(5)

A Simulation Toolkit for URN Resolution

by

Nancy Cheung

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirement for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science and

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 22, 1998

[June 1998]

Author_____

Department of Electrical Engineering and Computer Science
May 22, 1998

Certified by_____

Karen R. Sollins
Thesis Supervisor

Accepted by_____

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Simulation Toolkit for URN Resolution

by

Nancy Cheung

Submitted to the

Department of Electrical Engineering and Computer Science

May 22, 1998

In Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science and

Master of Engineering in Electrical Engineering and Computer Science

Abstract

The World Wide Web currently uses Uniform Resource Locators (URLs) to specify locations of Web resources. Since URLs contain the locations of the resources that they represent, they display poor longevity as the resources are moved over times. Uniform Resource Names (URNs), globally unique, persistent identifiers are developed as an alternative. Resolvers map URNs to resources they identify. A Resolver Discovery Service (RDS) is needed for determining the appropriate resolver for a URN. Various research studies have been done on URN resolution system and different RDS models have been developed. We design and implement a simulation toolkit for studying URN resolution system and analyzing the performance of different RDS models.

Thesis Supervisor: Dr. Karen R. Sollins
Title: Research Scientist, MIT Laboratory for Computer Science

Acknowledgements

I would like to give my sincere thanks to the following people:

Dr. Karen Sollins, my thesis and previous UROP advisor, for her support and guidance on this thesis and previous UROP projects over the past three years. I appreciate her for helping in every way possible: finding thesis topic and defining the scope, explaining concepts and ideas on URN resolution, pointing to helpful references, discussing issues on design and implementation, reading drafts and providing valuable comments, helpful conversation on issues unrelated to this work, and a lot more too numerous to mention.

Lewis Girod and Edward Slottow, members in my research group, for explaining many ideas and details about the RDS models they have developed, in which I am doing simulation on.

Dorothy Curtis, system administrator for my research group, for setting up computers, installing software I need, and answering my questions about UNIX.

Friends, for helping me get through some tough times and sharing other happy moments. I also thank them for occasionally distracting me from my work, taking me out when I should have been working on my thesis.

Last but not least, my parents, for their love and caring throughout these twenty-two years. They provided an excellent environment for me to grow up, and their influences on me are profound.

Table of Contents

Chapter 1: Introduction **9**
1.1 Background 10
1.2.1 Information Mesh 10
1.2.2 URN Resolution 11
1.2 Motivation 12
1.3 The Project 13
1.4 Roadmap 13
Chapter 2: Related Work **15**
2.1 REAL 15
2.2 NS 17
2.3 Summary 18
Chapter 3: Uniform Resource Name Resolution **21**
3.1 URL 22
3.2 URN 22
3.3 URN Resolution Framework 23
3.4 Different RDS Models 25
3.4.1 NAPTR 25
3.4.2 mu 26
3.4.3 Distributed RDS 28
3.5 Summary 29
Chapter 4: The Simulation Toolkit **31**
4.1 OTcl Linkage 32
4.1.1 Class Tcl 33
4.1.2 Class TclObject 34
4.1.3 Class TclClass 35
4.1.4 Class InstVar 35
4.2 Basic Modules 36
4.2.1 Class Simulator 36
4.2.2 Class Node 38
4.2.3 Class Link 40
4.2.4 Class Agent 41
4.3 URN Resolution System 44
4.3.1 Hint 45
4.3.2 Client 47
4.3.3 Server 49
4.3.4 Publisher 53
4.4 Summary 56
Chapter 5: Demonstration **57**
5.1 Overview of the model 57
5.2 Main Components 58
5.2.1 The Central database 59
5.2.2 The Replication Groups 62

5.2.3 The Distributed Database 64
5.3 Example 67
5.4 Summary 69
Chapter 6: Conclusion 71
6.1 Improvements 72
6.2 Future Work 73
References 75
Appendix A 77
Appendix B 79

Chapter 1

Introduction

The World Wide Web currently uses Uniform Resource Locators (URLs) to specify locations of Web resources. Since URLs contain the locations of the resources that they represent, they display poor longevity as the resources are moved over times. Uniform Resource Names (URNs), globally unique, persistent identifiers are developed as an alternative. URNs do not contain any information on the locations of the resources. Resolvers map URNs to resources they identify. A Resolver Discovery Service (RDS) is needed for determining the appropriate resolver for a URN. Various research studies have been done on URN resolution system and different RDS models have been developed.

Due to the increasing complexity and scale of the current computer networks, simulation has become an important tool for network researches. In this project, we design and implement a simulation toolkit for studying URN resolution system and analyzing the performance of different RDS models.

1.1 Background

Over the years, the World Wide Web has become the dominant information system. It relies heavily on the use of URLs to link network resources together. The major weakness is the poor longevity of the URLs. The Information Mesh [15], an alternative architecture for a global information system, has the goal of creating a robust, reliable, long-lasting infrastructure. URNs appear to be a good alternative to URLs and satisfy the goals of the Information Mesh.

1.2.1 Information Mesh

The Information Mesh [15] is an information infrastructure architecture designed primarily to support the longevity, mobility, and evolvability of information. The following summarizes the goals of the Information Mesh project:

- **Global Scope:** The Information Mesh should provide a general agreement on object referral in a highly scalable manner.
- **Ubiquity:** The Information Mesh should support all network-based applications wherever the information they access is located.
- **Heterogeneity:** The Information Mesh must support a broad set of network protocols and applications. The set includes the past implementation and likely future implementation.
- **Longevity:** Information and its identifiers should be able to survive indefinitely, or more practically speaking, for at least 100 years.

- **Mobility:** Information should be able to move not only from one physical location to another, but also from one administrative region to another.
- **Evolvability:** Future applications and a changing network may place new requirements on the Information Mesh. Therefore the Information Mesh must be able to evolve and adapt to these new requirements.
- **Resiliency:** The Information Mesh should be resilient to failure in accessing information. These failures may arise due to a variety of reasons such as hardware failures or expired pointer.

To satisfy the above goals, the Information Mesh needs persistent identifiers for naming objects and linking them together. The location of an object can change over times. On the other hand, a pure identifier of an object should be static. Objects named by pure identifiers can move and computer systems can change without their identifiers becoming invalid.

The information Mesh uses object identifiers (oids) to name objects. An oid identifies an object without indicating the location of the object. Hints are used to resolve an oid into one or more locations. URNs are a type of name that matches the Information Mesh requirements for oids well.

1.2.2 URN Resolution

Currently in the World Wide Web, Uniform Resource Locators (URLs) are used in retrieving resources. Typically, URLs serve as the addresses of resources in the Web by specifying a particular file on a particular machine. However, this makes it difficult to reorganize the files on the web server, or to replicate the resource on multiple machines

in order to distribute the load or to provide fault-tolerance. Uniform Resource Names (URNs), persistent, location-independent identifiers, are intended to overcome these problems. Unlike URLs, URNs are assigned to specify the identity of a resource, rather than its location. Given a URN, a client contacts a resolver to discover locations of the resource. By adding a layer of indirection, we can change the location of a resource, or add multiple locations, without changing the identifier. The system required for performing URN resolution would consist of two parts: the resolvers, and a Resolution Discovery Service (RDS), which finds the appropriate resolver for a URN.

1.2 Motivation

Performance analysis of computer networks is rapidly gaining importance as networks increase in size and geographical extent [9]. The size of the networks and the inherent complexity of network protocols complicate this analysis. Nowadays, researchers often use a simulation approach when conducting network performance analysis. Simulation offers a good method of studying computer networks since one can simulate the details of actual protocols and analyze the performance of different protocols on the network. Although there is considerable effort involved in building a simulator and substantial machine cost in running it, this approach is still the most attractive.

As simulation becomes a more popular tool for network performance analysis, various general-purpose network simulators have been developed by different research groups. These general-purpose network simulators make simulation easier by capturing characteristics of real network components and providing a modular programming environment, often composed by links, nodes, and existing protocol suites. For instance,

a link may contain transmission and propagation delay modules, and a node may contain routing tables, forwarding machinery, local agents, queuing objects, TTL objects, interface objects and loss modules. These modules provide a flexible environment to simulate network behavior. As a result, building a simulation toolkit based on one of those general-purpose network simulators would be a good way for analyzing the performance of different URN resolution systems.

1.3 The Project

Previous research studies on URN resolution system have emphasized the system requirements, basic architecture and the design of RDS models. However, few if any studies have been done on the performance analysis of URN resolution systems using the simulation approach. This thesis project will involve building a simulation toolkit for URN resolution using one of those general-purpose network simulators. With this simulation toolkit, we can then study URN resolution system in detail and compare the performance of different RDS models.

1.4 Roadmap

In the remainder of this document, Chapter 2 discusses two existing general-purpose network simulators in details and compares their features. Chapter 3 describes URN resolution in general and gives a framework for the resolution process. It also describes in details several different RDS models that have been developed. Chapter 4 presents the simulation toolkit itself. It includes both the design and implementation of different

modules that compose the toolkit. Chapter 5 gives a demonstration of the toolkit using a particular RDS model. Finally, Chapter 6 concludes this report and suggests future work

Chapter 2

Related Work

REAL and NS are two general-purpose network simulators that are commonly used by network researchers. REAL is a network simulator originally intended for studying the dynamic behavior of flow and congestion control schemes in packet-switched data networks. It provides users with a way of specifying such networks and to simulate their behavior. NS is a discrete event simulator targeted at networking research. NS provides substantial support for simulation of TCP, routing, and multicast protocols.

2.1 REAL

REAL (REalistic And Large) [9] is built using the NEST (NEtwork Simulation Testbed) package from Columbia University. The simulator takes as input a scenario, which is a description of network topology, protocols, workload and control parameters. Scenarios are described using NetLanguage, a simple ascii representation of the network. REAL

produces output statistics such as the number of packets sent by each source of data, the queuing delay at each queuing point, and the number of dropped and retransmitted packets.

The REAL simulator consists of two parts – a simulation server, and a display client. The server carries out simulations, and connects through a Berkeley UNIX socket to the client. The client maintains a 'simulation window' which reflects the current state of the simulation. A number of control panels allow simulation parameters to be set interactively. Users can also quickly build simulation scenarios with a point-and-click interface. This graphical display is very useful as it allows users to monitor the simulation in real time and makes the simulator fun to use.

REAL simulates a packet switched, store, and forward network similar to existing wide area networks such as the Xerox corporate net and the DARPA Internet. The network layer is datagram oriented, and packets can be lost or delivered out of sequence. The transport layer, which is modeled on the Transmission Control Protocol (TCP), provides reliable, sequenced packet transmission. The network consists of a set of sources that execute a transport protocol, gateways that route and schedule packets, and sinks that absorb packets sent to them, returning an acknowledgment for each packet received.

REAL provides around 30 modules (written in C) that exactly emulate the actions of several well-known flow control protocols (such as TCP), and 5 research scheduling disciplines (such as Fair Queuing and Hierarchical Round Robin). The modular design of the system allows new modules to be added to the system with little effort.

2.2 NS

NS [18] began as a variant of the REAL network simulator in 1989 and has evolved substantially over the past few years. The NS development effort is now an ongoing collaboration with the VINT project. NS is an event-driven network simulator. An extensible simulation engine is implemented in C++ [4] that uses MIT's Object Tool Command Language [20], OTcl (an object oriented version of Tcl) as the command and configuration interface.

