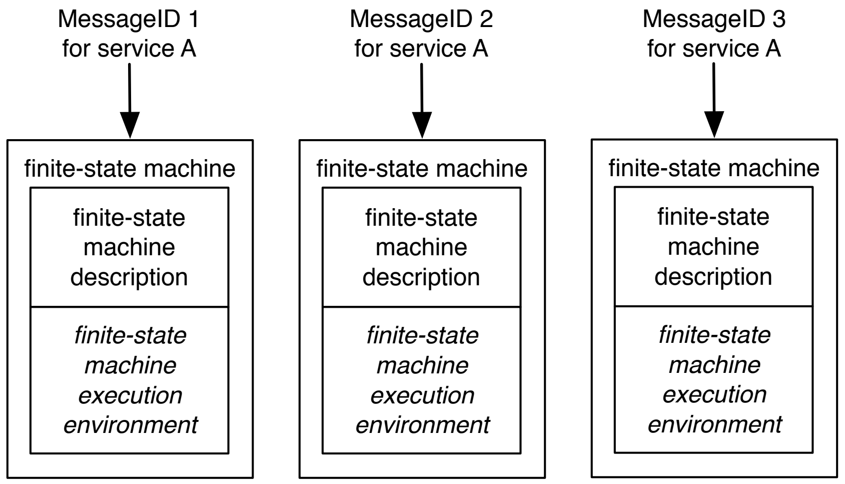
In the next section, we describe how finite-state machines are executed.

4.3.3 Finite-State Machine Execution

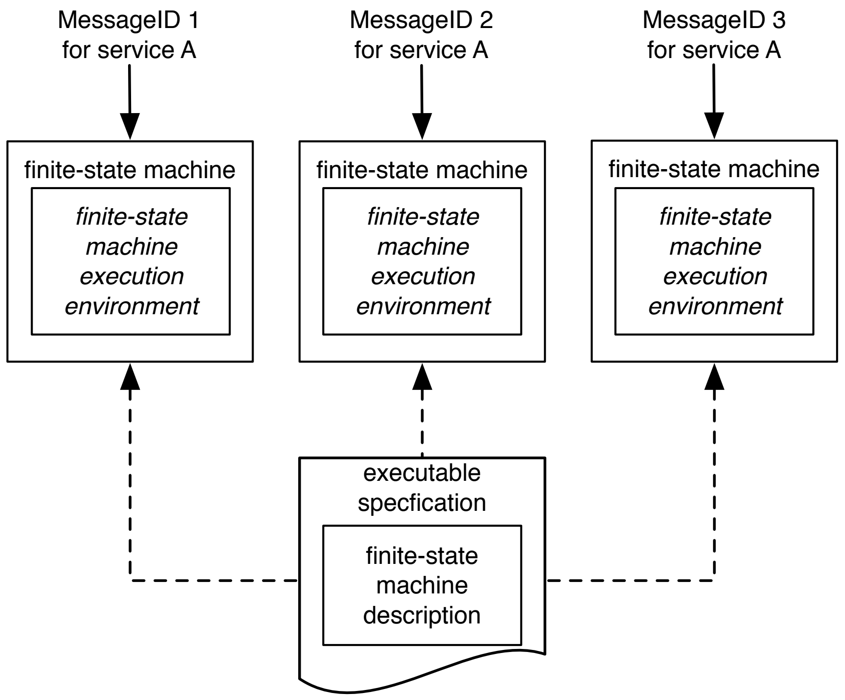
Our design decouples the description of the finite-state machine and the execution environment that executes the finite-state machine. An execution specification is shared between all finite-state machines of its service. This allows a single copy of certain service information (description of inputs, output, states, and state transitions described in the executable specification) to be shared among a large number of finite-state machines of the same service (Figure 4.10b), compared with the original MessageStore Figure 4.10a) where each finite-state machine must have its own individual copy.

MessageStore provides a finite-state machine execution engine that can execute finite-state machines. Given a finite-state machine specified by an executable specification, the execution engine is a software environment that receives input symbols, and performed state transitions and actions. There is a finite-state machine execution engine instance for each finite-state machine and its components are shown in Figure 4.11.

When a finite-state machine for a service is instantiated, it starts in the starting state specified in the executable specification and waits for input symbols. The overlay node communicates with a finite-state machine via its *Event Handler* and *Action Dispatcher* interfaces. Events (messages, timers) are delivered from the overlay node to the Event Handler to be processed. The Event Handler is the software realization of the events to input mappings as described in Section 4.2.1.3. The *Event Handler* filters the events based on the executable specification and produce input symbols for the finite-state machine. Basic events trigger input symbols E_{basic} and are stored in the history of basic events as described in Section 4.2.2. The Event Handler also checks if any composite


MessageID 1 for service A
MessageID 2 for service A
MessageID 3 for service A
finite-state machine
finite-state machine description
finite-state machine execution environment

(a) Finite-state machine description and execution environment coupled


MessageID 1 for service A
MessageID 2 for service A
MessageID 3 for service A
finite-state machine
finite-state machine execution environment
executable specfication
finite-state machine description

(b) Finite-state machine description and execution environment decoupled

Figure 4.10: Execution models for finite-state machines

events are built from with the input symbols E_{basic} as described in Section 4.2.1.3. If so, the first-order logic expressions for those composite events are evaluated using a first-order logic processor provided in MessageStore. This processor can query the history of

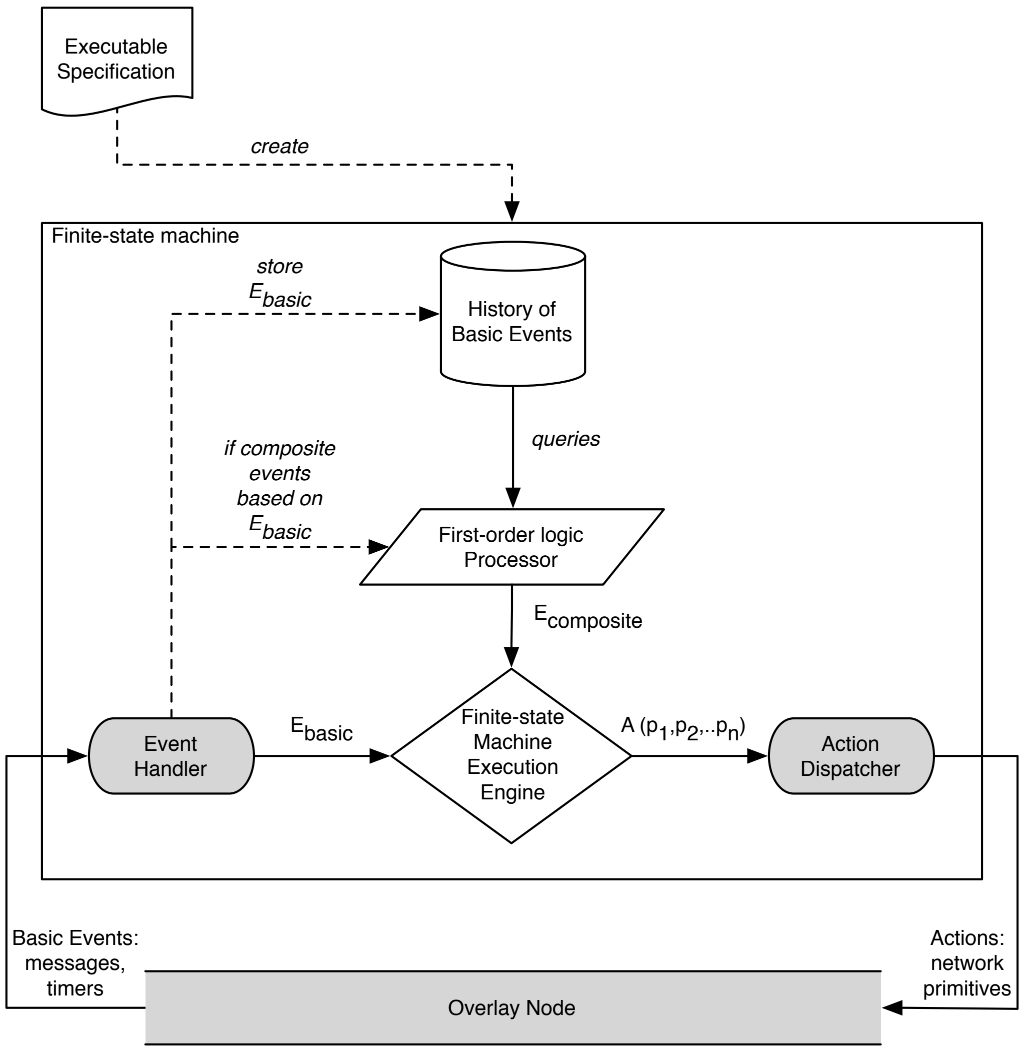


Figure 4.11: Finite-state machine execution engine and its interaction with the overlay node

basic events if needed.

Based on the finite-state machine's current state and the input symbols produced from the Event Handler, the finite-state machine execution engine checks if any state transitions need to be make and if any actions need to be performed.

Actions to be perfumed are defined in the executable specification. The finite-state machine produces output symbols A representing these actions. The Action Dispatcher is the software realization of the mapping from output symbols A to actions as described in Section 4.2.3.2. Output symbols A correspond overlay network primitives in MessageStore as described in Section 4.2.3.1. The Action dispatcher invokes the primitive identified by the name of the output symbols A along with its list of parameters $(p_1, p_2, ...p_n)$. The primitive then process the action asynchronously to the execution of the finite-state machine.

Chapter 5

Implementation

In this chapter, we discuss the implementation for application customizable data transfer services in the HyperCast overlay middleware software system. To accomplish this, we extend the existing MessageStore implementation in HyperCast 4.0. There are three design consideration that have influenced our implementation. First, the MessageStore should be service independent in the sense that it can be used to realize any service that can be specified as a finite-state machine using SCXML. Second, the implementation of the MessageStore should require no modifications to other HyperCast components. Thirdly, the design of the MessageStore should be modularized with clearly defined interfaces to other components. This allows each individual component to be upgraded or replaced with minimal modifications to other MessageStore components.

5.1 MessageStore Extensions

Realizing application customizable information services without modifications to the existing HyperCast software system requires that all modifications be made locally in MessageStore. There is no modification to how messages marked with a service are sent or received by the overlay node. Recall that MessageStore was originally designed as a component within the overlay socket to provide services without altering the processing of incoming and outgoing messages not using the services. In the context of HyperCast, MessageStore is an on-demand *processor* component that is invoked when messages marked with services are transferred in an application-layer overlay network.

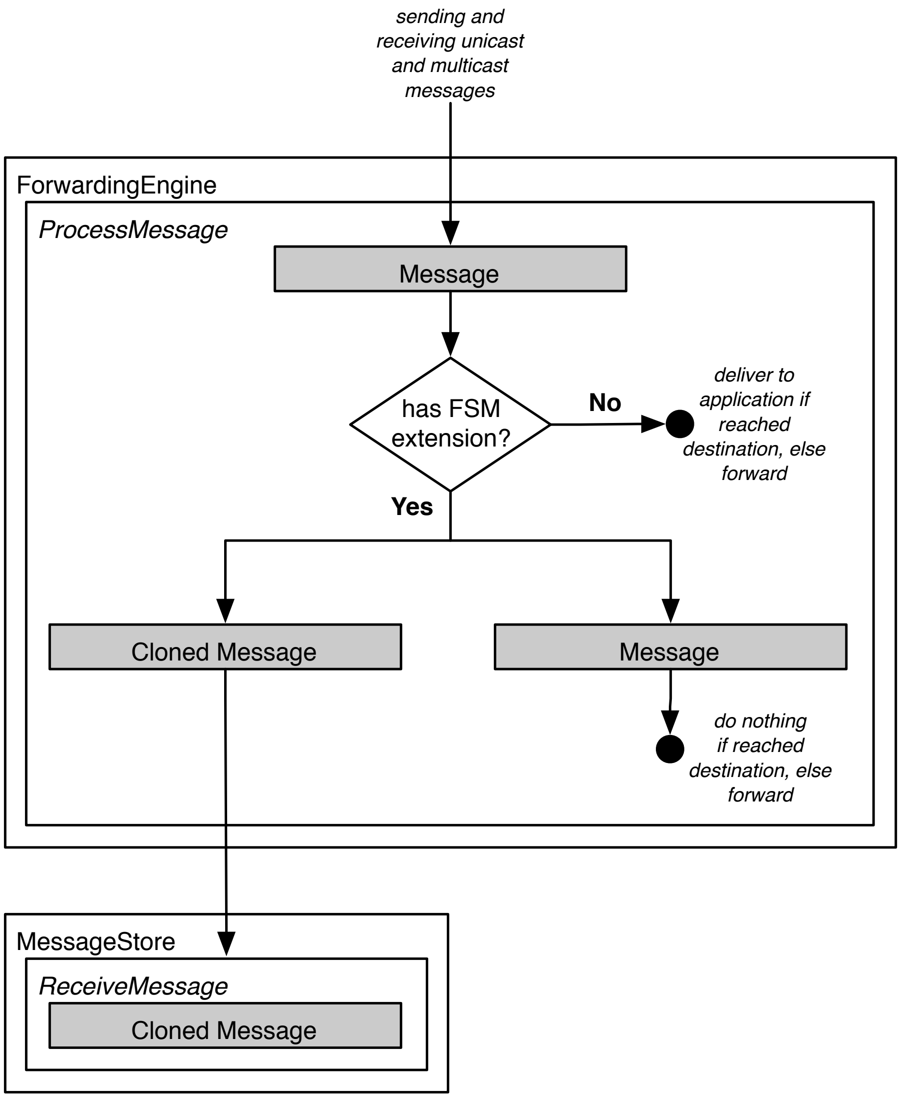

sending and receiving unicast and multicast messages
ForwardingEngine
ProcessMessage
Message
has FSM extension?
No
deliver to application if reached destination, else forward
Yes
Cloned Message
Message
do nothing if reached destination, else forward
MessageStore
ReceiveMessage
Cloned Message

Figure 5.1: Message handling in MessageStore

In Figure 5.1, the overlay node is either sending an outgoing message or receiving an incoming message (both unicast and multicast). The processing in the MessageStore is the same for both scenarios. The message is processed in the `processMessage` method of the ForwardingEngine. The ForwardingEngine checks if the message has an FSM extension (marked with a service). If the message is not marked with a service, the ForwardingEngine processes the message without involving the MessageStore. If the message is marked with a service, it is cloned and the cloned message is passed to the `receiveMessage` method of the MessageStore. A thread in the MessageStore processes the cloned message concurrently with the ForwardingEngine from this point onward. The flow is designed in this way since the processing time of a message in the MessageStore, may require a significant amount of processing. Thus, if the overlay node is not the intended destination of the message, forwarding the message after the processing is completed by the MessageStore may increase the delay. By cloning messages, messages can be forwarded to their next-hop destinations as soon as they are received by the ForwardingEngine. This processing flow exists in HyperCast 4.0 and was not altered.

The original message resumes processing in the ForwardingEngine but is only forwarded and not passed to the application in the overlay node. The MessageStore takes over responsibility of choosing when and how the message is passed to the application.

5.1.1 Implementation Overview

Our implementation does not change the core concept of the MessageStore, *i.e.*, it is a repository of data messages that executes finite-state machines for those messages. Also, we do not modify the MessageStore interface as viewed by other HyperCast components. Our implementation modifies the MessageStore by storing the services as executable specifications in SCXML and providing facilities to execute these services using a generic SCXML execution engine. An overview of the implementation is shown in Figure 5.2.

From an executable specification given as a SCXML document, MessageStore creates an finite-state machine Java object, referred to as a FSM object, which contains three major components. The first component is the SCXML engine which is able to directly execute the executable specification. The second component is the Event Handler which takes messages and timers from the MessageStore relating to this finite-state machine (FSM) object and triggers them as events in the SCXML engine. This is illustrated in Figure 5.3. The Event Handler takes a message arrival or timer expiration coming from MessageStore and checks event's attributes against the defined events in the executable specification for the name of the event. The defined event attributes, in the form of a XML DOM tree, is used as a filters on the message/timer. If a match is found, the name of the event is used to trigger the SCXML engine. The triggered events are stored in the History of Basic Events (See Section 4.2.1.1).

The third component is the Action Dispatcher which takes actions defined in the executable specification (embedded in the SCXML document) and executes the corresponding Java method pre-defined in the MessageStore's list of actions. The list of actions is the set of overlay network primitives discussed in the previous chapter. This is illustrated in Figure 5.4. The executable specification uses the SCXML custom actions interface to creates actions, in the form of XML documents, to be executed based on event arrivals and state transitions. The XML document is a DOM tree where the root element of the tree is the name of the Java method provided by MessageStore. The children elements

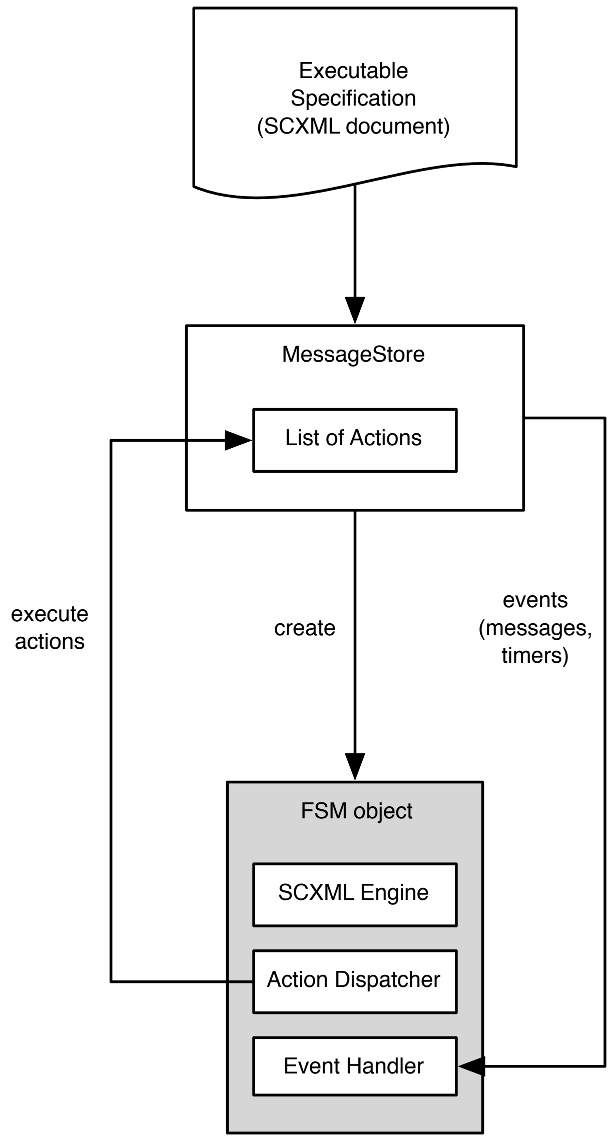


Figure 5.2: Overview of implementation in MessageStore to use executable specification for services

contains the list of parameters that maps to the Java method's parameters. The Action Dispatcher is responsible for mapping the tree elements to the actual Java method.

In the following sections, we describe our implementation in detail. First, we discuss how a service is defined as an executable specification in SCXML. Then, we describe the software components in the MessageStore to process the executable specifications. Lastly, we describe our mechanisms to deploy new services to MessageStore.

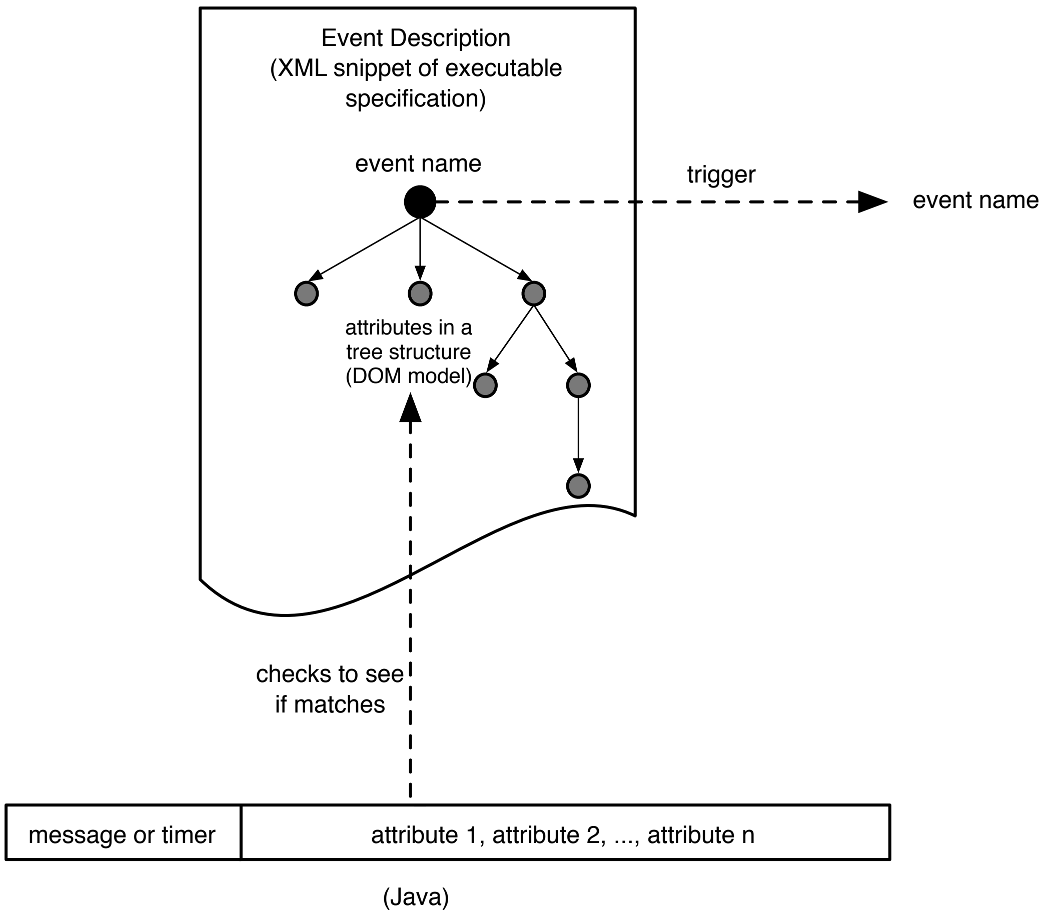


Figure 5.3: Mapping of a message arrival or timer expiration to an event in the executable specification

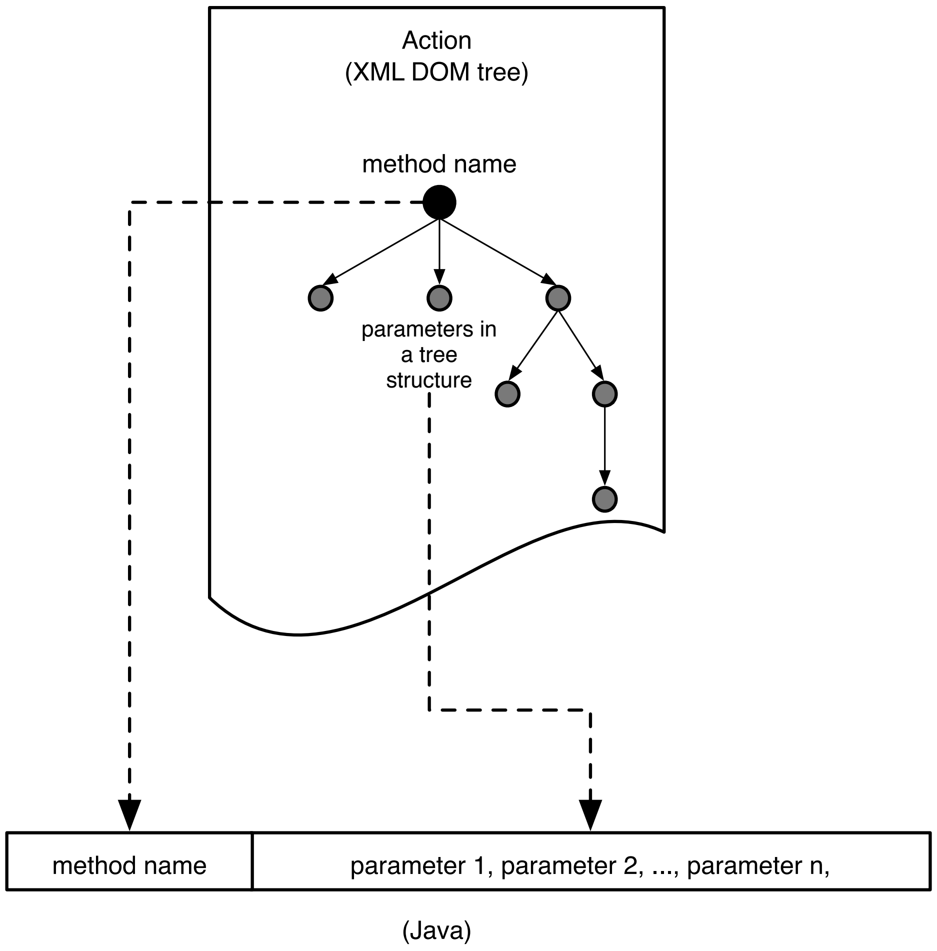

Action
(XML DOM tree)
method name
parameters in a tree structure
method name
parameter 1, parameter 2, ..., parameter n,
(Java)

Figure 5.4: Mapping of an action in the executable specification to a Java method

5.1.2 A Service as an Executable Specification

As discussed in the previous chapter, we specify services in the form of an executable specification. This is in contrast to the hard-coded finite-state machines which define services in the MessageStore in HyperCast 4.0. We modify MessageStore to use executable specifications in the form of SCXML documents. There is one SCXML documents for each service. The SCXML document can be parsed and executed by the Apache Commons SCXML engine. In this section, we describe in detail how we specify a service using SCXML.

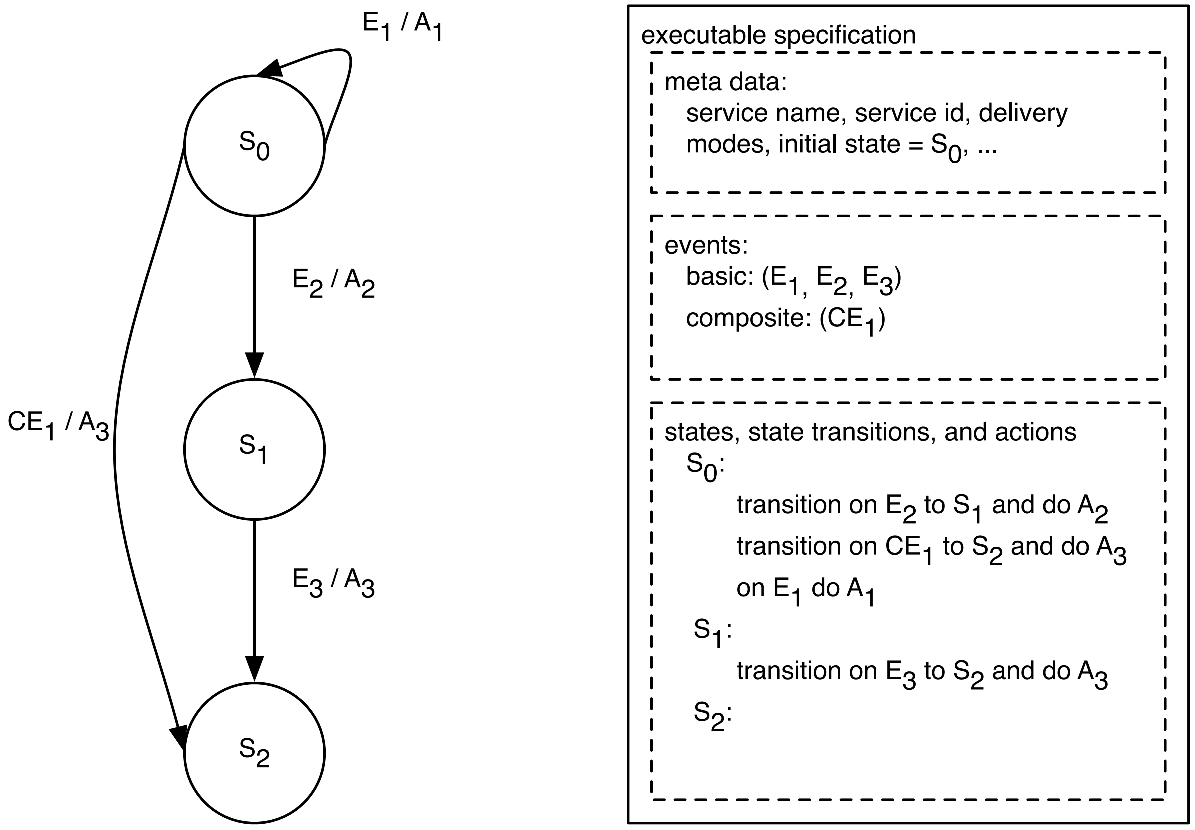

E_1 / A_1
S_0
E_2 / A_2
CE_1 / A_3
S_1
E_3 / A_3
S_2
executable specification
meta data:
service name, service id, delivery modes, initial state = S_0, ...
events:
basic: (E_1, E_2, E_3)
composite: (CE_1)
states, state transitions, and actions
S_0:
transition on E_2 to S_1 and do A_2
transition on CE_1 to S_2 and do A_3
on E_1 do A_1
S_1:
transition on E_3 to S_2 and do A_3
S_2:

Figure 5.5: The structure for an executable specification of a service defined by a finite-state machine

As illustrated in Figure 5.5, the executable specification is composed of 1) a declaration of meta-data, 2) a declaration of events $(E_1, CE_1, E_2, ...)$ where E_n represents a basic event and CE_n represents a composite event, 3) a declaration of states $(S_0, S_1, ...)$ and state transitions based on the events ($S_0 \rightarrow S_1$ on $E_2, ...$), and 4) a declaration of actions ($A_1, A_2, ...$) that are performed as a consequence of events and/or state transitions. The meta data definitions and the definition of events are sections in the SCXML document which provide the necessary information for the Event Handler and Action Dispatcher to process the service as described previously.

5.1.2.1 Header and Meta Data

The executable specification for a service contains meta data. The meta data is contained in the header and meta data section of the SCXML document. The header portion of the executable specification describes the XML version, SCXML version, namespaces, and initial state of the service. We use the Hop-to-Hop Acknowledgment service, described in Section 3.1.2.1, as an example SCXML document.

```
<?xml version="1.0"?>
<scxml  version="1.0"
        xmlns="http://www.w3.org/2005/07/scxml"
        xmlns:action="http://my.custom-actions.domain/action"
        initialstate="init">
```

The above SCXML snippet identifies the XML and SCXML versions as“1.0” and defines the default namespace (`xmlns`) and the `action` namespace (`xmlns:action`). The `action` namespace is the namespace for the SCXML element tag `<action:>`. The XML snippet enclosed within the `<action></action>` describe an action and its parameters corresponding to a Java method as described in Figure 5.4. The Action Dispatcher maps these actions to the list of network primitives in MessageStore. The `initialstate` attribute defines the initial state of the finite-state machine. Additional meta data is contained within two custom SCXML data elements `<data name="service">` and `<data name="metaops">` as shown below.

```
<datamodel>
    <data name="service">
        <serviceName>h2hack</serviceName>
        <serviceID>1</serviceID>
        <serviceType>message-oriented</serviceType>
        <serviceDeliveryMode>unicast,multicast</serviceDeliveryMode>
    </data>
    <data name="metaops">
        <messageStoreWillForwardMessage>false</messageStoreWillForwardMessage>
        <processIntermediateUnicastMessage>true</processIntermediateUnicastMessage>
    </data>
</datamodel>
```

For this example service, the service name is “h2hack”, and its identifier (`serviceID`) is 1. The service supports message-oriented delivery with delivery modes of unicast and multicast. Message-oriented delivery means that one finite-state machine is created per data message for a service. The first meta data defines that for this service, MessageStore does not forward messages. This means that a data message will be forwarded as soon as it is received by the overlay node and a cloned copy of the message will be processed in MessageStore (default behavior). If the tag `<messageStoreWillForwardMessage>` is set to true, it means that MessageStore is responsible for forwarding the message. In this case, a data message marked with this service is not cloned. This is useful in services such as Duplicate Elimination where a message is checked to see if it has already been received before forwarding. The second meta data determines if MessageStore will process unicast messages if the overlay node is an intermediate node between the source and destination. For some services, intermediate nodes need not process a service message. This is done to reduce overhead at intermediate nodes.

5.1.2.2 Events

Table 5.1: Events for Hop-to-Hop Acknowledgment Service

Event	Basic/Composite	Type
Payload message	basic	message
Local ACK message	basic	message
ACK request message	basic	message
NACK message	basic	message
Reset message	basic	message
Timeout$_{\text{ACK}}$	basic	timer
Timeout$_{\text{MaxACK}}$	basic	timer
Timeout$_{\text{NACK}}$	basic	timer
Timeout$_{\text{RST}}$	basic	timer
Timeout$_{\text{MaxRST}}$	basic	timer
Timeout$_{\text{MSG}}$	basic	timer
all ACKs received	composite	first-order logic expression on Local ACK messages

Events of an executable specification are either basic events or composite events. Recall that a basic event is either a message arrival or a timer expiration. Composite events are based on first-order logic expression evaluation of one or more basic events. For the Hop-to-Hop Acknowledgment service we have the 11 events (Table 5.1). There is one composite event, when all "Local ACK messages" have been received for a forwarded message. Which expressed that acknowledgement messages have arrived from all expected nodes.

In the SCXML document, events descriptions are enclosed within `<data name="events">` elements. We leverage the SCXML datamodel as these event descriptions are structured data. The SCXML datamodel express data as a DOM tree. The corresponding section in the executable specification is specified in SCXML as follows:

```
<datamodel>
    <data name="events">
        <Basic>
            <!-- messages -->
            <Event name="h2hnackmessage">
                <Message>
                    <MessageType>1</MessageType>
                </Message>
            </Event>
            <Event name="h2hackmessage">
                <Message>
                    <MessageType>2</MessageType>
                </Message>
            </Event>
            <Event name="h2hackreqmessage">
                <Message>
                    <MessageType>3</MessageType>
                </Message>
            </Event>
            <Event name="resetmessage">
                <Message>
                    <MessageType>4</MessageType>
                </Message>
            </Event>
            <Event name="payloadmessage">
                <!-- the message type is defined in HyperCast -->
                <!-- to be 0x80 in 8 bits for a Payload -->
                <!-- Message which is -128 decimal -->
                <Message>
                    <MessageType>-128</MessageType>
                </Message>
            </Event>
            <!-- timers -->
            <Event name="finaldelete">
                <!-- this is a timer set in the initial state to -->
                <!-- auto-delete the finite-state machine -->
                <!-- after a certain extended period of time -->
                <Timer>
                    <TimerIdentifier>1</TimerIdentifier>
                </Timer>
            </Event>
            <Event name="nacktimeout">
                <Timer>
                    <TimerIdentifier>2</TimerIdentifier>
                </Timer>
            </Event>
            <Event name="acktimeout">
                <Timer>
                    <TimerIdentifier>3</TimerIdentifier>
                </Timer>
            </Event>
            <Event name="maxacktimeout">
                <Timer>
                    <TimerIdentifier>4</TimerIdentifier>
                </Timer>
            </Event>
            <Event name="msgtimeout">
                <Timer>
                    <TimerIdentifier>5</TimerIdentifier>
                </Timer>
            </Event>
            <Event name="resettimeout">
                <Timer>
                    <TimerIdentifier>6</TimerIdentifier>
                </Timer>
            </Event>
            <Event name="maxresettimeout">
                <Timer>
                    <TimerIdentifier>7</TimerIdentifier>
                </Timer>
            </Event>
        </Basic>
    </data>
</datamodel>
```

Basic events are identified by the SCXML element `<Basic>`. Let us discuss a few events in detail. For example, the basic event named “payloadmessage” indicates that it is an event based on the receipt of a “Payload message”. The SCXML elements `<Message>` and `<MessageType>`, which are required attributes of the event named “payloadmessage”, indicate that the event is triggered by messages with a message type decimal value of 0x80. (The event named “payloadmessage” filters messages based on these two attributes.) Similarly, the basic event named “acktimeout” is an timer expiration event based on $\text{Timeout}_{\text{ACK}}$ (See Table 3.2). The SCXML elements `<Timer>` and `<TimerIdentifier>`, which are parameters for the event description named “acktimeout”, indicate that the event is triggered by a timer with identifier “1”.

For the Hop-to-Hop Acknowledgement service we have one composite event when the overlay node receives all “Local ACK messages” from all children nodes after the “Payload message” is forwarded to them. This event is triggered if the set of the source logical addresses of received “Local ACK messages” contains the set of logical addresses of children nodes.

This relationship is expressed in first-order logic as discuss in the Section 4.2.1.2. The express is as follows:

“For each neighbour d, there exists an acknowledgement message $E_{basic_{ACK}}$, such that $E_{basic_{ACK}}$ comes from d and there does not exist a timeout $E_{basic_{timeout}}$ for d indicating the maximum waiting time for the acknowledgment has expired for d.”

$\forall d : (\exists E_{basic_{ACK}} : from(d, E_{basic_{ACK}}) \cap \nexists E_{basic_{timeout}} : timeout(d, E_{basic_{timeout}}))$.

This composite event is defined as follows in the executable specification:

```
<Composite>
    <Event name="allh2hackmessagesreceived">
        <ConstituentBasicEvent>h2hackmessage</ConstituentBasicEvent>
        <XQuery>
            let $x := //Node[last()]/Children/LogicalAddress,
            let $y := //Event[@name="h2hackmessage"]/Message/SourceAddress return
            for $i in $x return
            if ($i = $y) then &lt;out&gt;true&lt;/out&gt; else &lt;out&gt;false&lt;/out&gt;
        </XQuery>
</Composite>
```

Composite events are specified within the SCXML element `<Composite>`. The above SCXML snippet identifies the name of the composite event as "allackmessagesreceived". This composite event's first-order logic express is evaluated when the basic event "h2hackmessage" is triggered. The first-order logic expression of the composite event is expressed using XQuery and XPath contained within the `<XQuery>` element (See Section 3.3).

The XQuery sets two variables x and y. The variable x is the set containing all the local node's children's logical addresses and the variable y is the set of all acknowledgement message's source logical addresses. Both of these data is queried from the History of Basic Events (See Section 4.2.2) using XPath. Recall all acknowledgement messages recorded as events when they arrive and the node information is updated when $updateNodeInfo$ (See Section 4.2.3.1)is invoked. The `last()` predicate indicates the last updated node information is to be used. The last two lines of the XQuery expression means that if for all local node's children logical addresses, if there is an acknowledgement with a matching source logical address then the expression evaluates to true, else false. Note that the result is outputted as `<out>true</out>` and `<out>false</out>` elements and the `&lt;` and `&gt;` symbols are the unicode equivalent of `<` and `>` which are illegal characters in an XML document.

The expression are evaluated using Saxon XQuery and XSLT processor libraries. A true result triggers the composite event "allackmessagesreceived".

5.1.2.3 States and State Transitions

The next part of the executable specification is the declaration of states and state transitions. The syntax of states and state transitions is defined by the SCXML specification. Below is the SCXML portion describing the states and state transitions for the Hop-to-Hop Acknowledgment service (See Figure 3.6).

```
<state id="init">
    <transition event="payloadmessage" target="waitack">
        ...
    </transition>
    <transition event="h2hackreqmessage" target="nopayload">
        ...
    </transition>
    <transition event="finaldelete">
        ...
    </transition>
</state>

<state id="waitack">
    <transition event="allh2hackmessagesreceived" target="done">
        ...
    </transition>
    <transition event="h2hackreqmessage">
        ...
    </transition>
    <transition event="h2hnackmessage">
        ...
    </transition> -->
    <transition event="resettimeout" target="reset">
        ...
    </transition>
    <transition event="acktimeout">
        ...
    </transition>
    <transition event="maxacktimeout" target="done">
        ...
    </transition>
</state>

<state id="nopayload">
    <transition event="payloadmessage">
        ...
    </transition>
    <transition event="h2hackreqmessage">
        ...
    </transition>
    <transition event="reset" target="reset">
        ...
    </transition>
    <transition event="nacktimeout">
        ...
    </transition>
    <transition event="maxnacktimeout" target="reset">
        ...
    </transition>
</state>

<state id="done">
```

```
        <transition event="h2hackreqmessage">
            ...
        </transition>
        <transition event="h2hnackmessage">
            ...
        </transition> -->
        <transition event="resettimeout" target="reset">
            ...
        </transition>
        <transition event="deletetimeout" target="init">
            ...
        </transition>
    </state>

    <state id="reset">
        <transition event="resettimeout">
            ...
        </transition>
         <transition event="maxresettimeoout" target="init">
            ...
        </transition>
     </state>
```

According to the finite-state machine of the Hop-to-Hop Acknowledgment service there are five states "init", "waitack", "reset", "nopayload" and "done" as defined by SCXML elements `<state id="...">`. Note that the "init" state was previously defined as the initial state in the header in Section 5.1.2.1. The `<transition ...>` element specifies the name of the event that will cause a change in state in the attribute `event`, and the target state of the transition in the attribute `target`. For example, the"init" state will transition to the"waitack" state when the event named "payloadmessage" (arrival of "Payload message") is triggered. Note that `<transition ...>` elements that do not contain a target are self-transitions.

5.1.2.4 Actions

Lastly we describe how actions are specified in the executable specification. Actions are defined by the `<action:>` element. Actions delegate processing to our own code that we interfaced with the Apache Commons SCXML engine through the provided `CustomActions` interface. This is described in more detail in Sections 5.1.3.2 and 5.1.4.4. The `<action:>` element can be embedded within `<state>` elements or within `<transition>` elements as shown below.

```
<state id="...">
    <onentry>
        <action:...>
    </onentry>
</state>

<state id="...">
    <onexit>
        <action:...>
    </onexit>
</state>

<state id="...">
    <transition event="..." target="...">
        <action:...>
    </transition>
</state>
```

Actions placed within `<state>` element use enclosing `<onentry>` or `<onexit>` tags. If an action is enclosed using the `<onentry>` tag, it will be performed upon every entry to the state. If an action is enclosed using the `<onexit>` tag, it will be performed every time when the state is exited. If the action is embedded within a state transition, it is performed when that state transition is performed.

An action is specified in SCXML by first giving it a name corresponding to a method in the MessageStore, *i.e.*, the overlay network primitives. Then the parameters for that method are set.

```
    <action:create  name="name of method in list of actions"/>
    <action:set     name="name of method in list of actions"
                    field="XPath expression"
                    value="string value"/>
    <action:set     name="name of method in list of actions"
                    field="XPath expression"
                    value="XPath expression"/>
    <action:execute name="name of method in list of actions"/>
```

The details of the implementation of actions are discussed later in this chapter.