The simulator is invoked via the NS interpreter. A simulation is defined by an OTcl script. In the script, a user needs to define a network topology and to configure the traffic sources and sinks. The user may create smaller and simpler topologies by writing the definition in OTcl by hand. For larger and more complex topologies, user can use one of several network topology generators. Both the GT-ITM topology generator [22] and the Tiers topology generator [3] currently support NS. After generating network topologies using these network topology generators, conversion programs are available for converting the topologies to NS format.

Unlike REAL, NS does not contain a graphic display that allows users to monitor the simulation in real time. However, the network animator ``nam'' has been developed to serve as a simple tool for animating packet trace data. Nam supports trace data derived as output from NS as well as from real network measurements such as tcpdump.

NS provides extensive support for collecting output or trace data on a simulation. Besides recording counts of various interesting quantities such as packet and byte arrivals and departures, it can also record each individual packet as it arrives, departs, or is dropped at a link or queue.

NS provides simulation researchers with an extensive, and extensible, library of network traffic generators, common algorithms and protocols so that researchers can save time on building common basic modules and perform performance analysis over a wide range of simulation environments. For example, it provides four schedulers implemented using different data structure, various scheduling algorithms, different types of TCP (Tahoe, Reno, Vegas, SACK), unicast and multicast routing protocols. NS also includes a small collection of mathematical functions used to implement random variable generation and integration.

2.3 Summary

This chapter described two existing general-purpose network simulators that are popular among network researchers. As mentioned before, general-purpose network simulators are very powerful tools. They are useful for network researchers who want to test ideas or newly designed protocols. They usually serve as building blocks and give researchers a head start when designing a network simulation. People can use the library of modules that a general-purpose network simulator provides to build basic components and then write their own modules according to their specific needs. In this way, they can save time on building common components and concentrate on the details of the specific components that they want to build.

The toolkit that we are going to describe in the next few chapters was built on NS. We think that NS is a better choice for us because it provides a much bigger library of modules than the one that REAL provides. Unlike REAL, NS does not have a convenient graphical interface that allows users to monitor the simulation in real time. However, NS

can support the building of network simulation in a much larger scale. It allows the use of network topology generators for users to create large and complex topologies. It comes with a huge library of modules including the basic components essential to network simulation as well as different types of protocols and algorithms.

Chapter 3

Uniform Resource Name Resolution

This chapter explains URN resolution systems in detail. It first starts by pointing out the deficiency of URLs, which are widely used in the current World Wide Web. As an alternative, URNs are designed to overcome the shortcomings of URLs. In order to use URNs, a resolution system is needed. A resolution framework is laid out later in this chapter to explain basic components that are important in URN resolution.

A URN resolution system usually consists of two parts: the resolvers, which store the information about how to resolve URNs into URLs, and a Resolution Discovery Service (RDS), which finds the appropriate resolver for a URN. In the end of this chapter, we describe three different RDS models that have been developed by researchers on URN resolution.

3.1 URL

Many different types of names used in network applications have limitations that make them unsuitable for a long-lived global infrastructure. As the World Wide Web becomes more and more popular, the importance of using truly global, long-lived names increases dramatically. Currently, the Web consists of pages linked together by Uniform Resource Locators (URLs). However, the URL naming scheme has a major weakness in supporting a long-lived global infrastructure. URLs incorporate names, such as email addresses, hostnames and pathnames, which have limited scope. The names display poor longevity as they identify a location instead of a resource, causing the problems such as dangling links and poor load distribution.

3.2 URN

Uniform Resource Names (URNs), long-lived, globally unique identifiers, which identify resources independent of network location, are proposed as an alternative way to reference network objects [2]. Ideally, URNs lack semantic information, such as protocols, hostnames, pathnames, and other volatile information to ensure their longevity. To overcome the shortcomings of URLs, URNs give a long-lived resource a permanent name by adding a layer of indirection. A system is needed to resolve URNs into a lower level of information, such as URLs. The remainder of this chapter describes how such a URN resolution system may look like and various important components such a system may need.

3.3 URN Resolution Framework

URNs are designed to be location independent. Therefore, in order to find the location of a resource named by a URN, we need a URN resolution service that knows the mapping between URNs and their locations. This service is essentially a global lookup table. Whenever a resource is moved, its URN remains unchanged and the only thing that needs to be modified is its mapping in the table. One thing to notice is that resources should be named by URNs that are pure identifiers, that is, names with an extremely general syntax giving no indication of how to find the appropriate resolution services. The less information a name contains, the longer it will last and the broader its applicability.

The URN resolution system can generally be divided into two major parts. The first part is the resolvers, which is a distributed set of servers that can resolve URNs into URLs or other lower level of information. Some resolvers may provide direct access to resources as well. The other part is the Resolution Discovery Service (RDS), which finds the appropriate resolver for a URN. Some RDS designs may also incorporate resolver functionality [13].

A general URN resolution framework was initially laid out in [13] and extended in [12]. The framework uses hints to indicate how to resolve a URN. A hint may be the URN of a resolver that may further resolve the URN or the address of such a resolver. A more detailed discussion of hints can be found in chapter 4.3.1. Although a hint will usually be correct, it is not guaranteed to be. The resolution process is simply trying to find the correct hint, starting from fast, unreliable sources to slow, reliable ones.

URN resolution consists of a number of different components. Clients make requests for resolving URNs, possibly through local proxies that provide an abstraction

for the resolution process. The proxy will likely consult a Resolver Discovery Service that will find the appropriate resolver. Publishers publish Web pages with URNs and hints on some storage servers. Naming authorities maintain the resolvers that map URNs into other information.

Within the resolution framework, a client begins by obtaining a URN from some source. To resolve the URN, the client will first try to consult the source since the source may cache hints for the URN. If the source has no hint or stale ones, the client will then have to contact the authoritative resolver for the URN. Since a URN does not contain information about its resolver, an RDS which contains a mapping of URNs to resolvers is needed.

The proxy server isolates clients from the RDS. This abstraction leaves the responsibility of choosing an appropriate RDS scheme to the proxy. The client makes a request to the local proxy server for resolving a URN. The proxy may have cached the hints for the URN and return them to the client. If there is no hint or stale ones in the cache, the proxy chooses the appropriate RDS, which returns hints containing resolution information, usually a resolver for the URN. In some cases, the hints may contain the actual URL for the URN. If a resolver is returned, the client makes a request to the resolver. In return, the resolver may resolve the URN to a URL. Alternatively, the authority over the URN could have been delegated to another resolver. In this case, the resolver would return a hint containing another resolver.

3.4 Different RDS Models

There are two extreme approaches for the design of an RDS, based on how the resolution responsibility is distributed. The first one is using a very deep model. It is similar to the DNS naming system, where delegation of resolution responsibility mirrors the structure of the name. The other one is using a very flat model in which the URN namespace is treated as one flat subspace. Authority may still be delegated, but all resolution responsibility remains with a central entity.

A number of different RDS models have been developed. The architecture of three different RDS schemes is discussed in detailed in the remainder of this section. The NAPTR system, one of the earliest RDS designs, uses the DNS to resolve a URN to an authoritative resolution server. mu (Mulitcast URN resolver) is an RDS model designed to support fine-grained caching and replication, which uses a Multicast Protocol for sharing cached data. In the last model, an RDS is divided into two main components, a distributed and replicated component to serve the needs of the URN users and a centralized and more conventional part for publishers to register hints.

3.4.1 NAPTR

NAPTR [2] was proposed to the Internet Engineering Task Force URI working group as a possible RDS model for URN resolution. NAPTR relies on a series of DNS lookups and rewrites of the URN to locate a resolver. It first extracts a part of a URN, creates a DNS name, and looks up its DNS NAPTR record, which contains two transformation fields. One is a new string to lookup. The other is a regular expression. When it is applied to the original URN, it will extract some part to lookup. To resolve a URN, a client

performs a series of transformations and queries until a failure occurs or a terminal NAPTR record which points to a resolver is returned.

The NAPTR proposal has a number of deficiencies. First, it does not promote longevity due to its reliance on DNS. Furthermore, NAPTR works best when the URN contains a hostname or at least has certain kinds of structure. If the URN namespace is very large and flat, NAPTR will not scale well. Finally, NAPTR relies on cryptic rewrite rules that make the delegation of naming authorities hard to determine and control.

3.4.2 mu

mu (Multicast URN resolver) [6] is an RDS design that is intended to support scalability and provide the flexibility required for the usage of persistent names. The word 'multicast' in the title refers to a multicast cache-sharing protocol allowing sets of cooperating servers to share cached and mirrored data. Multicast is also used in discovery. mu implements a distributed database designed to support fine-grained caching and replication while offering the degree of flexibility of delegation needed for URN resolution.

Within the resolution framework of mu, URNs fall into blocks of names. There is a naming authority for each block and the authority is delegated hierarchically. The naming authority is represented by a machine that holds the authoritative hints for URNs in that block. The blocks may be replicated. Hence there is a tree of authoritative servers holding authoritative information. For each node in the tree, there may be a multicast group of replicas of the information. The block information is grouped into zones for purposes of distribution of updates of block information. There is no central authoritative

source for correct information, but rather that is distributed among the authoritative servers. A server may be authoritative for more than one block and these blocks need not be near each other in the block hierarchy.

At the other end of the spectrum, each client talks to a local proxy server that caches hint information. Each such server is part of a "neighborhood". Each server in a neighborhood is responsible for replicating one or a number of blocks from the tree above. In being responsible for a block, a server becomes part of the multicast group for the zone containing that block. Among the servers in a neighborhood the whole tree is covered. If a server is responsible for a block, it is also responsible for all the blocks higher than it in the tree. The server for a block need not necessarily keep all the information in a zone; it may need to re-acquire information in a block, if it decides not to retain that information.

There are two kinds of activities and two kinds of multicast groups reflecting that. One is managing the infrastructure, the tree, the hints, etc. For this there is a multicast group for each zone, with a zone manager doing all the transmission to that group. When updates are made to the tree structure (new blocks added, etc.) or hint information updated (determined by the appropriate block authority), those updates will be sent to the zone manager. All the updates for a zone will be collected and sent out periodically by the zone manager to the multicast group for that zone.

The other kind of activity is resolving URNs. To resolve a URN, the client first tries the hints obtained from the source. If these hints do not work or there is no hint available, the client then sends a request to a local proxy server. The server may have cached hints for the URN. If there is no hint or stale ones, it will send a multicast to its

neighborhood. If one of its neighbors has the correct hints, it returns the hints to the server, which then returns them to the client. If none of the neighbors has the correct hints, then at least one of the members of the neighborhood is definitely responsible for the block containing that URN. Therefore, the responsible server(s) go directly to the authoritative server for that information. They then can cache that specific information, and also return it to the server helping the client. In order to do this, each server must know the mapping of all blocks for which it is responsible to authoritative servers, so this information is part of what the above mechanism is keeping up to date.

3.4.3 Distributed RDS

The third RDS model [12] is designed to support an extremely large and flat namespace of URNs. Special attention has been paid to the scalability problem. The design focus was on supporting mobility and longevity of URNs.

The basic architecture consists of two main components, the central database and the distributed database. The central database contains the official version of the delegation of naming authority and all subspace hints. It is not accessible to clients who want to resolve URNs. It only has interaction with publishers (or naming authorities) who want to register hints for new URNs or modify old ones.

The distributed database, on the other hand, is widely available to clients for URN resolutions. It determines the subspace into which a URN falls and returns the hints containing information about the resolver for the URN. It handles the issue of scalability by employing a high degree of distribution and replication. The database is structured as a B-tree of replication groups. Unlike the DNS, this hierarchy is not based on the

structure of the URN namespace. A B-tree always remains balanced and every leaf has a constant depth, thus minimizing the number of servers involved in a search.

Each node of the B-tree is a replication group that is responsible for a portion of the database. Each replication group is composed of a set of members that maintain the same sets of hints. One member in each group serves as the leader and acts for the group as a whole. The leader is responsible for tasks such as adding and removing members, receiving and distributing updates, and making sure the group does not get overload.

The central database sends updates to the root replication group, which in turn filter down to the appropriate places in the tree. URN resolution can be initiated at any member. To resolve a URN, a client begins by making a request to the local distributed database server, which serves as an intermediary between the user and the RDS. If the server does not have the hints in its cache, it carries out the lookup in the database by routing queries through the tree to a member at the replication group that contains the hints for the requested URN.

3.5 Summary

In this chapter, we gave the reader a general introduction to the basics of a URN resolution system. We presented a layout of the resolution framework and identified the important components in the general model of URN resolution. Resolvers resolve URNs into URLs. An RDS is needed to determine the appropriate resolver for an URN. Clients make requests for resolving URNs through proxy servers. Servers resolve the URNs for the clients using a particular RDS scheme. Servers return hints containing resolution information, possibly the resolvers, to clients. Publishers publish Web pages with URNs

and hints on servers. NAPTR, mu, and the Distributed RDS are three different RDS models that have been developed. We presented a brief summary of each model in the end of the chapter.

Chapter 4

The Simulation Toolkit

This chapter describes the simulation toolkit in detail. NS, a general-purpose network simulator serves as a base for our simulation toolkit to build on. NS is an object-oriented simulator, written in C++ [4], with an OTcl [20] interpreter as a frontend. It provides some basic modules serving as primitive building blocks. Nodes, links, and agents are three of core components that compose a network simulation. Every network consists of a number of nodes interconnected by links. Each node in the network has a unique address. A link connects two nodes, which can be either unidirectional or bi-directional. Agents are the objects that actively drive the simulation. Agents can be thought of as the processes or transport entities that run on nodes that may be end hosts or routers. Traffic sources and sinks, dynamic routing modules and the various protocol modules are examples of agents.

In addition to the basic components, our simulation toolkit consists of a number of additional modules that are required for a URN resolution system. Clients, servers and publishers are the three of the main components in the general URN resolution model. In this simulation toolkit, each resides on some node in the network. Two objects can communicate with each other by sending messages based on TCP connections only if the nodes on which they reside are connected by a link. Hint is another important component in a URN resolution system. Hints contain resolution information. They represent the primary type of messages that are being sent among servers, clients and publishers.

This chapter begins with a description of the OTcl interpreter that serves as the interface for NS and our simulation toolkit. Next, the basic modules provided by NS including nodes, links and agents are described. They are then followed by a detailed explanation of each of the additional modules that we built for running simulation on URN resolution.

4.1 OTcl Linkage

NS is an event-driven network simulator. It contains an extensible simulation engine implemented in C++ that uses MIT's Object Tool Command Language, OTcl (an object oriented version of Tcl) as the command and configuration interface. The simulator supports a class hierarchy in C++ (the compiled hierarchy), and a similar class hierarchy within the OTcl interpreter (the interpreted hierarchy). The two hierarchies are closely related to each other. From the user's perspective, there is a one-to-one correspondence between a class in the interpreted hierarchy and one in the compiled hierarchy.

The root of the interpreted hierarchy is the class TclObject. Users create new simulator objects through the interpreter. These objects are instantiated within the interpreter, and are closely mirrored by a corresponding object in the compiled hierarchy. The interpreted class hierarchy is automatically established through methods defined in the class TclClass. User-instantiated objects are mirrored through methods defined in the class TclOjbect.

There are a number of classes that compose the OTcl interpreter. The class Tcl contains the methods that C++ code will use to access the interpreter. The class TclObject is the base class for all simulator objects that are also mirrored in the compiled hierarchy. The class TclClass defines the interpreted class hierarchy, and the methods to permit the user to instantiate TclObjects. Finally, the class InstVar contains methods to access C++ member variables as OTcl instance variables.

4.1.1 Class Tcl

The class Tcl encapsulates the actual instance of the OTcl interpreter, and provides the methods to access and communicate with that interpreter. The class provides methods for the following operations:

- obtain a reference to the Tcl instance
- invoke OTcl procedures through the interpreter
- retrieve or pass back results to the interpreter
- report error situations and exit in a uniform manner
- store and lookup "TclObjects"
- acquire direct access to the interpreter

4.1.2 Class TclObject

The class TclObject is the base class for most of the other classes in the interpreted and compiled hierarchies. Every object in the class TclObject is created by the user from within the interpreter. An equivalent shadow object is created in the compiled hierarchy. The two objects are closely associated with each other. The following list describes some of the important procedures of this class.

Creating TclObject

By using **new{}**, the user creates an interpreted TclObject. The interpreter will execute the constructor **init {}** for that object. The **init{}** procedure will automatically create the compiled object and perform the initialization required in this class. Finally, **new{}** returns a handle to the interpreted object that can be used for further operations upon that object. Notice that the user simply performs operations on the interpreted object and therefore does not need to have a handle to the compiled object.

Deleting TclObject

The **delete** operation destroys the interpreted object, and the corresponding shadow object. The destructor automatically invokes the procedure **delete-shadow**, that in turn invokes the equivalent compiled method to destroy the shadow object.

Variable Bindings

The **set** operation allows users to set and change a value of a variable for a TclObject. It automatically updates the value of the variable for both the interpreted object and the corresponding compiled object.

Variable Tracing

TclObject also supports tracing of both C++ and Tcl instance variables. Users can use the **trace** method to trace variables for TclObjects. The first argument to the **trace** method must be the name of the variable. The optional second argument specifies the trace object that is responsible for tracing that variable. If the trace object is not specified, the object that owns the variable is responsible for tracing it.

4.1.3 Class TclClass

The class TclClass is a pure virtual class. Classes derived from this base class provide two functions: construct the interpreted class hierarchy to mirror the compiled class hierarchy; and provide methods to instantiate new TclObjects. Each such derived class is associated with a particular compiled class in the compiled class hierarchy, and can instantiate new objects in the associated class.

4.1.4 Class InstVar

The class InstVar defines the methods and mechanisms to bind a C++ variable in the compiled object to a specified OTcl variable in the equivalent interpreted Object. The binding is set up such that the value of the variable can be set or accessed either from within the interpreter, or from within the compiled code at all times.

An OTcl variable is created by specifying the name of the interpreted variable, and the address of the C++ variable in the compiled object. The constructor for the base class InstVar creates an instance of the variable in the interpreter, and then sets up a trap routine to catch all accesses to the variable through the interpreter.

Whenever the variable is read through the interpreter, the trap routine is invoked just prior to the occurrence of the read. The routine invokes the appropriate **get** function that returns the current value of the variable. The value is then used to set the value of the interpreted variable that is then read by the interpreter.

Likewise, whenever the variable is set through the interpreter, the trap routine is invoked just after the write is completed. The routine gets the current value set by the interpreter, and invokes the appropriate **set** function that sets the value of the compiled member to the current value set within the interpreter.

4.2 Basic Modules

This section describes a number of basic modules that NS has provided. The class Simulator provides a set of interfaces for configuring a simulation. The classes Node and Link are two important elements for defining the network topology. Agents represent endpoints where network-layer packets are constructed or consumed, and provide some functions helpful in developing transport-layer and other protocols.

4.2.1 Class Simulator

The class simulator provides a set of interfaces for configuring a simulation and for choosing the type of event scheduler used to drive the simulation. A simulation script, written in OTcl, generally begins by creating an instance of this class and calling various methods to create nodes, topologies, and configure other aspects of the simulation.

The simulator is event-driven. There are presently four different schedulers available in the simulator, each of which is implemented using a different data structure:

- **The List Scheduler** – uses a linked-list structure (default). The list is kept in time-order (earliest to latest)
- **The Heap Scheduler** – uses a heap structure. This structure is good for a large number of events, as insertion and deletion times are in O(log n) for n events [1].
- **The Calendar Queue scheduler** – uses a data structure analogous to a one-year desk calendar, in which events on the same month/day of multiple years can be recorded in one day.
- **The Real-Time Scheduler** – attempts to synchronize the execution of events with real-time.

A scheduler runs by selecting the next earliest event, executing it to completion, and returning to execute the next event. An event consists of a "firing time" and a handler function:

```
class Event {
public:
        Event* next_;                                   /* event list */
        Handler* handler_;              /* handler to call when event ready */
        double time_;                       /* time at which event is ready */
        int uid_;                                       /* unique ID */
        Event ( ) : time_(0), uid_(0) { }
};

/*
 * The base class for all event handlers. When an event's scheduled
 * time arrives, it is passed to a handler which must consume it.
 */
class Handler {
public:
        virtual void handle (Event* event) = 0;
};
```

Here is a simple example to show how to setup a simulation and select the type of scheduler used for the simulation:

```
...
set ns_ [new Simulator]
$ns_ use-scheduler Heap
$ns_ at 300.5 "$self complete_sim"
...
```

This OTcl code fragment first creates a simulation object, then changes the default scheduler implementation to be heap-based, and finally schedules the function $self complete_sim to be executed at time 300.5.

The Simulator class provides a number of methods used to set up the simulation. The following is a list of some of the methods:

- Simulator instproc now – return scheduler's notion of current time
- Simulator instproc at <args> – schedule execution of code at specified time
- Simulator instproc cancel <args> – cancel event
- Simulator instproc run <args> – start scheduler
- Simulator instproc halt – stop (pause) the scheduler
- Simulator instproc flush-trace – flush all trace object write buffers
- Simulator instproc create-trace type files src dst – create trace object
- Simulator instproc create_packetformat – set up the simulator's packet format

4.2.2 Class Node

Recall that each simulation requires a single instance of the class Simulator to control and operate that simulation. Besides keeping track of each element in the simulation, the Simulator class also provides instance procedures to create and manage the network topology including nodes, links, agents, etc. At a more detailed level, for example, the class Node provides methods to access and operate on individual nodes.

The basic primitive for creating a node is

```
set ns [new Simulator]
$ns node
```

The instance procedure **node** is used to construct a node. By default nodes are constructed for unicast simulations. In order to create nodes for multicast simulation, the class variable, EnableMcast_, should be set to 1, as:

```
Simulator set EnableMcast_ 1
```

before any nodes are created.

Procedures to configure an individual node can be classified into:

- Control functions
- Address and Port number management
- Agent management
- Adding neighbors

Control functions

$node entry returns the entry point for a node. This is the first element which will handle packets arriving at that node.