We now present some, not a complete set, of the actions specified in SCXML for the Hop-to-Hop Acknowledgment service.

Example Action 1: The following actions set a timer using the Java method `setTimer`:

```
<action:create name="setTimer"/>
<action:set name="setTimer" field="Timer/TimerIdentifier" value="2"/>
<action:set name="setTimer" field="Timer/TimerValue" value="5000"/>
<action:execute name="setTimer"/>
```

The above SCXML snippet creates an action (`<action:create...>`) named "setTimer" that sets (`<action:set...>`) the fields `TimerIdentifier` and `TimerValue` as "2" and 5000ms respectively. The action is then sent to the MessageStore to be executed when the (`<action:execute...>`) element is used. Note that in Hop-to-Hop Acknowledgement's specification, a timer identified by "2" correspond to a $\text{Timeout}_{\text{NACK}}$.

Example Action 2: The following SCXML snippet sends a "Local ACK Message".

```
<!-- send acknowledgement back -->
<action:create name="sendMessage"/>
<action:set name="sendMessage" field="Message"
    xpath="/HistoryOfEvents/Event[@name='payloadmessage'][last()]/Message"/>
<action:set name="sendMessage" field="Message/DeliveryMode" value="3"/>
<action:set name="sendMessage" field="Message/DestinationAddress"
    xpath="/HistoryOfEvents/Event[@name='payloadmessage'][last()]/Message/PreviousHopAddress"/>
<action:set name="sendMessage" field="Message/MessageType" value="2"/>
<action:execute name="sendMessage"/>
```

The "sendMessage" action corresponds to the method called `sendMessage` in the MessageStore. Once the action is created, parameters corresponding to the action are set. The parameter for the *sendMessage* method is a message contained in a DOM tree with root element `Message`. This message is first set (`<action:set...>`) as a copy of the last received "Payload message" in the History of Basic Events obtained by an XPath expression defined in the attribute `xpath` of the element `<action:set ...>`. Evaluation of the Xpath expression results in an XML snippet representing the last received "Payload message" embedded as a parameter in the action `sendMessage`.

The delivery mode, destination address, and message type of this message are then modified using `<action:set ...>`. The delivery mode is unicast. The destination address is changed to the previous-hop address of the last received “Payload message”. The message type is set to “2”, which corresponds to a “Local ACK message”.

Example Action 3: Similarly, below is an action to send a “NACK Message”.

```
<!-- send no-acknowledgement back -->
<action:create name="sendMessage"/>
<action:set name="sendMessage" field="Message"
    xpath="/HistoryOfEvents/Event[@name='payloadmessage'][last()]/Message"/>
<action:set name="sendMessage" field="Message/DeliveryMode" value="3"/>
<action:set name="sendMessage" field="Message/DestinationAddress"
    xpath="/HistoryOfEvents/Event[@name='payloadmessage'][last()]/Message/PreviousHopAddress"/>
<action:set name="sendMessage" field="Message/MessageType" value="1"/>
<action:set name="sendMessage" field="Message/Payload" value="No Acknowledgement"/>
<action:execute name="sendMessage"/>
```

Example Action 4: Finally, the action to retransmit the last received “Payload Message” to all children nodes (delivery mode of “1” indicates multicast) is shown below.

```
<!-- retransmit the payload to the children -->
<action:create name="sendMessage"/>
<action:set name="sendMessage" field="Message"
    xpath="/HistoryOfEvents/Event[@name='payloadmessage'][last()]/Message"/>
<action:set name="sendMessage" field="Message/DeliveryMode" value="1"/>
<action:set name="sendMessage" field="Message/HopLimit" value="1"/>
<action:execute name="sendMessage"/>
```

5.1.3 New MessageStore Objects

MessageStore has been modified to support two new objects that allow it to support processing of executable SCXML specifications. The first object, called `ExecSpec`, encapsulates the executable specification of a service. The second object, called `GenericFSM`, contains the finite-state machine that is executed by the Apache Commons SCXML engine. This object is generic since all services use instances of the same object. It is important to note the separation between the service specification (in object `ExecSpec`) and the code that the MessageStore needs to execute the service (in object `GenericFSM`). One `ExecSpec` object is created for each service and one `GenericFSM` object is created for each data message corresponding to that service (identified by the `<MessageID, ServiceID>` tuple.

5.1.3.1 ExecSpec Object

In MessageStore, we provide a hash-table called `serviceSpecifications` which is a repository for all loaded services in the MessageStore available to the application. Each entry in this hash-table is indexed by a unique `serviceID` and each entry contains an object representing the executable service specification (`ExecSpec` object).

To load a service, an executable specification in the form of a SCXML document is passed to the MessageStore method `initializePreDefinedService`. The SCXML document is checked against a locally stored XML schema, provided as a URI, using the method `validateSpecification` to ensure that the structure of the SCXML document is correct and all elements have the expected format.

If the SCXML document is valid the method parses the SCXML document using the Apache Commons SCXML library's `SCXMLParser` and creates an `ExecSpec` object. Conceptually, the `ExecSpec` object is the executable specification's object in MessageStore. The `ExecSpec` object contains the finite-state machine describing the behaviour of the service in the SCXML FSM object which can be executed using methods in the Apache

Commons SCXML engine. An `ExecSpec` object also extracts the meta-data, events, and actions defined in the SCXML document. As previously described, these data are defined and use by our implementation. For each service, an `ExecSpec` object is created and stored in the hash-table `serviceSpecifications`.

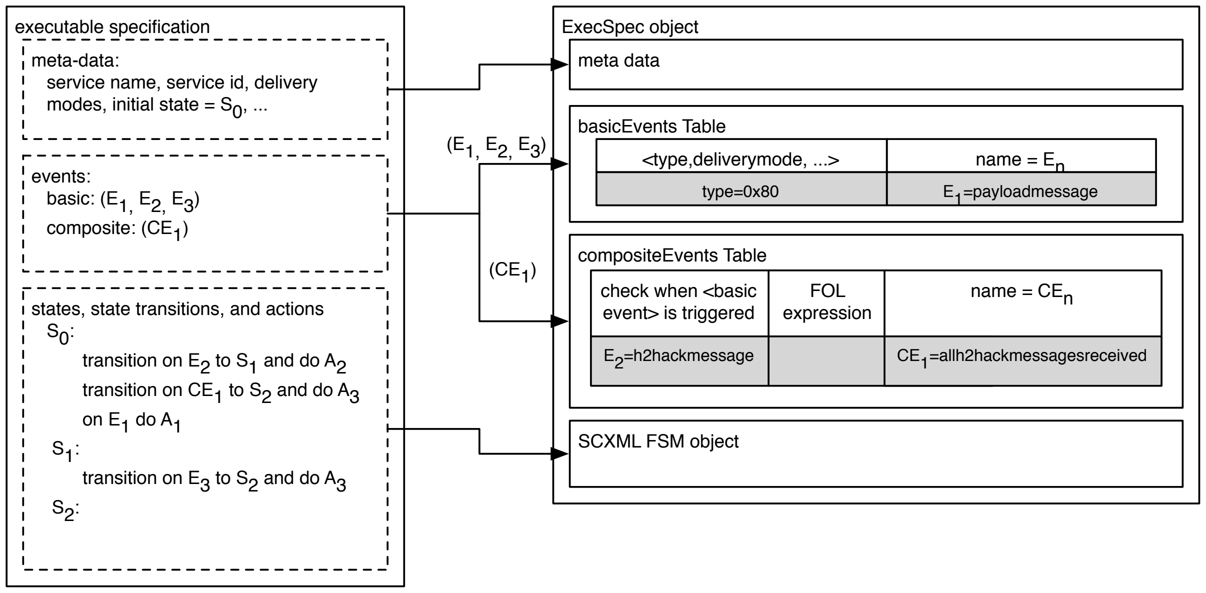

executable specification
meta-data:
service name, service id, delivery modes, initial state = S_0, ...
events:
basic: (E_1, E_2, E_3)
composite: (CE_1)
states, state transitions, and actions
S_0:
transition on E_2 to S_1 and do A_2
transition on CE_1 to S_2 and do A_3
on E_1 do A_1
S_1:
transition on E_3 to S_2 and do A_3
S_2:
(E_1, E_2, E_3)
(CE_1)
ExecSpec object
meta data
basicEvents Table
<type,deliverymode, ...>
name = E_n
type=0x80
E_1=payloadmessage
compositeEvents Table
check when <basic event> is triggered
FOL expression
name = CE_n
E_2=h2hackmessage
CE_1=allh2hackmessagesreceived
SCXML FSM object

Figure 5.6: Relationships of the SCXML representation of the executable specification to the ExecSpec object

Figure 5.6 describes the main data structures of the `ExecSpec` object. The meta data is stored in private member variables in the `ExecSpec` object. The list of events is stored in two tables, one for basic events and one fore composite events.

The data structure `basicEvents` is a hash-table that contains the list of basic events, consisting of the event names and the attributes of arrived message/expired timer that must match to trigger the event. For example, in the Hop-to-Hop Acknowledgment service, when a message arrives with message type equal to 0x80, the basic event named "payloadmessage" is triggered.

The data structure `compositeEvents` is a hash-table that contains the list of composite events identified by name, the names of basic events that will cause the composite events to be evaluated, and the first-order logic expression (FOL) that triggers the composite events. For the Hop-to-Hop Acknowledgement service, the composite event “allh2hackmessagesreceived” is checked every time a basic event “h2hackmessage” is triggered using the first-order logic expression defined in Section 5.1.2.2. If the first-order logic expression evaluates to true, the composite event is trigged.

The `ExecSpec` object contains an SCXML FSM object, created from the parsed SCXML document’s state and state transitions. This is the realization of the SCXML specification of states, state transitions, and actions in code. This object is used to create FSM instances by the `GenericFSM` object that can be executed by the Apache Commons SCXML engine as explained in the following section.

5.1.3.2 GenericFSM Object

Finite-state machine instances for a service in the new MessageStore are encapsulated by `GenericFSM` objects, which reference to an `ExecSpec` object.

When MessageStore receives a new message marked with a `ServiceID`, a `GenericFSM` object for that message can be created using the `ExecSpec`'s object obtained from the repository of all loaded services (the `serviceSpecifications` hash-table). The `createFSM` method instantiates a `GenericFSM` object of the corresponding service which takes the SCXML FSM object as a parameter. This `GenericFSM` object has access to all subcomponents of the corresponding `ExecSpec` object. The `GenericFSM` object also has access to the MessageStore and the overlay node.

The MessageStore creates and stores a new `GenericFSM` object for every data message received with a new `MessageID`. The `GenericFSM` object are stored in the MessageStore hash-table repository called `FSMStore` (This data structure existed previously but is now modified to store `GenericFSM` objects instead of service-specific FSM objects). The uniqueID of a GenericFSM object for a data message, used as the key in the hash-table, consists of both its `ServiceID` and `MessageID`.

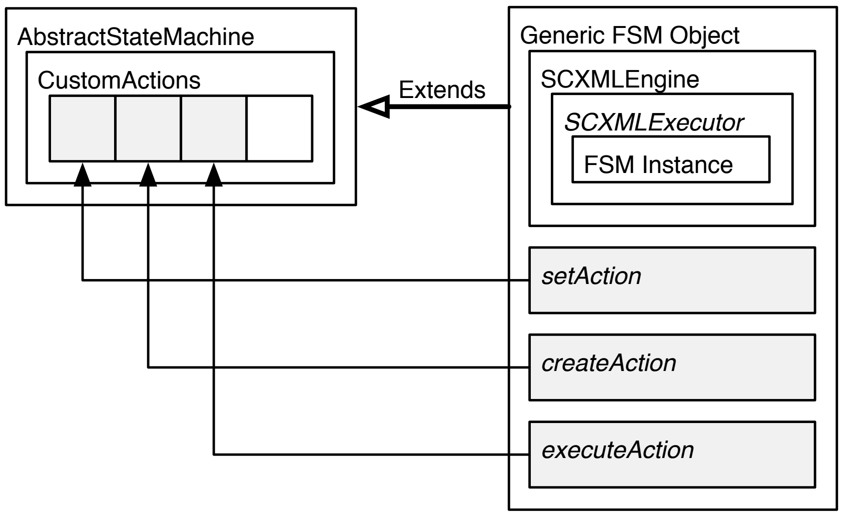


Figure 5.7: A GenericFSM object extends the AbstractStateMachine object and instantiate three actions

The `GenericFSM` object has many subcomponents. At its core, it contains a Apache Commons SCXML engine which has the methods to execute an instance of the SCXML FSM object contained in the `ExecSpec` object.

The `GenericFSM` object extends the Apache Commons SCXML `AbstractStateMachine` object. We provide three custom actions methods using the Apache Commons SCXML's custom action interface called `createAction`, `setAction`, and `executeAction` that enable the SCXML engine to process the elements`<action:create>`, `<action:set>`, and `<action:execute>` in the SCXML document of an executable specification. These methods are implemented in MessageStore and extend the Apache Commons SCXML's `Action` object. These three actions are instantiated upon the instantiation of the `GenericFSM` object. Figure 5.7 illustrates this.

Some other subcomponents in the `GenericFSM` object communicate with the Apache Commons SCXML engine. Their relationship can be see in Figure 5.8. The Event Handler is realized in method `handleEvent`, which receives events (messages and timers). It has access to both the `basicEvents` and `compositeEvents` tables to check if message arrivals and timer expirations should result in an event trigger in the FSM. The events to be triggered are passed to the Apache Commons SCXML engine. The Event Handler also stores each basic event that triggers in the `historyOfEvents` table. This table, representing the History of Basic Events, can be queried for information regarding past basic events.

Recall that an action is created and the parameters for that action are set by the SCXML engine using the `<action:create>` and `<action:set>` SCXML elements. For each action, an entry is created in the `ActionsQueue`. Each action has a name corresponding to a Java method in `MessageStoreActions`. The `ActionsQueue` buffers actions so they can be modified, *i.e.*, adding or modifying parameters queried from the History of Basic Events, anytime before executing them. This information is stored in the `ActionsQueue` until the SCXML element `<action:execute>` tag is evaluated by the

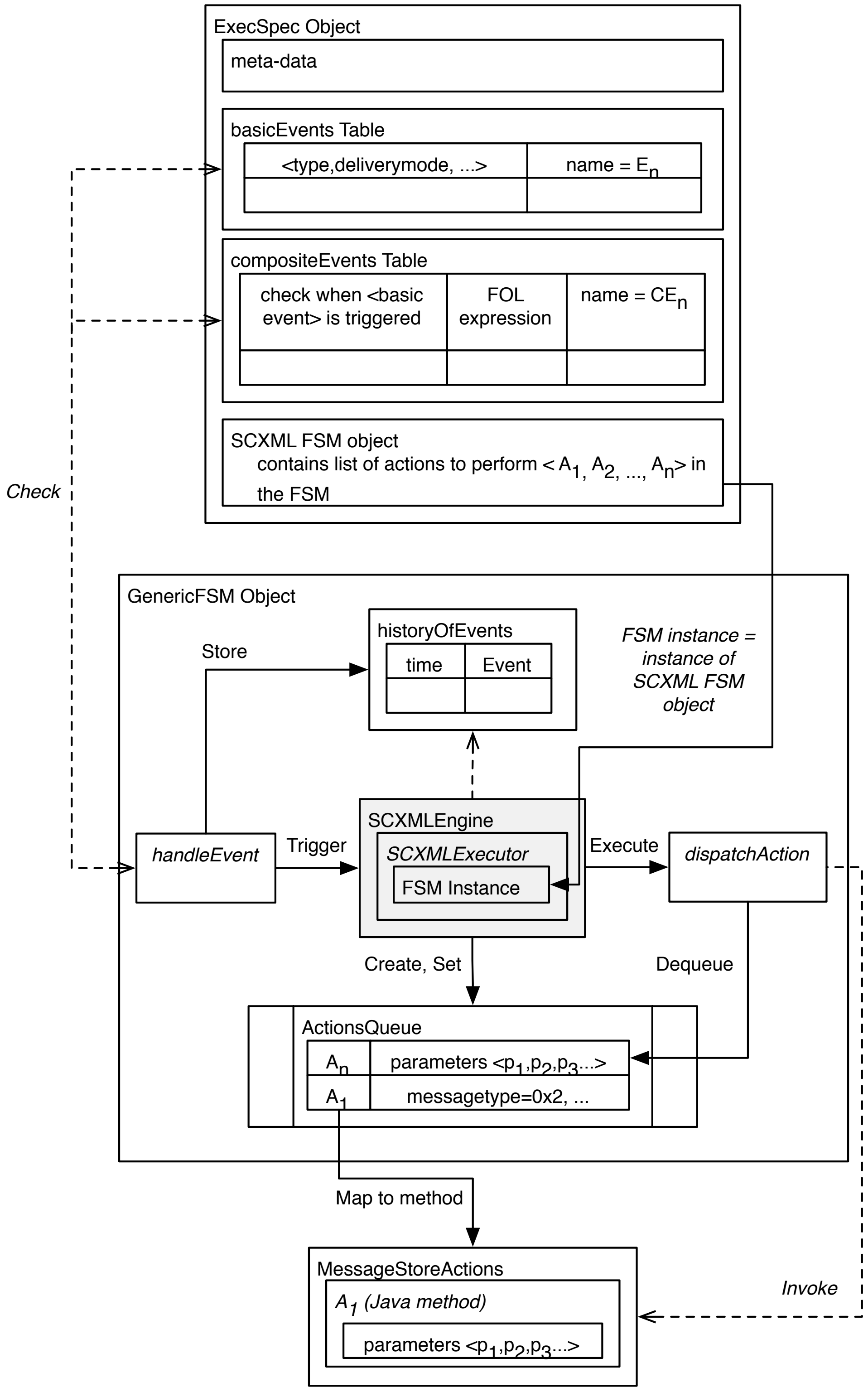


Figure 5.8: Component of GenericFSM object and interactions with ExecSpec object

Apache Commons SCXML engine. At that point, the processing of an action specified by its name in the SCXML element `<action:execute>` is passed to the Action Dispatcher.

The Action Dispatcher is realized in the method `dispatchAction`, which takes an action and instantiates the Java method in the `MessageStoreActions` outside of the `GenericFSMObject` and passing it all parameters of the action. The Java method then executes asynchronously to the `GenericFSMObject`.

5.1.4 Service Execution

In the next four sections, we discuss in detail the execution flow of services in MessageStore. In Section 5.1.4.1 and 5.1.4.2, we describe how messages arriving at the MessageStore and timer expirations in the MessageStore are passed as events to their corresponding finite-state machine instance. In Section 5.1.4.3, we describe how each `GenericFSM` object processes these events using the Event Handler. Lastly, in Section 5.1.4.4, we describe how actions are invoked by the Action Dispatcher.

5.1.4.1 Message Arrival Event

Figure 5.9 illustrates the steps involved in processing an arrived message at the MessageStore. The following steps occur:

1. The message is received at the MessageStore by the method `receiveMessage` (bottom of Figure 5.9). The message is checked to determine if it is a data message containing payload or a control message. The message is then written to the MessageStore FIFO queues `DataBuffer` (for data messages) or `ControlBuffer` (for control message).

2. Within the MessageStore, there is a MessageProcessor thread that reads from the queues `DataBuffer` and `ControlBuffer` when they have messages available and dequeues them to be processed.

3. In the MessageProcessor, messages are handled by the `processMessage` method. The `ServiceID` of the message is extracted and checked against the list of supported services using the `serviceSpecifications` hash-table. If the service does not exist in MessageStore, a request for the service's executable specification is sent, and the processing of the message is postponed until the specification arrives. If no service specification is found within an application specified time, the message is dropped. This will be discussed in detail in Section 5.2.

If the service is found, the `MessageID` of the message is extracted and checked against the MessageStore repository of stored `GenericFSM` objects. Recall that the `<MessageID, ServiceID>` tuple uniquely identifies a GenericFSM object for a data message.

If the repository does not contain a `GenericFSM` object with this uniqueID, it means the message is new. Then, a new `GenericFSM` object is created using the `createFSM` method and the Apache Commons SCXML engine is started for that instance. If the repository already contains the `GenericFSM` object, the reference to the existing `GenericFSM` object is retrieved using the `getFSM` method from the MessageStore repository.

4. The `processMessage` method also invoked the `createEvent` method which marshall the complete message into an `Event` object. The event object contains the message, as a XML DOM tree. For example, a data message of the Hop-to-Hop Acknowledgment service, after marshalling, will give the following XML document.

```
<Event>
    <Message>
        <MessageIdentifier>1</MessageIdentifier>
        <ServiceIdentifier>H2HACK</ServcieIdentifier>
        <DeliveryMode>1</DeliveryMode>
        <HopLimit>255</HopLimit>
        <SourceAddress>100,100</SourceAddress>
        <DestinationAddress>200,200</DestinationAddress>
        <PreviousHopAddress>100,100</PreviousHopAddress>
        <MessageType>128</MessageType>
        <Payload>Hello World</Payload>
        <RootAddress>100,100</RootAddress>
    </Message>
</Event>
```

Note that the `RootAddress` is a HyperCast message attribute and is not used in the finite-state machine, nevertheless, it is still an attribute of the message. The `Event` object is then passed to the `GenericFSM` object to be processed by method `handleEvent` as shown in Figure 5.8.

5. The `GenericFSM` object is stored (in case of a new uniqueID) or updated (in case

of an existing uniqueID) in the finite-state machine repository (`FSMStore`) of MessageStore.

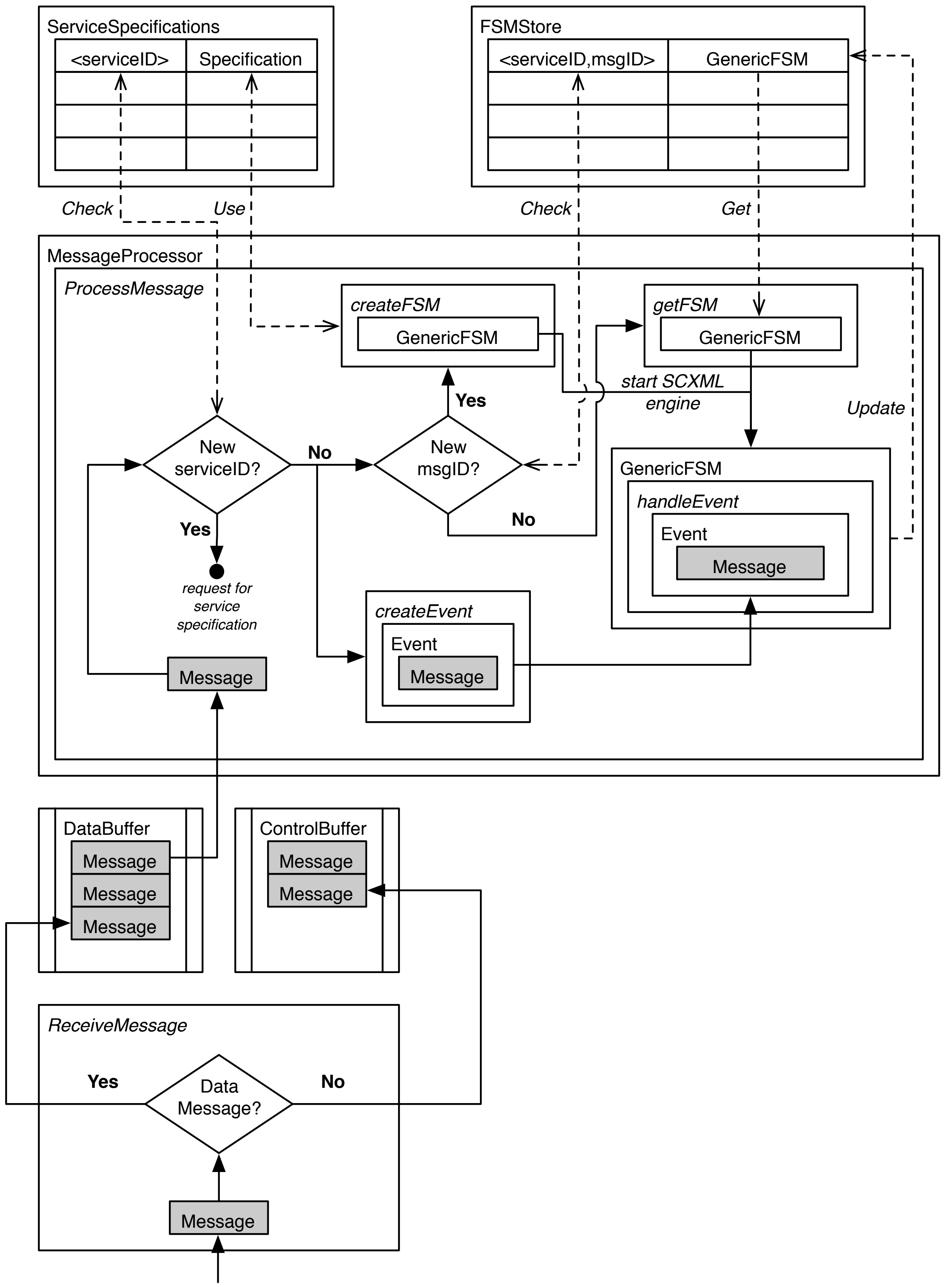


Figure 5.9: Processing of a message arrival event in MessageStore

5.1.4.2 Timer Expiration Event

Figure 5.10 illustrates the steps involved when a timer expires in MessageStore. The following steps occur:

1. When a timer expires, the overlay node gets notified by the `timerExpired` method. The method delivers the timer to the MessageStore (if MessageStore is instantiated) if the timer a MessageStore timer.

2. The MessageStore method `timerExpired` extracts the timer object reference to the `GenericFSM` object that issued the timer and attempts to obtain that `GenericFSM` object from the MessageStore finite-state machine repository `FSMStore`. If that `GenericFSM` object is not found, the timer expiration is ignored. If a `GenericFSM` object exists for the timer, a reference is obtained using the method `getFSM`.

3. The `timerExpired` method invokes the `createEvent` method to marshall the timer into an `Event` object which contains the timer as a XML DOM tree. For example, a Timeout$_{\text{ACK}}$ of the Hop-to-Hop Acknowledgment service will give the following XML document after marshalling.

   ```
   <Event>
       <Timer>
           <TimerIdentifier>3</TimerIdentifier>
       </Timer>
   </Event>
   ```

   The `Event` object is then passed to the `GenericFSM` object to be processed by the `handleEvent` method.

4. The `GenericFSM` object is updated and stored in the finite-state machine repository (`FSMStore`).

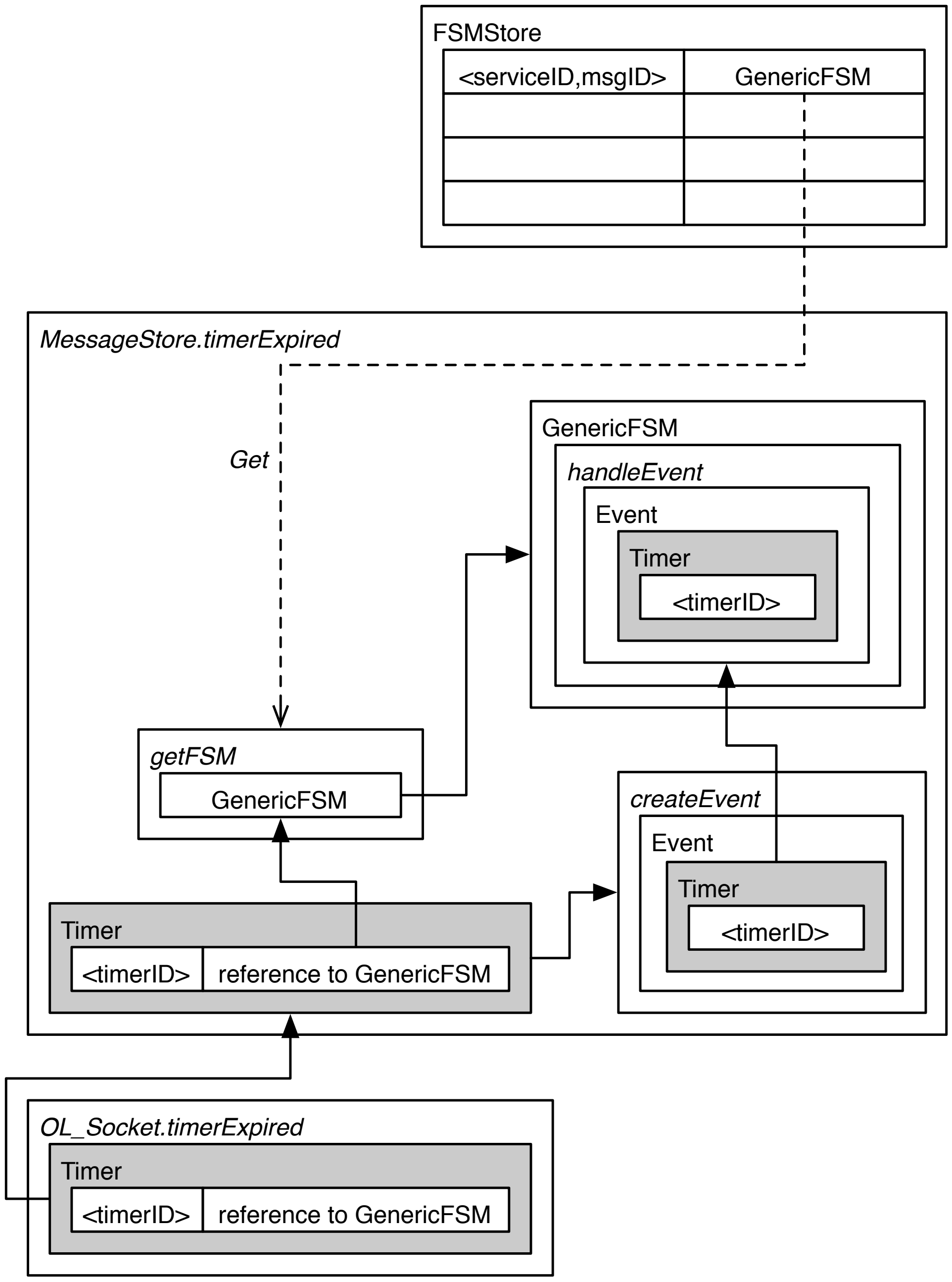


Figure 5.10: Processing of a timer expiration event in MessageStore

5.1.4.3 Handling Events

Figure 5.11 illustrates the steps taken by the `handleEvent` method within a `GenericFSM` object to process an event (`Event` object). The following steps occur:

1. The `Event` object is received by the `handleEvent` method of a `GenericFSM` object.

2. The `handleEvent` method checks the list of basic events ($E_1, E_2, ...$) of this service using its reference to the `basicEvents` table in its executable specification (`ExecSpec` object) to see if one or more basic events need to be triggered in the Apache Commons SCXML engine. If no basic event needs to be triggered, the event object is released. If one or more basic event are to be triggered, the names of the triggered events are obtained from the `basicEvents` table and are passed to the `fireEvent` method of the Apache Commons SCXML engine which adds the event's names names ($E_1, E_2, ...$) to the engine's event queue. The engine then triggers these events on the state machine. When the basic events are triggered, the event object that caused these basic events is saved in the `historyOfEvents` table with a timestamp.

3. The `handleEvent` method then checks the `compositeEvents` table in its executable specification object to see if there are any composite events ($CE_1, CE_2, ...$) that must be evaluated due to the already triggered basic events names ($E_1, E_2, ...$) resulting from the event object. If one or more composite events must be evaluated, the first-order logic XQuery expressions for those composite events are obtained from the `compositeEvents` table and evaluated using the `executeFOL` method from the Saxon XQuery and XLST processor libraries. These libraries were loaded upon instantiation of the MessageStore. If the first-order logic XQuery expression evaluates to true, the composite event names associated with it ($CE_1, CE_2, ...$) are passed to the `fireEvent` method of the Apache Commons SCXML engine which adds their names ($CE_1, CE_2, ...$) to the engine's event queue. Note that even if composite

events were triggered, the event object is not saved in the `historyOfEvents` table. Conceptually speaking, composite events are always composed of basic events. Therefore, if the entire history of basic events is logged, any composite event can be reconstituted by querying the `historyofEvents` table.

4. The SCXML engine processes the event names ($E_1, E_2, ...$) and ($CE_1, CE_2, ...$) and issues state transitions or actions using the `SCXMLExecutor`.

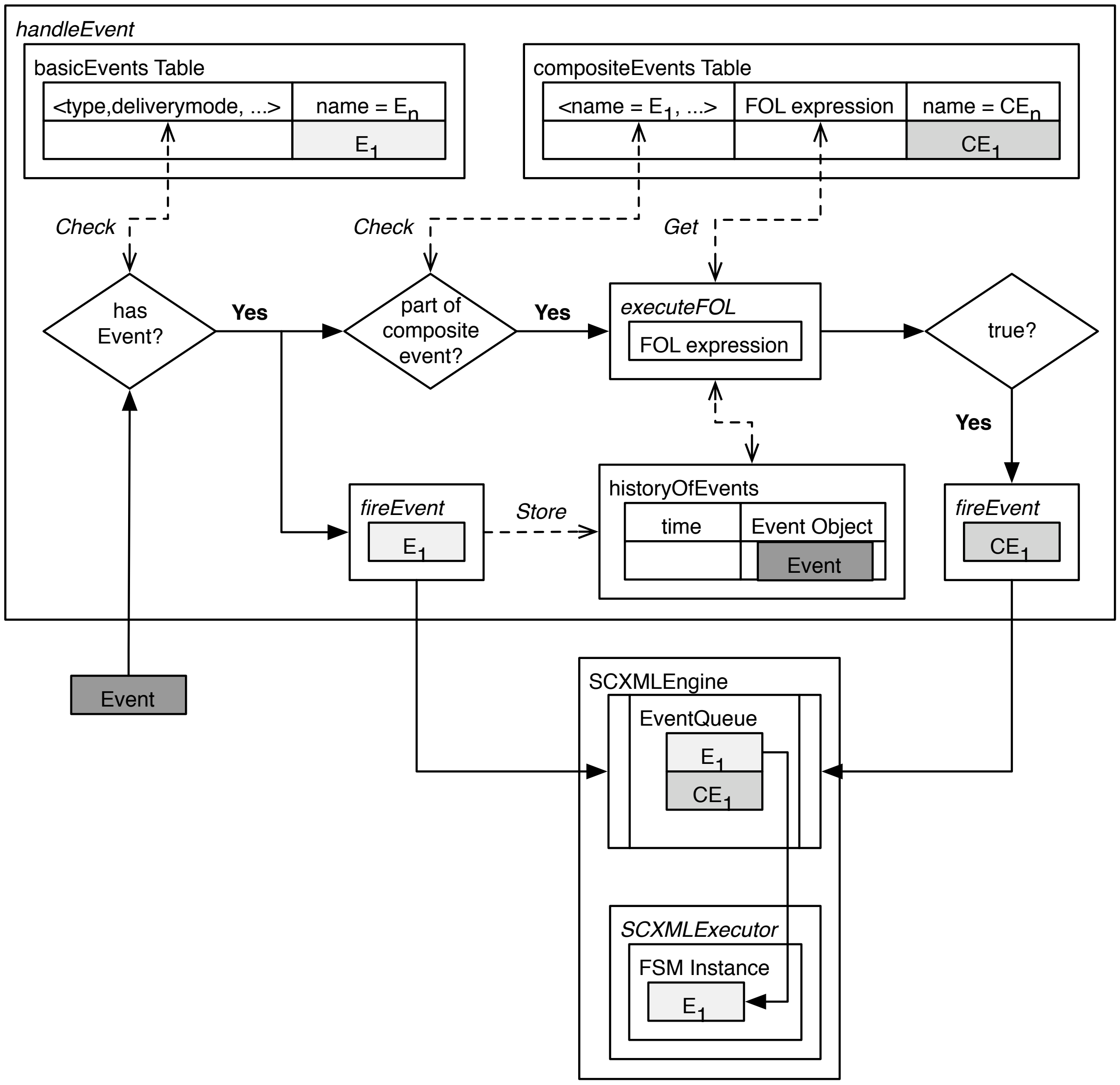


Figure 5.11: Event processing by the handleEvent method in GenericFSM Object

5.1.4.4 Executing Actions

Figure 5.12 illustrates the steps taken to create, set, and execute an action by the `GenericFSM` object. Recall that the actions are defined in XML snippets embedded in the executable specification stored as an SCXML FSM object in the `ExecSpec` object. The SCXML FSM object is passed as a reference to the `GenericFSM` object upon instantiation the `GenericFSM` object. The actions are mapped to pre-defined methods in MessageStore. The list of pre-defined methods are provided by the class `MessageStoreActions`. All methods in the `MessageStoreActions` class expect an XML document as parameter. This class is described in detail in Section 5.1.5.

The following steps occur in executing an action.

1. As previously described, the Apache Commons SCXML engine is aware of the types actions that need to be processed from SCXML document describing the executable specification. These are SCXML elements of the format `<action ...>`. As soon as one of these SCXML elements needs to be performed, the engine delegates processing to the custom actions interface `CustomActions`. We have implemented three methods in the `GenericFSM` object, `createAction`, `setAction`, and `executeAction` which handle the three SCXML elements `<action:create>`, `<action:set>`, and `<action:execute>` respectively. This was described in Section 5.1.3.2.

2. To execute an action, an action must be created (shown as Step 1 in Figure 5.12). The `<action:create ...>` element invokes the `createAction` method of the `GenericFSM` object. This method creates a new entry, in the form of an XML DOM tree, in the `ActionsQueue` table with the name of the action, A_n as the attribute of the root element. The name of the action must correspond to a method in the class `MessageStoreActions`.

3. Next the parameters for this action must be set (shown as Step 2 in Figure 5.12).

The `<action:set ...>` element invokes the `setAction` method of the `GenericFSM` object. This method updates the action named A_n with the list of parameters $(p_1, p_2, ...)$ in the `ActionsQueue` table. The parameters are added to the root element A_n as children nodes in the XML DOM tree. These parameters corresponds to input parameters for the method called A_n in the `MessageStoreActions` class. The XML DOM tree with the name of the action as the root element and parameters as children represents the action.

4. Finally, the action A_n is executed with the `<action:execute ...>` elements (shown as Step 3 in Figure 5.12) which passes control to the `executeAction` method of the `GenericFSM` object. The method dequeues the action with the root element attribute of A_n from the `ActionsQueue` and sends the action (XML DOM tree) to the `dispatchAction` method. The action is also stored, with a timestamp, in the `historyOfActions` table.

5. The `dispatchAction` method receives the action (XML DOM tree) extract from the action the method name A_n. Using Java reflection, `disptachAction` instantiates and invokes the method in the `MessageStoreAction` class with the method name equal to A_n and sends the remaining XML DOM tree as the parameter. If the method name A_n is not found, an exception is thrown and the action is discarded. This is shown in Figure 5.13.

6. The method A_n in `MessageStoreActions` extracts the input parameters in the form of an XML DOM tree and then proceeds to execute the action if the parameters are correct. The execution of the action is asynchronous to the execution of the `GenericFSM` object. Hence, the `GenericFSM` object does not know if this action succeeds or fails. This is also shown in Figure 5.13.

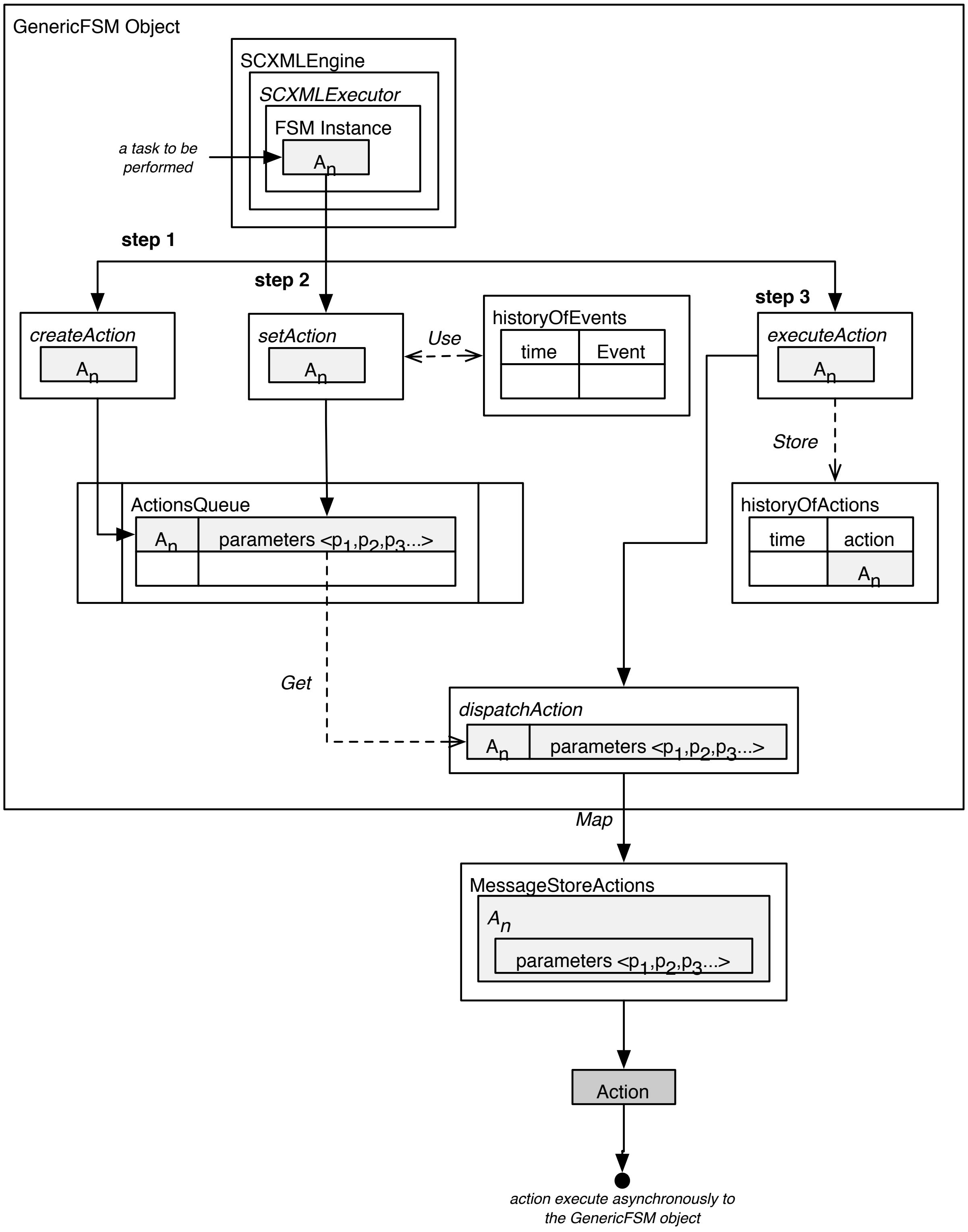


Figure 5.12: Creating an action in GenericFSM object using SCXML custom actions interface

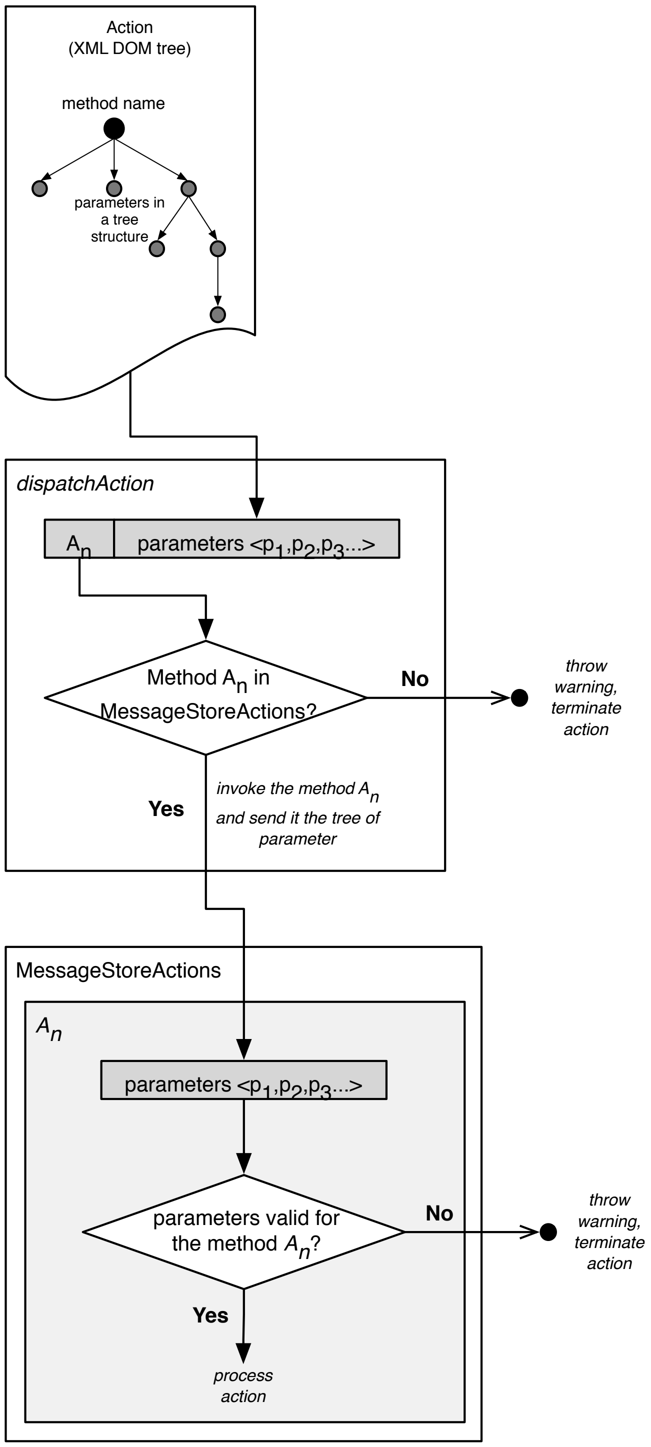

Action
(XML DOM tree)
method name
parameters in a tree structure
dispatchAction
A_n
parameters <p_1,p_2,p_3...>
Method A_n in MessageStoreActions?
No
throw warning, terminate action
Yes
invoke the method A_n and send it the tree of parameter
MessageStoreActions
A_n
parameters <p_1,p_2,p_3...>
parameters valid for the method A_n?
No
throw warning, terminate action
Yes
process action

Figure 5.13: Action processing by the dispatchAction method in GenericFSM Object

We again use Hop-to-Hop Acknowledgment example to show how an action, such as sending an acknowledgement message, is issued and processed. Recall from Section 5.1.2.4 that the following SCXML snippets create an action that sends a “Local ACK message” upon receipt of a “Payload message”.

```
<!-- send acknowledgement back -->
<action:create name="sendMessage"/>
<action:set name="sendMessage" field="Message"
    xpath="/HistoryOfEvents/Event[@name='payloadmessage'][last()]/Message"/>
<action:set name="sendMessage" field="Message/DeliveryMode" value="3"/>
<action:set name="sendMessage" field="Message/DestinationAddress"
    xpath="/HistoryOfEvents/Event[@name='payloadmessage'][last()]/Message/PreviousHopAddress"/>
<action:set name="sendMessage" field="Message/MessageType" value="2"/>
<action:execute name="sendMessage"/>
```

The `GenericFSM` object then takes the following step:

1. An action is first created.

   ```
   <action:create name="sendMessage"/>
   ```

   Creates an action with the root element of the XML DOM tree A_n = “sendMessage” and enqueues it on `ActionsQueue`. The name of the action corresponds to the method `sendMessage` in the `MessageStoreActions` class. At this point the `ActionsQueue` would have the following action. Note that the action does not yet have any parameters.

   ```
   <Action name="sendMessage">
   </Action>
   ```

2. The parameters for the action is set.

   ```
   <action:set name="sendMessage" field="Message"
      xpath="/HistoryOfEvents/Event[@name='payloadmessage'][last()]/Message"/>
   <action:set name="sendMessage" field="Message/DeliveryMode" value="3"/>
   <action:set name="sendMessage" field="Message/DestinationAddress"
      xpath="/HistoryOfEvents/Event[@name='payloadmessage'][last()]/Message/PreviousHopAddress"/>
   <action:set name="sendMessage" field="Message/MessageType" value="2"/>
   ```

   The first `<action:set ...>` element copies the XML DOM tree of the last received “Payload message” appends it to the newly created XML DOM tree of the action "sendMessage" under the child element `<Message>` assigned by the `field` attribute. The `setAction` method does this by taking the XPath expression in the field `xpath` and performing an XPath query of the `historyOfEvents` table searching for the last "Payload message" event. The result of the query is the XML DOM tree representing the last received "Payload message". Similarily, the next two `<action:set ...>` elements modify the delivery mode, destination address, and message type nodes of the XML DOM tree of the action "sendMessage" to have a delivery mode value of "3", destination address logical address equal to the previous-hop address, and a message type of "2". After these manipulations the `ActionsQueue` would have the following action.

```
<Action name="sendMessage">
    <Message>
        <MessageIdentifier>1</MessageIdentifier>
        <ServiceIdentifier>H2HACK</ServiceIdentifier>
        <DeliveryMode>3</DeliveryMode>
        <HopLimit>255</HopLimit>
        <SourceAddress>200,200</SourceAddress>
        <DestinationAddress>100,100</DestinationAddress>
        <PreviousHopAddress>200,200</PreviousHopAddress>
        <MessageType>2</MessageType>
        <RootAddress>100,100</RootAddress>
    </Message>
</Action>
```

The XML DOM tree representing the action "sendMessage" now contain a message as its children enclosed within the `<Message>` element. This representing a parameter for the "sendMessage" action that is a "Local ACK message".

3. The action is then executed.

```
<action:execute name="sendMessage"/>
```

Finally, the action A_n="sendMessage" is executed with the `<action:execute ...>` elements which passes control to the `executeAction` method in the `GenericFSM` object. The method finds the action with the name "sendMessage" using an XPath query and dequeues the action from the `ActionsQueue` table. The action "sendMessage" is then stored in the `historyOfActions` table. The action is then sent to the `dispatchAction` method.

4. The `dispatchAction` method extracts the action name "sendMessage", and using Java reflection, instantiates and invokes the `sendMessage` method in the `MessageStoreAction` class. The XML DOM tree of the "Local ACK message", enclosed in the`<Message>` element, is sent to `sendMessage` as the parameter.

5. The `sendMessage` method parses the XML DOM tree representing the "Local ACK message" creating a "Local ACK message" object and execute the action.

5.1.5 List of Supported Actions - Network Primitives

The set of actions provided by the class `MessageStoreActions` in MessageStore a set of common overlay network primitives. This set of primitives are the only methods any service may invoke. We believe we have equipped the `MessageStoreActions` primitives that are useful for many services, and which perform tasks easily understood by developers. In particular, all existing services of HyperCast 4.0 can be expressed. If new primitives are required for a service, they should be added to this `MessageStoreActions` class.

Table 5.2 lists the actions currently available in `MessageStoreActions`. All the parameter for these method is represented as a XML DOM tree described previously.

This list of supported actions corresponds to those defined Section 4.2.3.1. Note that the two message manipulation primitives $createMessage$ and $sendMessage$ are not present. These actions are provided by our custom action `createAction` and `setAction` using the SCXML element `<action:create...>` and `<action:set...>`. For example, to create a message, an action is created and the message parameters are set using `setAction`. Similarly, to modify a message, the message is first obtained using XPath query on the `historyOfEvents` and any fields can be changed using `setAction`.

Table 5.2: Supported Actions in MessageStore

Action	Description	Parameters
`sendDataMessage`	Sends a data message to the list of destination nodes with the specified delivery mode (unicast, multicast, ood). The invoking node sets its own logical address as the source logical address and previous-hop logical address. This method automatically adds the service's `ServiceID` and a `MessageID`. The message type is 0x80.	data message
`sendControlMessage`	Sends a control message to the destination node(s) with the specified delivery mode (unicast, multicast, ood). The invoking node sets its own logical address as the source logical address and previous-hop logical address. This method automatically adds the service's `ServiceID`. The `MessageID` correspond to a data message.	control message
`toApplication`	Passes a message to the application.	data message
`setTimer`	Sets a timer in the MessageStore. This method automatically sets the `ServiceID` and a reference in the timer to `GenericFSM` instance that created it.	MessageStore timer object (TimerID, duration in milliseconds)
`terminate`	Removes a `GenericFSM` instance from the MessageStore repository `FSMStore`.	uniqueID of the `GenericFSM` instance `<MessageID, ServiceID>` tuple
`updateNodeInfo`	Updates the local node information and stores the information in the `historyOfEvents` under the `<Node>` elements. Node information includes the local nodes logical address and other topology information (parents, children, neighbours, etc.).	-

5.1.6 Discussion of the Implementation

We designed the `GenericFSM` object to only process data expressed as XML DOM trees rather than Java objects. Events and actions are all in the form of XML DOM trees. Conceptually this means the `GenericFSM` object is a XML processor. An advantage of this approach is that the objects are not coupled with a specific implementation of the overlay middleware, *i.e.*, a message object is described XML rather than instantiated from an overlay specific Java class. Hence, the core component of *MessageStore* can be interoperable with different overlay networks.

Also, all the History of Basic Events repository stores Events expressed as XML DOM trees. Hence, this repository can be easily moved to a database or multiple parallel databases if needed.

5.1.7 Mechanisms to Improve MessageStore Performance

The new design MessageStore must instantiate two large libraries for each `GenericFSM` object: 1) Apache Commons SCXML engine and 2) Saxon XQuery and XLST processor (for processing first-order logic expressions). Since Java loads class libraries on-demand, performance degrades when these libraries are loaded. To mitigate this performance degradation we provide two mechanisms. First, we allow the pre-allocation of `GenericFSM` objects during the initialization of MessageStore. Second, we provide a mechanism to "warm-start" the Apache Commons SCXML engine to force loading of a portion of the required libraries. We do not "warm-start" the SAxon XQuery and XLST processor because involves the use of composite events.

5.1.7.1 Pre-allocation of Finite-State Machines

Figure 5.14 shows the pre-allocation of FSM instances during MessageStore initialization. The `<Preallocated>` parameter in the configuration file for the HyperCast overlay can be used by the application to set the number `GenericFSM` objects to be pre-allocated. If the value is not set or if the value is set to 0, no FSMs are pre-allocated. The pre-allocated `GenericFSM` objects are stored in the `preallocatedFSM` collection. Note that these pre-allocated FSMs are not started and they have no reference to any executable specification, *i.e.*, no reference to an `ExecSpec` object. Hence, the pre-allocated object can be used by any service.

When a new message is received by the MessageStore that requires a new instance of the `GenericFSM` object, the `processMessage` method first checks the pool of pre-allocated `GenericFSM` objects (stored in the collection `preallocatedFSM`) to see if there are any remaining. If there are pre-allocated `GenericFSM` objects available, one of the object is obtained. The `ExecSpec` object is identified by the `ServiceID` of the message and the Apache Commons SCXML engine is then started for the `GenericFSM` object.

This processing flow is shown in Figure 5.15.

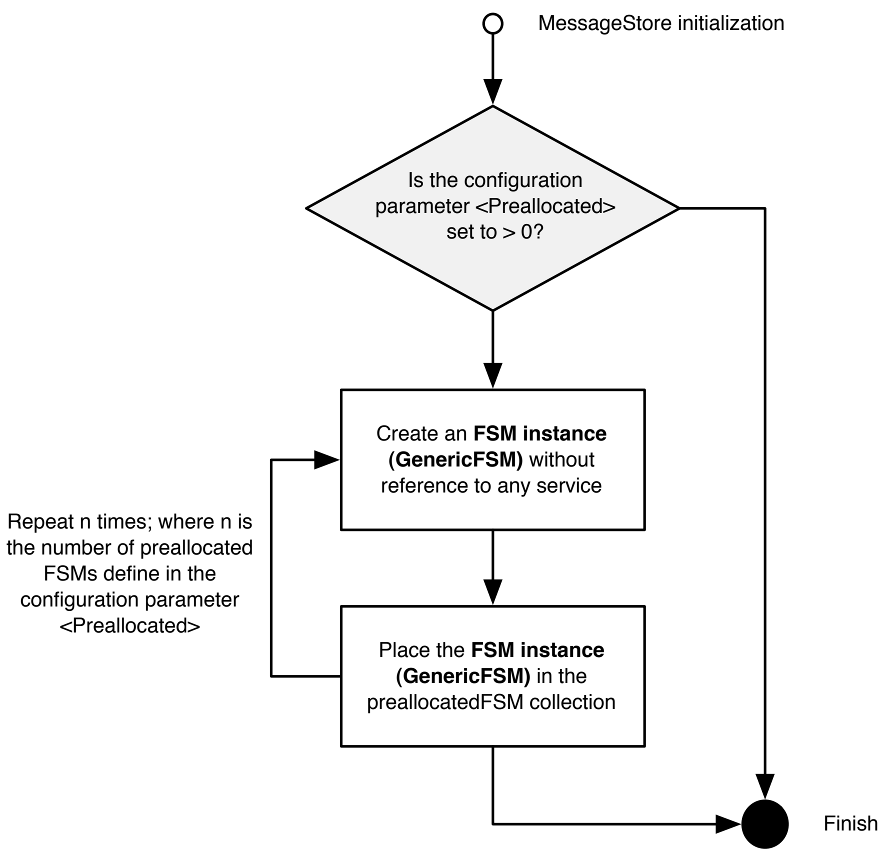

MessageStore initialization
Is the configuration parameter <Preallocated> set to > 0?
Create an FSM instance (GenericFSM) without reference to any service
Place the FSM instance (GenericFSM) in the preallocatedFSM collection
Repeat n times; where n is the number of preallocated FSMs define in the configuration parameter <Preallocated>
Finish

Figure 5.14: Pre-allocation of GenericFSM objects during MessageStore initialization

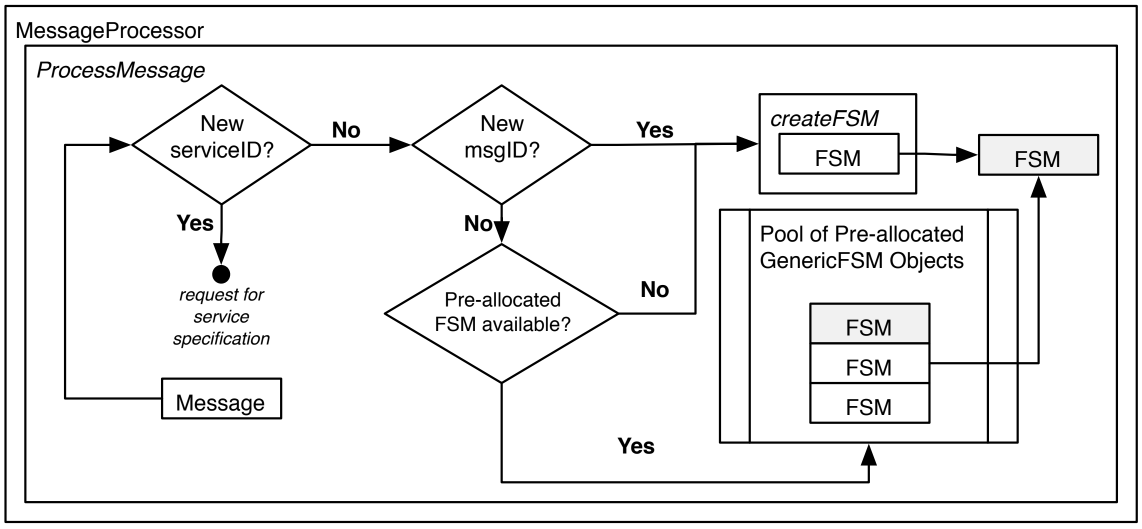

MessageProcessor
ProcessMessage
New serviceID?
No
Yes
request for service specification
Message
New msgID?
Yes
No
Pre-allocated FSM available?
No
Yes
createFSM
FSM
FSM
Pool of Pre-allocated GenericFSM Objects
FSM
FSM
FSM

Figure 5.15: How preallocated FSMs are used in MessageStore

In Chapter 6, the performance improvement due to this pre-allocated FSMs is evaluated. With a pre-allocation of 100 FSMs for a 1-hop transfer of 10,000 overlay messages of 1,024 bytes using the Hop-to-Hop Acknowledgement service, the delay of the first few messages processed by MessageStore was reduced by up to 100 milliseconds per message.

5.1.7.2 “Warm-starting” the Apache Commons SCXML Engine

We use the term “warm-start” to describe our approach to force some, but not all, required classes of the Apache Commons SCXML libraries to be loaded during the initialization of MessageStore prior to processing any messages. The Java Classloader, by default, uses on-demand loading of class libraries. This means that classes are not loaded into the Java Virtual Machine until they are actually used, even if the classes are instantiated. Since the Apache Commons SCXML is a set of large libraries, the on-demand loading degrades the performance of message processing in the MessageStore for the first few received messages. For the first few messages, the Java Classloader must load all classes required to process the message into the Java Virtual Machine before processing it. Our mechanism does not force all existing Apache Commons SCXML libraries to be loaded, since this would reduce available memory in the Java Virtual Machine. The “warm-start” mechanism attempts to load a core set of classes for the Apache Commons SCXML libraries needed to process the simplest finite-state machine within our context.

Figure 5.16 shows the “warm-start” of Apache Commons SCXML engine during MessageStore initialization.

The “warm-start” mechanism uses a SCXML document for a dummy service that is loaded into the MessageStore. The dummy service has a `ServiceID` of 0. The finite-state machine of the service has one state where it receives any message with its corresponding `ServiceID`. There are no actions or state transitions for the dummy service. The “warm-start” mechanism is used when the `<WarmStart>` parameter in the HyperCast configuration file is set to “true”. Upon MessageStore initialization, the parameter executable specification (`ExecSpec` object) is created from the SCXML document of the dummy service. Then 10 `GenericFSM` objects are instantiated and the Apache Commons SCXML engine is started for each `GenericFSM` object. Note that this is a forced instantiation. Under normal circumstances, the `GenericFSM` object is only instantiated if a message belonging to that service is received. We do not send or any receive messages,

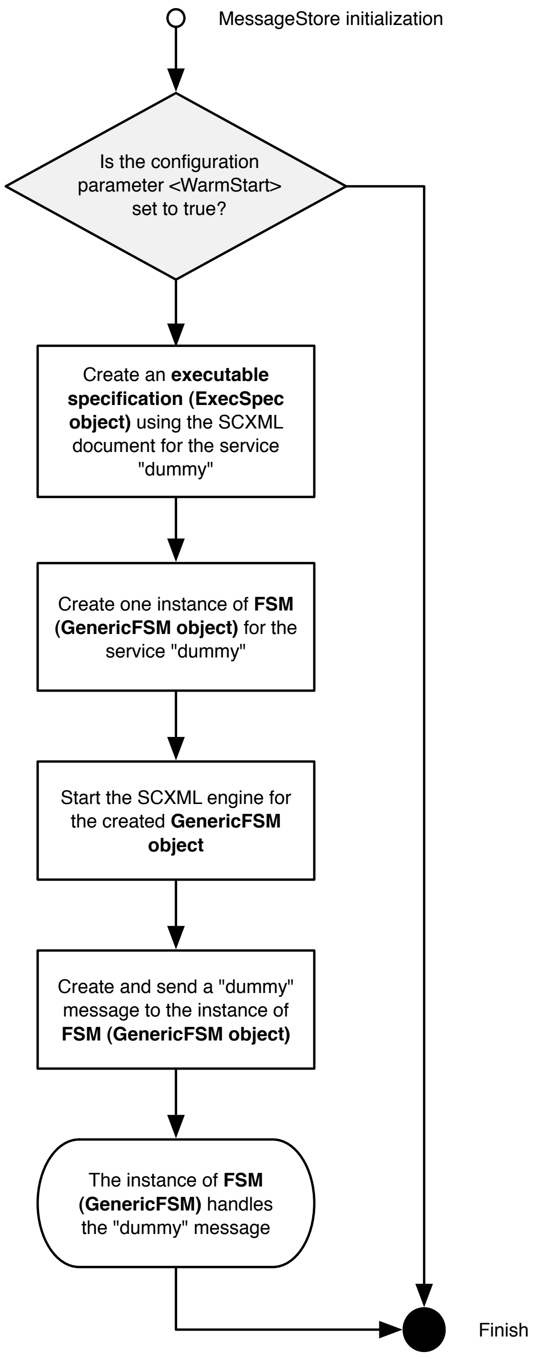

MessageStore initialization
Is the configuration parameter <WarmStart> set to true?
Create an **executable specification (ExecSpec object)** using the SCXML document for the service "dummy"
Create one instance of **FSM (GenericFSM object)** for the service "dummy"
Start the SCXML engine for the created **GenericFSM object**
Create and send a "dummy" message to the instance of **FSM (GenericFSM object)**
The instance of **FSM (GenericFSM)** handles the "dummy" message
Finish

Figure 5.16: Warm-start mechanism during MessageStore initialization

instead we create an event object representing a dummy message with a payload of 1,024 bytes and passes this event object directly to the `handleEvent` method of the `GenericFSM` object. The event is then processed by the Apache Commons SCXML engine.

In Chapter 6 we show the primary performance bottleneck is the processing within the Apache Commons SCXML engine. The “warm-start” mechanism can reduce the delay of the first few messages processed by MessageStore by few hundred milliseconds.

5.2 Service Deployment

There are two mechanism to deploy a new service to MessageStore. The first mechanism is to invoke the MessageStore API called `loadService` which takes a URI for an executable specification and deploys it in the MessageStore. The second mechanism involves the use of a *Services Server*. We do not send executable specifications directly as messages in the overlay since the SCXML documents can be extremely large.

5.2.1 Services Server

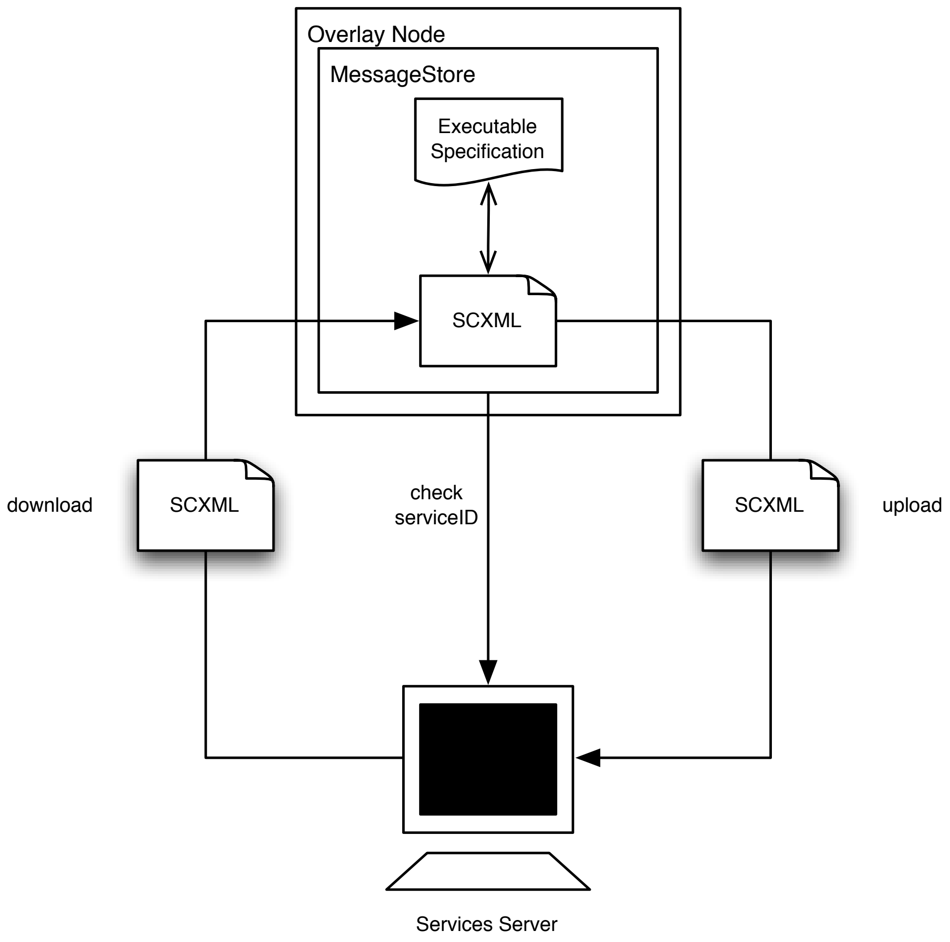


Figure 5.17: Interactions of MessageStore with the Services Server

If HyperCast is running in an environment where it has access to the Internet, the overlay nodes can interact with a special HTTP server called the "Services Server". This server is the same type of server that is available in HyperCast for overlay socket configurations. The server stores executable specifications identified by the `ServiceID` in SCXML document form. The interactions of MessageStore with the Services Server are shown in Figure 5.17. The location of the Services Server is configurable in the HyperCast configuration file. Executable specification can be loaded using the method `loadService` by the method with a URI to the SCXML document. Alternatively, HyperCast provides methods to interact with the Services Server and download and upload SCXML documents for services on-demand.

HyperCast contains the implementation of this Services Server. This server is a minimal implementation of an HTTP server and is run as a standalone application. The Services Server is accessible through a TCP port. This port number is configured in the HyperCast configuration file.

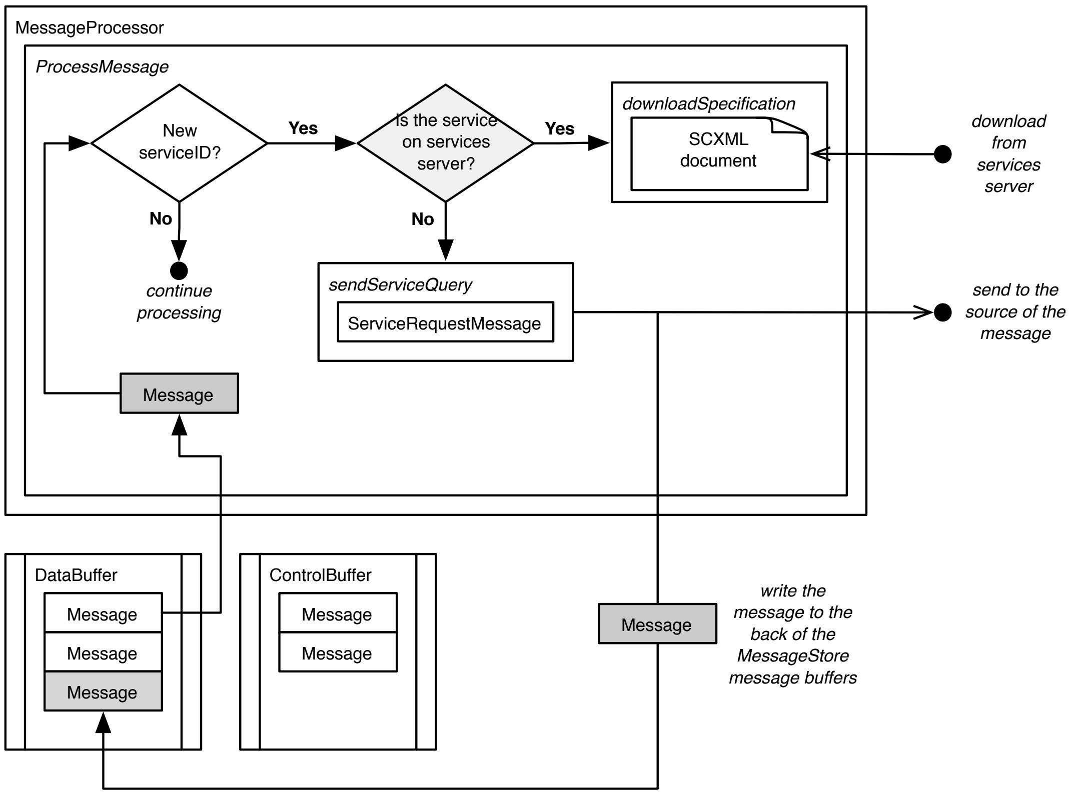


Figure 5.18: How MessageStore handles a message with a new serviceID

Next we describe how an overlay node without a specific service can obtain the executable specification from the Services Server. Figure 5.18 presents an illustration. When a message arrives at the MessageStore processing thread `processMessage` its `ServiceID` is extracted and checked. If MessageStore currently does not have the executable specification for that service, MessageStore attempts to contact the Services Server if its is configured in the HyperCast configuration XML file. If the Services Server cannot be contacted or doesn't exist, the message is dropped. If the Services Server is contacted successfully, MessageStore checks the server for an executable specification with he required `ServiceID`. If found, the MessageStore downloads the SCXML document using the method `downloadSpecification`. An `ExecSpec` object is then created from this SCXML document and the message is processed.

If the Services Server is contacted successfully but the `ServiceID` is not found, the MessageStore will attempt to contact the source node of the message in the overlay network for the executable specification. Since message created at the source node, it means it will likely have the executable specification for the service. To do this, a node that received a message marked with an unknown service (assuming the service is not found on the Services Server) sends a MessageStore control message called "Service Request message" to the source node of the unknown service message. This "Service Request message" is marked with the `ServiceID` of the unknown service and with a `MessageType` attribute of "-1". This corresponds to a special class of messages that when received by MessageStore, (since they are marked with a MessageStore extension) are processed completely within MessageStore (not delegated to `GenericFSM` object). Any overlay nodes that receives this "Service Request message" will extract the `ServiceID` to obtain the executable specification. If the executable specification is found, the SCXML document for that s`ServiceID` is uploaded to the Services Server.

After a node sends the "Service Request message", the unprocessed message is placed back to the end of the MessageStore message queues, so it can be processed again. When the message is processed again, MessageStore again checks if the Services Server has the executable specification for the message. This process continues until MessageStore obtains the executable specification or when a configurable (in HyperCast configuration file) number of tries are attempted.

Chapter 6

Evaluation

In this chapter, we present the evaluation of our design and implementation for customizable services for application-layer overlay networks. Our evaluation has four main objectives. First, we determine the bottlenecks in our implementation for the *Java-based* system and the *SCXML-based* system through various runtime benchmark experiments. The Java-based system refers to the existing implementation where each customizable service is defined and realized by a Java class object. The SCXML-based system refers to the new implementation where each customizable service is defined by an executable specification and executed by a generic SCXML engine. Second, we examine the sustainable performance for each of these system for single-hop, two-hop, and five-hop overlay message transfer. Thirdly, we compare the performance overhead of the Java-based system versus the SCXML-based system. Lastly, we evaluate the effectiveness of our systems for supporting customizable services for application-layer overlay networks.

Due to the large number of possible services our systems can support, we choose to focus in detail on two end-to-end message transfer services: 1) Hop-to-hop Acknowledgement (Section 3.1.2.1), and 2) End-to-end Acknowledgement (Section 3.1.2.2). These two services have sufficient complexity to allow detailed evaluation of our objectives for single-hop and multi-hop overlay message transfer scenarios.

For the evaluation, we conducted measurement experiments using a locally available testbed network at the University of Toronto. Before we present the experiments in detail, we describe the setup of the testbed network and our configuration for single-hop, two-hop and five-hop overlay message transfers.

6.1 Experiment Setup

6.1.1 Testbed Network

Our experiments were performed using a local testbed configured using *Emulab*. Emulab is an emulation system for creating and running large-scale networked and distributed system experiments using arbitrary user-defined network topologies [9]. Emulab provides a web interface for developing, managing, and debugging network systems. Emulab allows us to remotely manage and configure network topologies.

The Emulab testbed at the University of Toronto consists of 22 Dell PowerEdge 2950 III PCs each with two Quad-Core Intel Xeon X8 5400 series processors clocked at 2.00 GHz, and 4GB DDR2 RAM. Out of the 22 PCs, 2 machines are dedicated as control servers, one for the gateway to the remaining 20 machines, and one for the web server to access Emulab's web interface. Each PC has an Intel VT PCIe Quad-port Copper Gigabit Ethernet NIC with 4 Gigabit Ethernet interfaces proving 8 usable Ethernet interfaces per PC. Four 48-port Cisco Catalyst 4949-10GE switches interconnect the PCs.

Emulab supports a variety of operating systems. For our experiments, we use the Fedora Core 6 Linux distribution with the Linux 2.6.20.6 kernel version.

6.1.2 Network Topology

For our experiments, we set up networks consisting of up to 6 network nodes connected using bidirectional Ethernet links. The network topology for these 6 nodes is in the form of a daisy-chain as visualized in Figure 6.1. There are 6 network nodes (node-1, node-2, node-3, node-4, node-5, and node-6). Bidirectional links of 100 Mbps transfer capacity connect adjacent nodes (node-1 and node-2, node-2 and node-3, etc.). The current version of Emulab supports a maximum link capacity of 100 Mbps. These bidirectional links are set using Emulab to have 0 ms delay. This means there is no added delay, but there is still the transmission delay between adjacent nodes. As illustrated by Figure 6.1, each end of a

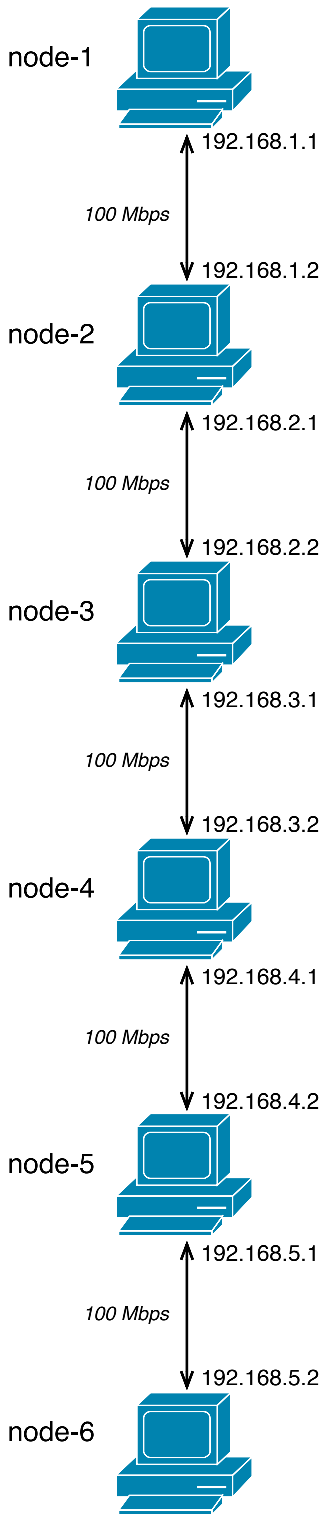


Figure 6.1: Network topology setup of experiments

bidirectional link connects to a network interface with a unique IP address. For example, the link between node-1 and node-2 connects the network interface of 192.168.1.1 and 192.168.1.2 and the link between node-2 and node-3 connects the network interfaces of 192.168.2.1 and 192.168.2.2.

We manually set the IP routing tables on each of the nodes such that packets are forwarded properly in the configured topology. In this manner, a packet sent from node-1 destined for node-6 must traverse the links between node-1 and node-2, node-2 and node-3, node-3 and node-4, node-4 and node-5, and node-5 and node-6 before reaching node-6.

The clocks for each of the 6 nodes are synchronized using NTP (Network Time Protocol). NTP is a protocol for synchronizing clocks over packet-switched networks [12]. The clocks across our experimental testbed have a jitter of less than 30μs. All nodes (node-1 through node-6) communicate with a central gateway server (ops.testbed.nrl) that serves as the master clock for the slave nodes (node-1 through node-6). A sample output below of the NTP peering list shows this relationship between the gateway (ops.testebed.nrl) and node-1.

```
[tony@node-1 ~]$ ntpq -p
     remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
*ops.testbed.nrl 128.100.102.201  3 u  911 1024  377    0.102    0.228   0.021
```

A *traceroute* from node-1 to node-6 displays the actual delays and route. The output is as follows.

```
[tony@node-1 ~]$ traceroute node-6
traceroute to node-6 (192.168.5.2), 30 hops max, 40 byte packets
 1  node-2-link12 (192.168.1.2)  1.366 ms  1.359 ms  1.353 ms
 2  node-3-link23 (192.168.2.2)  1.442 ms  1.439 ms  1.434 ms
 3  node-4-link34 (192.168.3.2)  3.325 ms  3.323 ms  3.319 ms
 4  node-5-link45 (192.168.4.2)  3.405 ms  3.403 ms  3.399 ms
 5  node-6-link56 (192.168.5.2)  3.394 ms  3.391 ms  3.386 ms
```

6.1.3 Overlay Network Topology using DT Protocol

We use HyperCast as the middleware to construct and maintain an application-layer overlay network of 6 overlay nodes on top of the physical network topology as described in Section 6.1.2. The overlay network topology is also a daisy chain as shown in Figure 6.2. Every physical node is assigned a unique logical address and every logical link corresponds to an existing physical link. Hence, for our experiments, the physical network topology and overlay network topology are identical. For our experiments, we use a structured overlay protocol to build a topology using the Delaunay Triangulation protocol. The protocol is referred to as the DT Buddy List protocol in the HyperCast middleware. Each node is assigned a logical address in the form of a (x,y) coordinate. In our experiment, Node-1 is assigned a logical address of (100,100), node-2 assigned a logical address of (200,200), node-3 assigned a logical addresses of (300,300), etc.

We configure the nodes such that node-n obtain its neighbourhood information (Buddy List) from node-$(n-1)$. We manually assign every logical address in the overlay network. Note that our nodes do not actually form a triangulation since it is manually configured to be in a daisy-chain. Additionally we configure the overlay network such that every node can only establish logical connections to its immediate neighbours. For example, node-1 can only connect with node-2 and node-3 only connect with node-2 in our overlay network. This was accomplished by making the logical links between adjacent nodes have different substrates identifiers mimicking different substrates (the actual substrates are all TCP/IP). This multi-substrate overlay setup [46] for HyperCast. For example, in Figure 6.2, node-1 connects to node-2 using TCP on substrate 1, node-2 connects to node-3 using TCP on substrate 2, etc. In this way, if the HyperCast application is not running on node-2, messages sent from node-1 to node-3 would never reach its destination. We created this setup because we want to control how overlay messages are forwarded from source to destination in our overlay topology at the application-layer. Emulab has a separate control network that can interfere with message delivery during multicast if we do not enforce these conditions. The result is that all overlay messages must traverse through node-1 to node-$(n-1)$ in order to reach node-n.

The neighbourhood information for node-2 is shown below. Node-2 has two neigh-

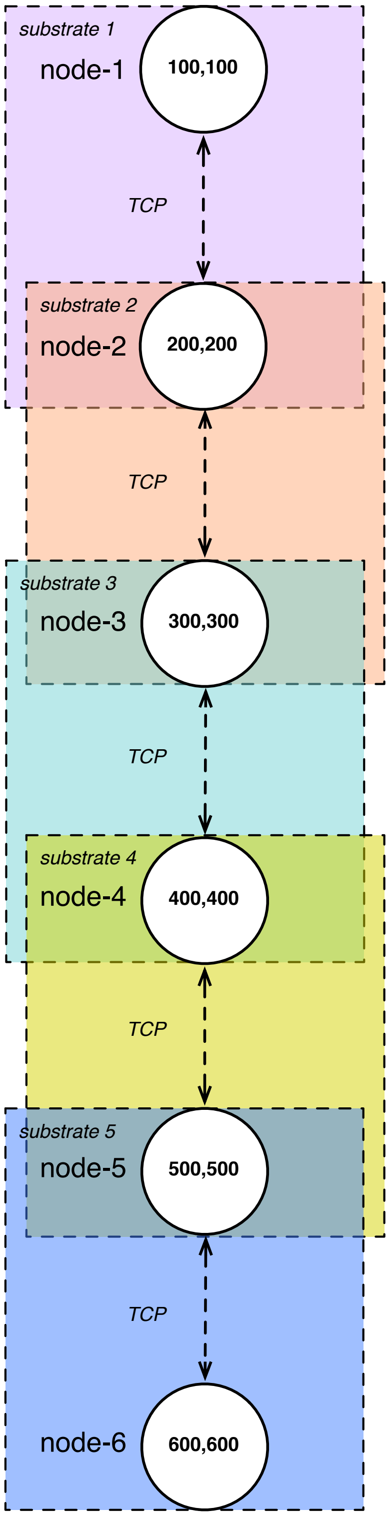


Figure 6.2: Application-layer overlay topology setup of experiments

bours (100,100) corresponding to node-1 and (300,300) corresponding to node-3.

```
Receiver Logical address is 200,200.

<?xml version="1.0" encoding="UTF-8"?>
<NeighborTable>
  <LogicalAddress>300,300</LogicalAddress>
  <SubstrateAddress>tcp2|192.168.2.2:9800</SubstrateAddress>
  <CW/>
  <CCW/>
</NeighborTable>
<?xml version="1.0" encoding="UTF-8"?>
<NeighborTable>
  <LogicalAddress>100,100</LogicalAddress>
  <SubstrateAddress>tcp1|192.168.1.1:9800</SubstrateAddress>
  <CW/>
  <CCW/>
</NeighborTable>
```

6.1.4 Underlay Substrate

For our experiments, we use TCP/IP substrates. Using TCP/IP guarantees there is no packet loss between the network interfaces of the overlay nodes during overlay messages transfer. Loss can only occur if buffers in the overlay socket or the MessageStore overflows. With our fixed configuration, overlay messages cannot be dropped due to topology changes. When we want to measure the impact of dropped messages, we induce loss by dropping messages at overlay nodes.