$node reset will reset all agents at the node.

$node enable-mcast convert a unicast node to a multicast node.

Address and Port number management

$node id returns the node number of the node. This number is automatically incremented and assigned to each node at creation by the class Simulator method $ns node. The class Simulator also stores an instance variable array, Node_, indexed by the node id, and contains a reference to the node with that id.

$node agent <port> returns the handle of the agent at the specified port.

$node reset <port> resets all agents attached to this node.

alloc-port returns the next available port number.

Agent management

$node attach <agent> attaches an agent of type <agent> to this node. It adds the agent to its list of agents_, and assign a port number to the agent.

$node detach <agent> detaches an agent of type <agent> from the list of agents_ of this node.

$node join-group <agent> <group> adds the agent specified by <agent> to the multicast host group specified by <group>.

Adding neighbors

$node neighbors returns a list of the neighbor node objects. Each node keeps a list of its adjacent neighbors in its instance variable neighbor_.

$node add-neighbor <node> adds a neighbor to its list.

4.2.3 Class Link

Link is another important aspect of defining the network topology. As mentioned before, the class Simulator provides methods for creating links to connect the nodes. The class Link provides instance procedures for manipulating the links.

The following describes the syntax for creating a simplex link between two nodes:

```
set ns [new Simulator]
set node0 [$ns node]
set node1 [$ns node]
$ns simplex-link $node0 $node1 <bandwidth> <delay> <queue_type>
```

The simple-link command creates a new unidirectional link between node0 and node1 with specified <bandwidth> and <delay> characteristics. The link uses a queue of type

<queue_type>. The procedure also adds a TTL checker to the link. The procedure duplex-link creates a bi-directional link between two nodes in the similar manner.

There are a number of instance variables that define a link:

- **head_** : entry point to the link, it points to the first object in the link.
- **queue_** : reference to the main queue element of the link.
- **link_** : reference to the element that actually models the link, in terms of the delay and bandwidth characteristics of the link.
- **ttl_** : reference to the element that manipulates the ttl in every packet.
- **drophead_** : reference to an object that is the head of a queue of elements that process link drops.

The instance procedures in the class Link include:

$link head returns the handle for head_.

$link queue returns the handle for queue_.

$link link returns the handle for the delay element, link_.

$link cost <cost-val> set the cost of this link to <cost-val>

$link cost? Returns the cost of this link. Default cost of link is 1, if no cost has been specified earlier.

4.2.4 Class Agent

Agents represent endpoints where network-layer packets are constructed or consumed, and provide some functions helpful in developing transport-layer and other protocols. A new protocol can be added by creating a class derived from Agent.

The class Agent has the following instance variables:

- **addr_** : node address of this agent.
- **dst_** : the destination of the packet being sent
- **size_** : packet size in bytes
- **type_** : type of packet
- **fid_** : the IP flow identifier
- **prio_** : the IP priority field
- **flags_** : packet flags
- **defttl_** : default IP ttl value

The values of the instance variables can be modified using the **set** function. For example:

```
Agent set addr_ 0
Agent set size 210
```

The class Agent supports packet generation and reception. The following member functions are implemented by the C++ Agent class:

Packet* allocpkt( ) allocates a new packet and assign its fields.

Packet* allocpkt(int) allocates a new packet with a data payload of n bytes and assign its fields.

void timeout(timeout number) calls a timeout when a packet expires.

void recv(Packet*, Handler*) receives a packet and performs a series of actions depending of the type of agent.

These functions are implemented by the OTcl Agent class:

$agent port returns the transport-level port of the agents. Ports are used to identify agents within a node.

$agent dst-addr returns the address of the destination node this agent is connected to.

$agent dst-port returns the port at the destination node that this agent is connected to.

$agent conncet <addr> <port> connects this agent to the agent identified by the address <addr> and port <port>. This causes packets transmitted from this agent to contain the address and port indicated, so that such packets are routed to the intended agent.

$agent attach-source <type> installs a data of type <type> in this agent. This will be explained in more detail later.

Agents are used in the implementation of protocols at various layers. There are several types of agents supported in the simulator. They are subclasses derived from the class Agent. This is a list of some of the common ones:

- **TCP** : a "Tahoe" TCP sender
- **TCP/Reno** : a "Reno" TCP sender
- **TCPSink** : a Tahoe or Reno TCP receiver
- **CBR** : connectioness protocol with sequence numbers
- **CBR/RTP** : an RTP sender and receiver
- **CBR/UDP** : an UDP sender and receiver
- **LossMonitor** : a packet sink which checks for losses
- **TCP/Message** : a protocol to carry textual messages based on TCP

Example:

The following OTcl code fragment demonstrates how to create a TCP agent and set it up:

```
set tcp [new Agent/TCP]                          ; create sender agent
$tcp set fid_ 2                                  ; set IP-layer flow ID
set sink [new Agent/TCPSink]                     ; create receiver agent
$ns attach-agent $node0 $tcp                     ; put sender on $node0
$ns attach-agent $node1 $sink                    ; put receiver on $node3
$ns connect $tcp $sink                           ; establish TCP connection
set ftp [new Source/FTP]                         ; create an FTP source application
$ftp set agent_ $tcp                             ; associate FTP with the TCP sender
$ns at 1.2 "$ftp start"                          ; arrange for FTP to start at time 1.2
```

4.3 URN Resolution System

As mentioned before, a URN resolution system can be generally divided into two major parts, the resolvers, which can resolve URNs into URLs or other lower level of information, and the Resolution Discovery Service (RDS), which finds the appropriate resolver for a URN. For this simulation toolkit, our main focus is in the RDS, as we want to investigate how different RDS models vary in performance. Therefore, we didn't implement the resolver part in this simulation toolkit. Instead, hints are used to indicate how to resolve a URN. The resolution process is simply trying to find the appropriate resolver or the correct hint for a URN.

This section presents a number of important components that compose a simple URN resolution system. Clients make requests for resolving URNs. Publishers publish information consisting of URNs and hints on some servers. Servers have a number of different responsibilities. They serve as storage for information, such as web pages that publishers publish. They also accept resolution requests from local clients and try to resolve the URNs for them. In addition, they store a list of hints for some URNs. These could be hints for popular URNs that a server chooses to cache. Alternatively, a server could be an owner for a URN and therefore keeps the official hint for that URN.

In the simulation toolkit, clients, servers and publishers reside on nodes in the network. They communicate with each other through the TCP protocol that NS provides. Two objects can communicate only if the nodes on which they reside are connected by a link.

4.3.1 Hint

A typical hint would be a mapping of a URN to the appropriate resolver for the URN. In some cases, a hint could also be a mapping of a URN to the actual URL. In general, a hint is just a list of ways indicating how a URN can be resolved.

Typically, a hint for a URN is first created by the publisher of the URN. The publisher puts the information that is needed for resolving the URN into the hint. It then puts both the URN and the hint on a server, which becomes the resolver of this URN.

A hint will be marked as expired when the resolution information it contains is no longer valid. A stale hint may be replaced by a newer hint. The server that owns the URN always tries to keep track of the latest hint for the URN.

Each hint contains the following information:

urn_ : This indicates which URN this hint is representing and serves as a key or identifier for this hint. Each URN is simply a name and implemented as a string.

timestamp_ : A time stamp tells when the hint was created and hence its age. During an update, a hint with a newer time stamp replaces a previous one with an older time stamp.

time-to-live_ : This gives an upper bound on the lifetime of a hint. Publishers will set the default maximum lifetime for the hints that they create.

expired_ : If a hint is no longer valid, hopefully it will be marked as expired and may be purged. If an expired hint exists, it may still be more useful than no hint at all.

resolution-data_ : This is a list of ways to resolve the URN. Each item can be the actual resolution, that is a mapping between an URN and the corresponding URL. It can also be a redirection to a resolver along with a protocol or another URN. It may be both a

resolver and a URN, in which case the given resolver should be queried about the given URN.

The following describes the OTcl syntax for creating a new hint.

set hint1 [new Hint <urn> <timestamp> <time-to-live>]

In order to create a new hint, one needs to input the urn that this hint is representing and the timestamp as the arguments. The time-to-live field is optional. If this is not entered by the user, a default value that is set by the publisher of this urn will be used.

The Hint class supplies a number of methods that allow the user to manipulate hints. Here is a list of some of the methods:

$hint add-data <resolution data> : add new resolution information for this hint to the list resolution-data_. This list is simply implemented as a linked list. New data are added to the front of the list.

$hint urn : returns the urn that this hint is representing, that is the value of urn_.

$hint time-stamp : returns the time when this hint was created, that is the value of timestamp_.

$hint time-to-live : returns the maximum lifetime of this hint, that is the value of time-to-live_.

$hint expired : returns the value of expired_. It returns true if this hint is expired and false if this hint is still valid.

$hint mark-expired : marks this hint as expired by changing the value of expired_ to true.

$hint get-data : returns the resolution-data_ list.

4.3.2 Client

Clients in this simulation toolkit represent users in the Web who want to obtain Web pages or other information. In this toolkit, we assume that clients somehow obtain URNs of Web pages from some sources and keep a bookmark list of URNs. To simulate the resolution process, it begins with a client just randomly picking a URN from the bookmark list and making a request for resolving that URN. In reality, it usually begins with a client obtaining a URN from some source. Since the source may have cached hints for the URNs that it provides, consulting the source would be the primary way for clients to resolve URNs. Clients would use the resolution system only when the source could not provide valid hints for the URNs.

The following is a list of state variables that a client has:

local-server_ : This points to the local server that serves this client. It can be viewed as a proxy server, which isolates the client from the RDS. The client makes requests for resolving URNs to this local server. The server may have cached the hints and return them to the client. Otherwise, the server is responsible for finding the appropriate hints using a particular RDS scheme.

bookmark_ : This is a list of URNs that the client keeps. This could be a bookmark list of Web pages or simply a list of URNs that the client frequently checks. The list is implemented as a linked list. Each entry contains a URN.

cnode_ : This is the node on which this client resides. A client can communicate with other servers only if the node on which it resides and the nodes on which the servers reside are connected.

ns_ : This refers to the simulator object that controls the whole simulation. This reference is needed by some of the client's methods that handle communications with servers using the protocols provided by the basic modules.

The following describes the OTcl syntax for creating a client:

set client1 [new client <local server> <node> <ns>]

In order to create a new client, one needs to input the local server for the client, the node on which the client resides, and the simulator object as arguments.

The main function of a client in our simulation toolkit is to make requests for resolving URNs. Typically, a client randomly picks a URN from its bookmark list and sends the request to its local server. It then waits for the local server to return the appropriate hints.

Here is a list of methods that the client class has:

$client cnode : returns the node on which this client resides.

$client local-server : returns the local server for this client, that is the value of local-server_.

$client bookmark : returns the list of URNs that bookmark_ contains. This is simply a linked list of URNs.

$client add-bookmark <urn> : adds the given URN to the bookmark list. New URN is added to the front of the list.

$client remove-bookmark <urn> : removes the given URN from the bookmark list. It has to search through the list from the beginning until it finds the given URN, which is not a very efficient algorithm.

$client pick-bookmark : returns a URN that is randomly picked from the bookmark list.