6.2 Overview of Experiments

6.2.1 Measurement Methodology

Table 6.1 shows the list of metrics that are of interest for our experiments.

To obtain metrics within HyperCast we built a detailed logging system. The logging system utilizes timestamps of nanosecond precision using the Java `System.nanoTime()`. Upon receiving and processing of a data message or control message, the MessageStore logs parameters: message processing time (T_{msg}), MessageStore data message buffer backlog (L_{data}), MessageStore control message buffer backlog ($L_{control}$), and MessageStore stored FSMs (N_{FSM}) in an external file.

We provide a TTCP [4] variant for overlay messages using HyperCast implemented in Java. TTCP is a utility for measuring network metrics popular on UNIX systems using the User Datagram Protocol (UDP) or Transmission Control Protocol (TCP) protocols. We use TTCP to set the transmission rate between overlay nodes and measure the network parameters such as sent time of messages (T_{sent}), received time of messages ($T_{received}$), transmission delay (D), throughput, etc.

6.2.2 Configuration Parameters

The following are configuration parameters that are fixed for our experiments:

- MessageStore parameters
  - **MessageStore Data Message Buffer Backlog (L_{data}):** maximum size of 10,000 messages for this droptail buffer.
  - **MessageStore Control Message Buffer Backlog ($L_{control}$):** maximum size of 10,000 messages for this droptail buffer.
  - **MessageStore Stored FSMs (N_{FSM}):** initial capacity of the hash table that stores FSMs in MessageStore of 1,000 buckets with a load factor of 0.75. The

Table 6.1: Definition of Measured Metrics

Metric	Description
Delay (D)	The elapsed time from when a data message is sent by the sender application (T_{sent}) to when it is received by the receiver application measured with millisecond precision.
Message Processing Time (T_{msg})	The elapsed time from when a data/control message arrives from the adapter to when it is delivered to the application. This is the total processing time in the overlay socket plus the MessageStore, including the time the message spends in the buffers. This metric is measured with nanosecond precision.
MessageStore Data Message Buffer Backlog (L_{data})	The number of data messages awaiting processing by the MessageStore.
MessageStore Control Message Buffer Backlog ($L_{control}$)	The number of control messages awaiting processing by the MessageStore.
MessageStore Stored FSMs (N_{FSM})	The number of FSMs stored in the MessageStore hash table. This metric is indicative of how many FSMs are created (one for each data message) waiting to receive control messages.
Received Time ($T_{received}$)	The time when a data message is received by the receiver application measured with millisecond precision.
Sequence Number (seq)	The sequence number of a sent data message.
Sent Time (T_{sent})	The time when a data message is sent by the sender application measured with millisecond precision.
Message Size (S)	The size of a data message in Bytes.
Throughput ($throughput$)	The throughput is calculated using a moving average of the 500 last received data messages. The throughput is calculated by the following formula: $Throughput_{i+500} = \frac{S_i + S_{i+1} + S_{i+2}, \ldots, S_{i+500}}{T_{received_{i+500}} - T_{sent_i}}$; $i > 0$ The units for throughput is kilobits-per-second (Kbps).

load factor is the ratio of the number of stored entries and the size of the hash table's array of buckets. The load factor is a measure of how full the hash table must be before its capacity is automatically increased.

- Hop-to-Hop Acknowledgement service and End-to-End Acknowledgement service parameters
  - $Timeout_{ACK}$: The maximum time a sender of a data message waits for an acknowledgment message before sending an acknowledgment request. The value is set to 1 second.
  - $Timeout_{Delete}$: The amount of time the sender of a data message keeps a stored FSM in its MessageStore after the FSM reaches completion. The value is set to 10 seconds.

6.2.3 Experiment Topologies

We conducted experiments using single-hop, two-hop, and five-hop overlay message transfer scenarios over the network setup in Section 6.1.3. Below we detail which nodes and links are used for each of the three transfer scenarios.

- **single-hop transfer**: we use node-1 and node-2 in the network topology. We transfer overlay messages from node-1 Sender (S) to node-2 Receiver (R) using overlay multicast.
- **two-hop transfer**: we use node-1, node-2 and node-3 in the network topology. We transfer overlay messages from node-1 Sender (S) to node-2 Receiver 1 ($R1$) and node-3 Receiver 2 ($R2$) using overlay multicast. Here node-2 is the intermediate node, *i.e.*, messages multicast to node-3 ($R2$) are forwarded by node-2 ($R1$).
- **five-hop transfer**: we use node-1 through node-6 in the network topology. We transfer overlay messages from node-1 Sender (S) to node-2 Receiver 1 ($R1$), node-

3 Receiver 2 ($R2$), node-4 Receiver 3 ($R3$), node-5 Receiver 4 ($R4$) and node-6 Receiver 5 ($R5$) using overlay multicast. Here node-2 through node-5 are the intermediate nodes.

6.2.4 List of Experiments

Table 6.2 and 6.3 provides an overview of the following sections in this chapter and descriptions of the experiments contained within those sections.

Table 6.2: List of Experiments for Hop-to-Hop Acknowledgement service

Section	Scenario	Description
6.3.1	single-hop	sequence number vs. time, delay vs. sequence number, throughput vs time) of the Java-based implementation and the SCXML-based implementation
6.3.2	single-hop	bottleneck analysis of the Java-based implementation
6.3.3	single-hop	bottleneck analysis of the SCXML-based implementation
6.3.4	single-hop	performance improvements of the SCXML-based implementation
6.3.5	single-hop	sustainable throughput and delay of the Java-based implementation and the SCXML-based implementation.
6.4.1	two-hop	sequence number vs. time, delay vs. sequence number, throughput vs time of the Java-based implementation and the SCXML-based implementation.
6.4.1.1	two-hop	bottleneck analysis of the Java-based implementation.
6.4.1.2	two-hop	bottleneck analysis of the SCXML-based implementation.
6.4.1.3	two-hop	sustainable throughput and delay of the Java-based implementation and the SCXML-based implementation.
6.4.2	five-hop	sustainable throughput and delay of the Java-based implementation and the SCXML-based implementation.

Table 6.3: List of Experiments for End-to-End Acknowledgement service

Section	Scenario	Description
6.5.1	single-hop	sequence number vs. time, delay vs. sequence number, throughput vs time of the Java-based implementation and the SCXML-based implementation.
6.5.2	single-hop	bottleneck analysis of the Java-based implementation.
6.5.3	single-hop	bottleneck analysis of the SCXML-based implementation.
6.5.4	single-hop	sustainable throughput and delay of the Java-based implementation and the SCXML-based implementation.
6.6.1	two-hop	sustainable throughput and delay of the Java-based implementation and the SCXML-based implementation.
6.6.2	five-hop	sustainable throughput and delay of the Java-based implementation and the SCXML-based implementation.

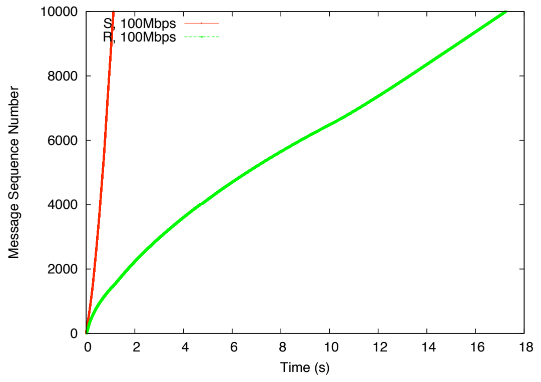
6.3 Hop-to-Hop Acknowledgment in Single-Hop Network

6.3.1 Java-based Implementation and SCXML-Based Implementation Performance at send rate of 100 Mbps, 10 Mbps, and 1 Mbps

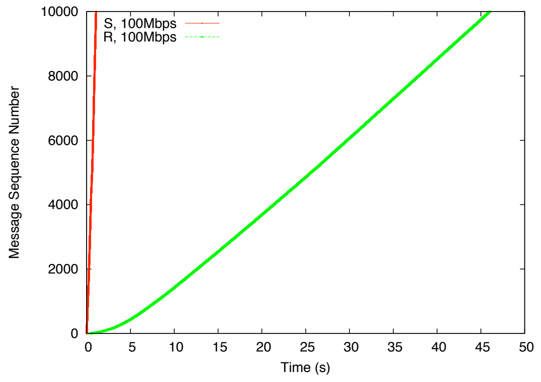
In this set of experiments, we study the performance of the Java-based and SCXML-based implementations of the Hop-to-Hop Acknowledgement service. The experiments consist of a single-hop transfer of 10,000 overlay messages with 1,024 bytes payloads from node-1 (Sender S) to node-2 (Receiver R). The sending rates are set to 100 Mbps, 10 Mbps, and 1 Mbps. 100 Mbps is the maximum link rate in our Emulab network.

Figure 6.3 shows the overlay data message sequence numbers (seq) as a function of time for the Java-based ((a),(c),(e)) and SCXML-based ((b),(d),(f)) implementations with send rates of 100 Mbps, 10 Mbps, and 1 Mbps. The red data points in the figures are the sent times (T_{sent}) of each sequence number of data messages at the Sender S measured in milliseconds. These data points are repositioned such that the sent time of the first data message is at time 0s. The green data points represent the received times ($T_{received}$) of each sequence number of data messages at the Receiver R measured in milliseconds. These data points are also repositioned such that the first data point is the difference between received time ($T_{received}$) and the sent time (T_{sent}) of the first data message (with sequence number 1).

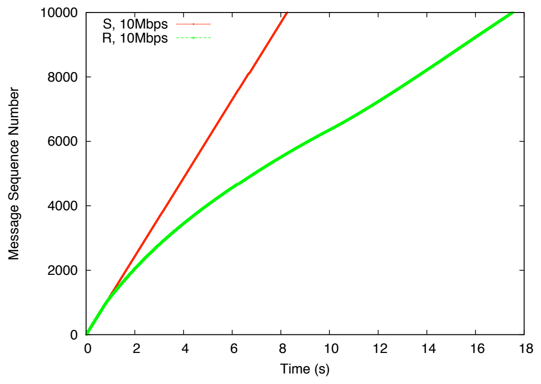
For both the Java-based implementation and SCXML-based implementation, at 100 Mbps and 10 Mbps, the receivers experience delays, as seen by the increasing difference between the sending time and received time for each message. This delay is the total time each message spends in the overlay socket (buffer and processing) and in the MessageStore (buffer and processing). We show in Section 6.3.2 and 6.3.3 that the increasing delays are the result of increasing MessageStore buffer backlog when messages arrive at an unsustainable rate. From this it is evident that both implementations cannot sustain a rate of 10 Mbps and above. At a sending rate of 1 Mbps, both implementations perform well with no noticeable delay between the sent time and received time.



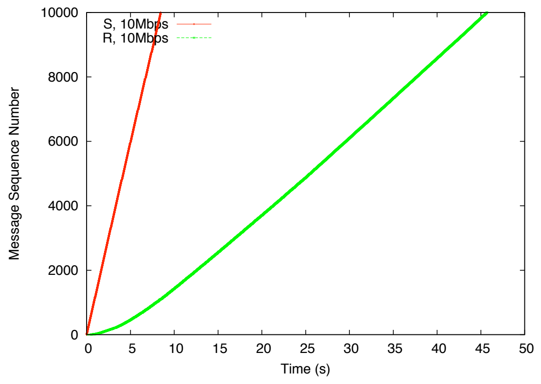
(a) Java-based, send rate = 100Mbps



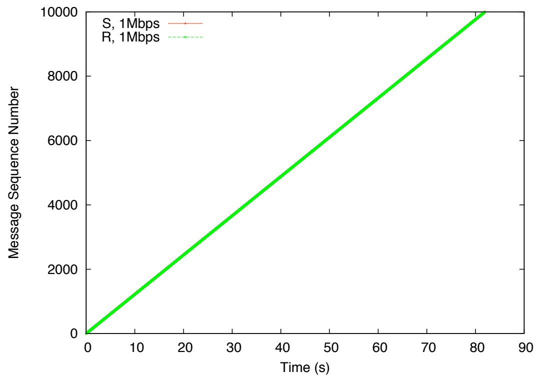
(b) SCXML-based, send rate = 100Mbps



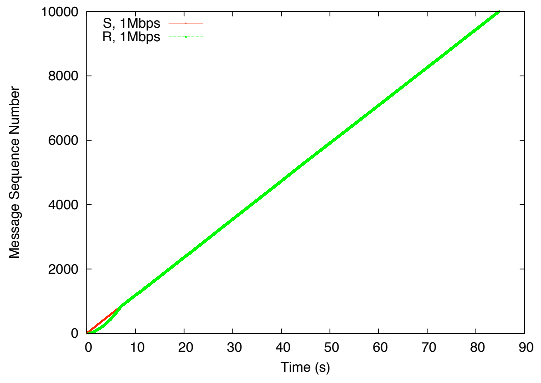
(c) Java-based, send rate = 10Mbps



(d) SCXML-based, send rate = 10Mbps



(e) Java-based, send rate = 1Mbps

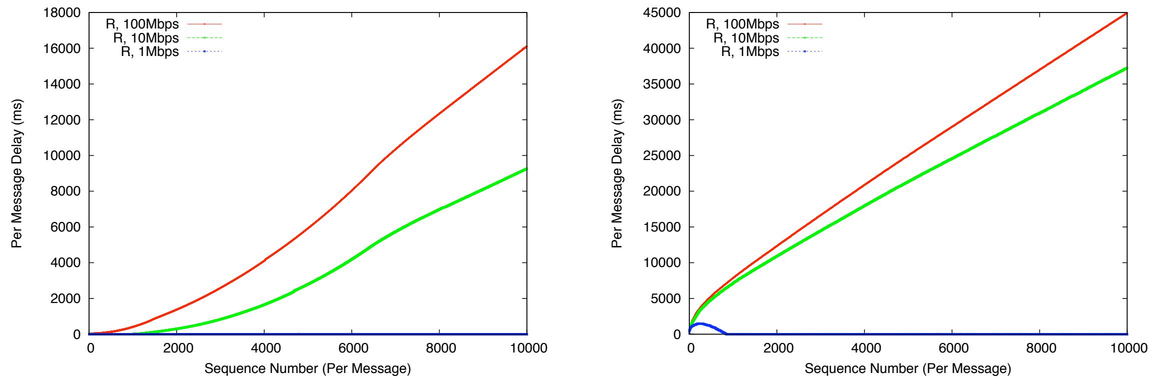


(f) SCXML-based, send rate = 1Mbps

Figure 6.3: Sequence number versus time for Hop-to-Hop Acknowledgement

Figure 6.4 shows the per message delay (D) as a function of the sequence number (seq) for data messages for both the Java-based and SCXML-based implementations. The delay is the total delay from when the message is sent at the sender to when it received by the application at the receiver measured in milliseconds. The red, green, and blue data points represent sending rates of 100 Mbps, 10 Mbps, and 1 Mbps respectively.

From these two figures, we see that at 100 Mbps and 10 Mbps sending rates, both implementations show increasing per message delays. At a sending rate of 1 Mbps, the Java-based implementation show negligible a per message delay. The SCXML-based implementation show per message delay for the first 1,000 messages at approximately 1 s.



(a) Java-based

(b) SCXML-based

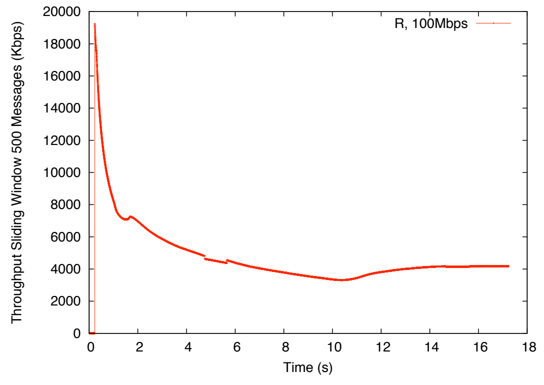
Figure 6.4: Per message delay versus time for Hop-to-Hop Acknowledgement

Next we examine how the delays are reflected in the throughput at the receivers. We show the receiver throughput as a function of time in Figure 6.5 for both the Java-based ((a),(c),(e)) and SCXML-based ((b),(d),(f)) implementations with send rates of 100 Mbps, 10 Mbps, and 1 Mbps. The *throughput* is displayed on the y-axis of the figures and is calculated according to the formula presented in Table 6.1. The first data point for throughput is calculated at the received time of the 500th message. Time is represented on the x-axis. Similar to previous plots, the time represents the received times ($T_{received}$) of each sequence number of data messages at the Receiver R. These data points are repositioned such that the first data point is the difference between received time ($T_{received}$) and the sent time (T_{sent}) of first the data message.

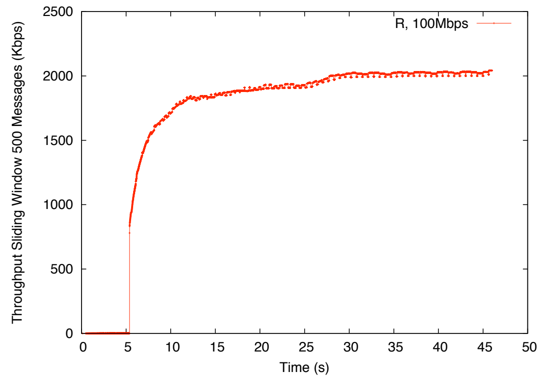
At 100Mbps and 10Mbps sending rate for the Java-based implementation, the receiver throughput initially has a value of approximately 20 Mbps and 10 Mbps, respectively, and then decreases to under 4 Mbps around 10s. Beyond 10s, the throughput rises to a steady-state value of slightly above 4 Mbps. For the sending rate of 1 Mbps (Figure 6.5(e)) for the Java-based implementation, the receiver throughput matches exactly the sending rate of 1 Mbps. These plots seem to indicate the sustainable throughput for the Java-based implementation is approximately 4 Mbps. Note that the initial data point for throughput (calculated with the 500th received message) occurs later for lower sending rates since a lower sending rate results in longer times until the 500th message is received.

For the SCXML-based implementation, there is a different pattern for the throughput observed with sending rate of 100 Mbps and 10 Mbps. The receiver throughput is low in the beginning and steadily rises to a steady-state throughput of around 2 Mbps. At 1 Mbps sending rate (Figure 6.5(f)), the SCXML-implementation shows a spike in the throughput between 5 s and 10 s, and then maintains a steady-state throughput equivalent to the sending rate. For the SCXML-based implementation, the first calculated data point for sending rates of 100 Mbps, 10 Mbps, and 1 Mbps occurs at approximately the same time (around 5s) independent of the sending rate. This indicates that the performance of the SCXML-based implementation encounters a performance bottleneck at the beginning of an experiment. In contrast, the Java-based implementation suggest that the system is bottlenecked at a later time due to the high initial throughput and the steady decrease to steady-state. If we compare Figure 6.3 with Figure 6.5 we see evidence that the Java-based implementation performs faster in the beginning, while the SCXML-based implementation performs slower in the beginning.

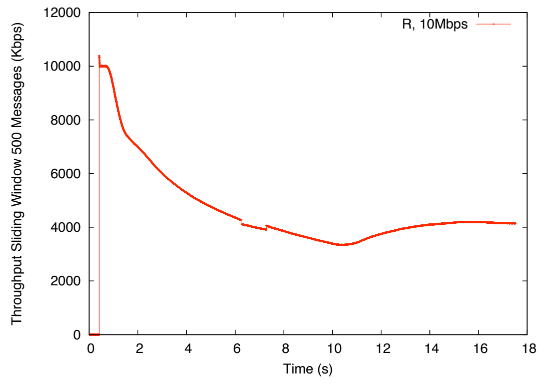
A closer examination of the initial 1,000 message received in Figure 6.5(f) shows that the throughput spike encountered between 5 s and 10 s corresponds to the per message delay pattern seen in Figure 6.4(b). Initially, there is a large per message delay which slowly decreases to small delays after 1,000 messages. Since the throughput is calculated with a sliding window of 500 messages, the decrease in the elapsed time between consecutively received messages (less per message delay as the sequence number increases) after a large initial delay results in this burst pattern in the throughput.



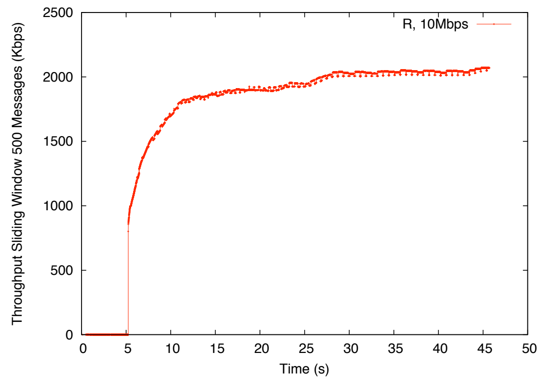
(a) Java-based, send rate = 100Mbps



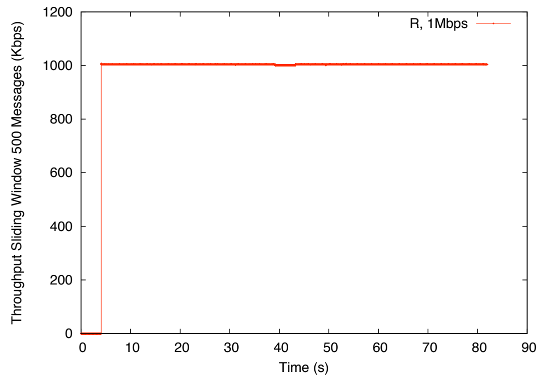
(b) SCXML-based, send rate = 100Mbps



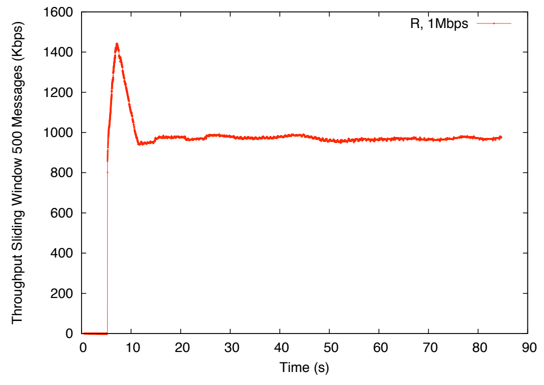
(c) Java-based, send rate = 10Mbps



(d) SCXML-based, send rate = 10Mbps



(e) Java-based, send rate = 1Mbps



(f) SCXML-based, send rate = 1Mbps

Figure 6.5: Throughput versus time for Hop-to-Hop Acknowledgement

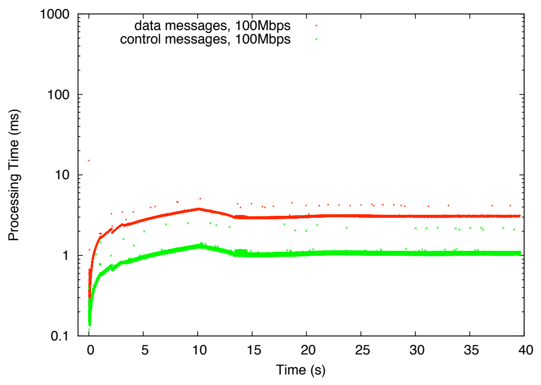
6.3.2 Bottleneck Analysis of Java-based Implementation

We next determine the performance bottlenecks for Hop-to-Hop Acknowledgement service by profiling 1) the per message processing time (T_{msg}), 2) the MessageStore buffers (L_{data} and $L_{control}$) and, 3) the number of stored FSMs (N_{FSM}) in the MessageStore. The metrics are described in Table 6.1. The bottleneck analysis is based on the same set of experiments detailed in Section 6.3.1.

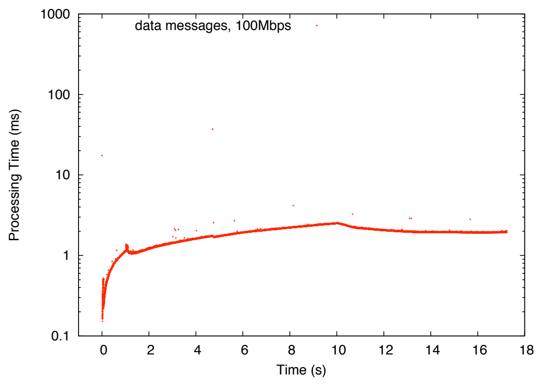
First we examine the Java-based implementation. In Figure 6.6 the MessageStore per message processing time for the Java-based implementation of Hop-to-Hop Acknowledgement is shown for send rates of 100 Mbps, 10 Mbps, and 1 Mbps. The x-axis time in these figures is the same metric as those in Section 6.3.1 and Section 6.3.4. The left column figures (Figure 6.6(a),(c),(e)) show processing times at the Sender (S) and the right column figures (Figure 6.6(b),(d),(f)) show the processing times at the Receiver (R). The red data points represent data messages and the green data points represent control messages (in this service they are acknowledgement messages). Hence, the receiver does not process any control messages. The sender processes data messages after they are forwarded; that is, messages are forwarded first, and are then processed at the MessageStore to ensure fast forwarding as explained in Chapter 5. In the case of Hop-to-Hop Acknowledgement, the sender also waits to receive acknowledgement messages from the receiver.

From the message processing times we first note that control messages take less time to process than data messages because data messages have a payload. On average, it takes MessageStore, at steady-state, approximately 0.5 ms - 1 ms to process a control message and 1.5 ms - 2 ms to process a data messages; depending on the send rate and if the data is measured at the sender or receiver. Processing of data messages at the sender appears to take more time than at the receiver. This is because the sender needs to perform more computation since it also receives control messages. The processing times for unsustainable send rates (10 Mbps and 100 Mbps) is also longer than for the sustainable send rate (1 Mbps). The general pattern in these figure exhibit that the MessageStore processing time starts at approximately 0.1 ms and steadily increases to the steady-state value around the 10s mark for both the receiver and senders. This correlates directly to the higher initial throughput seen in Figure 6.5(a),(c) and the receiver reaching the steady-state throughput at 10 s.

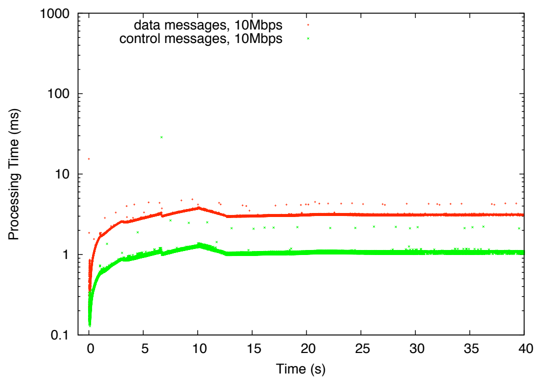
The data shown in Figure 6.6 shows very little variance. There are outliers in the plots but due the the large number of data points for each figure (10,000), these constitute a small percentage. We have examined these outliers and have found no link between them and our design and implementation. We thus believe these data points are caused by the standard Java libraries. The likely explanations are that these outliers are caused by data structure resizing and/or Java garbage collection. We observe outliers in all experiments in this Chapter and conclude with the same explanations.



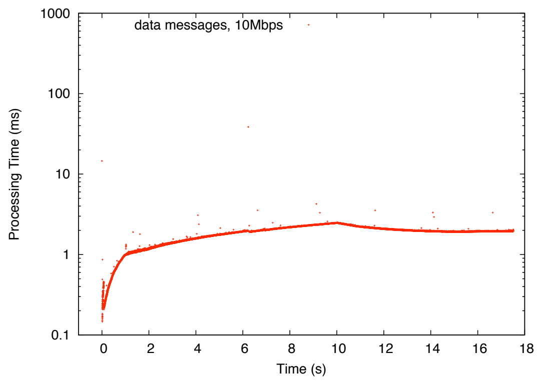
(a) Sender (S), send rate = 100Mbps



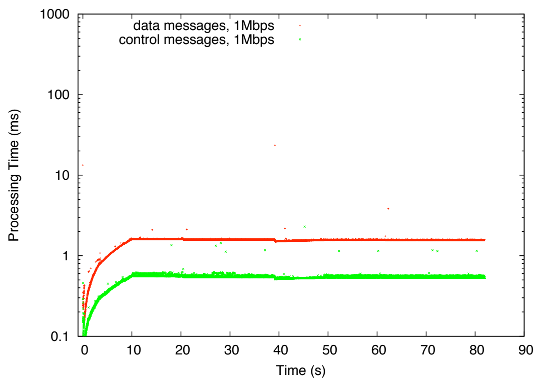
(b) Receiver (R), send rate = 100Mbps



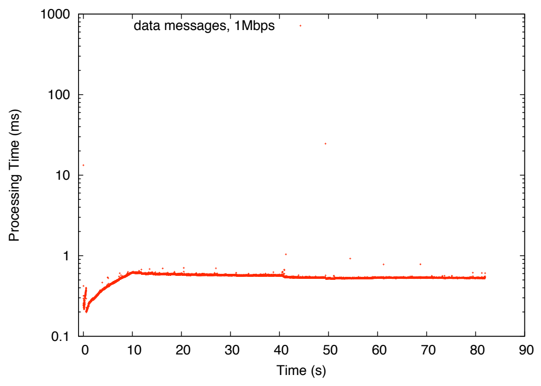
(c) Sender (S), send rate = 10Mbps



(d) Receiver (R), send rate = 10Mbps



(e) Sender (S), send rate = 1Mbps



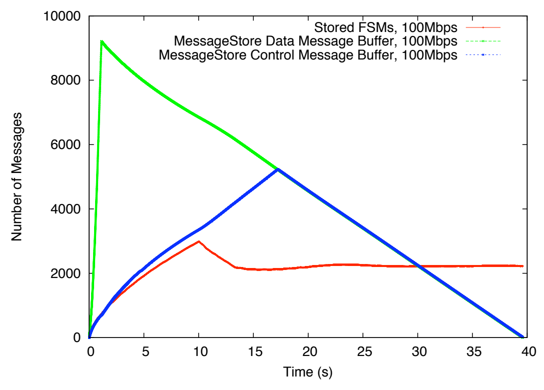
(f) Receiver (R), send rate = 1Mbps

Figure 6.6: MessageStore per message processing time versus time for Hop-to-Hop Acknowledgement of the Java-based implementation

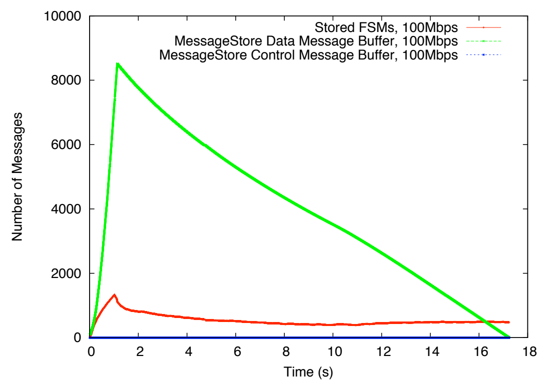
In Figure 6.7 we present the size of the MessageStore buffers (L_{data} and $L_{control}$) and the number of stored FSMs (N_{FSM}) in the MessageStore as a function of time. The x-axis time in these figures is the same metric as those in previous sections. The left column figures (Figure 6.7(a),(c),(e)) show the Sender (S) and the right column figures (Figure 6.7(b),(d),(f)) show the Receiver (R). The red data points represent the number of stored FSMs (N_{FSM}) in the MessageStore's hash table. The green data points represent the number of backlogged messages in the MessageStore data message buffer (L_{data}). The blue data points represent the number of backlogged messages in the MessageStore control message buffer ($L_{control}$).

These plots show that at send rates of 100 Mbps and 10 Mbps, a large backlog of data messages is built up at both the sender and and the receiver. At a rate of 100 Mbps, almost all of 10,000 data messages are buffered before being processed since MessageStore cannot process messages at that rate. A the lower rate of 10 Mbps, less than 50% of data messages are buffered. At a rate of 1Mbps, there is no backlog of data messages at the sender or receiver. For control messages received at the sender, the plots show a large backlog of messages in the buffer for unsustainable send rate of 100 Mbps and 10Mbps and no backlog of messages at a send rate of 1 Mbps. We notice that the downslope of green and blue curves are nearly identical. This downslope is the rate at which the messages are being dequeued from the MessageStore data and control message buffers. It appears that this rate is bounded to an upper value of approximately 200 messages per second.

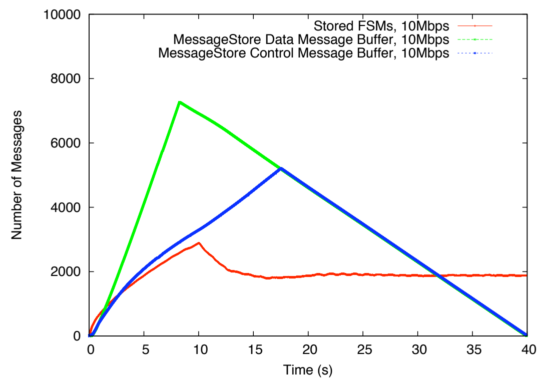
Looking at the red data points, the number of stored FSMs (N_{FSM}) in the MessageStore, there appears to be a correlation between the number of stored FSMs (Figure 6.7) and the processing time for each message (Figure 6.6) for both the sender and receiver. We see that when the MessageStore has no stored FSMs, the processing time is low. As the number of stored FSMs increase, so does the message processing time. For the senders, the number of stored FSMs reaches a maximum at around 10 s seen in Figures



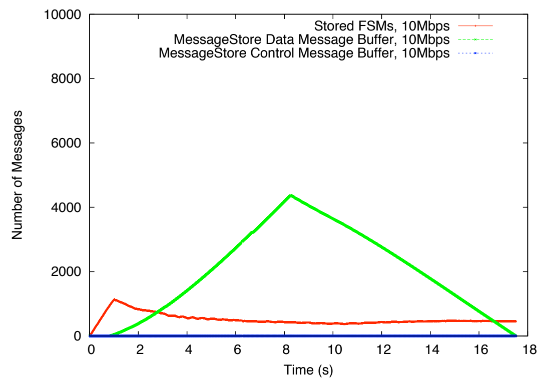
(a) Sender (S), send rate = 100Mbps



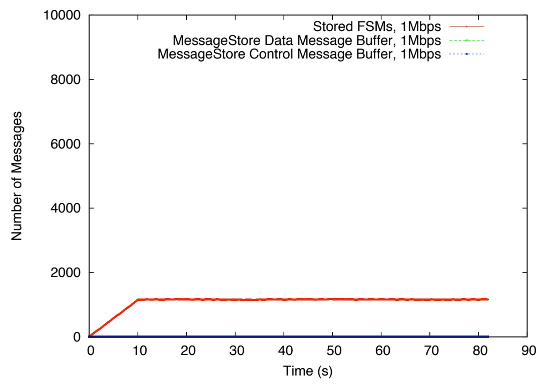
(b) Receiver (R), send rate = 100Mbps



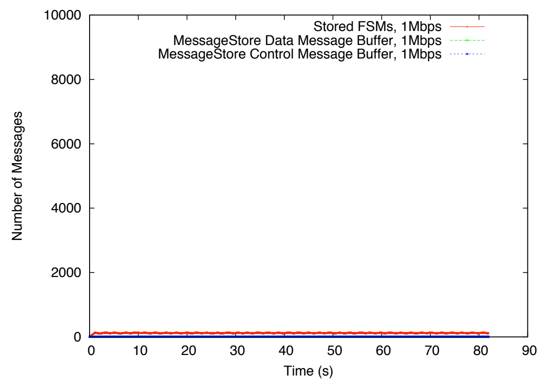
(c) Sender (S), send rate = 10Mbps



(d) Receiver (R), send rate = 10Mbps



(e) Sender (S), send rate = 1Mbps



(f) Receiver (R), send rate = 1Mbps

Figure 6.7: Backlog of messages buffered in MessageStore for data and control message, and the number of stored FSMs in MessageStore versus time for Hop-to-Hop Acknowledgement for the Java-based implementation

6.7(a),(c),(e) and achieves steady-state thereafter. This is reflected in the message processing times, for both data and control messages, seen in Figures 6.6(a),(c),(e) where the processing time steadily increases until after 10 s where steady-state is reached. At the senders, the FSMs gets removed when acknowledgement messages corresponding to the data messages are received. The FSMs then reaches its final state and are removed after a timer expires ($Timeout_{Delete}$), this timer is set at 10s as described in Section 6.2.2. Hence steady-state is reached at around 10 s. For the receivers, the number of stored FSMs reaches a maximum at around 1s seen in Figure 6.7(b),(d) and achieves steady-state thereafter.

This trend is also reflected in the message processing times of data messages, seen in Figures 6.6(b),(d),(f) where the processing time steadily increases until around the 1-1.5s mark where steady-state is reached. Note that the receivers do not receive any control messages in this set of experiments. The receivers do not wait for acknowledgement messages and terminates as soon as the FSMs reaches its final state. Hence steady-state is reached much earlier at around 1 s - 1.5 s. Steady-state is not reached immediately since there is a large per message delay initially as shown in Figure 6.4.

We argue that the correlation between the message processing time and the number of stored FSMs in the MessageStore is due to the hash table access times. Since the FSMs are stored in the MessageStore in a hash-table of 10,000 buckets with a load factor of 0.75, it takes longer to search, retrieve, and update any existing or new FSMs as the hash table becomes fuller.

To confirm this we conduct a supplementary set of experiments with send rate of 100 Mbps, 10 Mbps, and 1 Mbps without ever removing any stored FSMs (*i.e.*, no FSMs are removed once the acknowledgements are received at the sender from the receiver).

6.3.2.1 The effect of the number of stored FSMs in MessageStore on per message processing time

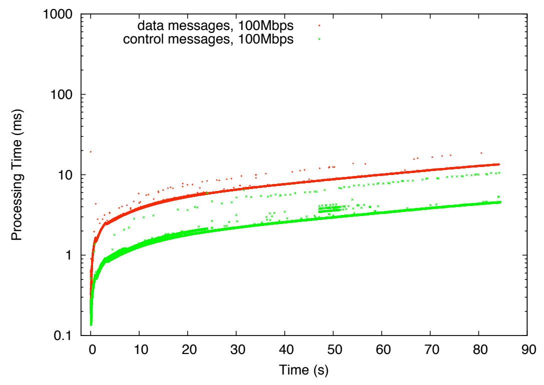
Figure 6.8, Figure 6.9, Figure 6.10 show the MessageStore processing time, MessageStore data and control message backlog, MessageStore stored FSMs, and throughput as a function of time without ever removing an allocated FSMs. These figures are displayed in the same manner as those in figures of the previous two sections.

The figures confirms the hypothesis that, as the number of stored FSMs increases in MessageStore (red curve in Figure 6.9), the per message processing time increases accordingly (Figure 6.8), which results in a continuously decreasing throughput (Figure 6.10) over time.

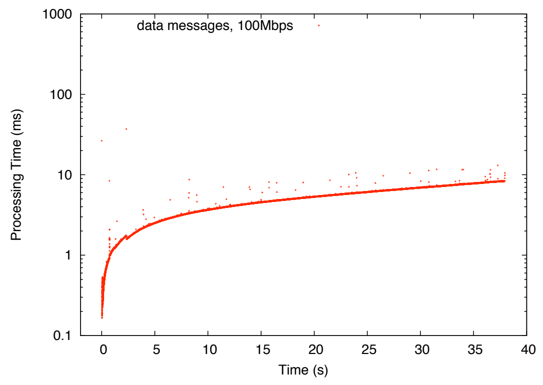
If we compare Figure 6.9 and Figure 6.7, we see that if we do not remove the FSMs, the stored number of FSMs in MessageStore does not reach steady-state (red curve Figure 6.9) for all send rates at both the Sender (S) and Receiver (R). This negatively affects the performance of our systems if we compare the message processing times between Figures 6.6 and 6.8. When the FSMs are removed the message processing times reaches steady-state (Figures 6.6). In contrast, if the FSMs are not removed, the message processing times continues to increase and never reaches steady-state (Figures 6.8).

This is reflected in the throughput, without removing the FSMs, the throughput (Figure 6.10(a),(b)) does not reach steady-state at rates of 100 Mbps and 10 Mbps compared to Figures 6.5(a),(c).

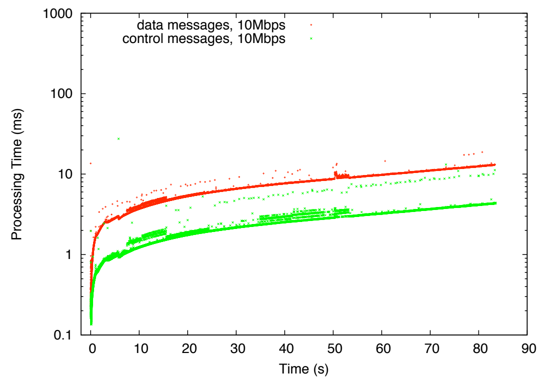
After evaluating the measurements for the Java-based implementation of Hop-to-Hop acknowledgement, we have determined that the primary performance overhead for the system is the number of stored FSMs in the MessageStore. This number of FSMs is directly related to the rate of which messages are processed in MessageStore.