$client request <urn> : sends a request for resolving the given URN to the client's local server. To resolve a URN, a client opens a TCP connection to a port of the node on which the local server resides and sends the URN.

Currently, when a client receives the hint returned by the local server, it does not do anything with it since we do not implement the resolver part of the URN resolution system in our simulation toolkit. In the future, if the resolver part is added to the simulation toolkit, a client can actually use the resolution information given by the hint to arrive at the actual URL for the URN.

4.3.3 Server

Servers in this toolkit have two main functions. First, servers store URNs and hints that are published by publishers. Publishers can update or remove the URNs and hints that they have published on the servers. The other function of servers is to resolve URNs requested by clients. Each server caches a list of hints for URNs that are frequently requested by clients. When a server receives a request for resolving an URN, it first checks to see if the hint for the URN is cached. If there is no hint or only stale ones in the cache, the server tries to resolve the URN using a specific RDS scheme. Finally, it returns the resulting hint to the client and it may cache the hint as well.

Each server has the following list of variables:

cached-hints_ : This is a list of hints that this server chooses to cache. A server may choose to cache hints for the URNs that are frequently requested by its clients. When a server receives a request from a client for resolving a URN, it first checks the cache-

hints_ list. If the list does not contain the appropriate hint, the server then tries to resolve the URN by finding the appropriate hint using a specific RDS scheme.

stored-data_ : This is a list of URNs and the corresponding hints that are published on this server by publishers. It is implemented as a linked list. Each entry contains a URN and a hint.

snode_ : This is the node on which this server resides. A server can communicate with other objects such as clients, publishers and other servers only if the node on which it resides and the nodes on which the other objects reside are connected.

ns_ : This refers to the simulator object that controls the whole simulation. This reference is needed by some of the server's methods that handle communications with other objects using the protocols provided by the basic modules.

The following describes the OTcl syntax for creating a server:

set server1 [new server <node> <ns>]

In order to create a new server, one needs to input the node on which the server resides, and the simulator object as arguments.

Here is a list of methods that the server class provides:

$server snode : returns the node on which this server resides.

$server cache-hint <hint> : adds the given hint to the cached-hints_ list. The new hint is added to the front of the list.

$server remove-cached-hint <urn> : removes the hint for the given URN from the cached-hints_ list. It searches through the list until the hint for the given URN is found, and then removes it. It does not do anything if the hint for the given URN is not found in the list.

$server find-cached-hint <urn> : searches through the cached-hints_ list for the hint corresponding to given URN. It returns the hint for the given URN if it is found. Otherwise, it returns null.

$server update-cached-hint <hint> : replaces an old hint in the cached-hints_ list with the given new hint. It searches through the list for the hint that is the older version of the given hint, and replaces it. If no older version of the given hint is found in the list, it simply adds the given hint to the list.

$server update-cached-hint <urn> <resolution data> : updates the cached hint for the given URN by adding the given resolution data to the hint. It searches through the cached-hints_ list for the appropriate hint and updates it by calling the add-data method provided by the hint class. If no appropriate hint is found in the list, it simply creates a new hint based on the given URN and resolution data, and adds it to the cache.

$server add-data <urn> <hint> : adds an entry containing the given URN and hint to the stored-data_ list. The new entry is added to the front of the list.

$server remove-data <urn> : removes the given URN and the corresponding hint from the stored-data_ list. It searches through the list for the entry containing the given URN, and removes it. It does not do anything if no matched entry is found.

$server find-data <urn> : searches through the stored-data_ list for the entry containing the given URN, and returns the corresponding hint. If no matched entry is found, it returns null.

$server update-data <urn> <hint> : replaces the old hint for the given URN with the given hint. It searches through the stored-data_ list for the entry containing the given

URN, and replaces its hint with the given one. If no matched entry is found, it simply adds a new entry containing the given URN and hint to the list.

$server update-data <urn> <resolution data> : updates the hint for the given URN by adding the given resolution data to the hint. It searches through the stored-data_ list for the entry containing the given URN, and updates its hint by calling the add-data method provided by the hint class. If no matched entry is found, it simply creates a new entry based on the given URN and resolution data, and adds it to the list.

$server resolve <urn> <client> : resolves the given URN and sends the appropriate hint back to the client using a TCP connection. The server first checks the cached-hints_ list. If the hint for the given URN is not found in the cache, the server resolves the URN using a specific RDS model.

In the current implementation, the cached-hints_ list and the stored-data_ list have no limit on capacity except for the memory constrain of the servers. Alternatively, we can limit the capacity of the each list by setting a maximum length for each list. A new entry is added to the front of the list, and therefore, the oldest entry is at the end. When a new entry has to be added to a list that is already full, the last entry in the list will be removed so that new entry can be added, making it a First-In-First-out queue.

We chose linked list to be the data structure for storing URNs and hints because it provides easy and fast implementation. In addition, insertion to the list is quick as a new item is simply added to the front of the list. However, deletion and searching are not as efficient especially for lists containing large number of items. In the case when users want to do simulations on a large scale, the linked list could be replaced by other more

sophisticated data structure such as a heap or a binary tree that provides more efficient algorithms for deletion and searching.

4.3.4 Publisher

The main function of a publisher is to publish new Web pages or other information on a storage server. The publisher needs to create a URN for the information it wants to publish. In addition, it also creates the corresponding hint containing information on how to resolve the URN. The publisher sends the URN and the hint to the server. It also keeps a list for all the URNs and hints that it has published. When a hint is updated, the publisher updates its own list and sends the update to the server.

In this toolkit, since we do not implement the resolver part of the resolution process, we simulate the publication process by having the publisher publish only the URN and the hint on the server. No actual web pages are stored on the server. The server just keeps a list of URNs and the corresponding hints. In reality, a publisher would put a web page on a storage server, and publish the URN and the hint on another server, or more accurately, a resolver (a server that stores resolution information). In this toolkit, we only implement one kind of server, since only resolution information is stored.

Here is a list of variables that a publisher has:

server_ : This points to the server on which this publisher publishes Web pages or other information. A publisher publishes a web page on a server by sending the URN and the corresponding hint to the server.

publication_ : This is a list of URNs and hints for all the Web pages or other information that this publisher has published. This is implemented as a simple linked list. Each entry contains a URN and the corresponding hint.

pnode_ : This is the node on which this publisher resides. A publisher can communicate with the server on which it wants to publish information only if the node on which it resides and the node on which the server resides are connected.

ns_ : This refers to the simulator object that controls the whole simulation. This reference is needed by some of the publisher's methods that handle communications with the server using the protocols provided by the basic modules.

The following describes the OTcl syntax for creating a publisher:

set publisher1 [new publisher <server> <node> <ns>]

In order to create a new publisher, one needs to input the server on which this publisher will publish information, the node on which the publisher resides, and the simulator object as arguments.

Here is a list of methods that the publisher class provides:

$publisher create-urn : creates a new urn and publishes it on the server that is chosen by this publisher, which is specified by the variable server_. The name for a new urn will be automatically assigned. It then calls the create-hint method to create the hint for this URN. Finally, the URN and the hint are sent to the server using a TCP connection. The publisher also adds the URN and the hint to its publication list. It returns the newly created hint.

$publisher remove-urn <urn> : removes the given urn and the corresponding hint from the server. It sends a request for removing the URN to the server using a TCP connection.

The server will remove the URN and the corresponding hint from its database. The publisher also removes the URN and the hint from its publication list.

$publisher create-hint <urn> : creates a hint for the given URN. The publisher needs to specify information such as time stamp and maximum lifetime for the hint. The publisher also puts the resolution information in the hint.

$publisher update-hint <hint> : performs update for the given hint. It finds the old version of the given hint in its publication list and replaces it with the given hint. It also sends the update to the server on which the hint was published using a TCP connection.

$publisher pnode : returns the node on which this publisher resides.

$publisher server : returns the server on which this publisher publishes Web pages or other information, that is the value of server_.

$publisher publication : returns the list of URNs and hints that publication_ contains. This is a simple linked list. Each entry of the list contains a URN and the corresponding hint.

$publisher add-publication <urn> <hint> : adds the given URN and hint to the publication list. The new entry is added to the front of the list.

$publisher remove-publication <urn> : removes the given URN and the corresponding hint from the publication list. It has to search through the list from the beginning until it finds the given URN, which is not a very efficient algorithm.

$publisher find-hint <urn> : searches through the publication list for the entry containing the given URN, and returns the corresponding hint for this URN. If the given URN is not found in the list, it returns null.

4.4 Summary

NS provides a number of basic modules that are essential to network simulation. They provide methods for building topologies and support for common protocols. They also specify the OTcl interface for users to define simulation environment and details. These modules serve as the building blocks for our simulation toolkit. To simulate a URN resolution system, several important components have to be added. Hints store the resolution information. Clients, servers and publishers reside on nodes that are connected by links. They communicate with each other in the resolution process through the TCP protocol provided by NS.

Chapter 5

Demonstration

By adding additional components, the simulation toolkit can perform simulation on URN resolution using different RDS schemes. This chapter presents a demonstration of one of the RDS models that we have mentioned in chapter 3.4, the Distributed RDS. We have created a simple implementation of the Distributed RDS to demonstrate the use of the simulation toolkit.

5.1 Overview of the model

The Distributed RDS model [12] is designed to support the longevity of URNs in a scalable and reliable manner. The basic architecture consists of two main components, the central database and the distributed database. The central database contains the authoritative version of the hints for all URNs. It is only accessible by publishers for

registering and updating hints. It has no interaction with clients who want to resolve URNs. This isolation can protect the central database from attack.

On the other hand, the distributed database has less protection and higher availability than the central database. It is designed for handling a large number of resolutions. It employs a high degree of distribution and replication to achieve scalability. The distributed database is composed of replication groups arranged in a hierarchical structure. A B-tree structure is used in our implementation. Each replication group is a collection of member servers that maintain the same set of hints for URNs within a certain subspace. One member in each group serves as the leader and acts for the group as a whole.

The central database sends updates to the root replication group, and the updates are filtered down to the appropriate replication group in the tree. URN resolution can be initiated at any member server. Servers route queries through the tree to the replication group containing the needed hints.

5.2 Main Components

This section describes the three main components of the Distributed RDS model: the central database, the distributed database and the replication groups. Similar to the clients, the servers, and the publishers, the central database resides on a node in the simulation. This node should be connected to all the nodes on which the publishers reside on so that the publishers can communicate with the central database to register and update hints using TCP connections.

A replication group consists of a number of servers with one of them being the leader. We need to modify the server class described in the last chapter by redefining the resolve method. The detail of the resolution process will be described later in this chapter. Each server now has to know the replication group in which it belongs. We also have to add to the server class a few methods for communicating with other members in the same replication group.

The distributed database is composed of replication groups arranged in a B-tree structure. The leader of each replication group acts for the group as a whole. Requests and updates are propagated up or down in the tree by the leaders of the replication groups. A leader of a replication group communicates with the leader of its parent or one of its children replication groups using a TCP connection. The central database sends updates to the leader of the root replication group using TCP connections. The updates are then propagated down the tree to the appropriate replication groups.