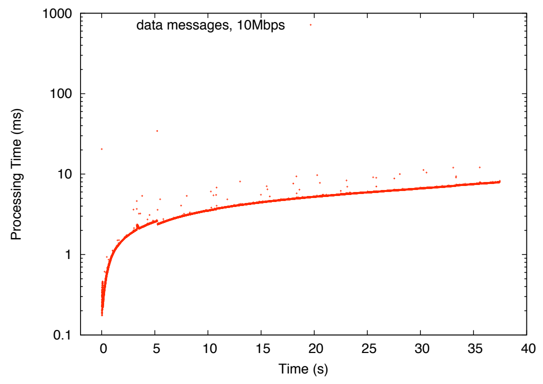
(a) Sender (S), send rate = 100Mbps



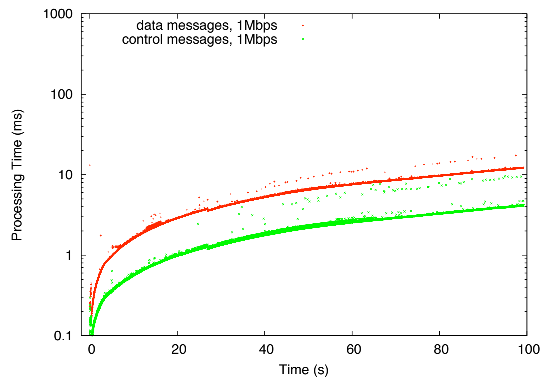
(b) Receiver (R), send rate = 100Mbps



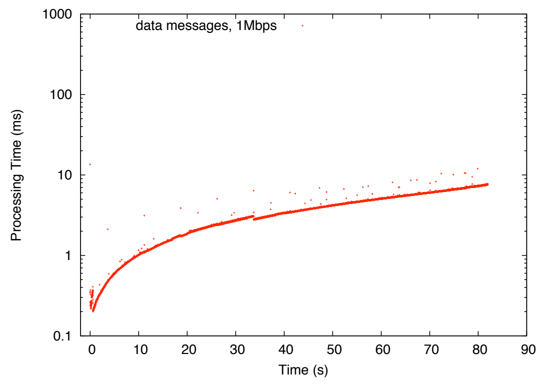
(c) Sender (S), send rate = 10Mbps



(d) Receiver (R), send rate = 10Mbps

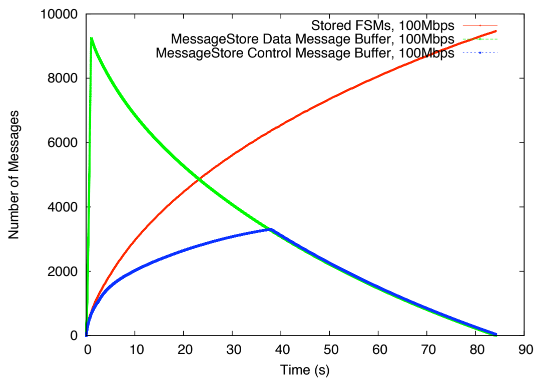


(e) Sender (S), send rate = 10Mbps

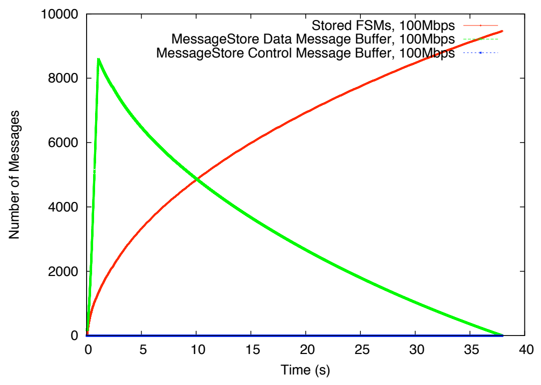


(f) Receiver (R), send rate = 10Mbps

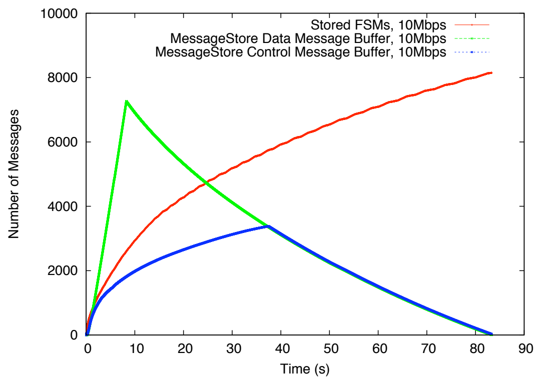
Figure 6.8: MessageStore per message processing time versus time for Hop-to-Hop Acknowledgement of the Java-based implementation without removing any MessageStore stored FSMs



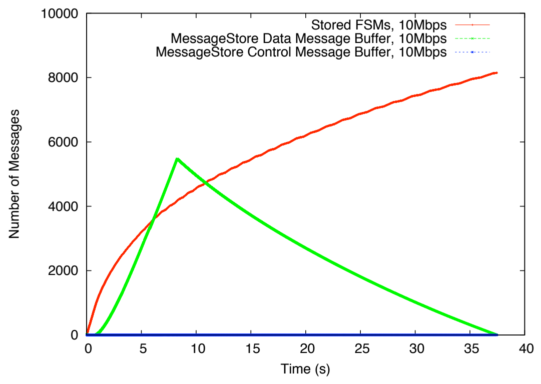
(a) Sender (S), send rate = 100Mbps



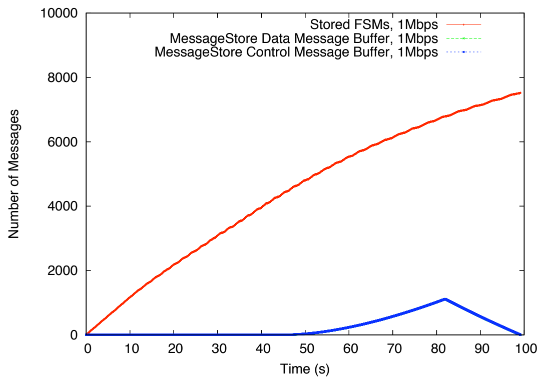
(b) Receiver (R), send rate = 100Mbps



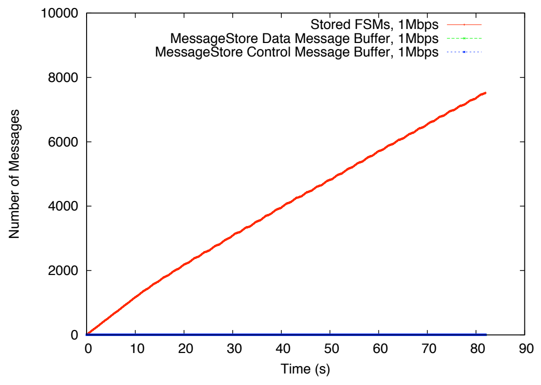
(c) Sender (S), send rate = 10Mbps



(d) Receiver (R), send rate = 10Mbps

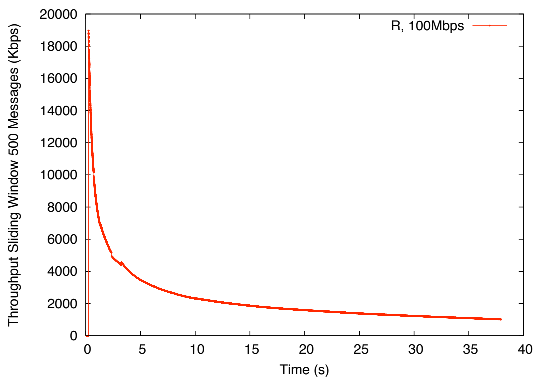


(e) Sender (S), send rate = 1Mbps

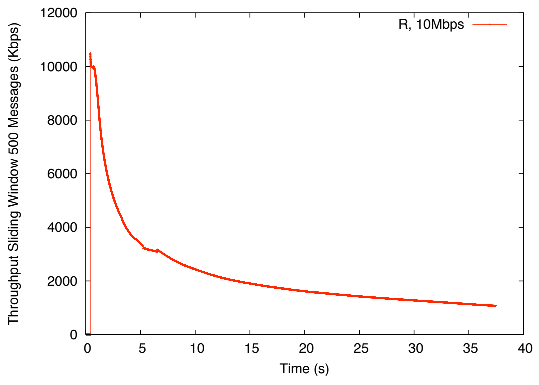


(f) Receiver (R), send rate = 1Mbps

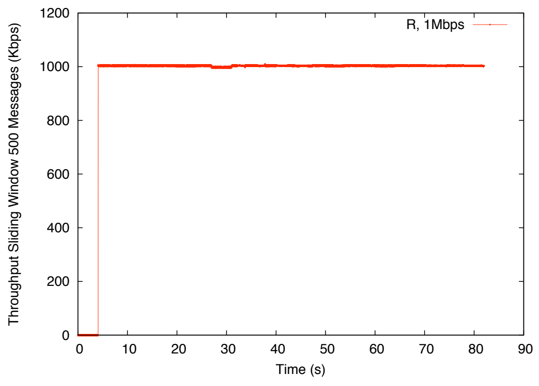
Figure 6.9: Backlog of messages buffered in MessageStore for data and control message, and the number of stored FSMs in MessageStore versus time for Hop-to-Hop Acknowledgement for the Java-based implementation without removing any MessageStore stored FSMs



(a) Java-based, send rate = 100Mbps



(b) Java-based, send rate = 10Mbps



(c) Java-based, send rate = 1Mbps

Figure 6.10: Throughput versus time for Hop-to-Hop Acknowledgement for the Java-based implementation without removing any MessageStore stored FSMs

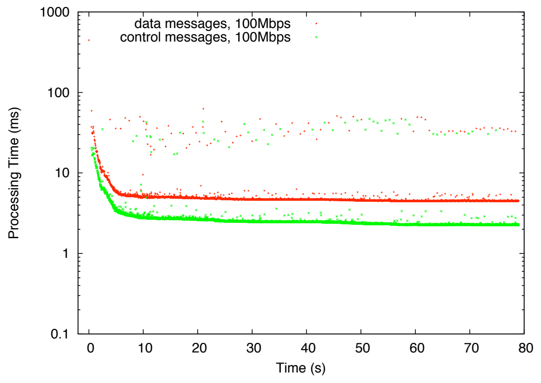
6.3.3 Bottleneck Analysis of SCXML-based Implementation

In this section we analyze the bottlenecks in the performance of the SCXML-based implementation of Hop-to-Hop Acknowledgement. We profile 1) the per message processing time (T_{msg}), 2) the MessageStore buffers (L_{data} and $L_{control}$), and 3) the number of stored FSMs (N_{FSM}) in the MessageStore. The metrics are described in Table 6.1. The bottleneck analysis is based on the same set of experiments as detailed in Section 6.3.1.

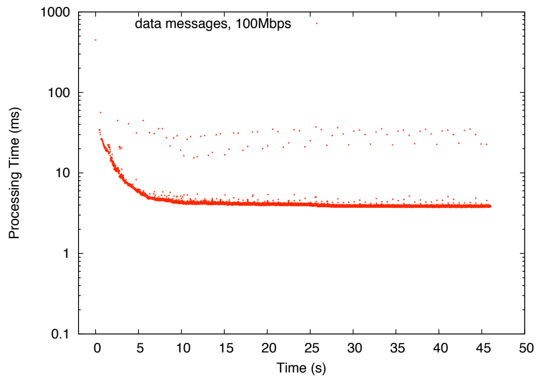
In Figure 6.11 the MessageStore per message processing time for the SCXML-based implementation of Hop-to-Hop Acknowledgement is shown for send rate of 100 Mbps, 10 Mbps, and 1 Mbps. The x-axis time in these figures is the same metric as those in Section 6.3.1 and Section 6.3.4. The left column figures (Figure 6.11(a),(c),(e)) show processing times at the Sender (S) and the right column figures (Figure 6.11(b),(d),(f)) show the processing times at the Receiver (R). The red data points represent data messages and the green data points represent control messages (in this service they are acknowledgement messages) in the same manner as Section 6.3.2.

For the same reason as in the Java-based implementation, the sender S takes longer to process messages. The control messages also take less time to process than data messages. Compared to the Java-based implementation shown in Figure 6.6, the SCXML-based implementation show a different trend for per message processing times. It takes the sender (Figures 6.11(a),(c),(e)) approximately 2.5 ms to process a control message and approximately 5 ms to process a data message. It takes the receiver (Figures 6.11(b),(d),(f)) approximately 5ms to process data messages. The message processing time is independent of the send rate. Here, at all send rates, the sender finishes processing messages at around 80s; this indicates the sustainable processing rate is around 1Mbps at the sender. Independent of the sending rate, the first message takes approximately 450 ms to process, and the initial bulk of messages before 5 s takes significant more time than the steady-state processing time reached around the 10 s mark.

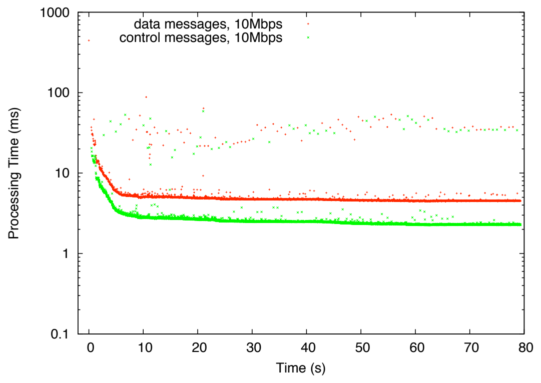
The SCXML-based implementation consists of multiple software modules as described



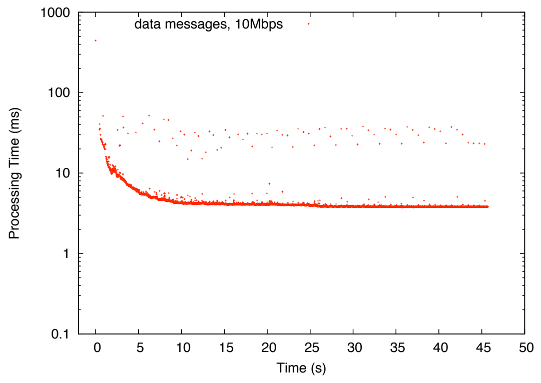
(a) Sender (S), send rate = 100Mbps



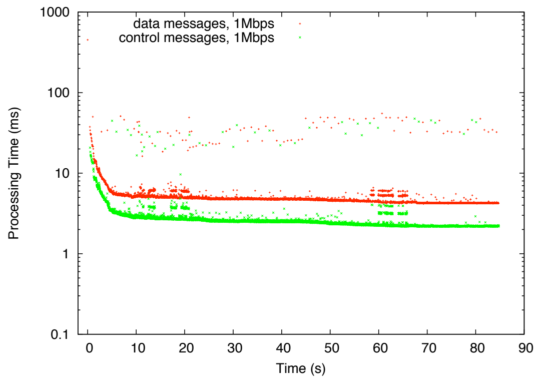
(b) Receiver (R), send rate = 100Mbps



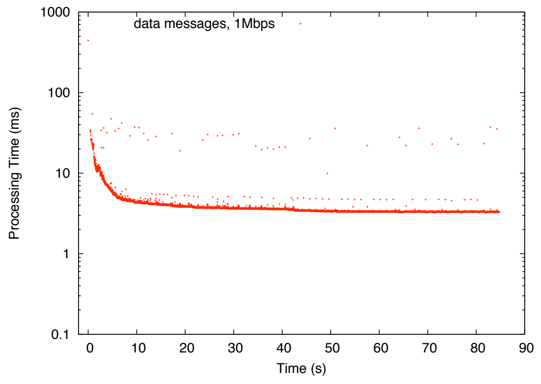
(c) Sender (S), send rate = 10Mbps



(d) Receiver (R), send rate = 10Mbps

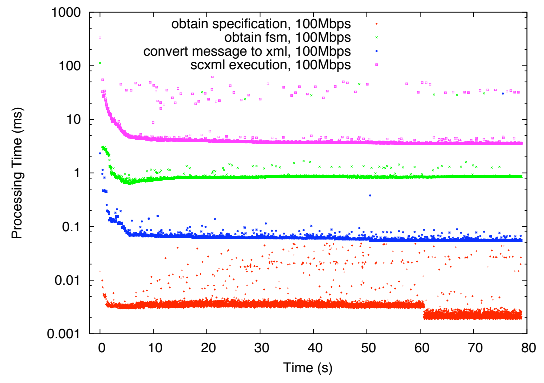


(e) Sender (S), send rate = 1Mbps

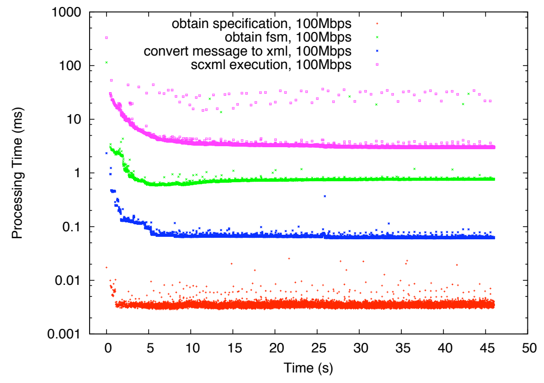


(f) Receiver (R), send rate = 1Mbps

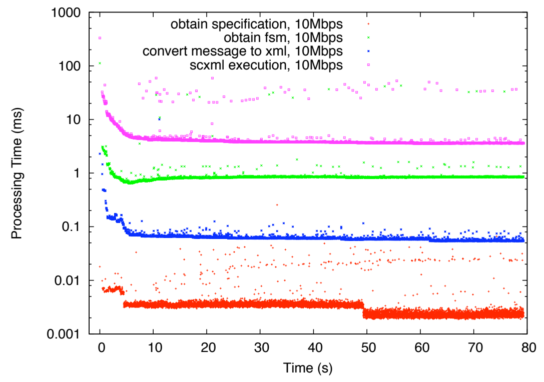
Figure 6.11: MessageStore per message processing time versus time for Hop-to-Hop Acknowledgement for the SCXML-based implementation



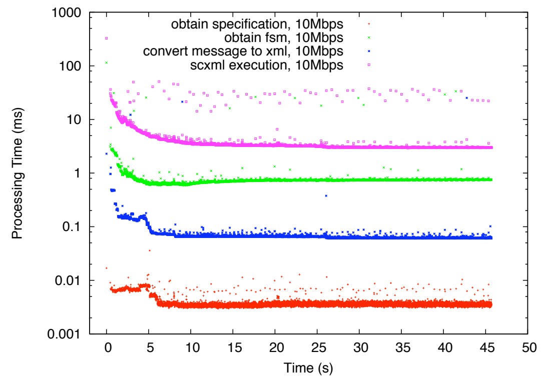
(a) Sender (S), send rate = 100Mbps



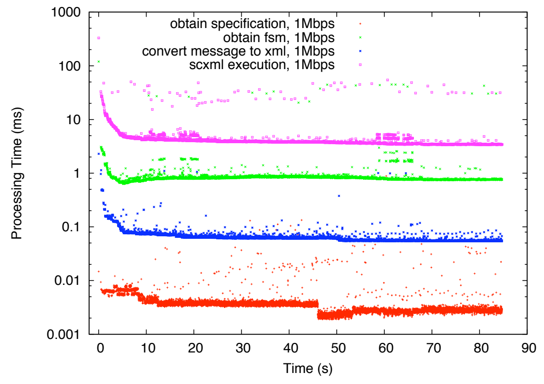
(b) Receiver (R), send rate = 100Mbps



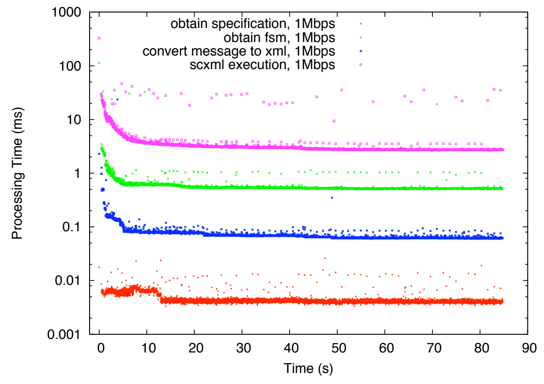
(c) Sender (S), send rate = 10Mbps



(d) Receiver (R), send rate = 10Mbps



(e) Sender (S), send rate = 1Mbps



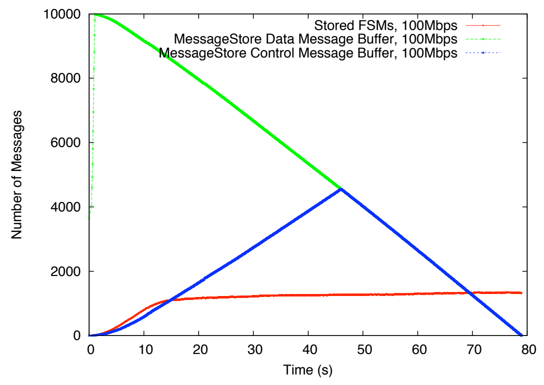
(f) Receiver (R), send rate = 1Mbps

Figure 6.12: Component breakdown of MessageStore per message processing time versus time for Hop-to-Hop Acknowledgement for the SCXML-based implementation

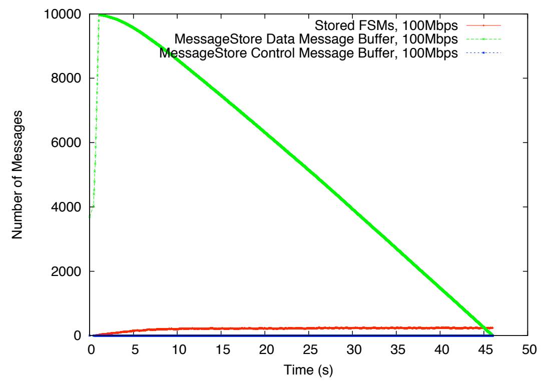
in Chapter 5. To analyze the performance of the system, we show additional timestamps for processing times of each of these modules with nanosecond precision. We show the results in Figure 6.12. These set of six figures is the same set of experiments as Section 6.3.1 and supplements Figure 6.11. The x and y axis have the same metrics and the left columns (Figure 6.12(a),(c),(e)) show processing times at the Sender (S) for data messages and the right column figures (Figure 6.12(b),(d),(f)) show the processing times at the Receiver (R) for data messages. The red data points represent the time need for MessageStore to extract the `ServiceID` of a data message and obtain the appropriate executable specification for that service. The green data points represent the amount of time MessageStore takes to obtain the FSM for the data message from the MessageStore hash table. The blue data points represent the amount of time MessageStore takes to convert the message object to an `Event` object as described in Chapter 5. The purple data points represent the amount of time spent by the Apache Commons SCXML engine to execute the event.

From the figure, we see that the message processing times is dominated by the Apache Commons SCXML engine processing. Up to 80% of processing time is consumed by the SCXML engine execution. We can conclude that the performance for the SCXML-based implementation of Hop-to-Hop Acknowledgement is limited by the performance of the SCXML execution engine. The large initial processing times measured for the SCXMl engine processing correlates to the throughput seen in Figure 6.5. The throughput is low in the beginning of an experiment and steadily increases to the steady-state value.

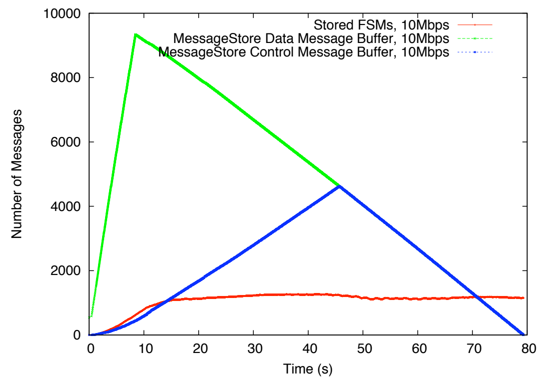
As with our analysis of the Java-based implementation, in Figure 6.13 we present the size of the MessageStore buffers (L_{data} and $L_{control}$) and the number of stored FSMs (N_{FSM}) in the MessageStore as a function of time. The figures in the left column (Figure 6.13(a),(c),(e)) show the Sender (S) and the figures in the right column (Figure 6.13(b),(d),(f)) show the Receiver (R). The red data points represent the number of stored FSMs (N_{FSM}) in the MessageStore's hash table. The green data points repre-



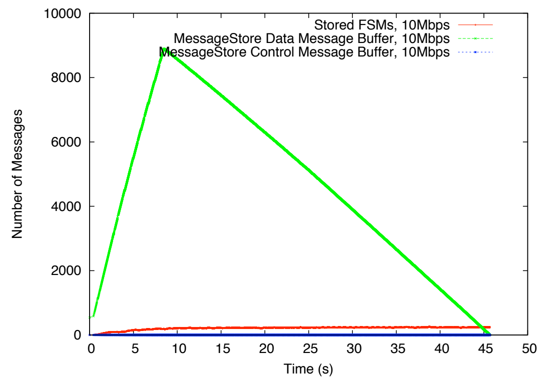
(a) Sender (S), send rate = 100Mbps



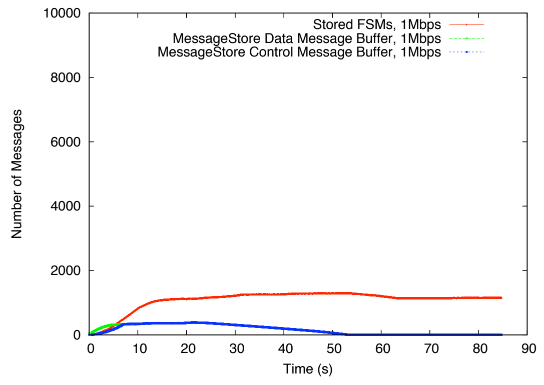
(b) Receiver (R), send rate = 100Mbps



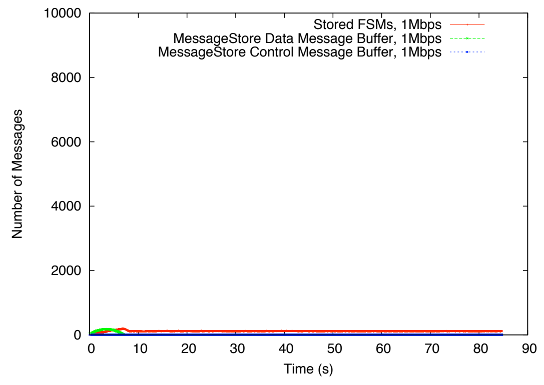
(c) Sender (S), send rate = 10Mbps



(d) Receiver (R), send rate = 10Mbps



(e) Sender (S), send rate = 1Mbps



(f) Receiver (R), send rate = 1Mbps

Figure 6.13: Backlog of messages buffered in MessageStore for data and control message, and the number of stored FSMs in MessageStore versus time for Hop-to-Hop Acknowledgement for the SCXML-based implementation

sent the number of backlog messages in the MessageStore data message buffer (L_{data}). The blue data points represent the number of backlogged messages in the MessageStore control message buffer ($L_{control}$).

As with the Java-based implementation, these plots show that at send rates of 100 Mbps and 10 Mbps, nearly all data messages are backlogged at the MessageStore buffers at both the sender and and the receiver. At a rate of 1 Mbps, there is very little backlog of data messages at the sender or receiver. The size of the backlog for the SCXML-based implementation at each time instant is larger than those seen in the Java-based implementation (Figure 6.9). This is to be expected since the SCXML-based implementation takes longer to process a message. The downslope of the green and blue curves are nearly identical in Figure 6.13(a),(c). This downslope is the rate at which the messages are being dequeued from the MessageStore data and control message buffers. It appears that this rate is bounded to an upper value of approximately 120 messages per second for the SCXML-based implementation. This is 60% of the rate of the Java-based implementation.

A point of interest in Figure 6.13(a),(c),(e) is that the red curve is nearly identical in all three send rates. This is not the case for the Java-based implementation. This means that the number of stored FSMs in MessageStore is not significantly affected by the send rate, but rather depends on the rate at which MessageStore processes these messages. This is because the time spent in the execution of the FSM by the SCXML engine is significantly longer than searching, retrieving, and updating any stored FSMs (less than an order of magnitude). This is confirmed by the Figure 6.12 which shows that the message processing time is dominated by the SCXML engine processing. Thus, in the SCXML-based implementation, the performance bottleneck is not governed by the number of stored FSMs in MessageStore as it is in the case for the Java-based implementation. In the case of the SCXML-based implementation, it appears that the pricinple bottleneck is the SCXML engine execution.

From Figure 6.11 we know that the per message processing time is initial large and reaches steady-state after about 10s. This results in the throughput patterns seen in Figure 6.5(b),(d),(f). After investigating, we determined that the cause of the large initial message processing time in the MessageStore is caused by the lazy class loading of the Java Virtual Machine described in Section 5.1.7. Since Java only loads classes on-demand, we observe an extremely large processing time for the first few messages, since many SCXML engine classes must be loaded to process these messages. It takes some time for the Java Virtual Machine to use locality to determine which classes should be cached. Eventually all required classes are cached and class-loading overhead ceases to be a major issue. In Section 6.3.4, we provide two methods to improve the large start-up overhead of the SCXML-based implementation.

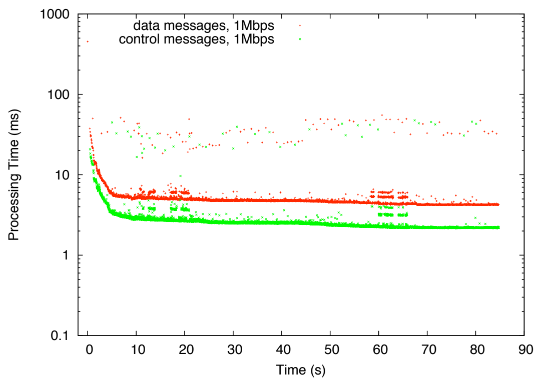
6.3.4 Performance Improvements for SCXML-Based Implementation

In this section, we present experiments that show how the performance for the SCXML-based implementation of Hop-to-Hop Acknowledgement can be improved. First, using the method "preallocation of FSMs" (Section 5.1.7.1) we reduce the class-loading overhead when creating new FSMs by using preallocated FSMs prior to the start of message transfer. This reduces the processing time to create a FSM from an executable specification. Second, we "warm-start" the SCXML engine (Section 5.1.7.2) by sending 10 dummy messages of the dummy service to force some SCXML classes to be loaded. Even though not all required SCXML classes will be loaded using this method, a significant performance improvement will be achieved.

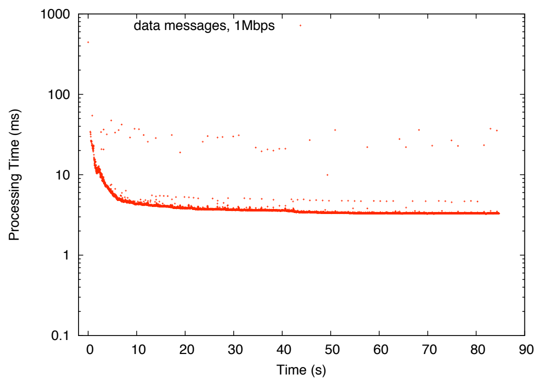
As in previous experiments, the experiment consists of transferring 10,000 overlay messages with 1,024 bytes of payload using the service Hop-to-Hop Acknowledgement from node-1 (Sender S) to node-2 (Receiver R). We use a send rate of 1 Mbps. We chose this rate since it is sustainable for the SCXML-based implementation as seen in previous experiments. Our objective is to compare the performance to a system without performance tuning. We first show measurements of the system with 0, 100, 200 preallocated FSMs and without warm-start in Figure 6.14. Then we show measurements of the system with 0, 100, 200 preallocated FSMs with warm-start in Figure 6.15. Finally we show the initialization overhead of these two mechanisms in Table 6.6.

6.3.4.1 Performance improvements using preallocation of FSMs

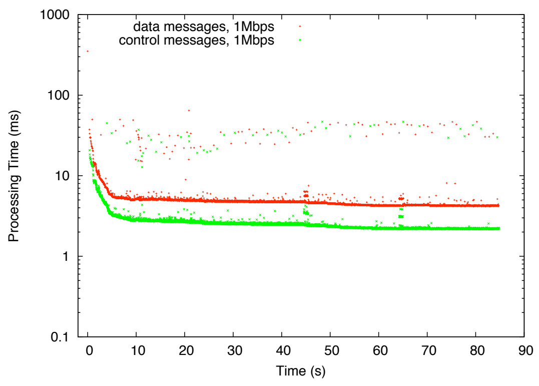
The measurements of MessageStore per message processing time as a function of time for preallocating 0, 100, and 200 FSMs without warm-start are shown in Figures 6.14. The figures in the left column (subfigures (a),(c),(e)) show processing times at the Sender (S) and the figures in the right column (subfigures (b),(d),(f)) show the processing times at



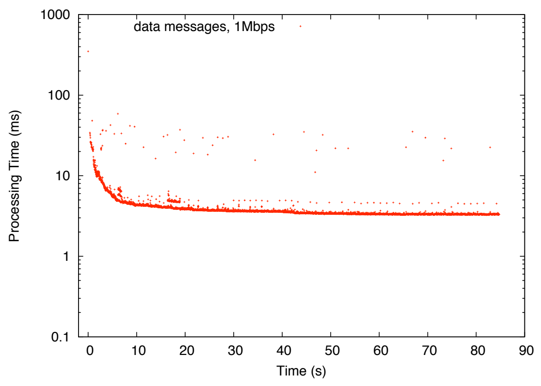
(a) Sender (S), preallocated = 0, without warm-start



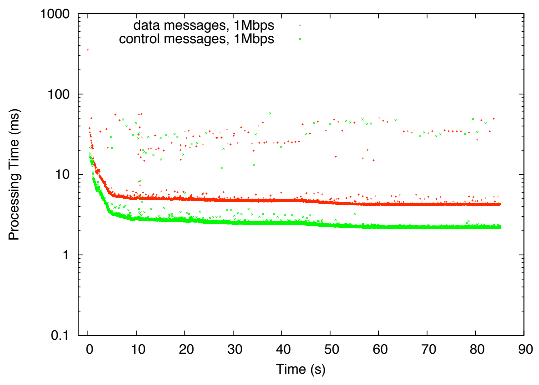
(b) Receiver (R), preallocated = 0, without warm-start



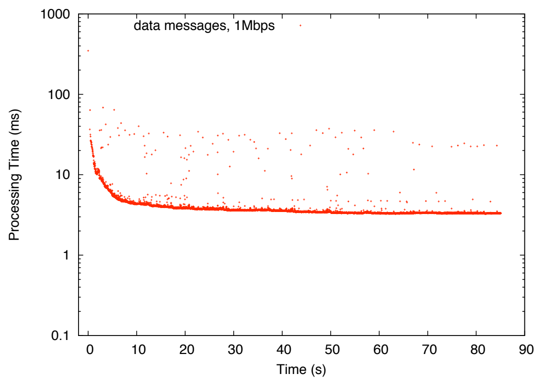
(c) Sender (S), preallocated = 100, without warm-start



(d) Receiver (R), preallocated = 100, without warm-start



(e) Sender (S), preallocated = 200, without warm-start



(f) Receiver (R), preallocated = 200, without warm-start

Figure 6.14: MessageStore per message processing time versus time for Hop-to-Hop Acknowledgement for the SCXML-based implementation with 0, 100, 200 preallocated FSMs

the Receiver (R). The red data points represent data messages and the green data points represent control messages.

Table 6.4: MessageStore per message processing time of first 5 data messages at the Sender (S) and Receiver (R) without warm-start

Pre-allocated FSMs	Message #	Sender (S)	Receiver (R)
0	1	452 ms	444 ms
	2	376 ms	342 ms
	3	341 ms	332 ms
	4	312 ms	317 ms
	5	300 ms	297 ms
100	1	352 ms	350 ms
	2	375 ms	343 ms
	3	334 ms	330 ms
	4	298 ms	314 ms
	5	299 ms	296 ms
200	1	355 ms	347 ms
	2	376 ms	366 ms
	3	334 ms	337 ms
	4	301 ms	316 ms
	5	301 ms	249 ms

We also present detailed data for the first 5 data messages. Table 6.4 show the MessageStore per message processing time of first 5 data messages at the Sender (S) and Receiver (R) for preallocating 0, 100, and 200 FSMs without warm-start.

From Figure 6.14(a),(b) and Table 6.4, we see that without preallocation of FSMs and no warm-start, the processing time of the first data message at the sender and receiver is approximately 450 ms. The processing of the next four messages steadily decreases to approximately 300 ms. With a preallocation of 100 FSMs (Figure 6.14(c),(d)) and Table 6.4, the processing time of the first data message at the sender and receiver is reduced to 350 ms. The next four messages do not show a significant decrease in processing time compared to 0 pre-allocated FSMs. Increasing to 200 preallocated FSMs (Figure 6.14(e),(f)) and Table 6.4, does not further reduce the processing time of the first data message. The preallocation of FSMs does not seem to significantly reduce processing time for data messages after the first. Also note that this mechanism does not affect control messages since in our setup, only data messages will cause a FSM to be created.

6.3.4.2 Performance improvements using warm-start of SCXML engine

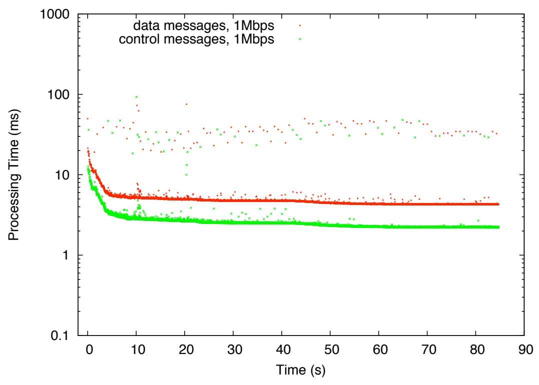
The measurements of MessageStore per message processing time as a function of time for preallocating 0, 100, and 200 FSMs with warm-start are shown in Figures 6.15. The figures in the left column (subfigures (a),(c),(e)) show processing times at the Sender (S) and the figures in the right column (subfigures (b),(d),(f)) show the processing times at the Receiver (R). The red data points represent data messages and the green data points represent control messages.

Table 6.5: MessageStore per message processing time of first 5 data messages at the Sender (S) and Receiver (R) with warm-start

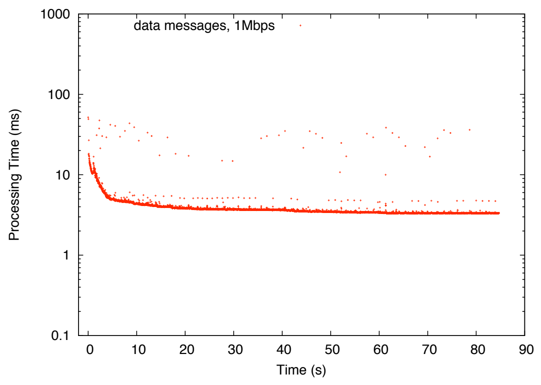
Pre-allocated FSMs	Message #	Sender (S)	Receiver (R)
0	1	49.8 ms	51.7 ms
	2	21.5 ms	49.1 ms
	3	19.0 ms	17.9 ms
	4	19.7 ms	18.3 ms
	5	19.2 ms	26.7 ms
100	1	55.0 ms	51.7 ms
	2	20.5 ms	48.2 ms
	3	18.8 ms	19.6 ms
	4	19.5 ms	19.3 ms
	5	17.9 ms	16.7 ms
200	1	54.9 ms	53.3 ms
	2	19.2 ms	17.4 ms
	3	18.8 ms	17.1 ms
	4	19.2 ms	17.2 ms
	5	18.0 ms	16.3 ms

Again, We present detailed data for the first 5 data messages. Table 6.5 show the MessageStore per message processing time of first 5 data messages at the Sender (S) and Receiver (R) for preallocating 0, 100, and 200 FSMs with warm-start.

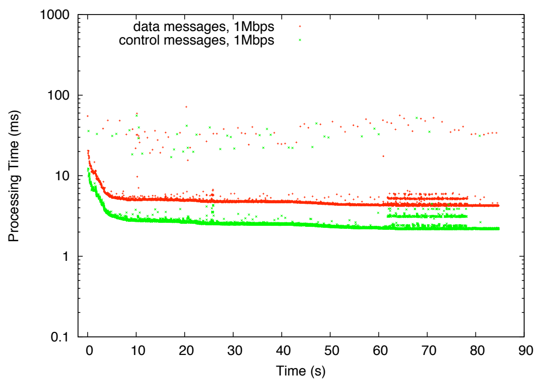
From Figure 6.15(a),(b) and Table 6.5 we see that without preallocation of FSMs and with warm-start, the processing time of the first data message at the sender and



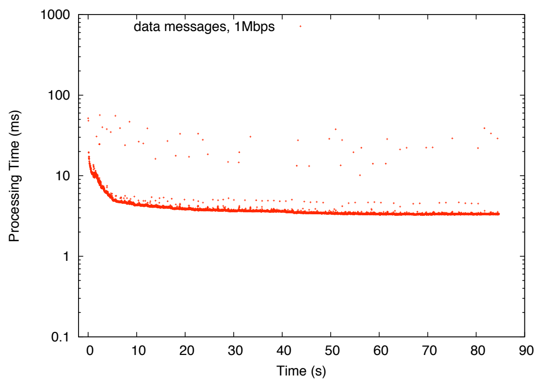
(a) Sender (S), preallocated = 0, with warm-start



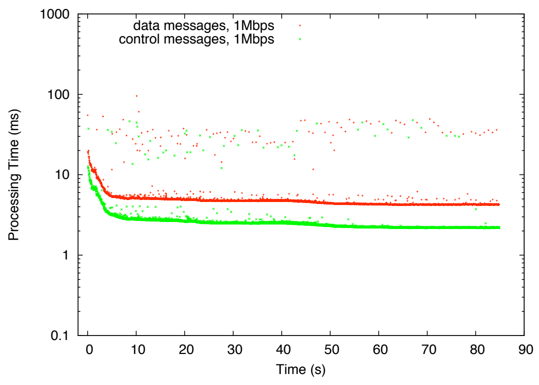
(b) Receiver (R), preallocated = 0, with warm-start



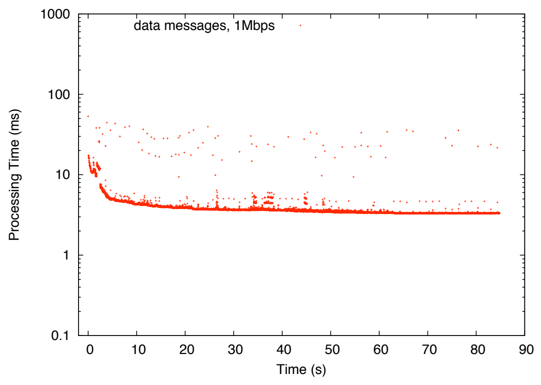
(c) Sender (S), preallocated = 100, with warm-start



(d) Receiver (R), preallocated = 100, with warm-start



(e) Sender (S), preallocated = 200, with warm-start



(f) Receiver (R), preallocated = 200, with warm-start

Figure 6.15: MessageStore per message processing time versus time for Hop-to-Hop Acknowledgement for the SCXML-based implementation with warm-start and 0, 100, 200 preallocated FSMs

receiver is approximately 50 ms. The subsequent messages also show significantly reduced processing times compared the times without warm-start. The processing of the next four messages steadily decreases to approximately 19 ms. However, a preallocation of 100 FSMs (Figure 6.15(c),(d)) or 200 FSMs (Figure 6.15(e),(f)) the does not further reduce the processing time of the first five data message's at the sender and receiver when using warm-start.

In the case of warm-start, the amount of FSMs preallocated have negligible impact on the performance of the system as seen by the nearly identical message processing times for warm-start with 0, 100, and 200 preallocated FSMs. This is because when using warm-start, the mechanism sends 10 dummy messages to the SCXML engine and FSMs must be created for those dummy messages invoking the same mechanism of FSM creation as that the preallocating FSMs mechanism. Hence, when sung warm-start, preallocating FSMs becomes redundant. In conclusion, warm-start is the mechanism we advocate improve performance.

This set of experiments validates that the large initial performance overhead for the SCXML-based implementation is primarily due to on-demand class loading by the Java Virtual Machine; whether its the FSM object or the SCXML engine.