5.2.1 The Central database

The central database is the place where all the official hints for URNs are stored. Since it does not interact with a huge number of clients who wish to resolve URNs but just the publishers who manage the official hints, the demands on this central database and the amount of communications it has with others is low. As a result, we have decided to implement the central database as a linked list structure for easy and fast implementation. Although searching is extremely slow and inefficient in a linked list structure, this would not be a problem for us since searching is not a frequent operation in the central database and we do not intend to build a very large database for the purpose of this demonstration.

On the other hand, the most frequent operation for the central database would be insertion of new hints, which can be done in O(1) time for a linked list.

Currently, the central database sends an update to the root replication group in the distributed database whenever it receives an update request from a publisher. Hence, only one update will be sent to the root replication group at a time. Alternatively, the central database can process updates on an interval basis, and send bulk updates to the root replication group once every period of time. In this case, the central database would need to keep track of all the updates made by publishers in each period, possibly implemented using a linked list structure. For demonstration purpose, in this case, we have chosen to implement the simpler single updates.

The central database class has the following variables:

data_ : This is a list of the authoritative version of hints for all URNs. This list is implemented as a linked list. A publisher who wants to publish a new URN has to register with the central database by adding the official hint for the URN to this list. Publishers can also update or remove hints from this list when necessary. For a publisher to communicate with the central database, the nodes on which the publisher and the central database reside must be connected by a link. They communicate through a TCP connection.

count_ : This keeps track of the total number of hints in the central database.

distdb_ : This points to the distributed database. The central database needs to send updates of hints to the root replication group of the distributed database when it receives update requests from publishers.

cdbnode_ : This is the node on which the central database resides on.

ns_ : This refers to the simulator object that controls the whole simulation. This reference is needed by some of the methods that handle communications with other objects using the protocols provided by the basic modules.

The following describes the OTcl syntax for creating the central database:

set cdb1 [new cdb <distdb> <node> <ns>]

In order to create the central database, one needs to input the distributed database object, the node on which the central database resides, and the simulator object as arguments.

Here is a list of methods that the central database class provides:

$cdb count : returns the total number of hints stored in the database.

$cdb add <hint> : adds the given hint to the data_ list. The new hint is added to the front of the list.

$cdb remove <urn> : removes the hint for the given URN from the data_ list. It searches through the list until the hint for the given URN is found, and then removes it. It does not do anything if the hint for the given URN is not found in the list.

$cdb find <urn> : searches through the data_ list for the hint corresponding to the given URN. It returns the hint for the given URN if it is found. Otherwise, it returns null.

$cdb update <hint> : replaces an old hint in the data_ list with the given new hint. It also calls the send-update method to send this update to the root replication group. This is a method called by publishers who want to update the official hints for the URNs that they have published. It searches through the list for the hint that is the older version of the given hint, and replaces it. If no older version of the given hint is found in the list, it simply adds the given hint to the list.

$cdb send-update <hint> : sends the given hint to the root replication group using a TCP connection. The root replication group will propagate the update down to the appropriate replication group in the tree.

Notice that we have to modify a number of methods provided by the publisher class so that a publisher notifies the central database when publishing or updating URNs and hints.

5.2.2 The Replication Groups

Each replication group represents a node in the tree of the distributed database. It is composed of member servers that maintain the same set of hints for URNs falling into a certain subspace. A subspace is represented by two endpoints where each endpoint is a URN. One member server in each group serves as the leader and acts for the group as a whole. Every update for this replication group is sent to the leader. The leader automatically sends the update to each member. For simple implementation, every request for resolving a URN forwarding to this replication group is also received by the leader. The leader automatically assigns one of the member servers to respond if the URN falls into the subspace for this replication group. Otherwise, the request will be forwarded to the parent or one of the children replication groups depending on the subspace in which the URN falls.

Notice that we need to modify the server class described in the last chapter so that a server knows about the replication group in which it belongs. We also need to implement additional methods for a server to communicate with other members in the same replication group.

Here is a list of variables for a replication group:

subspace_ : The subspace that this replication group represents. This is represented by a pair of URNs, [urn1, urn2].

members_ : This is a list of member servers in this replication group.

leader_ : This points to the member server who serves as the leader of this group.

parent_ : This points to the parent replication group of this group in the tree.

children_ : This is a list containing the children replication groups of this group in the tree.

The following describes the OTcl syntax for creating a replication group:

```
set rgroup1 [new rgroup]
```

No argument is needed at the creation of a replication group. When this newly created replication group is added to the distributed database, the distributed database will set the subspace_, parent_ and children_ variables accordingly.

The class rgroup provides the following methods:

$rgroup subspace : returns the subspace that this replication group represents.

$rgroup parent : returns the parent replication group.

$rgroup children : returns the list of children replication groups.

$rgroup set-subspace <urn1> <urn2> : specifies the subspace for this replication group, with the given URNs begin the endpoints.

$rgroup set-parent <rgroup> : sets the variable parent_ to given replication group.

$rgroup set-child <rgroup> : adds the given replication group to the children_ list.

$rgroup remove-child <rgroup> : removes the given replication group from the children_ list.

$rgroup add-member <server> : adds the given server to the members_ list.

$rgroup remove-member <server> : removes the given server from the members_ list.

$rgroup set-leader <server> : makes the given server the leader of this group.

$rgroup leader : returns the leader of this group.

$rgroup update <hint> : performs updates if the given hint corresponds to an URN falling into the subspace of this replication group. The leader who receives the updated hint sends the hint to each member server using a TCP connection. If the given hint does not fall into the subspace of this replication group, the leader will forward the update to the leader of its parent or one of its children replication groups using a TCP connection.

$rgroup resolve <urn> : resolves the given URN if it falls into the subspace of this replication group. The leader who receives this request assigns one of the member servers to respond by sending the request to the member using a TCP connection. If the given URN does not fall into the subspace of this replication group, the leader will forward the request to the leader of the parent or one of the children replication groups using a TCP connection. The resolve method provided by that replication group will be invoked.

5.2.3 The Distributed Database

The distributed database is implemented using a B-tree structure. The algorithms for manipulating a B-tree can be found in [1]. Modifications to the traditional algorithms have to be made for the distributed database. The details are described in [12]. Each node in the tree is a replication group, consisting of member servers that maintain the same set of hints for URNs within a certain subspace.

Updates from the central database are sent to the leader of root replication group through TCP connections. Thus, the nodes on which the central database and the leader server of the root replication group reside must be connected by a link. The updates propagate down the tree to the appropriate replication group. Requests for resolving URNs can be initiated from any server. The request starts from the replication group in which the server belongs and propagates up or down the tree to the appropriate replication group that can handle the request.

The distributed database has a pointer to the root of the tree:

root_ : This points to the root replication group in the tree. This is actually the entry point to the whole distributed database. Since each replication group has pointers to its parent and children replication groups, the whole distributed database is linked together.

The following describes the OTcl syntax for creating the distributed database:

```
set distdb1 [new distdb]
```

The class distdb provides the following methods:

$distdb root : returns the root replication group in the distributed database.

$distdb add-node <rgroup> : inserts a new replication group to the distributed database. It automatically assigns a subspace for this replication group. It searches down the tree to find the appropriate place for this node. A new node always becomes a leaf of the tree. It also has to set the parent and children variables for the nodes appropriately.

$distdb find-node <urn> : searches down the tree of the distribution database for the replication group that is responsible for the subspace in which the given URN falls. The resulting replication group is returned.

$distdb find-node <hint> : searches down the tree of the distribution database for the replication group that is responsible for the subspace in which the given hint falls. The resulting replication group is returned.

$distdb central-update <hint> : distributes the given updated hint sent by the central database to the appropriate replication group. The central database calls this method to send an update to the leader of the root replication group using a TCP connection. The update method provided by the rgroup class is then invoked by the root replication group so that the update can propagate down the tree to the appropriate replication group.

Notice that the distdb class does not provide any method for resolving URNs. Methods for resolving URNs are provided by the rgroup class and have been given in the last section. Generally, when a server receives a request for resolving a URN from a client, the resolve method provided by the server class is invoked. This method has to be redefined for each different RDS model.

For this distributed RDS model, when a server calls its resolve method on a URN, it first checks its cache. If the hint for the URN is not found in the cache, it calls the resolve method provided by the replication group in which it belongs. The request will propagate up to the parent or down to one of the children replication groups depending on the subspace in which the URN falls. Eventually, the request will reach the appropriate replication group and the response will be sent back directly to the original server who made the request. The server will then send the response back the client. Besides redefining the resolve method for the server class, we also have to modify the server class so that it contains the information about the replication group in which it belongs.

5.3 Example

In order to provide the reader a better understanding of the syntax for creating a simulation using our simulation toolkit, we have included an example of OTcl script for running a demonstration of the simulation using the Distributed RDS model. A major part of the script is setting up the network and creating the main components. At the end of the script, a few events including requests for resolving URNs and updates are scheduled at different times. The output of the simulation is given in Appendix A. It is given in a form of trace data stored in a file. The traces record all the packets as they arrive, depart or are dropped.

OTcl script for running a simulation:

```
set ns [new Simulator]

#Create nodes
set cdnode [$ns node];
set cnode1 [$ns node]; set cnode2 [$ns node]; set cnode3 [$ns node]
set snode1 [$ns node]; set snode2 [$ns node]; set snode3 [$ns node]
set pnode1 [$ns node]; set pnode2 [$ns node]; set pnode3 [$ns node]

#Connects all nodes together using simple link
#This is a newly defined method that connects each node to all the other nodes
$ns connect-all $cnode11 $cnode2 $cnode3 $snode1 $snode2 $snode3 $pnode1 $pnode2 $pnode3 $cdnode

#Create a trace and arrange for all the trace events to be dumped to "out.tr"
set f [open out.tr w]
$ns trace-all $f

#Create servers
#Specify the node on which the server resides and the simulator object for each server
set server1 [new server $snode1 $ns]
set server2 [new server $snode2 $ns]
set server3 [new server $snode3 $ns]

#Create clients
#Specify the local server, the node on which the client resides, and the simulator object for each client
set client1 [new client $server1 $cnode1 $ns]
set client2 [new client $server2 $cnode2 $ns]
set client3 [new client $server3 $cnode3 $ns]

#Create publishers
#Specify the server that the publisher publishes URNs and hints on, the node on which it resides, and the
#simulator object for each publisher
```

```
set publisher1 [new publisher $server1 $pnode1 $ns]
set publisher2 [new publisher $server2 $pnode1 $ns]
set publisher3 [new publisher $server3 $pnode1 $ns]

#Create replication groups, add member servers, and set leader for each group
set rgroup1 [new rgroup]; $rgroup1 add-member $server1; $rgroup1 set-leader $server1
set rgroup2 [new rgroup]; $rgroup2 add-member $server2; $rgroup2 set-leader $server2
set rgroup3 [new rgroup]; $rgroup3 add-member $server3; $rgroup3 set-leader $server3

#Create distributed database
#Inserts replication groups. This will set the subspace, parent and children for each replication group
set distdb1 [new distdb]
$distdb1 add-node $rgroup1; $distdb1 add-node $rgroup2; $distdb1 add-node $rgroup3

#Create central database
#Specify the distdb object, the node on which the cdb resides and the simulator object.
set cdb1 [new cdb $distdb1 $cdnode $ns]

#Schedule the publishers for creating URNs and hints
#create-urn performs a number of operations. The names of the URNs are automatically assigned.
#Publishers publish URNs and hints on their own #servers by sending them through TCP. Publishers also
#register new hints with the central database by #sending them through TCP. Publishers adds URNs and
#hints to its publication list. New hints are returned.
$ns at 1.0 "set hint1 [$publisher1 create-urn]"
$ns at 2.0 "set hint2 [$publisher2 create-urn]"
$ns at 3.0 "set hint3 [$publisher3 create-urn]"

#A Hint contains the name of the URN with the resolution information and other data.
#We can extract the name of the URN from a hint.
set urn1 [$hint1 urn]; set urn2 [$hint2 urn]; set urn3 [$hint3 urn]

#Schedule the clients for making requests for resolving URNs
$ns at 4.0 "$client1 request $urn1"
$ns at 5.0 "$client2 request $urn1"
$ns at 6.0 "$client3 request $urn2"

#Mark a hint as expired
$hint3 mark-expired

#Schedule the publisher for updating the expired hint
#Schedule the client for making requests for resolving URN
$ns at 7.0 "$publisher3 update-hint $hint3"
$ns at 8.0 "$client2 request $urn3"

#Run the simulation for 10 seconds
$ns at 10.0 "exit 0"
$ns run
```

5.4 Summary

This chapter demonstrates the use of the simulation toolkit by implementing a simple version of the Distributed RDS model for URN resolution. Since deletion in a B-tree is a fairly complicated and expensive operation to implement, we have omitted the implementation of the deletion of node in the distributed database. The central database currently handles one update at a time. Alternatively, the central database can be made to handle bulk updates. This would require modification on the update method for the distributed database so that the root replication group can propagate multiple updates down the tree at the same time. To show the reader the syntax for creating a simulation, an example of OTcl script for running a simulation is given at the end of this chapter.

Chapter 6

Conclusion

In this paper, we described a toolkit for doing simulation on URN resolution. This toolkit is built on NS, a general-purpose network simulator. NS is an event-driven network simulator, consisting of an extensible simulation engine implemented in C++ with a command and configuration interface in OTcl. It provides a number of basic modules that serve as the building blocks for our simulation toolkit. These modules provide primitive functions such as defining network topologies, configuring parameters for simulation and usage of various protocols. Although it takes a certain amount of time to understand how the NS program is organized and structured before one can use NS to build one's own simulation, NS is a powerful and convenient tool for building network simulation.

In building a simulation for URN resolution, we added a number of additional components to NS. Hints contain the important information indicating how to resolve a URN. Clients make requests for resolving URNs through local servers. Servers have the

functions of storing URNs and hints and resolving URNs for clients by consulting an RDS. Publishers publish information with URNs and hints on servers and maintain the URNs and hints.

To demonstrate the use of the toolkit, we implemented one of the RDS models, the Distributed RDS, to show how a simulation on URN resolution can actually be done. The Distributed RDS consists of two main components. The central database contains the official version of hints for all URNs and is accessible only by publishers. The distributed database, consisting of replication groups of servers arranged in a hierarchy, is widely available to clients for URN resolutions.

6.1 Improvements

There are a number of improvements that can be made to the simulation toolkit to improve efficiency and enhance features. In the current implementation, linked list is frequently used to store objects such as URNs and hints. It is a data structure that allows fast and easy implementation. It is good if the number of items stored in the list is small. It also provides quick insertion to the list. However, deletion and searching are not very efficient especially for lists containing large number of items. As a result, if one wants to build a simulation on URN resolution in a large scale, a more sophisticated data structure such as heap or binary tree should be used.

Since deletion is such an expensive operation due to our choice of data structure, it does not happen frequently in our current implementation. Instead of being deleted, hints are marked as expired. Obsolete URNs and hints remain in the lists permanently. This may make the size of lists grow as more and more URNs and hints are created in the

simulation and cause problems when the simulation is allowed to run for a long time. To fix this problem, operations can be added to regularly check and delete obsolete items in the lists. Of course, a long-term solution would be to change the data structure.

The central database in the Distributed RDS currently handles only single updates. It sends updates to the root replication group in the distributed database one at a time. To improve efficiency, one can modify the central database to handle bulk updates. In order to do this, the algorithm for updating employed by the root replication group will have to be changed so that multiple updates can be propagated down the tree at the same time.

6.2 Future Work

The current toolkit only supports simulation on URN resolution using the Distributed RDS model. The toolkit can be extended to model other RDS schemes such as NAPTR and mu. Additional components will have to be added. Configurations should be provided for users to choose different RDS models and perform various kinds of simulations. This can help RDS researchers to compare different RDS schemes in their performances and other aspects.

In additional to study URN resolution in isolation and analyze the performance of individual RDS scheme, one can also study the effect of URN resolution on regular network traffic. How does URN resolution affect peak hour traffic? What is the effect of URN resolution on some standard models of TCP behaviors? These are important questions to be answered if URNs are really to be adopted by the community at large.

References

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990.

[2] R. Daniel and M. Mealling, *Resolution of Uniform Resource Identifiers using the Domain Name System*, RFC 2168, June 1997. Available from <ftp://ds.internic.net/rfc/rfc2168.txt>

[3] M. Doar, *A Better Model for Generating Test Networks,* IEEE Global Telecommunications Conference/GLOBECOM'96, London, November 1996. Available from <http://www.geocities.com/ResearchTriangle/3867/publications.html>

[4] S. C. Dewhurst and K. T. Stark, *Programming in C++*, 2nd Edition, Prentice Hall, 1993.

[5] K. Fall and K. Varadhan, *NS Notes and Documentation*, November 1997. Available from <http://www-mash.cs.berkeley.edu/ns/ns-documentation.html>

[6] L. D. Girod, *mu: Multicast URN resolution*, LCS MIT, February 1998. Available from <http://mu.lcs.mit.edu/>

[7] L. Girod, *MURL: A Scalable URN Resolution System Based on Multicast*, LCS MIT, November 1996

[8] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley, 1996. Available from <http://www.javasoft.com/docs/language_specification.html>

[9] S. Keshav, *REAL: A Network Simulator*, University of California, Berkeley, December 1988. Available from <http://www.cs.cornell.edu/skeshav/real/overview.html>

[10] S. Myers, *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*, Addison-Wesley, 1992.

[11] J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1996.

[12] E. C. Slottow, *Engineering a Global Resolution Service*, M.Eng thesis, MIT, June 1997. Available from <http://ana.lcs.mit.edu/anaweb/people/edward/thesis/>

[13] K. R. Sollins, *Architectural Principles of Uniform Resource Name Resolution*, RFC 2276, January 1998. Available from <ftp://ds.internic.net/rfc/rfc2276.txt>

[14] K. R. Sollins, *Requirements and a Framework for URN Resolution Systems*, Internet Draft, March 1997. Available from <ftp://ds.internic.net/internet-drafts/draft-ietf-urn-req-frame-01.txt>

[15] K. R. Sollins and J. R. Van Dyke, *Linking in a Global Information Architecture*, Proceedings of the Fourth International World Wide Web Conference, Boston, MA, December, 1995. Available from <http://ana-www.lcs.mit.edu/anaweb/html-papers/links.html>

[16] W. R. Stevens, *TCP/IP Illustrated, Volume 1*, Addison-Wesley, 1996.

[17] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1997.

[18] *UCB/LBNL/VINT Network Simulator NS*, EECS Berkeley, January 1998. Available from <http://www-mash.cs.berkeley.edu/ns/>

[19] J. Weber, *Special Edition Using JAVA 1.1*, Que, 1997.

[20] D. Wetherall, *Extending Tcl for Dynamic Object-Oriented Programming*, July 1995. Available from <ftp://ftp.tns.lcs.mit.edu/pub/otcl/>

[21] D. Wetherall, *OTcl – MIT Object Tcl: The FAQ and Manual*, September 1995. Available from <ftp://ftp.tns.lcs.mit.edu/pub/otcl/>

[22] E. W. Zegura, K. Calvert and S. Bhattacharjee, *How to Model an Internetwork*, Proceedings of IEEE Infocom, San Francisco, CA, 1996. Available from <http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html>

Appendix A

Simulation Result for the Example in Chapter 5.3

Trace Format:

Each event can be an enqueue operation (indicated by "+" in the first column), a dequeue operation (indicated by "-"), or a packet drop (indicated by "d"). The simulated time (in seconds) at which each event occurred is listed in the second column. The next two fields indicate between which two nodes the tracing is happening. The next field is a descriptive name for the type of packet seen. The next field is the packet's size. The next four characters represent special flag bits that may be enabled. Presently, they are not used. The subsequent two fields indicate the packet's source and destination node addresses, respectively. The following field indicates the sequence number. Only those Agents interested in providing sequencing will generate sequence numbers. It is not used in our example. The last field is a unique packet identifier. Each new packet created in the simulation is assigned a new, unique identifier.

```
+ 1.08572 7 4 tcp 1000 ---- 148 124 0 0
- 1.08572 7 4 tcp 1000 ---- 148 124 0 0
+ 1.28464 7 0 tcp 1000 ---- 148 172 0 1
- 1.28464 7 0 tcp 1000 ---- 148 172 0 1
+ 1.34922 0 4 tcp 1000 ---- 172 148 0 2
- 1.34922 0 4 tcp 1000 ---- 172 148 0 2
+ 2.02956 8 5 tcp 1000 ---- 156 132 0 3
- 2.02956 8 5 tcp 1000 ---- 156 132 0 3
+ 2.04968 8 0 tcp 1000 ---- 156 172 0 4
- 2.04968 8 0 tcp 1000 ---- 156 172 0 4
+ 2.07386 0 4 tcp 1000 ---- 172 124 0 5
- 2.07386 0 4 tcp 1000 ---- 172 124 0 5
+ 2.24351 4 5 tcp 1000 ---- 124 132 0 6
- 2.24351 4 5 tcp 1000 ---- 124 132 0 6
+ 3.10923 9 6 tcp 1000 ---- 164 140 0 7
- 3.10923 9 6 tcp 1000 ---- 164 140 0 7
+ 3.13844 9 0 tcp 1000 ---- 164 172 0 8
- 3.13844 9 0 tcp 1000 ---- 164 172 0 8
+ 3.31038 0 4 tcp 1000 ---- 172 124 0 9
- 3.31038 0 4 tcp 1000 ---- 172 124 0 9
+ 3.66721 4 6 tcp 1000 ---- 124 140 0 10
- 3.66721 4 6 tcp 1000 ---- 124 140 0 10
+ 4.03855 1 4 tcp 1000 ---- 100 124 0 11
- 4.03855 1 4 tcp 1000 ---- 100 124 0 11
+ 4.05938 4 1 tcp 1000 ---- 124 100 0 12
```

```
- 4.05938 4 1 tcp 1000 ---- 124 100 0 12
+ 5.00129 2 5 tcp 1000 ---- 108 132 0 13
- 5.00129 2 5 tcp 1000 ---- 108 132 0 13
+ 5.20495 5 4 tcp 1000 ---- 132 124 0 14
- 5.20495 5 4 tcp 1000 ---- 132 124 0 14
+ 5.23947 4 5 tcp 1000 ---- 124 132 0 15
- 5.23947 4 5 tcp 1000 ---- 124 132 0 15
+ 5.58612 5 2 tcp 1000 ---- 132 108 0 16
- 5.58612 5 2 tcp 1000 ---- 132 108 0 16
+ 6.11830 3 6 tcp 1000 ---- 116 140 0 17
- 6.11830 3 6 tcp 1000 ---- 116 140 0 17
+ 6.13583 6 4 tcp 1000 ---- 140 124 0 18
- 6.13583 6 4 tcp 1000 ---- 140 124 0 18
+ 6.24268 4 5 tcp 1000 ---- 124 132 0 19
- 6.24268 4 5 tcp 1000 ---- 124 132 0 19
+ 6.43286 5 6 tcp 1000 ---- 132 140 0 20
- 6.43286 5 6 tcp 1000 ---- 132 140 0 20
+ 6.46979 6 3 tcp 1000 ---- 140 116 0 21
- 6.46979 6 3 tcp 1000 ---- 140 116 0 21
+ 7.01926 9 6 tcp 1000 ---- 164 140 0 22
- 7.01926 9 6 tcp 1000 ---- 164 140 0 22
+ 7.18472 9 0 tcp 1000 ---- 164 172 0 23
- 7.18472 9 0 tcp 1000 ---- 164 172 0 23
+ 7.21565 0 4 tcp 1000 ---- 172 124 0 24
- 7.21565 0 4 tcp 1000 ---- 172 124 0 24
+ 7.28436 4 6 tcp 1000 ---- 124 140 0 25
- 7.28436 4 6 tcp 1000 ---- 124 140 0 25
+ 8.18357 2 5 tcp 1000 ---- 108 132 0 26
- 8.18357 2 5 tcp 1000 ---- 108 132 0 26
+ 8.24592 5 4 tcp 1000 ---- 132 124 0 27
- 8.24592 5 4 tcp 1000 ---- 132 124 0 27
+ 8.27494 4 6 tcp 1000 ---- 124 140 0 28
- 8.27494 4 6 tcp 1000 ---- 124 140 0 28
+ 8.35835 6 5 tcp 1000 ---- 140 132 0 29
- 8.35835 6 5 tcp 1000 ---- 140 132 0 29
+ 8.41869 5 2 tcp 1000 ---- 132 108 0 30
+ 8.41869 5 2 tcp 1000 ---- 132 108 0 30
```