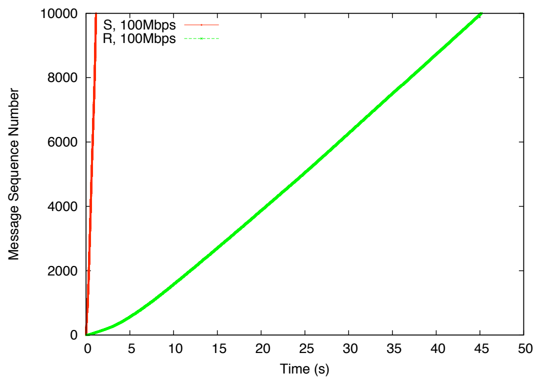
6.3.4.3 Performance at send rate of 100 Mbps, 10 Mbps, and 1 Mbps using warm-start

Next, we repeat the experiments in Section 6.3.1 using the warm-start for the SCXML-based implementation of Hop-to-Hop Acknowledgement. The experiments are identical as that of Section 6.3.1 consists of transferring 10,000 overlay messages with 1,024 bytes of payload of the service Hop-to-Hop Acknowledgement from node-1 (Sender S) to node-2 (Receiver R) with send rates of 100 Mbps, 10 Mbps, and 1 Mbps. Figures 6.16, 6.17, 6.18 are similar to Figure 6.3, 6.4, 6.5 with the same x and y axis, but run with the warm-start mechanism.

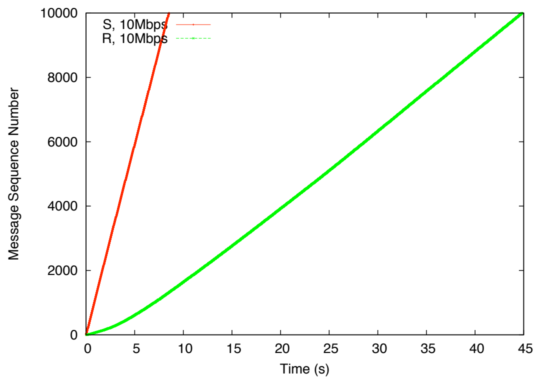
Send rates of 100 Mbps and 10 Mbps remains unsustainable as seen by the increasing delays between the sent time (red) and received time (green) at the receiver in Figure 6.16(a),(b). At the send rate of 1 Mbps (Figure 6.16(c)), we see a smaller initial delay (the difference between the sending time (S) and the received time (R)) for each message compared to the SCXML-based implementation without warm-start in Figure 6.3(f).

This is also evident in the delay plot in Figure 6.17(a) at the 1 Mbps sending rate. Compared to the system without warm-start, the initial delay (delay of messages 1-500) at the sending rate of 1 Mbps is approximately halved. This is due to the decrease in the initial delay of the system caused by warm-start as described in Section 6.3.4.2.

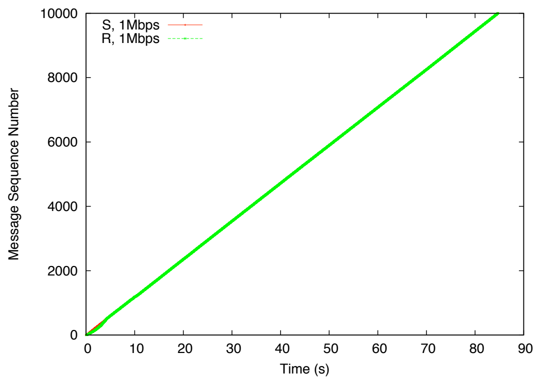
This decrease in the initial delay is also reflected in the throughput as seen in Figure 6.18. Even though at 100 Mbps and 10 Mbps (Figure 6.18(a)(b)) send rates the system is still unsustainable, we see that the first data point recorded with the 500th received message is at an earlier time compared to the system without warm-start for all send rates. At 100 Mbps send rate, Figure 6.5(b) shows the first data point slightly after 5 s while Figure 6.18(a) shows the first data point slightly before 5s. At 10 Mbps send rate, Figure 6.5(d) also shows the first data point slightly after 5 s while Figure 6.18(b) shows the first data point at approximately 4.5 s. This means that the initial 500 messages are processed faster by the receiver with warm-start. Also, the initial peak seen at the send



(a) SCXML-based, with warm-start, send rate = 100Mbps

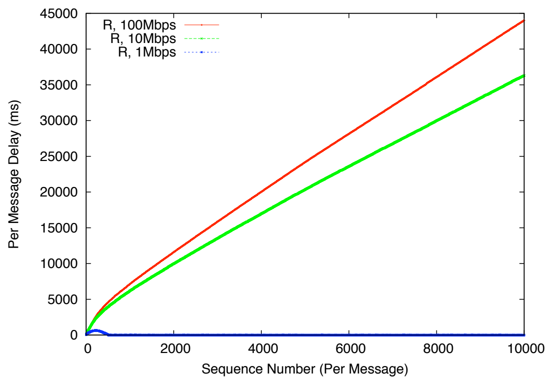


(b) SCXML-based, with warm-start, send rate = 10Mbps



(c) SCXML-based, with warm-start, send rate = 1Mbps

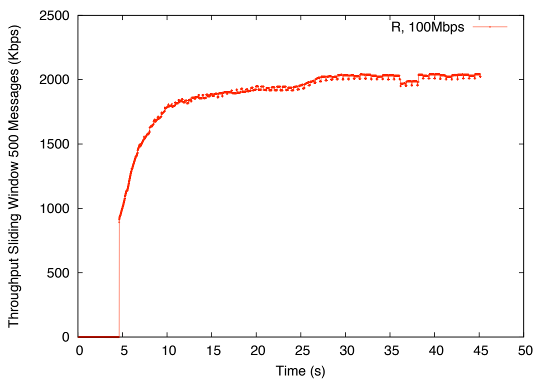
Figure 6.16: Sequence Number versus time for Hop-to-Hop Acknowledgement with warm-start for the SCXML-based implementation



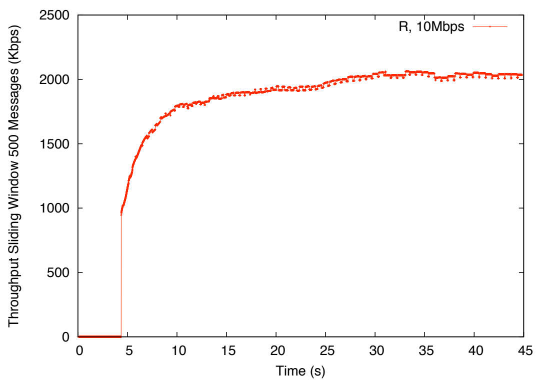
(a) SCXML-based, with warm-start

Figure 6.17: Per message delay versus time for Hop-to-Hop Acknowledgement with warm-start for the SCXML-based implementation

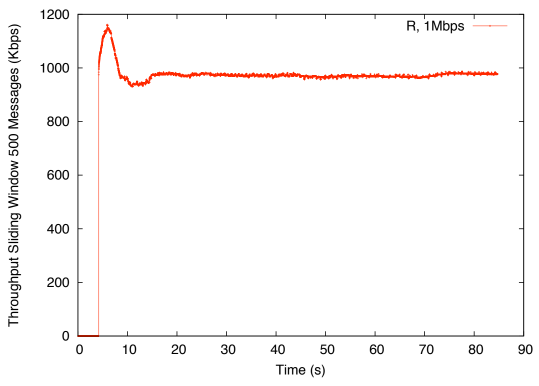
rate of 1Mbps (Figure 6.18(c)) between the 5s and 10s mark is reduced to a maximum peak value of 1.2 Mbps from the 1.4 Mbps seen in Figure 6.18(f). It is important to note that the long term steady-state throughput is unchanged with warm-start. This is expected since the warm-start mechanism only improves the initial overhead due to Java class loading. The principle bottleneck remains the SCXML engine processing.



(a) SCXML-based, with warm-start, send rate = 100Mbps



(b) SCXML-based, with warm-start, send rate = 10Mbps



(c) SCXML-based, with warm-start, send rate = 1Mbps

Figure 6.18: Throughput versus time for Hop-to-Hop Acknowledgement with warm-start for the SCXML-based implementation

6.3.4.4 Overhead of performance improving mechanisms

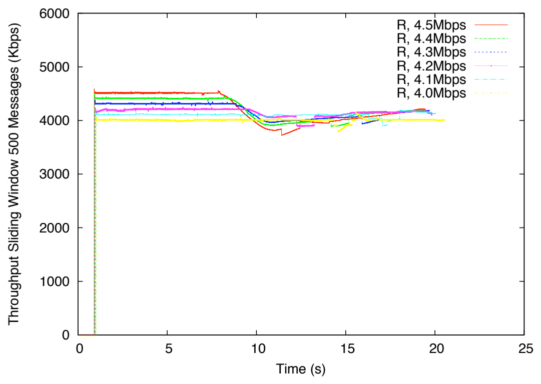
Table 6.6: Overhead of Performance Improving Mechanisms

Mechanism	Avg. Processing Time over 10 trials
preallocation of 100 FSMs	114.7 ms
warm-start of SCXML engine	763.2 ms

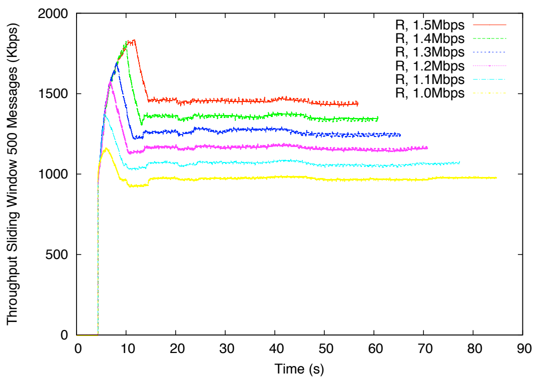
The mechanisms 1) preallocation of FSMs and 2) warm-starting the SCXML engine are performed during the initialization of MessageStore. These mechanisms introduce overhead that increases the time from when the overlay socket starts and when it finishes initialization. We used nanosecond precision timestamps to measure the time required to perform these two mechanisms. We collected data for 10 trials and average them to display in the Table 6.6. It can be seen that the warm-start of SCXML-engine consumes considerable processing time.

6.3.5 Sustainable Throughput and Delay

The next set of experiments are performed to more precisely determine the sustainable performance of the Java-based implementation and the SCXML-based implementation (with warm-start) in a single-hop transfer of 10,000 overlay messages with 1,024 bytes payloads from node-1 (Sender S) to node-2 (Receiver R) using the Hop-to-Hop Acknowledgment service. We vary the send rate close to the estimated sustainable send rate from Section 6.3.1 (approximately 4.0 Mbps for Java-based implementation and approximately 1.0 Mbps for SCXML-based implementation).



(a) Java-based

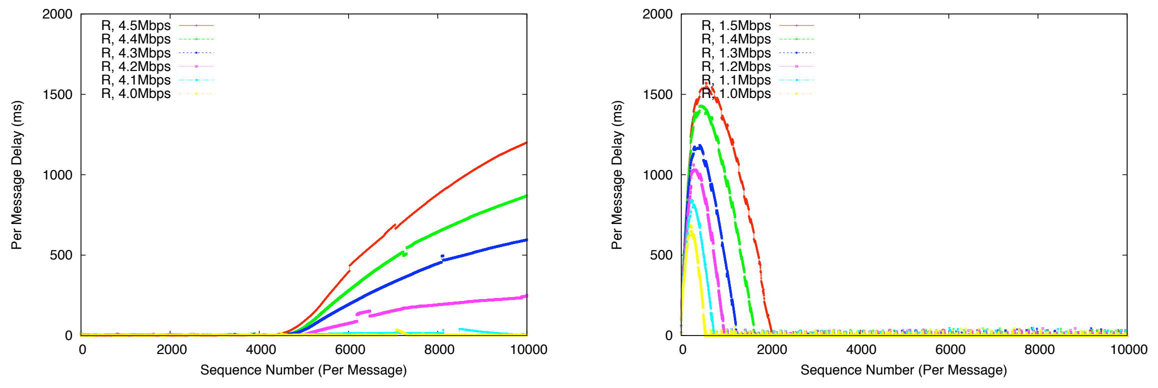


(b) SCXML-based, with warm-start

Figure 6.19: Throughput versus time for Hop-to-Hop Acknowledgement

Figure 6.19 shows that the throughput as a function of time for both the Java-based implementation (with varying send rates between 4.0 Mbps-4.5 Mbps) and the SCXML-based implementations (with varying send rates between 1.0 Mbps-1.5 Mbps). The throughput is displayed on the y-axis of the figures and is calculated according to the formula presented in Table 6.1. The first data point for throughput is calculated at the received time of the 500th message. The x-axis represents time.

Figure 6.20 shows the corresponding per message delays. The delay is the total delay from when the message is sent at the sender to when it received by the application at the receiver measured in milliseconds.



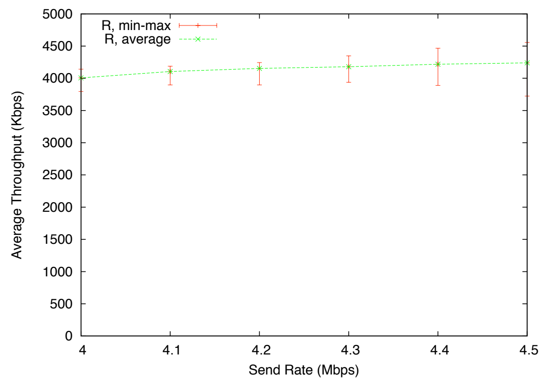
(a) Java-based (b) SCXML-based, with warm-start

Figure 6.20: Per message delay versus time for Hop-to-Hop Acknowledgement

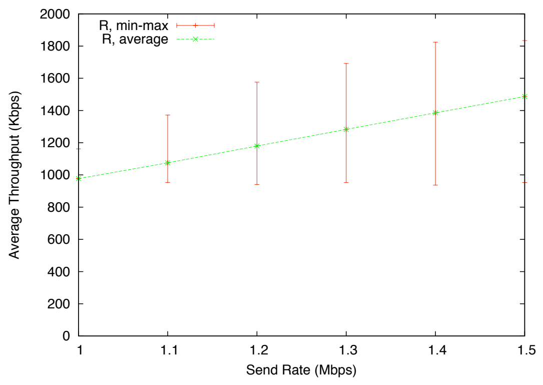
From these figure, it is evident that for the Java-based implementation, the system is sustainable at or below 4.0 Mbps. Above 4.0 Mbps, the system shows increasing per message delays. For the SCXML-based implementation, since the system is bottlenecked in the beginning, the system does not show increasing message delays over time up to 1.5Mbps. However, faster sending rates results in a larger number of initial messages experiencing very higher initial delay. For practical purposes, a sending rate of less than 1.0 Mbps is recommended as any faster sending rate results in too high of an initial performance overhead.

In Figures 6.21 and Figure 6.22, we plot the average throughput and average per message delay for the same set of experiments over all data messages with error-bars indicating the minimum and maximum value. The average delay in Figures 6.22 clearly illustrates that the sustainable performance is around 4.0 Mbps for the Java-based implementation and around 1.0 Mbps for the SCXML-based implementation. At higher rates the average delays increase sharply.

We also note that the higher the send rate, the more variability in the throughput (indicated by the greater range of the minimum and maximum throughput values) in the system for both implementations.

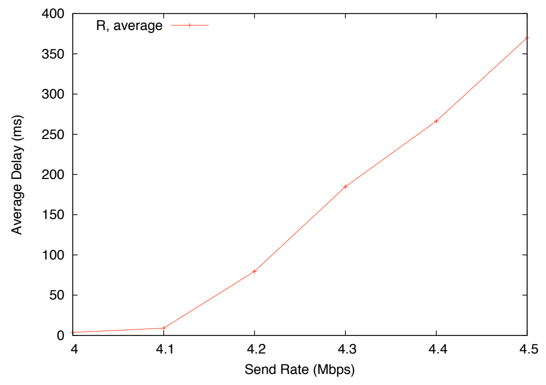


(a) Java-based

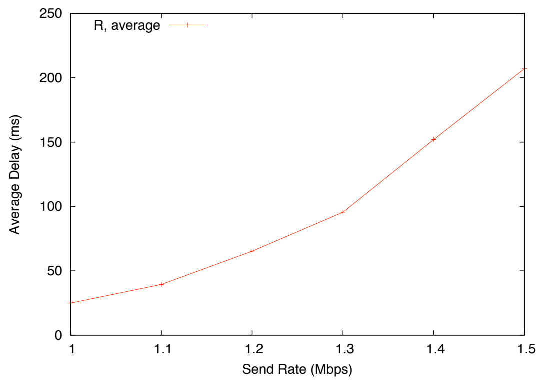


(b) SCXML-based, with warm-start

Figure 6.21: Average throughput versus time for Hop-to-Hop Acknowledgement



(a) Java-based



(b) SCXML-based, with warm-start

Figure 6.22: Average per message delay versus time for Hop-to-Hop Acknowledgement

6.4 Hop-to-Hop Acknowledgment in Multi-Hop Network

In this set of experiments, we study and compare the performance of the Java-based and the SCXML-based implementations of the Hop-to-Hop Acknowledgement service in a multi-hop scenario.

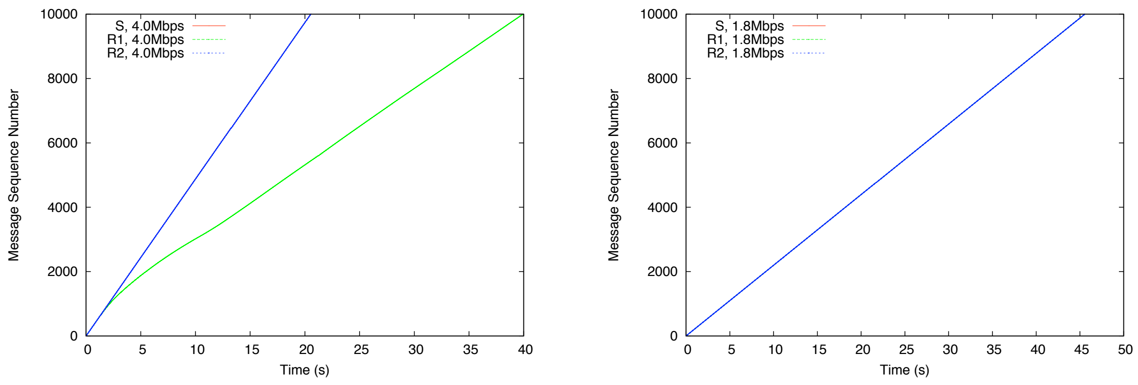
6.4.1 Two-Hop Transfer for Java-based Implementation (at send rate of 4.0 Mbps) and SCXML-Based Implementation (at send rate of 1.0 Mbps) Performance

This set of experiments consists of transferring 10,000 overlay messages with 1,024 bytes of payload of the service hop-to-hop acknowledgement from node-1 (Sender S) to node-2 (Receiver 1 $R1$) and node-3 (Receiver 2 $R2$) using overlay multicast. Due to the overlay topology settings, messages from node-1 (Sender) to node-3 (Receiver 2) must pass through the intermediate node-2 (Receiver 1), hence forming a two-hop transfer. We present similar figures as described in Section 6.3.1 for the single-hop scenario.

The send rates are the determined sustainable rates for single-hop (Section 6.3.5), *i.e.*, 4.0 Mbps for the Java-based implementation and 1.0 Mbps for the SCXML-based implementation.

Figures 6.23 and 6.24 show the overlay data message sequence numbers (seq) as a function of the time for the Java-based and SCXML-based implementations respectively. The red data points in the figures are the sent times of each sequence number of data messages at the Sender S measured in milliseconds. The green data points represent the received times ($T_{received}$) of each sequence number of data messages at Receiver 1 $R1$ measured in milliseconds. The blue data points represent the received times ($T_{received}$) of each sequence number of data messages at Receiver 2 $R2$ measured in milliseconds.

Figure 6.23(a) shows that the send rate of 4.0 Mbps, which is sustainable for single-hop overlay message transfer, is not sustainable for the two-hop scenario. The graph for node-3 (Receiver 2 $R2$) overlaps with the graph for node-1 (Sender S). However, the graph for node-2 (Receiver 1 $R1$) shows increasing per message delays as evident in the unsustainable scenario as previous explained in Section 6.3.1. Note that node-2 is the intermediate node in the overlay setup between the node-1 and node-3. Through experimentation with different send rates, we determined that the sustainable rate for the two-hop system is approximately 1.8 Mbps, which we in Figure 6.23(b).

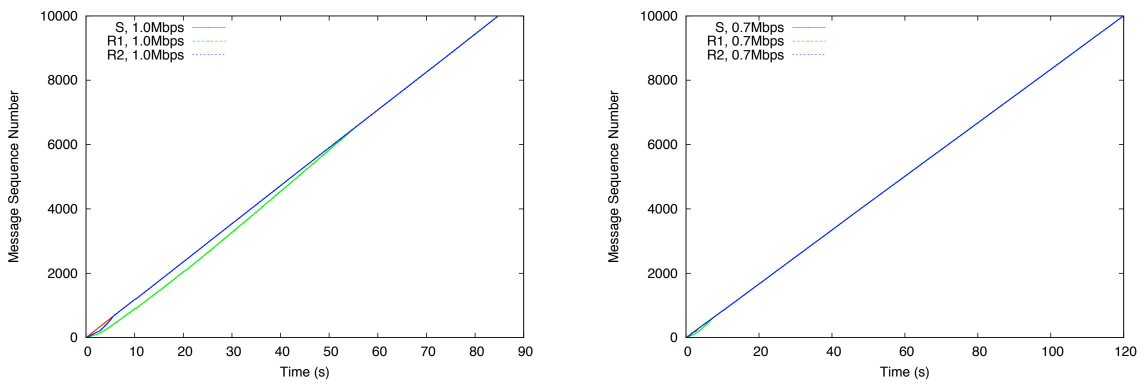


(a) Java-based, send rate = 4.0Mbps (b) Java-based, send rate = 1.8Mbps

Figure 6.23: Sequence number versus time for Hop-to-Hop Acknowledgement for the Java-based implementation

For the SCXML-based implementation, Figure 6.24(a) shows much overlap between the graphs for node-1 (Sender S), node-2 (Receiver 1 $R1$) and node-3 (Receiver 2 $R2$). Figure 6.24(a) seems to indicate that the single-hop sustainable rate of 1.0 Mbps is sustainable for the two-hop system. Node-2 experiences per message delays, however the delays do not increase over time. At a lower send rate of 0.7 Mbps shown in Figure 6.24(b), the system is sustainable and the graph for node-2 does not show per message delays.

Next we examine how delays are reflected in the throughput at the receivers. We



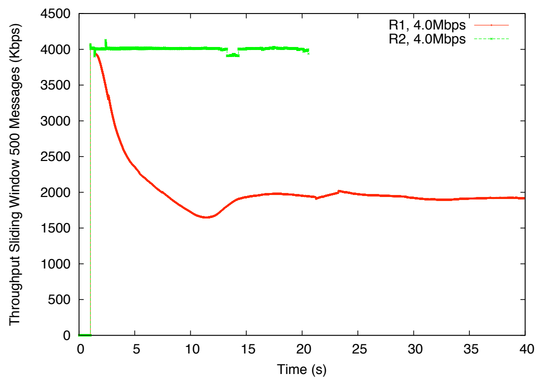
(a) SCXML-based, send rate = 1.0Mbps (b) SCXML-based, send rate = 0.7Mbps

Figure 6.24: Sequence number versus time for Hop-to-Hop Acknowledgement for the SCXML-based implementation

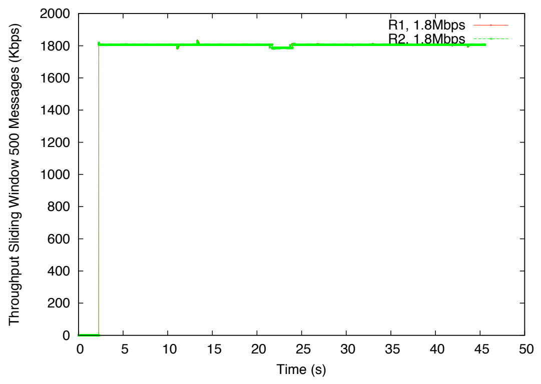
show the receiver throughput as a function of time in Figure 6.25 for both the Java-based implementation and in Figure 6.26 for the SCXML-based implementations. The *throughput* is displayed on the y-axis of the figures and is calculated according to the formula presented in Table 6.1. The first data point for throughput is calculated at the received time of the 500th message. The time is represented on the x-axis.

In Figure 6.25(a), we see that at a send rate of 4.0 Mbps for the Java-based implementation, the throughput at the intermediate node-1 (Receiver 1 $R1$) starts at 4.0 Mbps and decreases steadily to a steady-state value of approximately 1.8 Mbps. From our analysis of the single-hop scenario, we know this means node-1 cannot sustain a send rate of 4.0 Mbps. At a send rate of 1.8 Mbps, the throughput is equal to the send rate.

For the SCXML-based implementation shown in Figure 6.26(a), we see the send rate of 1.0 Mbps is sustainable to a degree (the stead-state value dos not deviate significantly from 1.0 Mbps) but there are some fluctuations in the throughput. We determined that a lower send rate of 0.7 Mbps is more sustainable for the SCXML-based implementation. This is illustrated by Figure 6.26(b) where the steady-state throughput measured at both receivers matches the send rate of 0.7 Mbps.

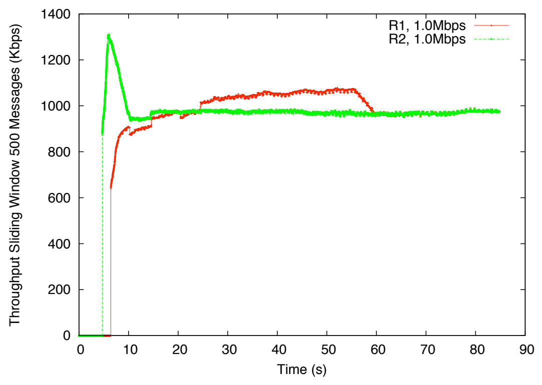


(a) Java-based, send rate = 4.0Mbps

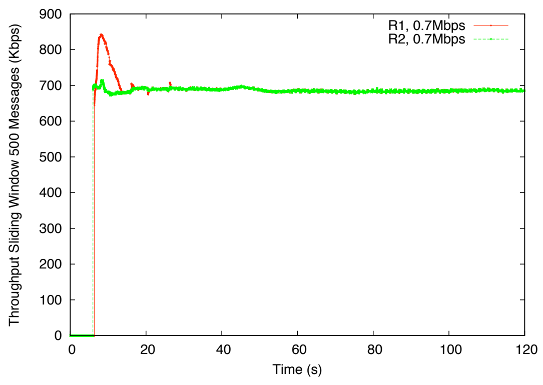


(b) Java-based, send rate = 1.8Mbps

Figure 6.25: Throughput versus time for Hop-to-Hop Acknowledgement for the Java-based implementation



(a) SCXML-based, send rate = 1.0Mbps



(b) SCXML-based, send rate = 0.7Mbps

Figure 6.26: Throughput versus time for Hop-to-Hop Acknowledgement for the SCXML-based implementation

6.4.1.1 Bottleneck Analysis of Java-based Implementation

We now determine the performance bottlenecks for Hop-to-Hop Acknowledgement service in the multi-hop setting by profiling the per message processing time (T_{msg}) as described in Table 6.1. The bottleneck analysis is based on the same set of experiments detailed in Section 6.4.1.

First we examine the Java-based implementation. In Figure 6.27 the MessageStore per message processing time for the Java-based implementation of Hop-to-Hop Acknowledgement is shown for send rates of 4.0 Mbps (subfigures (a),(c),(e)) and 1.8 Mbps (subfigures (b),(d),(f). The first row is for node-1 (Sender S), the second is for node-2 (Receiver 1 $R1$), and the third row is for node-3 (Receiver 2 $R2$). The red data points represent data messages and the green data points represent control messages (in this service they are acknowledgement messages). We notice that only node-3 does not process any control messages as it does not need to receive acknowledgement as it is a leaf node in our setup.

By comparing the results to Figure 6.6, we see that the processing time for each MessageStore message at node-1 (Sender S) and node-2 (Receiver 1 $R1$) in the two-hop scenario is almost identical to that of node-1 (Sender S) in the single-hop scenario. Whereas the processing time at node-3 (Receiver 2 $R2$) for the two-hop scenario is similar to that of node-2 (Receiver R) in the single-hop scenario. We know from our analysis for single-hop scenarios that the Java-based implementations bottlenecked by the number of stored FSMs in MessageStore. For the two-hop scenario for Hop-to-Hop Acknowledgement, acknowledgement messages are received at node-1 (Sender S) and node-2 (Receiver 1 $R1$), but no at node-3(Receiver 2 $R2$). We also know from our previous analysis that if a node does not receive acknowledgements, the messages are processed faster. Node-3 does not need to maintain a FSM in its MessageStore because it does not have to wait for an acknowledgement from a downstream node.

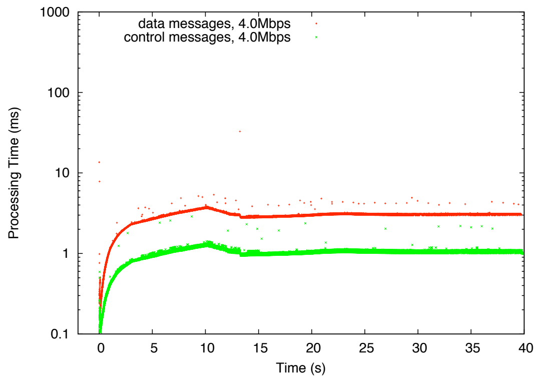
In the two-hop scenario, node-2 (Receiver 1 $R1$) is the intermediate node between the sender and receiver, hence it 1) receives data messages from the sender (node-1),

2) sends acknowledgement back to the sender (node-1), 3) forwards the message to its downstream node (node-3), and 4) waits and processes acknowledgements from its downstream node (node-3). This means the intermediate node-2 performs the most computational tasks. From this we can explain why the system could not sustain the 4.0 Mbps sustainable single-hop rate. In multi-hop scenarios, the performance of the Java-based implementation of Hop-to-Hop Acknowledgement is determined by the performance of the intermediate nodes between the sender and last receiver(s). This sustainable rate is determined previously to be approximately 1.8 Mbps.

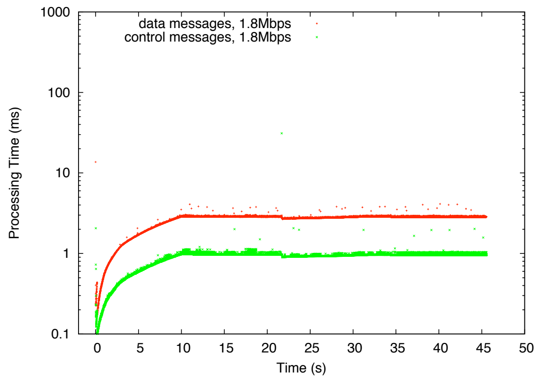
6.4.1.2 Bottleneck Analysis of SCXML-based Implementation

Next, we examine the SCXML-based implementation. In Figure 6.28 the MessageStore per message processing time for the SCXML-based implementation of Hop-to-Hop Acknowledgement is shown for send rates of 1.0Mbps (subplots (a),(c),(e)) and 0.7Mbps (subplots (b),(d),(f). The first row is for node-1 (Sender S), the second is for node-2 (Receiver 1 $R1$), and the third row is for node-3 (Receiver 2 $R2$). The red data points represent data messages and the green data points represent control messages (in this service they are acknowledgement messages).

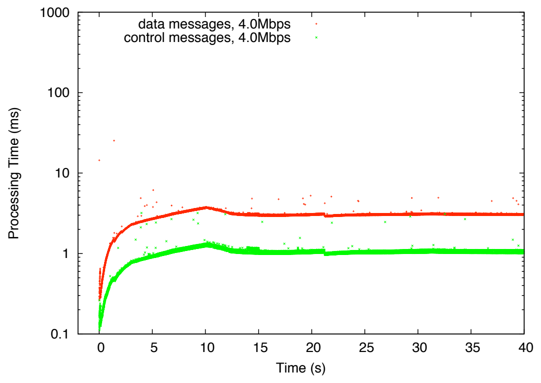
By comparing the results to Figure 6.11, we see that the processing time for each MessageStore message at all nodes in the two-hop scenario is almost identical to that of in the single-hop scenario. Recall our analysis for single-hop scenarios that the SCXML-based implementations is bottlenecked by the processing time of the SCXML engine. Hence we can see that the message processing times are not impacted significantly by the number of hops or the send rate. However, as previously explained in Section 6.4.1.1, the intermediate node-2 performs more computational tasks than the other two nodes due to the semantics of Hop-to-Hop Acknowledgement. Hence, the sustainable rate for single-hop scenario 1.0 Mbps may cause fluctuations in the throughput of the system in multi-hop scenarios as seen by Figure 6.26. Unlike the Java-based implementation,



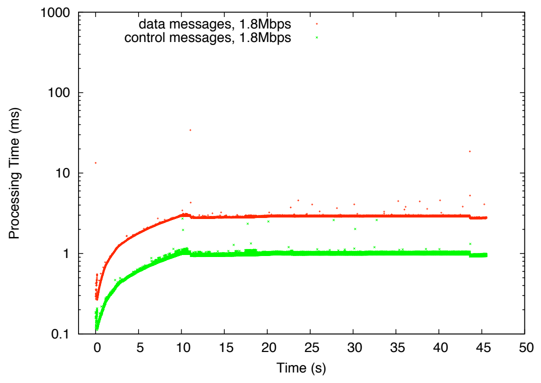
(a) Sender (S), send rate = 4.0Mbps



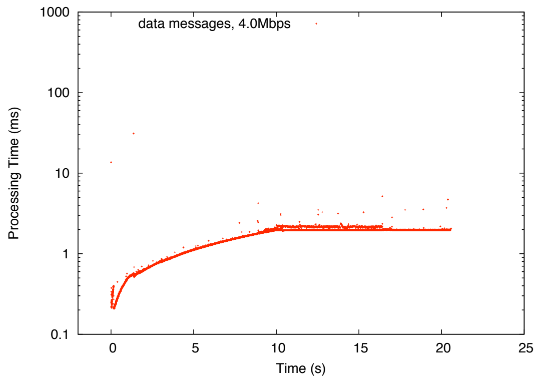
(b) Sender (S), send rate = 1.8Mbps



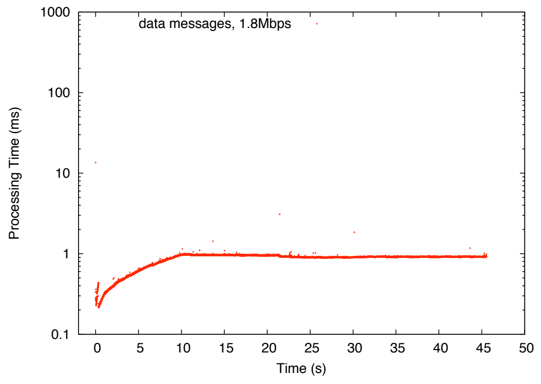
(c) Receiver 1 (R1), send rate = 4.0Mbps



(d) Receiver 1 (R1), send rate = 1.8Mbps



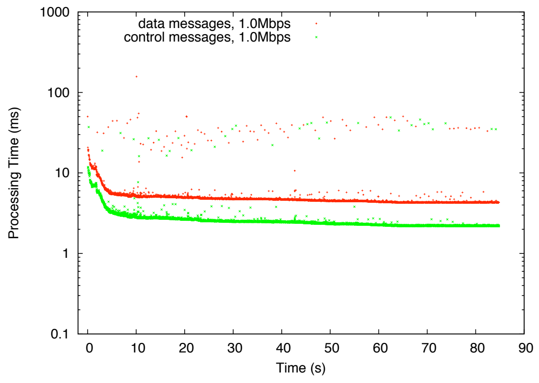
(e) Receiver 2 (R2), send rate = 4.0Mbps



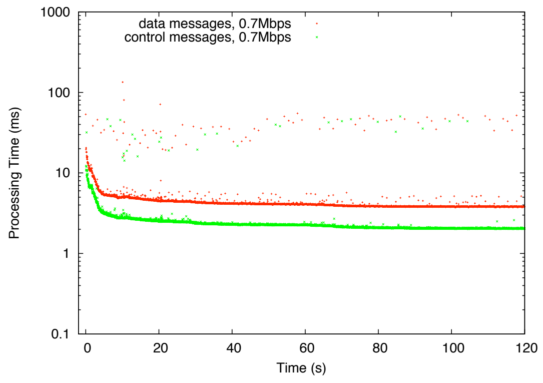
(f) Receiver 2 (R2), send rate = 1.8Mbps

Figure 6.27: MessageStore per message processing time versus time for Hop-to-Hop Acknowledgement for the Java-based implementation

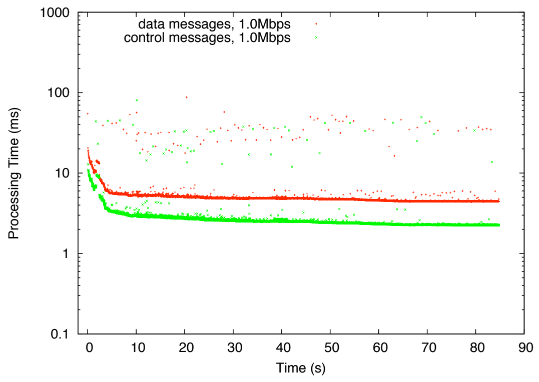
a smaller reduction in send rate from 1.0 Mbps to 0.7 Mbps (30% reduction) results in stable throughput.



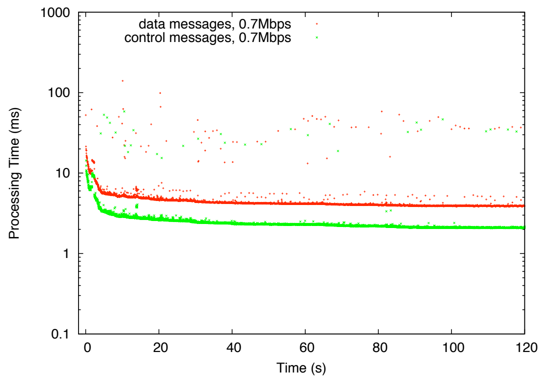
(a) Sender (S), send rate = 1.0Mbps



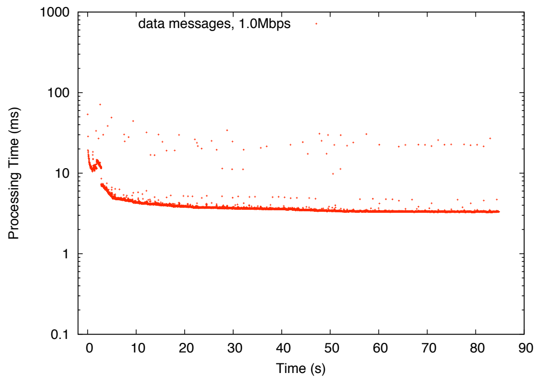
(b) Sender (S), send rate = 0.7Mbps



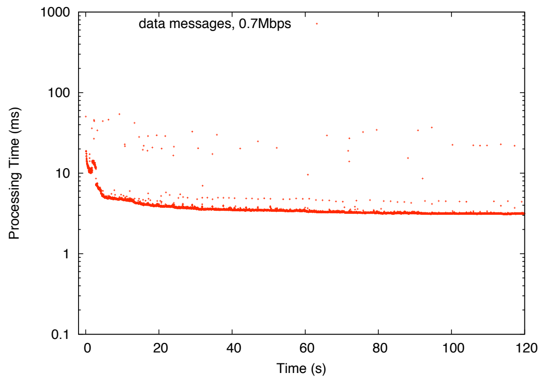
(c) Receiver 1 (R1), send rate = 1.0Mbps



(d) Receiver 1 (R1), send rate = 0.7Mbps



(e) Receiver 2 (R2), send rate = 1.0Mbps



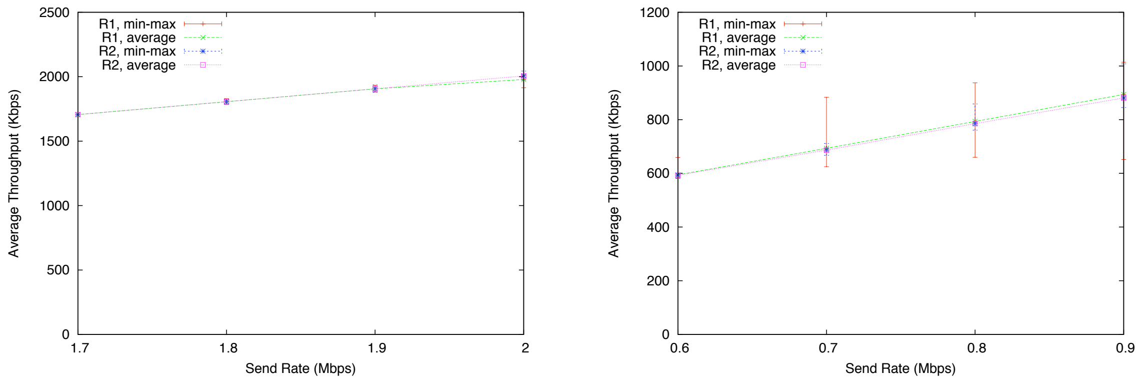
(f) Receiver 2 (R2), send rate = 0.7Mbps

Figure 6.28: MessageStore per message processing time versus time for Hop-to-Hop acknowledgement for the SCXML-based implementation

6.4.1.3 Sustainable Throughput and Delay

This set of experiments are performed to more precisely determine the sustainable performance of the Java-based and SCXML-based implementation (with warm-start) in a two-hop transfer of 10,000 overlay messages with 1,024 bytes payloads from node-1 (Sender S) to node-2 (Receiver 1 $R1$) and node-3 (Receiver 2 $R2$) using multicast and he Hop-to-Hop Acknowledgment service. We vary the send rate close to the estimated sustainable send rate from Section 6.4.1 (1.8 Mbps for the Java-based implementation and 0.7 Mbps for the SCXML-based implementation).

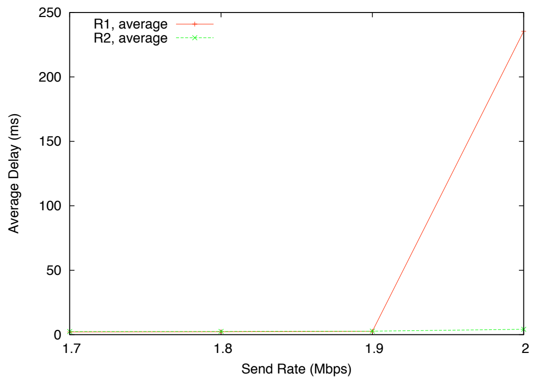
In Figures 6.29 and 6.30, we plot the average throughput and average per message delay. The average delay plot illustrates that the sustainable performance is 1.9 Mbps for Java-based Hop-to-Hop Acknowledgement and 0.7 Mbps for SCXML-based Hop-to-Hop Acknowledgement.



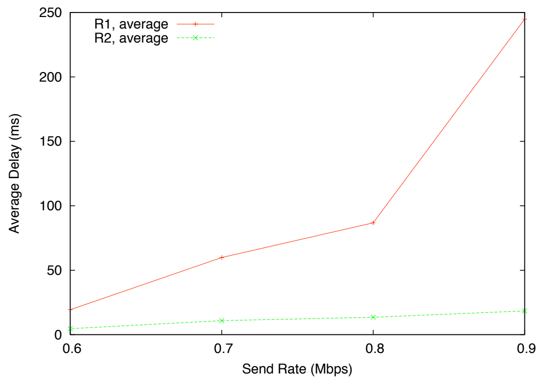
(a) Java-based (b) SCXML-based, with warm-start

Figure 6.29: Average throughput versus time for Hop-to-Hop Acknowledgement

In Figures 6.29 and 6.30, we plot the average throughput and average per message delay with error-bars indicating the minimum and maximum value. The average delay in Figures 6.30 illustrates that the sustainable performance is approximately 1.9 Mbps for the Java-based implementation and around 0.7 Mbps for the SCXML-based implementation. At rate higher than stated, the average delays increase sharply.



(a) Java-based

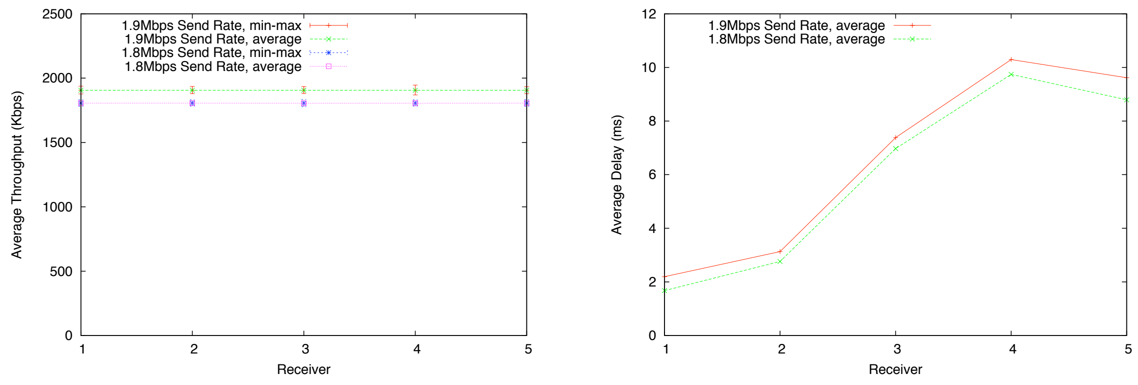


(b) SCXML-based, with warm-start

Figure 6.30: Average per message delay versus time for Hop-to-Hop Acknowledgement.

6.4.2 Five-Hop Transfer for Java-based Implementation (at send rate of 1.8 Mbps) and SCXML-Based Implementation (at send rate of 0.7 Mbps) Performance

In this set of experiments we send 10,000 overlay messages of 1,024 bytes payloads from node-1 Sender S to Receiver ($R1$, $R2$, $R3$, $R4$, and $R5$) corresponding to node-2 to node-6 in a five-hop transfer for the Java-based implementation and the SCXML-based implementation (with warm-start). We set the sending rate close to the estimated sustainable sending rate from Section 6.4.1.3 (1.8 Mbps for Java-based implementation and 0.7 Mbps for the SCXML-based implementation) to determine the sustainable performance for a five-hop message transfer.

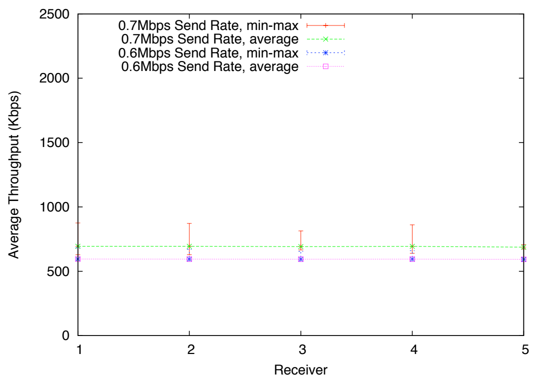


(a) Average throughput (b) Average per message delay

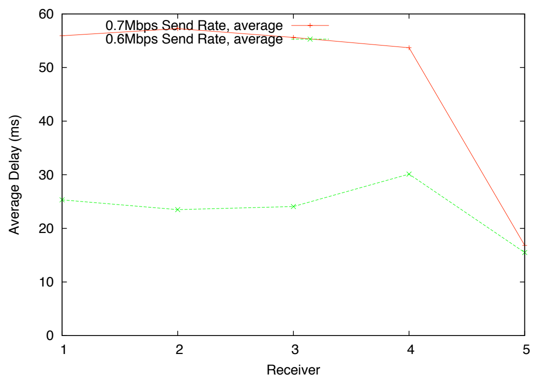
Figure 6.31: Average throughput and average delay for each receiver for Hop-to-Hop Acknowledgement using the Java-based implementation

In Figures 6.31 we plot the average throughput and average per message delay for each of the receivers for the Java-based implementation. Figure 6.32, we plot the average throughput and average per message delay for each of the receivers for the SCXML-based implementation.

In both implementations, we can see that the two-hop sustainable transfer rate is



(a) Average throughput



(b) Average per message delay

Figure 6.32: Average throughput and average delay for each receiver for Hop-to-Hop Acknowledgement using the SCXML-based implementation

also sustainable for the five-hop scenario. Hence the sustainable rate for the two-hop scenario extends to multi-hop scenarios. With the Hop-to-Hop Acknowledgement service, any intermediate nodes only need to process messages from their immediate neighbours. Therefore, the number of hops does not have an effect on the sustainable performance beyond two-hops. Note that the last node (node-6, Receiver 5 $R5$) in this setup generally has a lower per average message delay since there is less processing at the terminating/leaf node, as this node does not need to wait for acknowledgement messages as previously discussed.

6.5 End-to-End Acknowledgment in Single-Hop Network

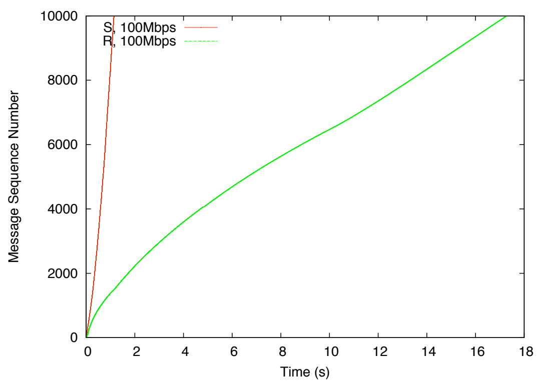
6.5.1 Java-based Implementation and SCXML-Based Implementation Performance at send rate of 100 Mbps, 10 Mbps, and 1 Mbps

In the previous sections we examined the performance of the Hop-to-Hop Acknowledgement service. We now apply the same experimental methodology to a different service, End-to-End Acknowledgement. Recall from Section 3.1.2.2, that unlike the Hop-to-Hop Acknowledgement service where acknowledgements are sent immediately to the upstream node upon receiving a data message, acknowledgements in the End-to-End Acknowledgement service are sent back to the source, where intermediate nodes aggregate all acknowledgement messages from downstream nodes.

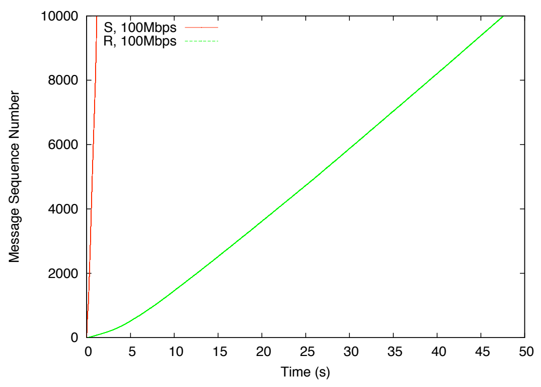
For the SCXML-based implementation of End-to-End Acknowledgement, we always use the "warm-start" mechanism. The single-hop experiments are as described in Section 6.3.1 where a single-hop transfer of 10,000 overlay messages with 1,024 bytes payloads from node-1 (Sender S) to node-2 (Receiver R) at sending rates of 100 Mbps, 10 Mbps, and 1 Mbps.

Figure 6.33 shows the overlay data message sequence numbers (seq) as a function of the time for the Java-based ((a),(c),(e)) and the SCXML-based ((b),(d),(f)) implementations at send rates of 100 Mbps, 10 Mbps, and 1 Mbps. The red data points in the figures are the sent times of each sequence number of data messages at the Sender S measured in milliseconds. The green data points represent the received times of each sequence number of data messages at the Receiver R measured in milliseconds.

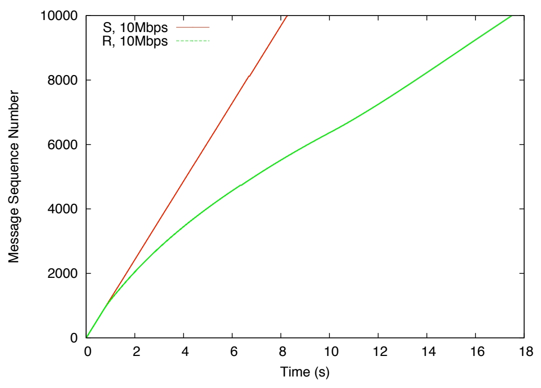
Figure 6.33(a),(c),(e) and Figure 6.34(a) for End-to-End Acknowledgement service look similar to Figure 6.3(a),(c),(e) and Figure 6.4(a) for Hop-to-Hop Acknowledgement



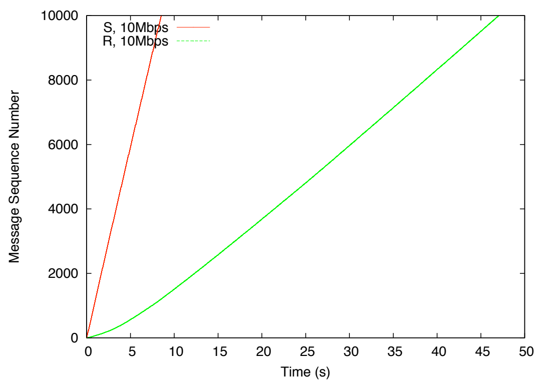
(a) Java-based, send rate = 100 Mbps



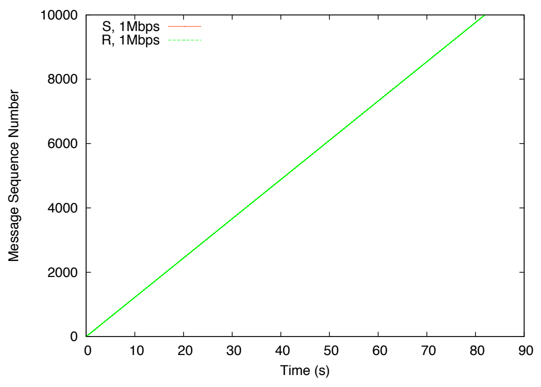
(b) SCXML-based, send rate = 100 Mbps, with warm-start



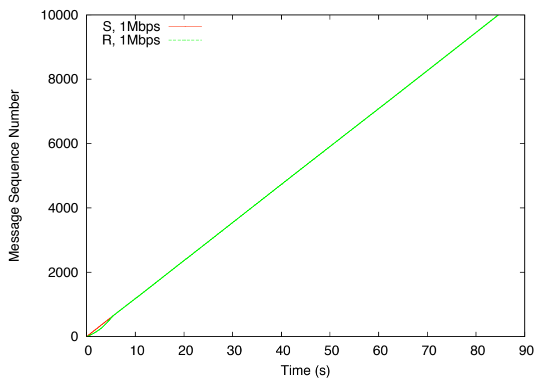
(c) Java-based, send rate = 10 Mbps



(d) SCXML-based, send rate = 10 Mbps, with warm-start



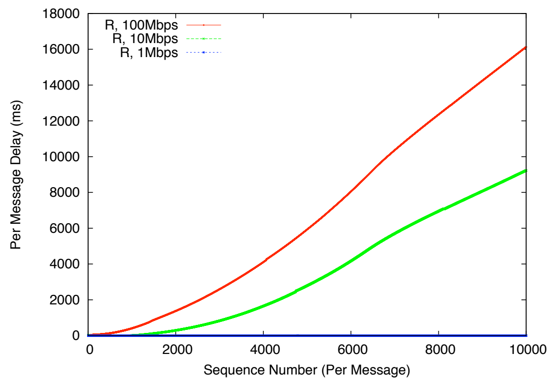
(e) Java-based, send rate = 1 Mbps



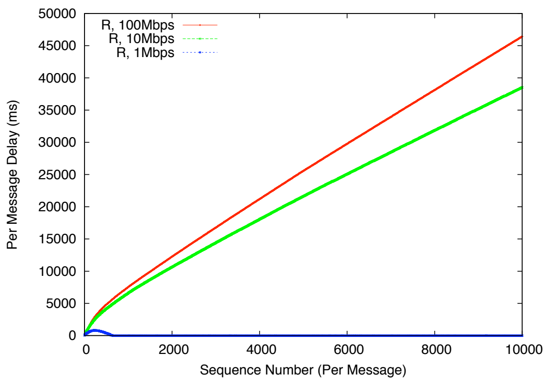
(f) SCXML-based, send rate = 1 Mbps, with warm-start

Figure 6.33: Sequence number versus time for End-to-End Acknowledgement

service (Java-based implementation). Figure 6.33(b),(d),(f) and Figure 6.34(b) for End-to-End Acknowledgement service look similar to Figure 6.16(a),(b),(c) and Figure 6.17(a) for Hop-to-Hop Acknowledgement service (SCXML-based implementation with warm-start). The performance characteristic for both services are very similar. End-to-End Acknowledgement service also cannot sustain send rates of 100 Mbps and 10 Mbps.



(a) Java-based



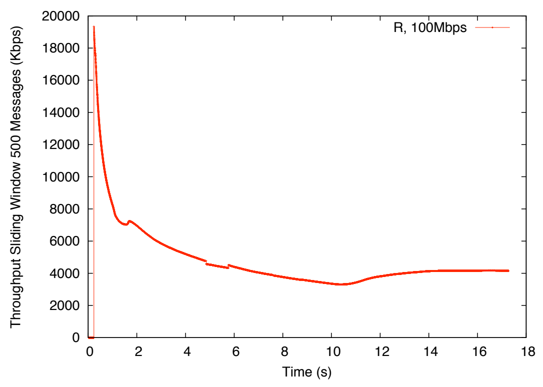
(b) SCXML-based, with warm-start

Figure 6.34: Per message delay versus time for End-to-End Acknowledgement

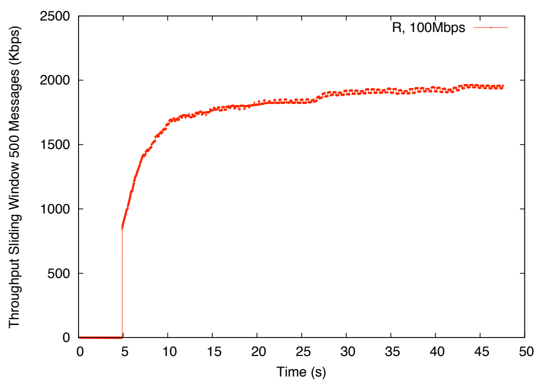
Figure 6.34 shows the per message delay as a function of the sequence number for data messages for the Java-based and the SCXML-based implementations. The delay is the total delay from when the message is sent at the sender to when it received by the application at the receiver measured in milliseconds. The red, green, and blue data points represent sending rates of 100 Mbps, 10 Mbps, and 1 Mbps respectively.

Figure 6.35 show receiver throughput as a function of time for the Java-based ((a),(c),(e)) and the SCXML-based ((b),(d),(f)) implementations with send rates of 100 Mbps, 10 Mbps, and 1 Mbps. The first data point for throughput is calculated at the received time of the 500th message. Time is represented on the x-axis.

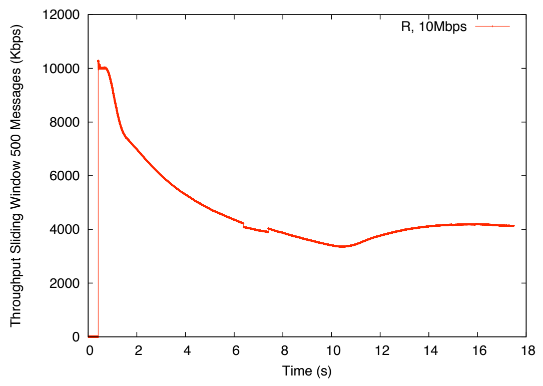
The throughput plots also show nearly identical trends as for Hop-to-Hop Acknowledgement service (Figure 6.5(a),(c),(e) for Java-based implementation and Figure 6.18(a),(b),(c) for the SCXML-based implementation with warm-start). From the throughput plots, we can approximate the sustainable rate of 4.0 Mbps for the Java-based implementation and 1.0 Mbps for the SCXML-based implementation for the End-to-End Acknowledgement service. These are approximately the same values determined for Hop-to-Hop Acknowledgement service in Section 6.3.1.



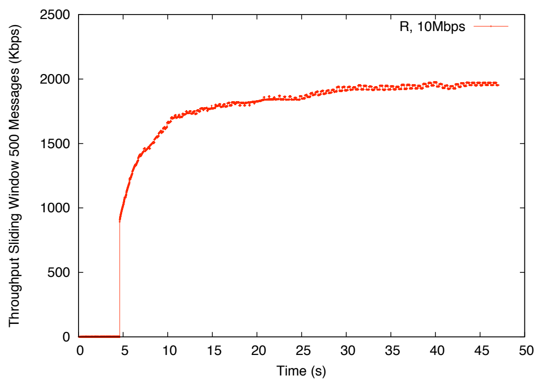
(a) Java-based, send rate = 100Mbps



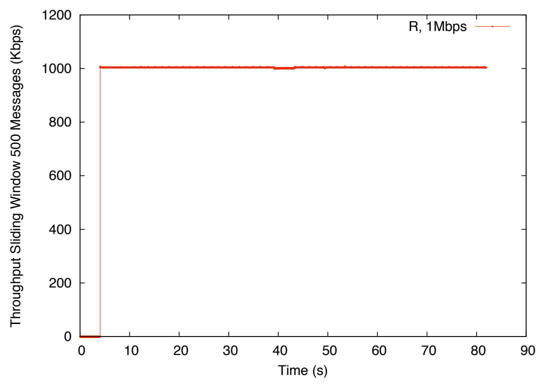
(b) SCXML-based, send rate = 100Mbps, with warm-start



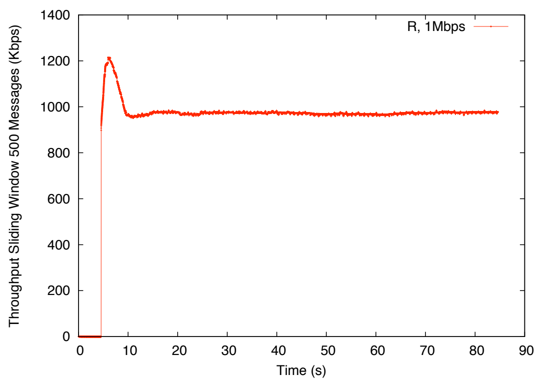
(c) Java-based, send rate = 10Mbps



(d) SCXML-based, send rate = 10Mbps, with warm-start



(e) Java-based, send rate = 1Mbps



(f) SCXML-based, send rate = 1Mbps, with warm-start

Figure 6.35: Throughput versus time for End-to-End Acknowledgement

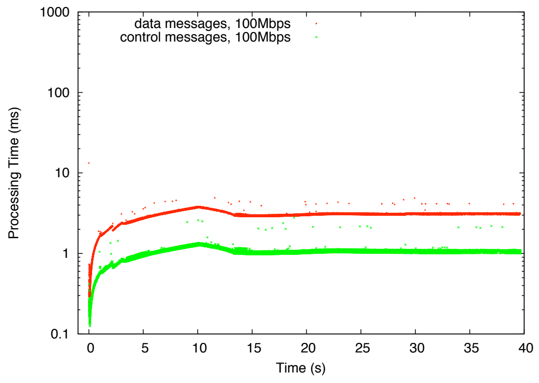
6.5.2 Bottleneck Analysis of Java-based Implementation

Just like Hop-to-Hop Acknowledgement, we now determine the performance bottlenecks for End-to-End Acknowledgement service by profiling 1) the per message processing time (T_{msg}), 2) the MessageStore buffers (L_{data} and $L_{control}$) and 3) the number of stored FSMs (N_{FSM}) in the MessageStore. The bottleneck analysis is based on the same set of experiments detailed in Section 6.5.1.

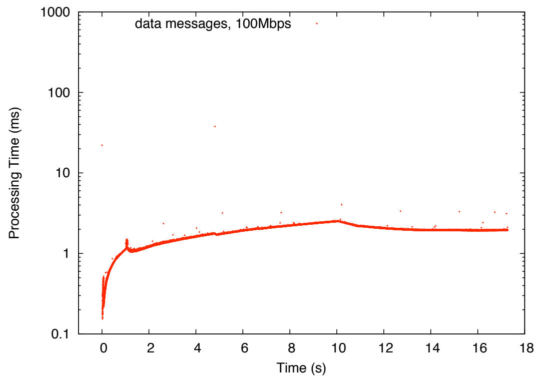
First we examine the Java-based implementation. In Figure 6.36 the MessageStore per message processing time for the Java-based implementation of End-to-End Acknowledgement is shown for send rates of 100 Mbps, 10 Mbps, and 1 Mbps. The figures in the left column (Figure 6.36(a),(c),(e)) show processing times at the Sender (S) and the figures in the right column(Figure 6.36(b),(d),(f)) show the processing times at the Receiver (R). The red data points represent data messages and the green data points represent control messages (acknowledgement messages). The receiver does not process any control messages.

We observe just like Hop-to-Hop Acknowledgement service, in End-to-End Acknowledgement service the control messages take less time to process than data messages. On average, it takes MessageStore, at steady-state, approximately 0.5 ms - 1 ms to process a control message and 1.5 ms - 2 ms to process a data messages. Again, processing of data messages at the sender take more time than at the receiver since the sender also receives control messages. The processing times for unsustainable send rates (10 Mbps and 100 Mbps) is also longer than for the sustainable send rate (1 Mbps). The general pattern in these figure exhibit that the MessageStore processing time starts at approximately 0.1 ms and steadily increases to the steady-state value around the 10s mark for both the receiver and senders. This correlates directly to the higher initial throughput seen in Figure 6.35(a),(c) and the receiver reaching the steady-state throughput at 10 s.

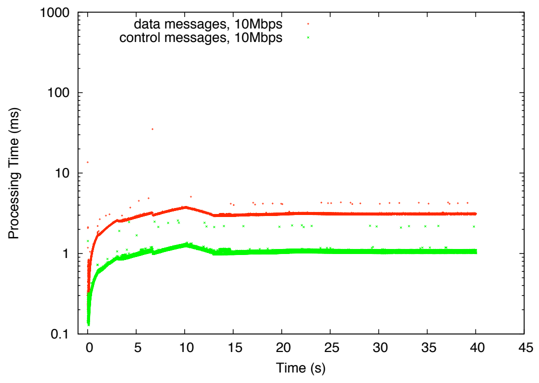
For the Java-based implementation, Figure 6.36 look nearly identical to that of the Hop-to-Hop Acknowledgement service (FIgure 6.6). This is expected, as the performance



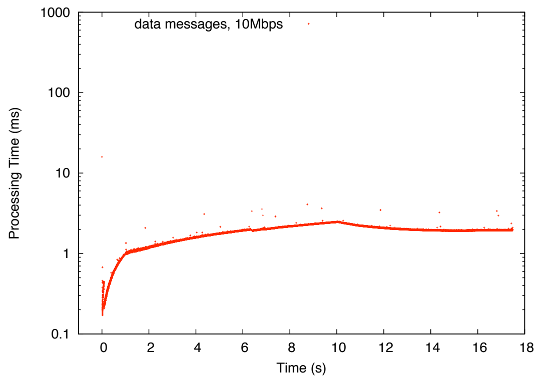
(a) Sender (S), send rate = 100Mbps



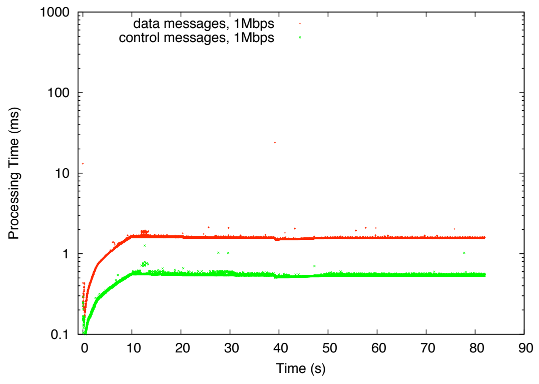
(b) Receiver (R), send rate = 100Mbps



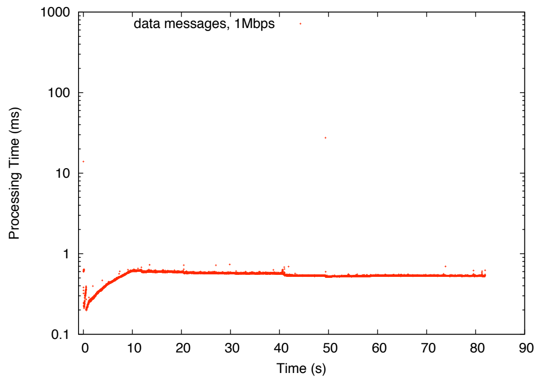
(c) Sender (S), send rate = 10Mbps



(d) Receiver (R), send rate = 10Mbps



(e) Sender (S), send rate = 1Mbps



(f) Receiver (R), send rate = 1Mbps

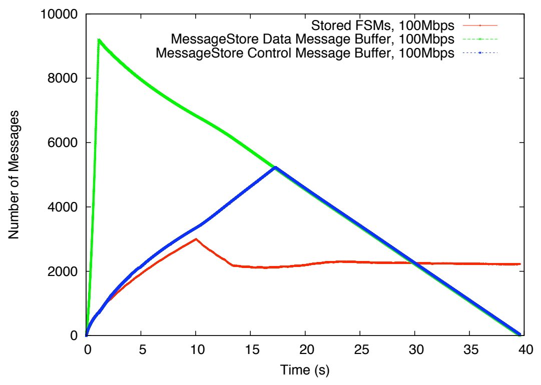
Figure 6.36: MessageStore per message processing time versus time for End-to-End Acknowledgement of the Java-based implementation

should not depend on the individual services.

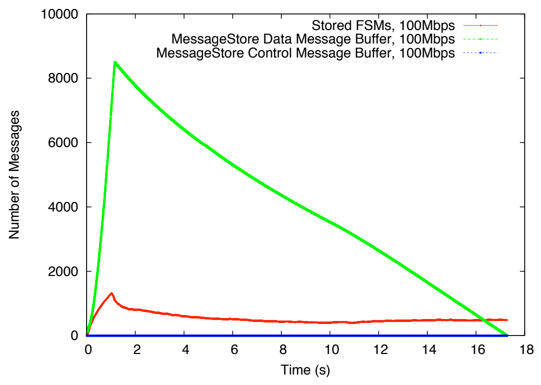
In Figure 6.37 we present the size of the MessageStore buffers (L_{data} and $L_{control}$) and the number of stored FSMs (N_{FSM}) in the MessageStore as a function of time. The figures on the left column (Figure 6.7(a),(c),(e)) show the Sender (S) and the figures on the right column (Figure 6.7(b),(d),(f)) show the Receiver (R). The red data points represent the number of stored FSMs (N_{FSM}) in the MessageStore's hash table. The green data points represent the number of backlogged messages in the MessageStore data message buffer (L_{data}). The blue data points represent the number of backlogged messages in the MessageStore control message buffer ($L_{control}$).

These plots show that at send rates of 100 Mbps and 10 Mbps, a large backlog of data messages is built up at both the sender and and the receiver. At a rate of 100 Mbps, almost all of 10,000 data messages are buffered before being processed since MessageStore cannot process messages at that rate. A the lower rate of 10 Mbps less than 50% of the data messages are buffered. At a rate of 1 Mbps, there is no backlog of data messages at the sender or receiver. For control messages only received at the sender, the plots show a large backlog of messages in the buffer for unsustainable send rate of 100 Mbps and 10 Mbps and no backlog of messages at a send rate of 1Mbps. The rate at which the messages are being dequeued from the MessageStore data and control message buffers and is approximately bounded to an upper value of approximately 200 messages per second.

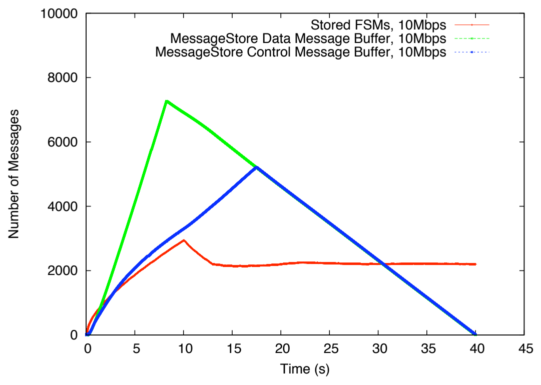
From the red data points, the number of stored FSMs (N_{FSM}) in the MessageStore, there appears to be a correlation between the number of stored FSMs (Figure 6.37) and the processing time for each message (Figure 6.36) for both the sender and receiver. The processing time increase with increasing number of stored FSMs. For the senders, the number of stored FSMs reaches a maximum at around 10 s seen in Figures 6.37(a),(c),(e) and achieves steady-state thereafter. This is reflected in the message processing times, for both data and control messages, seen in Figures 6.36(a),(c),(e). For the receivers, the number of stored FSMs reaches a maximum at around 1 s seen in Figure 6.37(b),(d),(f)



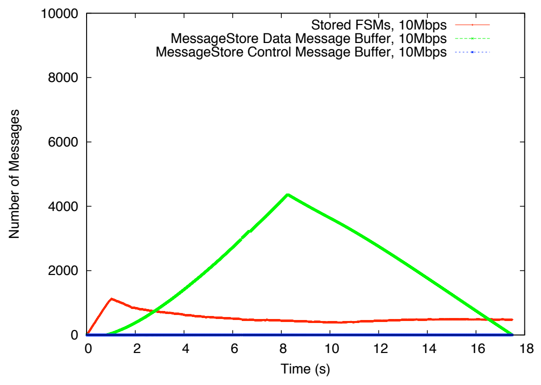
(a) Sender (S), send rate = 100Mbps



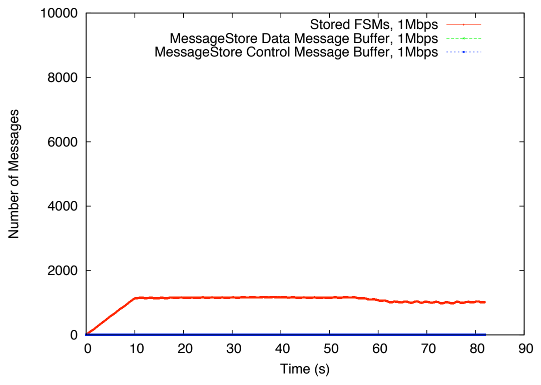
(b) Receiver (R), send rate = 100Mbps



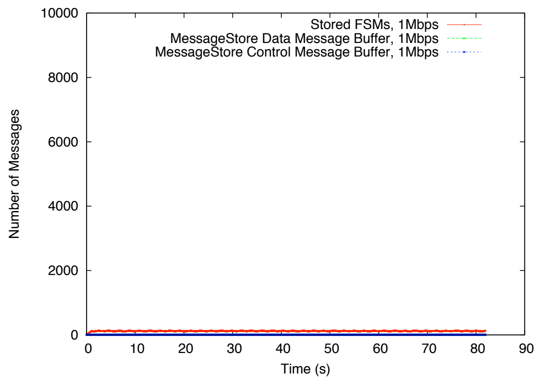
(c) Sender (S), send rate = 10Mbps



(d) Receiver (R), send rate = 10Mbps



(e) Sender (S), send rate = 1Mbps



(f) Receiver (R), send rate = 1Mbps

Figure 6.37: Backlog of messages buffered in MessageStore for data and control message, and the number of stored FSMs in MessageStore versus time for End-to-End Acknowledgement for the Java-based implementation

and achieves steady-state thereafter. At the senders, the FSMs gets removed when acknowledgement messages corresponding to the data messages are received. The FSMs then reaches its final state and are removed after a timer expires ($Timeout_{Delete}$), this timer is set at 10 s as described in Section 6.2.2. Hence, steady-state is reached at around 10s.

This trend is also reflected in the message processing times of data messages, seen in Figures 6.36(b),(d),(f) where the processing time steadily increases until around the 1 s-1.5 s mark where steady-state is reached. The receivers do not wait for acknowledgement messages and terminates as soon as the FSMs reaches its final state. Hence steady-state is reached much earlier. Steady-state is not reached immediately since there is a large per message delay initially as shown in Figure 6.34.

From the nearly identical performance behaviour of Hop-to-Hop Acknowledgement service (Section 6.3.2.1) and End-to-End Acknowledgement service, we conclude that the primary performance overhead for the Java-based implementation is the number of stored FSMs in the MessageStore.

6.5.3 Bottleneck Analysis of SCXML-based Implementation

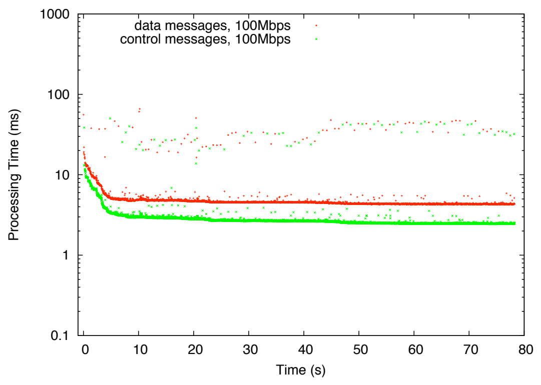
In this section we analyze the bottlenecks in the performance of the SCXML-based implementation of End-to-End Acknowledgement. We profile 1) the per message processing time (T_{msg}), 2) lthe MessageStore buffers (L_{data} and $L_{control}$) and 3) the number of stored FSMs (N_{FSM}) in the MessageStore. The bottleneck analysis is based on the same set of experiments detailed in Section 6.5.1.

In Figure 6.38 the MessageStore per message processing time for the SCXML-based implementation of End-to-End Acknowledgement is shown for send rate of 100 Mbps, 10 Mbps, and 1 Mbps. The figures on the left column (Figure 6.38(a),(c),(e)) show processing times at the Sender (S) and the figures on the right column (Figure 6.38(b),(d),(f)) show the processing times at the Receiver (R). The red data points represent data messages and the green data points represent control messages.

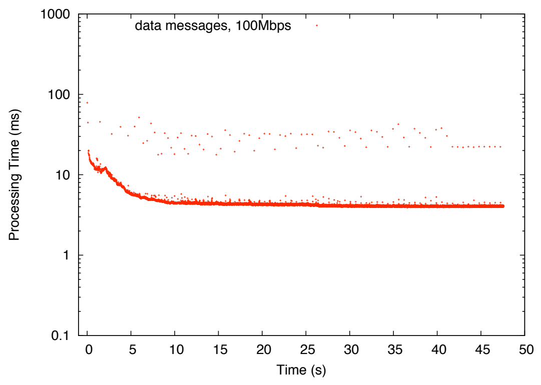
For the same reason as the Java-based implementation, the sender takes longer to process messages. Control messages also take less time to process than data messages. It takes the sender (Figures 6.38(a),(c),(e)) approximately 2.5 ms to process control messages and approximately 5 ms to process data messages. It takes the receiver (Figures 6.38(b),(d),(f)) approximately 5 ms to process data messages. At all send rates, the sender finishes processing messages at around 80 s indicating the sustainable processing rate is around 1 Mbps at the sender.

Independent of the sending rate, the first message takes approximately 50 ms to process. All these trends are nearly identical to those seen in the Hop-to-Hop Acknowledgement service with warm-start (Figure 6.15).

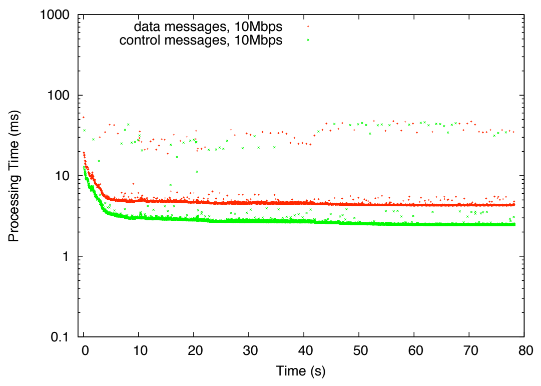
In Figure 6.39 we present the size of the MessageStore buffers (L_{data} and $L_{control}$) and the number of stored FSMs (N_{FSM}) in the MessageStore as a function of time. The figures on the left column (Figure 6.13(a),(c),(e)) show the Sender (S) and the figures on the right column (Figure 6.13(b),(d),(f)) show the Receiver (R). The red data points represent the number of stored FSMs (N_{FSM}) in the MessageStore's hash table. The



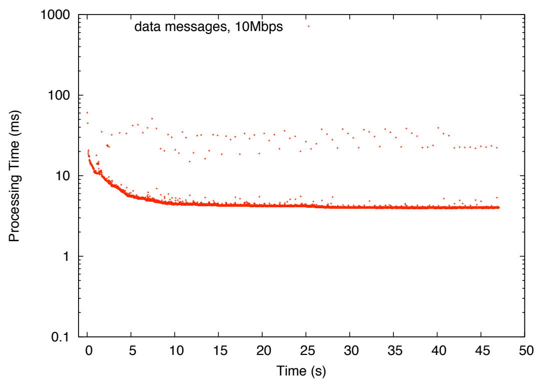
(a) Sender (S), send rate = 100Mbps, with warm-start



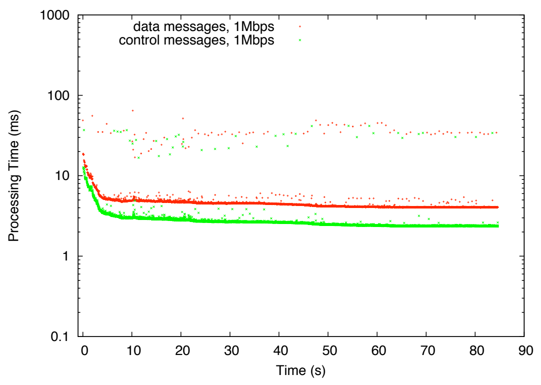
(b) Receiver (R), send rate = 100Mbps, with warm-start



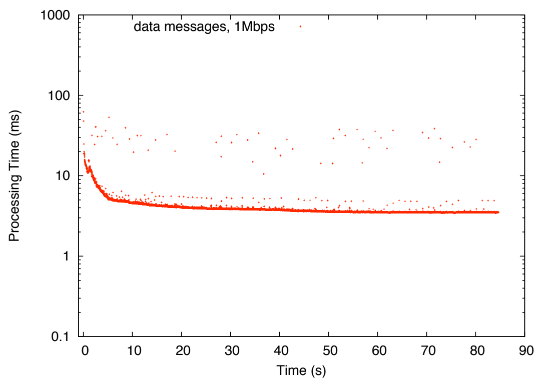
(c) Sender (S), send rate = 10Mbps, with warm-start



(d) Receiver (R), send rate = 10Mbps, with warm-start

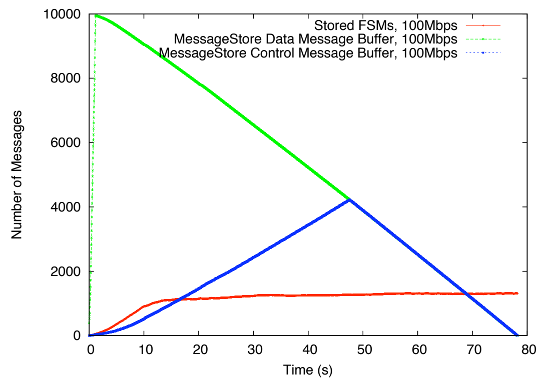


(e) Sender (S), send rate = 1Mbps, with warm-start

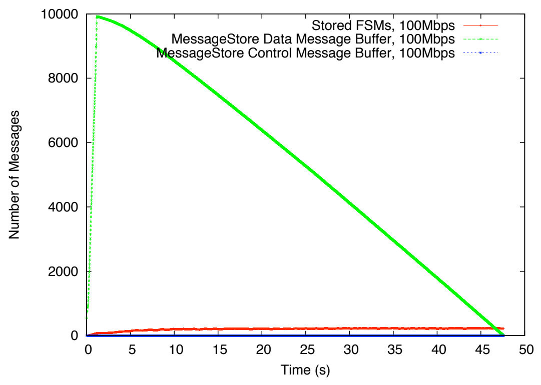


(f) Receiver (R), send rate = 1Mbps, with warm-start

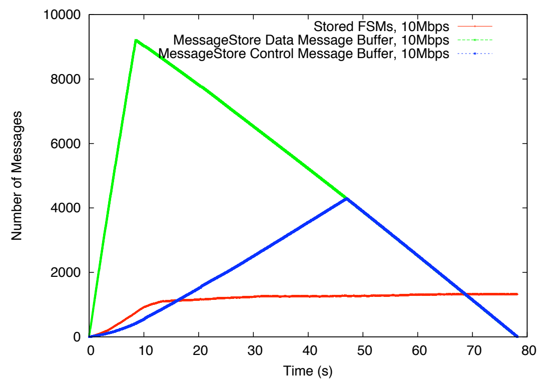
Figure 6.38: MessageStore per message processing time versus time for End-to-End Acknowledgement for the SCXML-based implementation



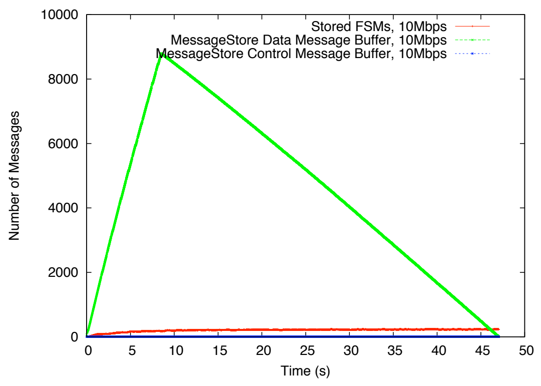
(a) Sender (S), send rate = 100Mbps, with warm-start



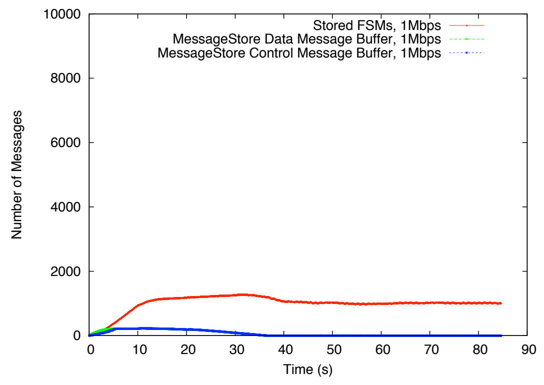
(b) Receiver (R), send rate = 100Mbps, with warm-start



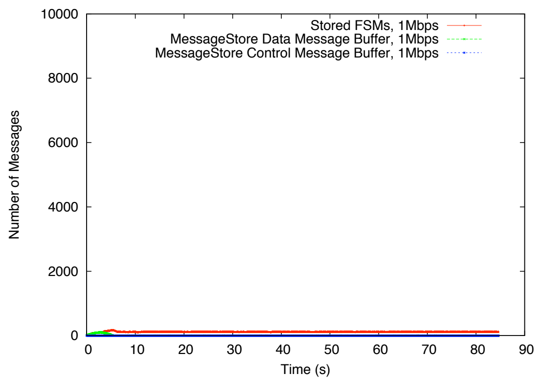
(c) Sender (S), send rate = 10Mbps, with warm-start



(d) Receiver (R), send rate = 10Mbps, with warm-start



(e) Sender (S), send rate = 1Mbps, with warm-start



(f) Receiver (R), send rate = 1Mbps, with warm-start

Figure 6.39: Backlog of messages buffered in MessageStore for data and control message, and the number of stored FSMs in MessageStore versus time for End-to-End Acknowledgement for the SCXML-based implementation

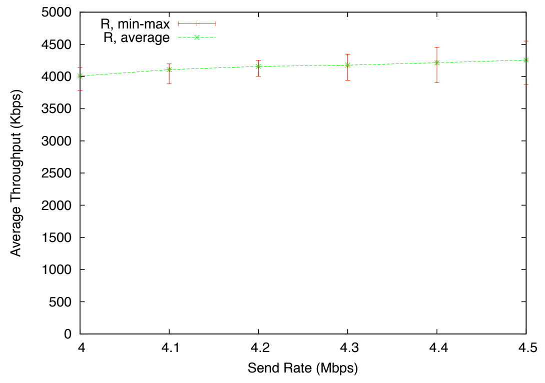
green data points represent the number of backlogged messages in the MessageStore data message buffer (L_{data}). The blue data points represent the number of backlogged messages in the MessageStore control message buffer ($L_{control}$).

As with the Java-based implementation, these plots show that at send rates of 100 Mbps and 10 Mbps, nearly all data messages are backlogged at the MessageStore buffers at both the sender and and the receiver. At a rate of 1 Mbps, there is no backlog of data messages at the sender or receiver. The size of the backlog for the SCXML-based implementation at each time instant is larger than those seen in the Java-based implementation (Figure 6.37) since the SCXML-based implementation takes longer to process a message. The rate at which the messages are being dequeued from the MessageStore data and control message buffers appears bounded to an upper value of approximately 115 messages per second when using the SCXML-based implementation.

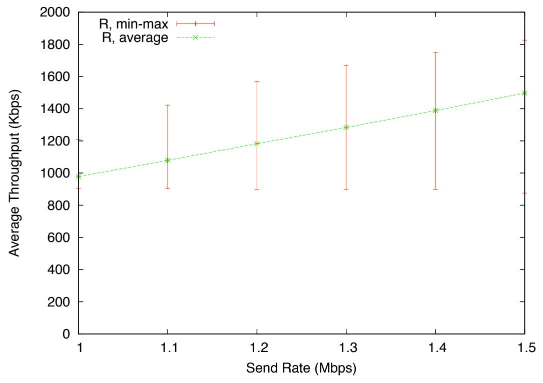
The number of stored FSMs does not appear to affect the MessageStore processing time for messages. We know from out analysis of SCXML-based implementation for Hop-to-Hop Acknowledgement service (Section 6.3.3) that the performance is dominated by the SCXML engine processing. The bottleneck here, again, is the SCXML engine execution.

6.5.4 Sustainable Throughput and Delay

Th next set of experiments are performed to more precisely determine the sustainable performance of the Java-based implementation and SCXML-based implementation (with warm-start) in a single-hop transfer of 10,000 overlay messages with 1,024 bytes payloads from node-1 (Sender S) to node-2 (Receiver R) using the End-to-End Acknowledgment service. We vary the send rate close to the estimated sustainable send rate from Section 6.5.1 (approximately 4.0 Mbps for Java-based implementation and approximately 1.0 Mbps for SCXML-based implementation).



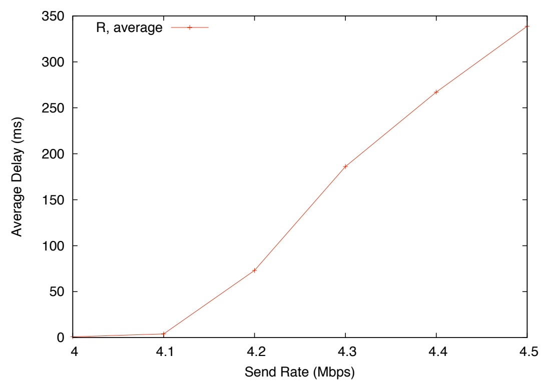
(a) Java-based



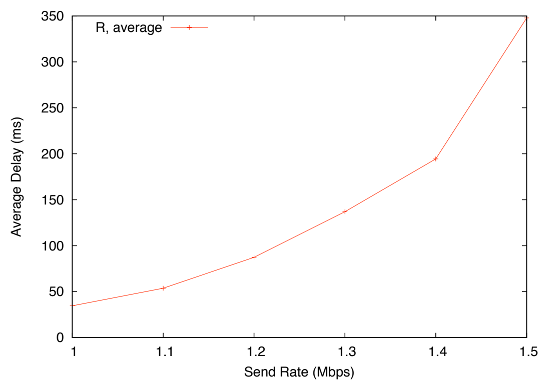
(b) SCXML-based, with warm-start

Figure 6.40: Average throughput versus time for End-to-End Acknowledgement

In Figures 6.40 and Figure 6.41, we plot the average throughput and average per message delay for the same set of experiments over all data messages with error-bars indicating the minimum and maximum value. The average delay in Figures 6.41 illustrates that the sustainable performance is around 4.0 Mbps for the Java-based implementation and around 1.0 Mbps for the SCXML-based implementation. At higher rates, the average delays increase sharply. With higher send rates, more variability in the throughput (indicated by the greater range of the minimum and maximum throughput values) is seem for both implementations.



(a) Java-based



(b) SCXML-based, with warm-start

Figure 6.41: Average per message delay versus time for End-to-End Acknowledgement

These performance results are again nearly identical to the Hop-to-Hop Acknowledgement found in Section 6.3.5.

6.6 End-to-End Acknowledgment in Multi-Hop Network

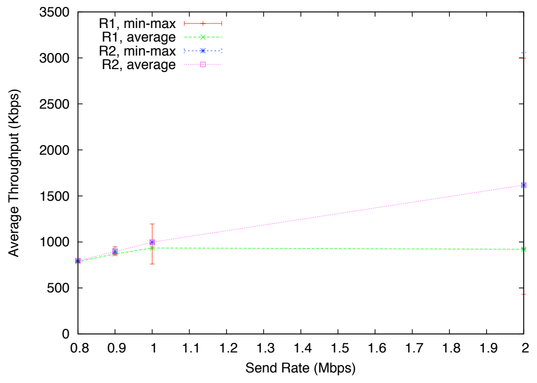
In this set of experiments, we study and compare the performance of the Java-based and the SCXML-based implementations of the End-to-End Acknowledgement service in a multi-hop scenario.

6.6.1 Two-Hop Transfer for Java-based Implementation (at send rate of 4.0 Mbps) and SCXML-Based Implementation (at send rate of 1.0 Mbps) Performance

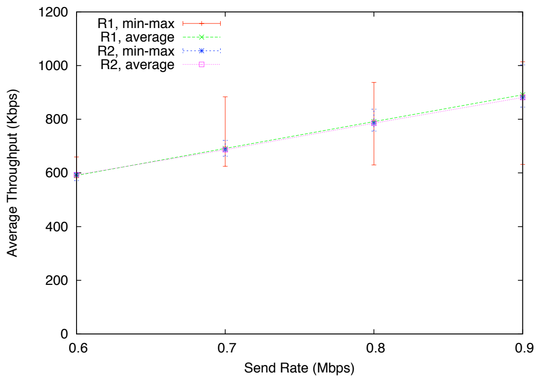
In this set of experiments we send 10,000 messages of 1024 bytes from node-1 Sender (S) to Receiver ($R1$) and Receiver ($R2$) corresponding to node-2 and node-3 in a two-hop transfer using the Java-based implementation and the SCXML-based implementation (with warm-start). We initially varied the sending rate close to the estimated sustainable sending rate for single-hop from Section 6.5.4 (4.0 Mbps for Java-based implementation and 1.0 Mbps for SCXML-based implementation). We found these rates are not sustainable for the two-hop scenario.

In Figure 6.42 and Figure 6.43, we plot the average throughput and average per message delay. For the Java-based implementation, we send messages at rates of 2.0 Mbps ,1.0 Mbps, 0.9 Mbps, and 0.7 Mbps. For the SCXML-based implementation, we send messages at rates of 0.6 Mbps, 0.7 Mbps, 0.8 Mbps, and 0.9 Mbps. The plots show that the sustainable performance for the Java-based implementation is approximately 0.9 Mbps and the sustainable performance for the SCXML-based implementation is approximately 0.7Mbps.

Compared with the Hop-to-Hop Acknowledgment service which sustained a send rate of approximately 1.9Mbps for the Java-based implementation (Section 6.4.1.3), the End-

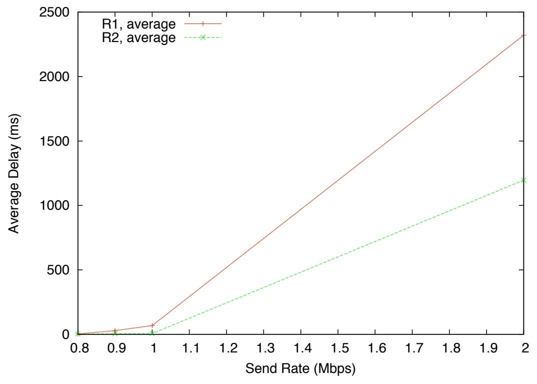


(a) Java-based

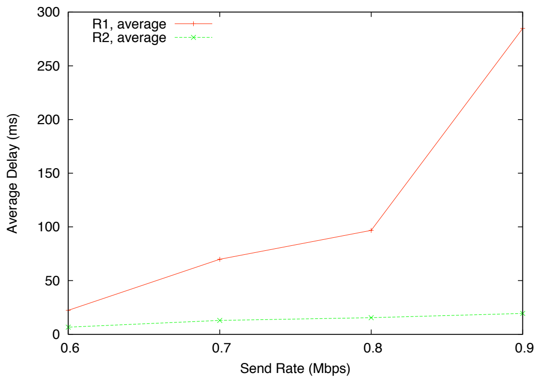


(b) SCXML-based, with warm-start

Figure 6.42: Average throughput versus time for End-to-End Acknowledgement



(a) Java-based



(b) SCXML-based, with warm-start

Figure 6.43: Average per message delay versus time for End-to-End Acknowledgement

to-End Acknowledgment service showed a significant drop in the sustainable rate for the two-hop scenario. This phenomenon can be explained due to the semantics of End-to-End Acknowledgement service. We have previous determined that the performance of the Java-based implementation is bottlenecked by the number of stored FSMs in MessageStore. Unlike Hop-to-Hop Acknowledgement where the acknowledgement messages are from the downstream node single-hop, in End-to-End Acknowledgment service must wait for all downstream node(s) to send an aggregated acknowledgment. Thus any node that has downstream node(s) must store a FSM until an aggregated acknowledgment message is received. This results in more FSMs stored in a node using the End-to-End Acknowledgement service compared to the Hop-to-Hop Acknowledgement service if it has more than one downstream node.

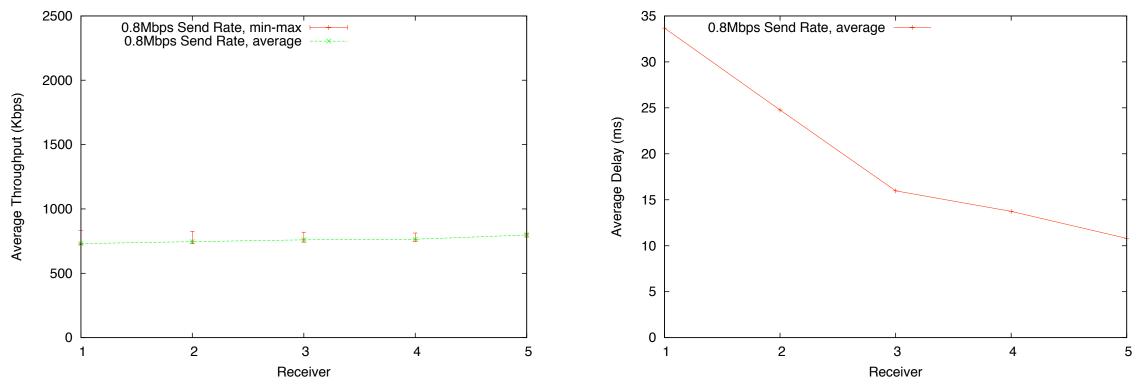
For example, assume we multicast data messages in a n node daisy-chain topology $(N_1, N_2, ..., N_i, ..., N_{n-1}, N_n)$ where N_1 is the sender, N_i is an intermediate node. In the Hop-to-Hop Acknowledgement service, node N_i stores and wants for an acknowledgement message to come from its downstream node N_{i+1}. If the time it waits for this acknowledgment is t then the FSM is stored for $t + Timeout_{Delete}$ until it gets removed. This time is independent of how many downstream nodes node N_i has. In contrast, in the End-to-End Acknowledgement service, node N_i stores and wants for an aggregated acknowledgement message to come from its downstream node N_{i+1}, and node N_{i+1} stores and wants for an aggregated acknowledgement message to come from its downstream node N_{i+2}, etc. Thus if the time a node waits for this acknowledgment is t then the FSM is stored for $t \times m + Timeout_{Delete}$ until it gets removed; where m is the number of downstream nodes.

We can see that for a single-hop scenario involving only two nodes, both the Hop-to-Hop Acknowledgement service and the End-to-End Acknowledgement service show similar results. However for multi-hop scenarios, the performance of a node using the Java-based implementation of End-to-End Acknowledgment service depends on the number of nodes downstream. More downstream nodes result in a longer waiting time for an aggregated acknowledgement which increases the number of stored FSMs in MessageStore. This results in longer per message processing times.

Evidently this effect is not present for the SCXML-based implementation since the principle bottleneck for sustainable performance is the SCXML engine execution, and not the number of stored FSMs in MessageStore.

6.6.2 Five-Hop Transfer for Java-based Implementation (at send rate of 0.8 Mbps) and SCXML-Based Implementation (at send rate of 0.7 Mbps) Performance

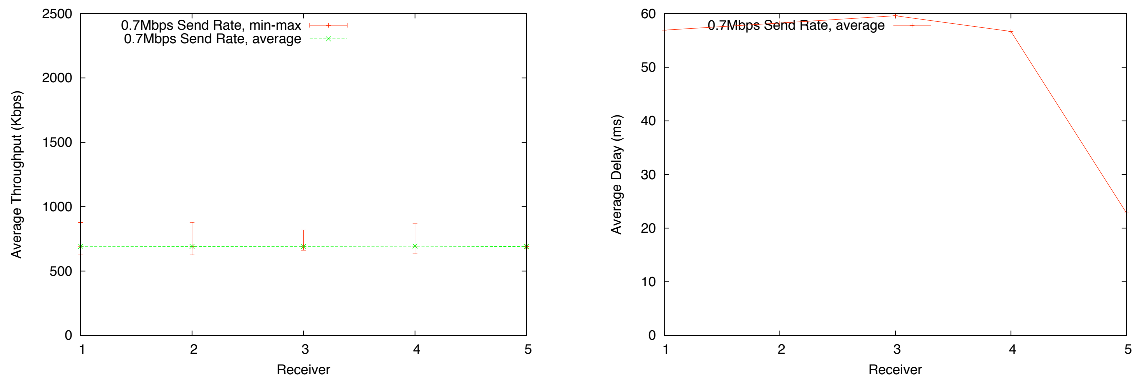
In this set of experiments we send 10,000 overlay messages of 1,024 bytes payloads from node-1 Sender S to Receiver ($R1$, $R2$, $R3$, $R4$, and $R5$) corresponding to node-2 to node-6 in a five-hop transfer for the Java-based implementation and the SCXML-based implementation (with warm-start). We set the sending rate close to the estimated sustainable sending rate from Section 6.6.1 (0.8 Mbps for Java-based implementation, and 0.7 Mbps for SCXML-based implementation) to determine the sustainable performance for five-hop message transfer.



(a) Average throughput (b) Average per message delay

Figure 6.44: Average throughput and average delay for each receiver for End-to-End Acknowledgement using the Java-based implementation

In Figures 6.44 we plot the average throughput and average per message delay for each of the receivers for the Java-based implementation. Figure 6.45, we plot the average throughput and average per message delay for each of the receivers for the SCXML-based implementation. We can see that the two-hop sustainable transfer rate is also sustainable for the five-hop scenario.



(a) Average throughput

(b) Average per message delay

Figure 6.45: Average throughput and average delay for each receiver for End-to-End Acknowledgement using the SCXML-based implementation

However, as explained earlier, the delay experienced at each node is dependent on the number of downstream nodes for the Java-based implementation. This added delay is significant for the Java-based implementation. Here we see that the average delay for the Java-based implementation shows that upstream nodes experience longer delays compared to downstream nodes (Figures 6.44(b)). For the SCXML-based implementation this effect is not significant and we observe constant delays across all receivers ($R1$, $R2$, $R3$, $R4$, and $R5$) as seen in Figures 6.45(b)).

Note that the terminating/leaf node will always have lower per message processing time since it never needs to wait for an acknowledgement message.

6.7 Memory Usage

In this section, we profile the real-time heap memory usage of our system for the two services, Hop-to-Hop Acknowledgement and End-to-End Acknowledgment. In these experiments we set the Java memory parameters as follows:

- `-Xms` 0241: initial Java heap size of 1024MB
- `-Xmx` 0241: maximum Java heap size of 1024MB

We did not tune the garbage collector and used default Java garbage collection.

6.7.1 Hop-to-Hop Acknowledgement Service in Single-Hop Network

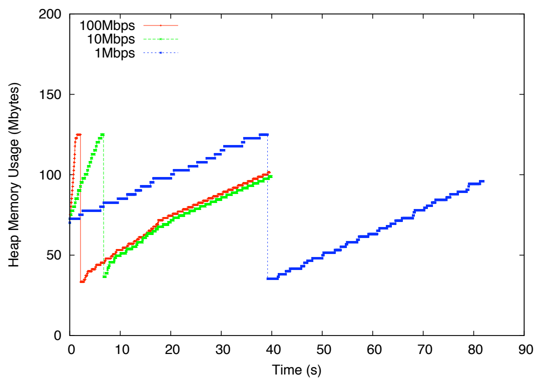
First we profile the heap memory usage for the Hop-to-Hop Acknowledgement service using the Java-based implementation and the SCXML-based implementation with warm-start. The experiments consist of a single-hop transfer of 10,000 overlay messages with 1,024 bytes payloads from node-1 (Sender S) to node-2 (Receiver R). We performance the experiment at sending rates of 100 Mbps, 10 Mbps, and 1 Mbps.

Figure 6.46 shows the heap memory usage as a function of the time at the Sender S ((a)) and the Receiver R ((b)) with send rates of 100 Mbps, 10 Mbps, and 1 Mbps. The time is represented on the x-axis. Similar to previous plots, the time represent the received times of each sequence number of data messages at the Receiver R. The y-axis show the amount of free memory in the Java Virtual Machine. The heap memory is measured directly with embedded profiling code using the Java `runtime` class. The method call used is:

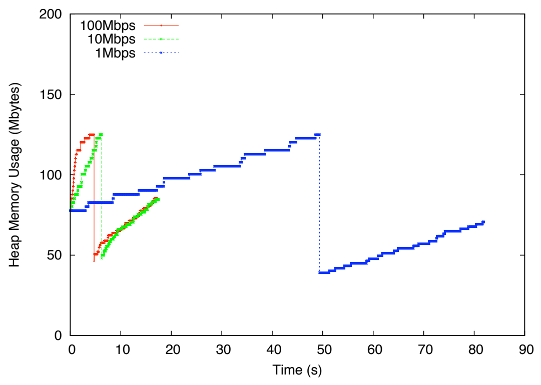
```
runtime.totalMemory() - runtime.freeMemory()
```

The red, green, and blue data points show the heap memory usage at send rates of 100 Mbps, 10 Mbps, and 1 Mbps, respectively. From Figure 6.46 we see that the memory profile is consistent with most Java applications. The drops in the memory usage are caused by the Java garbage collector that periodically frees the used memory. Here the Java garbage collection appears to be executed whenever memory usage reaches approximately 125MB. Depending on the send rate, this value is reached at different times. Evidently a faster send rate will result this value being reached faster as Java objects are created at a higher rate. We can approximate the rate which memory is allocated by examining the slope in Figure 6.46(a). For example, at a sustainable send rate 1 Mbps (blue data points), the Sender S have an slope measured to be approximately $\frac{100MB}{40s} = 2.5MB/s$. A send rate of 1Mbps means that we are approximately sending 125 messages per second since each message is 1 KB and 1 Mbps send rate means 125 KB per second. This results in a memory usage of approximately 20 KB/message processed at the Sender S. Similarly, the memory usage rate at the Receiver R for a send rate of 1Mbps can be calculated to be approximately 16 KB/message. This is as expected since we have previously determined that the sender needs to wait for acknowledgement messages from the receiver resulting in more finite-state machine processing.

Next we profile the heap memory usage for the SCXML-based implementation. Figure 6.47 shows the heap memory usage as a function of the time at the Sender S ((a)) and the Receiver R ((b)) with send rates of 100 Mbps, 10 Mbps, and 1 Mbps. The red, green, and blue data points show the heap memory usage at send rates of 100 Mbps, 10 Mbps, and 1 Mbps, respectively. From Figure 6.47 we see that the memory profile is increasing and does not drop. Closer investigation reveals that the garbage collection is functioning correctly. The “oscillations” that occur frequently in the data points are caused by the Java garbage collector. Note that the difference between the upper values and lower values in the data at any time is approximately 125MB. This is is because the Java garbage collector is clearing objects off the heap memory when its memory usage



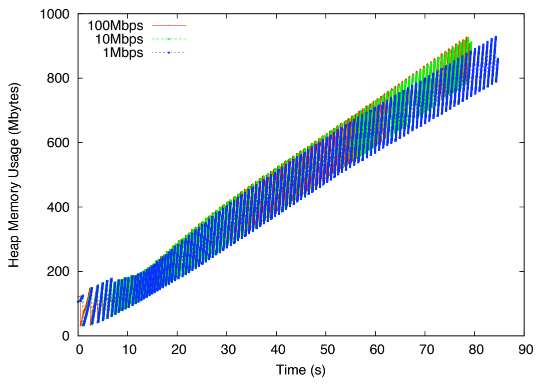
(a) Sender (S)



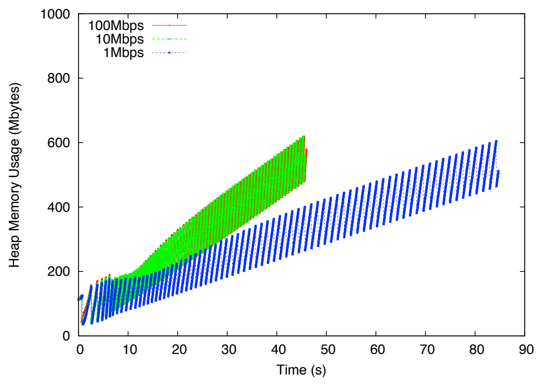
(b) Receiver (R)

Figure 6.46: Heap memory usage versus time for Hop-to-Hop Acknowledgement for the Java-based implementation

exceeds 125MB just like in the Java-based implementation.



(a) Sender (S), with warm-start



(b) Receiver (R), with warm-start

Figure 6.47: Heap memory usage versus time for Hop-to-Hop Acknowledgement for the SCXML-based implementation

The growth in memory is caused by an area of memory that the Java garbage collector does not clear unless the JVM is about to run out of memory. The SCXML-based implementation use Java Reflection which uses the Java Classloader. By using Java Reflection we store those reflective classes in the *permanent generation* area of the memory which is a part of the heap. The Java documentation [13] defines permanent generation as: "The permanent generation is used to hold reflective data of the VM itself such as class objects and method objects. These reflective objects are allocated directly into the permanent generation, and it is sized independently from the other generations." These reflective classes, being in permanent generation, will not be garbage-collected unless the JVM is running out of memory. Note that the permanent generation will never cause a `java.lang.OutOfMemoryError` (memory leak) since these classes are *soft references.* As described by Java [13]: "Soft reference objects, which are cleared at the discretion of the garbage collector in response to memory demand. Soft references are most often used to implement memory-sensitive caches. ... All soft references to softly-reachable objects are guaranteed to have been cleared before the virtual machine throws an OutOfMemoryError. Otherwise no constraints are placed upon the time at which a soft reference will be cleared or the order in which a set of such references to different objects will be cleared. Virtual machine implementations are, however, encouraged to bias against clearing recently-created or recently-used soft references."

Since we specified out initial and max heap memory size to be 1024 MB, the permanent generation area of the memory is allowed to grow until it reaches 1024MB, which then will be cleared by the Java garbage collector. We verified this by running the same experiment with different initial and max heap memory sizes and found that the garbage collector will only clear the permanent generation area of memory once the set maximum heap size is reached by our system. Below we show an example of the Sender S for the same experiment as that of Figure 6.47 but with a initial and max heap size set at 512 MB.

From Figure 6.48 we clearly see that the memory is cleared by the Java garbage collector once it exceeds 512 MB at approximately 45 s.

Now, we can use the same approach we used in the analysis of the Java-based implementation to approximate the rate of memory usage for the SCXML-based implementa-

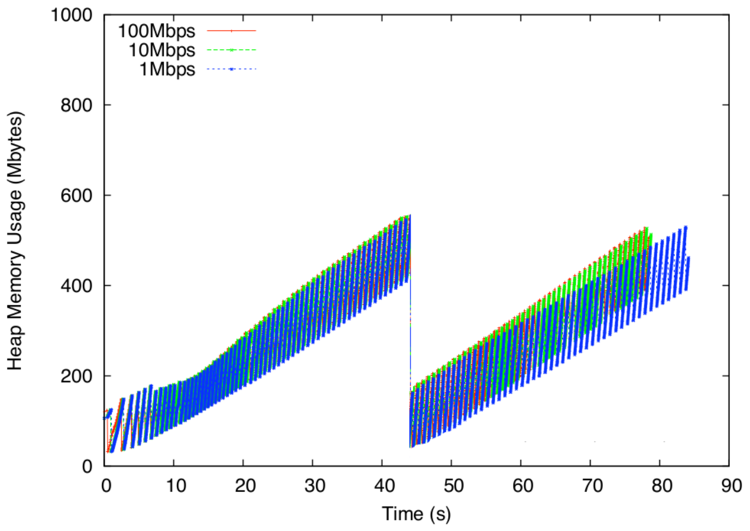


Figure 6.48: Heap memory usage versus time for Hop-to-Hop Acknowledgement for the SCXML-based Implementation (Sender (S), with warm-start, with initial and max heap size of 512MB)

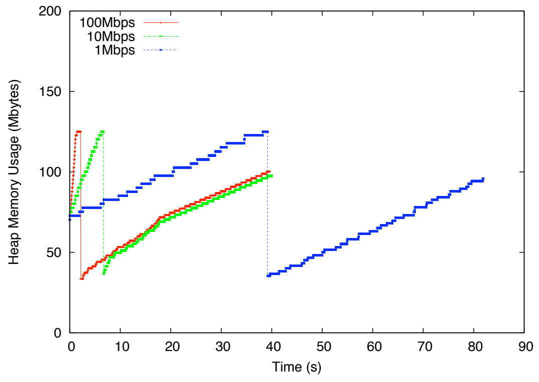
tion.

For example in Figure 6.47(a), at a sustainable send rate 1 Mbps (blue data points), the Sender S have an slope measured to be approximately $\frac{950MB}{85s} = 11.2MB/s$. A send rate of 1 Mbps means that we are approximately sending 125 messages per second since each message is 1 KB and 1 Mbps send rate means 125 KB per second. This results in a memory usage of approximately 89 KB/message processed at the Sender S. Similarly, the memory usage rate at the Receiver R for a send rate of 1 Mbps can be calculated to be approximately 54 KB/message. Again the receiver uses less memory.

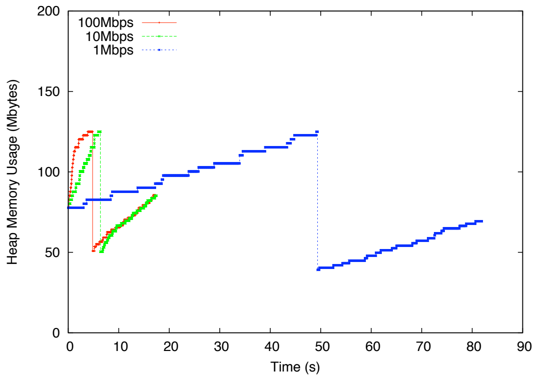
6.7.2 End-to-End Acknowledgement Service in Single-Hop Network

Next we profile the heap memory usage for End-to-End Acknowledgement for the Java-based implementation and SCXML-based implementation with warm-start. The experiments also consist of a single-hop transfer of 10,000 overlay messages with 1,024 bytes payloads from node-1 (Sender S) to node-2 (Receiver R). We performance the experiment at sending rates of 100 Mbps, 10 Mbps, and 1 Mbps.

Figure 6.49 shows the heap memory usage as a function of the time at the Sender S ((a)) and the Receiver R ((b)) with send rates of 100 Mbps, 10 Mbps, and 1 Mbps for the Java-based implementation in the same manner as Figure 6.46. Figure 6.50 shows the heap memory usage as a function of the time at the Sender S ((a)) and the Receiver R ((b)) with send rates of 100 Mbps, 10 Mbps, and 1 Mbps for the SCXML-based implementation in the same manner as Figure 6.47.



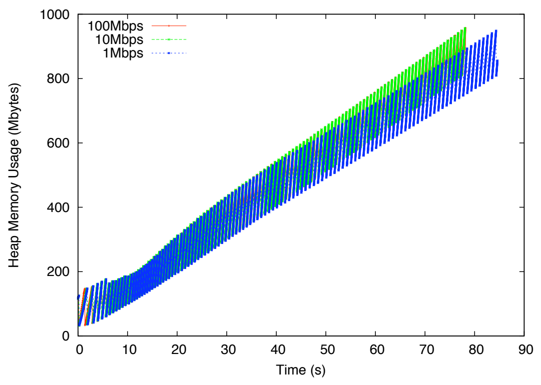
(a) Sender (S)



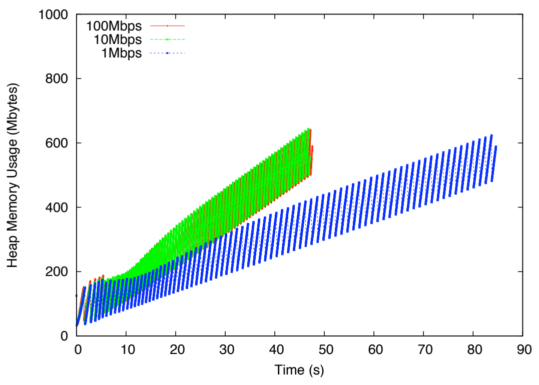
(b) Receiver (R)

Figure 6.49: Heap memory usage versus time for Hop-to-Hop Acknowledgement for the Java-based implementation

Both of these figures show nearly identical memory usage characteristics as the Hop-to-Hop Acknowledgment service. This is as expected since our system's bottlenecks does not vary with different services.



(a) Sender (S), with warm-start



(b) Receiver (R), with warm-start

Figure 6.50: Heap memory usage versus time for End-to-End Acknowledgement for the SCXML-based implementation

Chapter 7

Conclusions and Future Work

In the past decade significant research on overlay networks has focused on the control-plane of overlay networks rather than the data-plane. We propose a way to efficiently extend the data-plane in the form of data delivery services. We believe our work aids in the building of a more flexible architecture for overlay networking that supports innovation in data delivery.

7.1 Conclusions

In this thesis, we presented a mechanism for network applications to deploy customizable data delivery services into an overlay middleware system. Our mechanism allows custom data delivery service to be declared as executable specifications and dynamically deployed and executed on other overlay nodes by simply delivering messages marked with a service identifier.

We designed an approach to specify data delivery services as finite-state machines expressed using the XML markup language for state-machines, SCXML. These finite-state machines respond to two network events: message arrivals and timer expirations. We used first-order logic to describe complex events which are composited of multiple message arrivals and/or timer expirations events.

We also proposed a set of network primitives that are common to data delivery services. Although this set of network primitives may not be exhaustive we showed that it is sufficient to describe actions performed by complex data delivery services such as "Hop-to-Hop Acknowledgement" and "End-to-End Acknowledgement".

We developed a software prototype which can use a generic finite-state machine execution engine (Apache Commons SCXML) to execute the executable specifications of data delivery services in overlay middleware. We presented experiments that showed that our approach can sustain between 25% - 35% of the maximum throughput of a hard-coded data delivery service.

We proposed two mechanism to improve the performance of our implementation: 1) a preallocation of finite-state machines and 2) a "warm-starting" of the generic finite-state machine execution engine during initialization of the overlay node. It was found that these mechanism reduce per message delays but did not affect the sustainable throughput of our system.

We believe that the developed approach to describe, execute, and deploy custom data delivery services enables a flexible and more robust approach for developing and deploying new data delivery semantics. Our research may enable a new perspective in the design of networking software and protocols for the future of overlay networking.

7.2 Future Work

This thesis offers opportunities to be extended and continued in the following directions:

1. **Expressiveness of Services**: We provided a method for describing data delivery services in terms of a specification using first-order logic and a set of network primitives. We argued that the network primitives are sufficient to realize a wide variety of services of varying complexity. It remains to be investigated which services our approach can and cannot specify. We believe that congestion control algorithms at the application-layer can be specified using our approach. More work is needed to provide evidence that determine the limitations of our approach.

2. **Extension to Other Network Layers**: Our prototype addresses data delivery services in the application-layer. However, our methodology can be extended to other network layers. For example, transport protocols can be naturally described as a finite-state machine and could be expressed as an executable specification. Extension of this approach and its performance implications on other network layers remains to be investigated.

3. **Implications to Software Design**: In a broader sense, our approach is not limited to network services. One can envision that other softwares derived from a finite-state machine can also be expressed as an executable specification. For example, it may be feasible to design a flexible layer residing just above the hardware that executes all software programs from executable specifications. In this case, an operating system may be defined as an executable specification.

4. **Formal Verification of Services**: One advantage of describing a service in terms as a finite-state machine is that they can be formally verified using finite-state automata verification tools. We used a simple XML schema to validate the markup and data of our executable specifications. However, our scheme did not verify the

behaviour of the services (no infinite-loops in the finite-state machine, the finite-state machine does not attempt to flood the network, etc.). It is worthwhile to design a method to formally verify executable specifications before their deployment and execution. If this can be done, the executable specification is executed within a "sandbox".

5. **Flexible Deployment Mechanisms**: Currently, if overlay nodes receive a message marked with a service identifier that it does not have the executable specification for, it attempts to download the executable specification from the Services Server. This centralized approach can have scalability issues and robustness issues if the Services Server cannot sustain the rate of request and/or if the services server goes down. It is possible to design more sophisticated decentralized strategies to deploy executable specifications. However, due to the large size of these executable specifications, it is not clear whether a decentralized strategy is efficient. This remains to be examined.

6. **Performance Improvements**: The biggest factor that limits our the performance of our prototype appear to be XML parsing and processing. It is feasible to redesign the system to not use XML (using proprietary format and processing technologies). However, we feel that the XML standard along with available XML parsing and processing technologies makes XML attractive. As better XML technologies becomes available, they can be leveraged to improve the performance of our system.

7. **Memory-Constrained Systems**: From our experiments we showed that our flexible approach requires significantly more memory compared to hard-coded Java classes of services. The impact of this on memory constrained devices such as mobile phones, PDAs, may need to be investigated.

Bibliography

[1] http://commons.apache.org/scxml/.

[2] http://graphml.graphdrawing.org/.

[3] http://graphviz.org/.

[4] http://linux.die.net/man/1/ttcp.

[5] http://schemas.microsoft.com/vs/2009/dgml/.

[6] http://www.comm.utoronto.ca/hypercast/.

[7] http://www.comm.utoronto.ca/hypercast/design.html.

[8] http://www.comm.utoronto.ca/hypercast/design/messageformatsv5.pdf.

[9] http://www.emulab.net.

[10] http://www.fim.uni-passau.de/en/fim/faculty/chairs/theoretische-informatik/projects.html.

[11] http://www.gupro.de/gxl/.

[12] http://www.ntp.org.

[13] http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html.

[14] http://www.saxonica.com/welcome/welcome.xml.

[15] http://www.w3.org/dom/.

[16] http://www.w3.org/tr/2005/wd-scxml-20050705/.

[17] http://www.w3.org/xml/.

[18] http://www.w3schools.com/schema/default.asp.

[19] http://www.w3schools.com/xpath/default.asp.

[20] http://www.w3schools.com/xquery/default.asp.

[21] www.bittorrent.com.

[22] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The switchware active network architecture. *IEEE Network: The Magazine of Global Internetworking*, 12(3):29–36, May 1998.

[23] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. The case for resilient overlay networks. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HOTOS '01, page 152, Washington, DC, USA, 2001. IEEE Computer Society.

[24] D. P. Anderson. Automated protocol implementation with rtag. *IEEE Transactions on Software Engineering*, 14:291– 300, Mar. 1988.

[25] J. Barwise. *An introduction to first-order logic.* Handbook of Mathematical Logic. Studies in Logic and the Foundations of Mathematics, Amsterdam: North-Holland, 2nd edition, 1982.

[26] G. Berry. Proof, language, and interaction. chapter The foundations of Esterel, pages 425–454. MIT Press, Cambridge, MA, USA, 2000.

[27] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. An architecture for active networking. In *Proceedings of the IFIP TC6 seventh international conference on High performance netwoking VII*, HPN '97, pages 265–279, London, UK, UK, 1997. Chapman & Hall, Ltd.

[28] P.G. Bridges, G.T. Wong, M. Hiltunen, R.D. Schlichting, and M.J. Barrick. A configurable and extensible transport protocol. *IEEE/ACM Transactions on Networking*, 15(6):1254 –1265, Dec. 2007.

[29] K. L. Calvert, J. Griffioen, and S. Wen. Lightweight network support for scalable end-to-end services. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '02, pages 265–278, New York, NY, USA, 2002. ACM.

[30] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating efficient protocol code from an abstract specification. *IEEE/ACM Transactions on Networking*, 5:514–524, Aug. 1997.

[31] Y. Chu, S.G. Rao, S. Seshan, and H. Zhang. A case for end system multicast. *IEEE Journal on Selected Areas in Communications*, 20(8):1456 – 1471, 2002.

[32] D.C. Feldmeier, A.J. McAuley, J.M. Smith, D.S. Bakin, W.S. Marcus, and T.M. Raleigh. Protocol boosters. *IEEE Journal on Selected Areas in Communications*, 16(3):437 –444, Apr. 1998.

[33] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Plan: a packet language for active networks. *ICFP '98 Proceedings of the third ACM SIGPLAN international conference on Functional programming*, 34(1):86–93, 1998.

[34] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[35] B. Li, J. Guo, and M. Wang. ioverlay: a lightweight middleware infrastructure for overlay application implementations. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, Middleware '04, pages 135–154, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[36] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *SOSP '05 Proceedings of the twentieth ACM symposium on Operating systems principles*, 39(5):75–90, 2005.

[37] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110 – 143, May 1992.

[38] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. Opendht: a public dht service and its uses. *SIGCOMM '05 Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, 35(4):73–84, 2005.

[39] A. Rodriguez, C. Killian, S. Bhat, D. Kostić, and A. Vahdat. Macedon: methodology for automatically creating, evaluating, and designing overlay networks. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[40] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.

[41] S. R. Srinivasan, J. W. Lee, E. Liu, M. Kester, H. Schulzrinne, V. Hilt, S. Seetharaman, and A. Khan. Netserv: dynamically deploying in-network services. In *Proceed-*

ings of the 2009 workshop on Re-architecting the internet, ReArch '09, pages 37–42, New York, NY, USA, 2009. ACM.

[42] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

[43] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *Comm. Mag.*, 35(1):80–86, 1997.

[44] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *ACM SIGCOMM Computer Communication Review*, 37(5):81–94, 2007.

[45] K. J. Turner. *Using Formal Description Techniques: An Introduction to Estelle, Lotos, and SDL.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1993.

[46] M. Valipour. Cross-substrate advertisement: Building overlay cross-substrate advertisement: Building overlay networks for heterogeneous environments. Master's thesis, University of Toronto, 2010.

[47] S. T. Vuong, A. C. Lau, and R. I. Chan. Semiautomatic implementation of protocols using an estelle-c compiler. *IEEE Transactions on Software Engineering*, 14(3):384–393, 1988.

[48] D. J. Wetherall. *Service introduction in an active network.* PhD thesis, Massachusetts Institute of Technology, 1999. AAI0800686.

[49] D. J. Wetherall and D. L. Tennenhouse. The active ip option. In *Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*, EW 7, pages 33–40, New York, NY, USA, 1996. ACM.

[50] J. Zander and R. Forchheimer. Softnet - an approach to high level packet communications. In *Proceedings of the AMRAD Conference*, 1983.

[51] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2006.