Appendix B

User Guide on the Simulation Toolkit

This document contains a brief description on how to use the Simulation Toolkit. A summary on the basic commands and the additional modules required for simulating URN resolution is given. Users can refer to the NS User Manual for a description on the basic modules provided by NS and the related commands. The NS User Manual can be found at http://www-mash.cs.berkeley.edu/ns/.

The first step in the simulation is to acquire an instance of the Simulator class. It is created by the following command:

```
set ns [new Simulator]
```

Next, a network topology has to be defined. Basically, this involves creating some nodes and connecting them with links. For example, the command for creating a new node called node1 looks like:

```
set node1 [$ns node]
```

The following command creates a simplex link connecting two existing nodes, node1 and node2, with bandwidth specified by bw in bits per second, delay specified by d in seconds, and queuing discipline specified by type:

```
$ns simplex-link <node1> <node2> <bw> <d> <type>
```

The details for creating nodes and links can be found in the NS User Manual.

Trace data can be collected during a simulation. It can be displayed directly or stored in a file. It records each individual packet as it arrives, departs, or is dropped at a link or queue. Users can refer to the NS User Manuel for the details on the trace format. The following command creates a trace and arranges for all the trace events to be dumpled in the file "out.tr":

```
Set f [open out.tr w]
$ns trace-all $f
```

The following sections describe the additional modules required for simulating URN resolution. The modules include hint, client, server, publisher, central database, replication group and distributed database. A brief summary on the variables and the methods for each module is given. Users can refer to Chapter 4 and 5 in the thesis for a detailed description on each module.

Hint

Variables:

urn_
The URN that this hint represents.

timestamp_
A time stamp tells when the hint was created.

time-to-live_
An upper bound on the lifetime of a hint.

expired_
Set to true if the hint is expired. Set to false if the hint is valid.

resolution-data_
A list of resolution information.

The following command creates a new hint:

```
set hint1 [new hint <urn> <timestamp> <time-to-live>]
```

Methods:

$hint add-data <resolution data>
Returns the list resolution-data_. Adds new resolution information to resolution-data_.

$hint urn
Returns the value of urn_.

$hint time-stamp
Returns the value of timestamp_.

$hint time-to-live
Returns the value of time-to-live_.

$hint expired
Returns the value of expired_.

$hint mark-expired
Returns nothing. Set the value of expired_ to true.

$hint get-data
Returns the resolution-data_ list.

Client

Variables:

local-server_
The local server for the client.

bookmark_
A list of URNs that the client has bookmarked.

cnode_
The node on which the client resides.

ns_
The simulator object

The following command creates a new client:

set client1 [new client <local server> <node> <ns>]

Methods:

$client cnode
Returns the value of cnode_.

$client local-server
Returns the value of local-server_.

$client bookmark
Returns the value of bookmark_.

$client add-bookmark <urn>
Returns the value of bookmark_. Adds the given URN to the bookmark_ list.

$client remove-bookmark <urn>
Returns nothing. Removes the given URN from the bookmark_list.

$client pick-bookmark
Returns a URN that is randomly picked from the bookmark_ list.

$client request <urn>
Returns nothing. Sends a request for resolving the given URN to the local server.

Server

Variables:

cached-hints_
A list of hints that the server chooses to cache.

stored-data_
A list of URNs and the hints that are published on the server by publishers.

snode_
The node on which the server resides.

ns_
The simulator object
.
The following command creates a new server:

```
set server1 [new server <node> <ns>]
```

Methods:

$server snode
Returns the value of snode_.

$server cache-hint <hint>
Returns a pointer to the hint. Adds the given hint to the cached-hints_ list.

$server remove-cached-hint <urn>
Returns nothing. Removes the hint for the given URN from the cached-hints_ list.

$server find-cached-hint <urn>
Returns the hint for the given URN if it is found in the cached-hints_ list. Returns null otherwise.

$server update-cached-hint <hint>
Returns a pointer to the updated hint. Replaces the old hint in the cached-hints_ list with the given new hint.

$server update-cached-hint <urn> <resolution data>
Returns a pointer to the updated hint. Adds the given resolution data to the hint for the given URN.

$server add-data <urn> <hint>
Returns nothing. Adds the given URN and hint to the stored-data_ list

$server remove-data <urn>
Returns nothing. Removes the given URN and the corresponding hint from the stored-data_ list. Does nothing if the given URN is not found.

$server find-data <urn>
Returns the hint for the given URN if it is found. Returns null otherwise.

$server update-data <urn> <hint>
Returns nothing. Replaces the old hint for the given URN with the given hint.

$server update-data <urn> <resolution data>
Returns nothing. Adds the given resolution data to the hint for the given URN.

$server resolve <urn> <client>
Returns nothing. Resolves the given URN and sends the appropriate hint back to the client using a TCP connection.

Publisher

Variables:

server_
The server on which the publisher publishes URNs and hints.
.
publication_
A list of URNs and that the publisher has published.

pnode_
The node on which the publisher resides.

ns_ :
The simulator object

The following command creates a new publisher:

```
set publisher1 [new publisher <server> <node> <ns>]
```

Methods:

$publisher create-urn
Returns the hint for the newly created URN. Creates a new URN with the hint and publishes it on server_ and the central database. Adds the URN and the hint to the publication_ list.

$publisher remove-urn <urn>

Returns nothing. Removes the given urn and the corresponding hint from server_, the central database and the publication_ list.

$publisher create-hint <urn>
Returns the newly created hint for the given URN.

$publisher update-hint <hint>
Returns the updated hint. Updates the given hint in the publication_ list, the server, and the central database.

$publisher pnode
Returns the value of pnode_.

$publisher server
Returns the value of server_.

$publisher publication
Returns the value of publication_

$publisher add-publication <urn> <hint>
Returns nothing. Adds the given URN and hint to the publication_ list.

$publisher remove-publication <urn>
Returns nothing. Removes the given URN and the corresponding hint from the publication_ list.

$publisher find-hint <urn>
Returns the hint for the given URN if it is found. Returns null otherwise.

Central Database

Variables:

data_
A list of the authoritative version of hints for all URNs.

count_
The total number of hints in the central database.

distdb_
The distributed database

cdbnode_
The node on which the central database resides on.

ns_
The simulator object

The following command creates the central database:

```
set cdb1 [new cdb <distdb> <node> <ns>]
```

Methods:

$cdb count
Returns the value of count_.

$cdb distdb
Returns the value of distdb_.

$cdb cdbnode
Returns the value of cdbnode_.

$cdb add <hint>
Returns nothing. Adds the given hint to the data_ list.

$cdb remove <urn>
Returns nothing. Removes the hint for the given URN from the data_ list.

$cdb find <urn>
Returns the hint for the given URN if it is found. Returns null otherwise.

$cdb update <hint>
Returns the updated hint. Updates the given hint in the data_ list and sends the update to the distributed database.

$cdb send-update <hint>
Returns nothing. Sends the given hint to the root replication group in the distributed database.

Replication Group

Variables:

subspace_
The subspace that thereplication group represents.

members_
A list of member servers in the replication group.

leader_
The member server who serves as the leader of the group.

parent_
The parent replication group of this group in the tree.

children_
A list of the children replication groups of this group in the tree.

The following command creates a new replication group:

```
set rgroup1 [new rgroup]
```

Methods:

$rgroup subspace
Returns the value of subspace_.

$rgroup parent
Returns the value of parent_.

$rgroup children
Returns the value of children_.

$rgroup set-subspace <urn1> <urn2>
Returns nothing. Set the value of subspace_ to [urn1 urn2].

$rgroup set-parent <rgroup>
Returns nothing. Sets the variable parent_ to given replication group.

$rgroup set-child <rgroup>
Returns nothing. Adds the given replication group to the children_ list.

$rgroup remove-child <rgroup>
Returns nothing. Removes the given replication group from the children_ list.

$rgroup add-member <server>
Returns nothing. Adds the given server to the members_ list.

$rgroup remove-member <server>
Returns nothing. Removes the given server from the members_ list.

$rgroup set-leader <server>
Returns nothing. Sets the variable leader_ to the given server.

$rgroup leader
Returns the value of leader_.

$rgroup update <hint>
Returns nothing. Performs updates if the given hint corresponds to an URN falling into the subspace of this replication group. Otherwise, forwards the update to the parent or one of its children replication groups.

$rgroup resolve <urn>
Returns the hint for the given URN if the URN falls into the subspace of this replication group. Otherwise, returns null and forwards the request to the parent or one of the children replication groups.

Distributed Database

Variables:

root_
The root replication group in the distributed database.

The following command creates the distributed database:

```
set distdb1 [new distdb]
```

Methods:

$distdb root
Returns the value of root_.

$distdb add-node <rgroup>
Returns nothing. Inserts the given replication group to the distributed database.

$distdb find-node <urn>
Returns the replication group that is responsible for the subspace in which the given URN falls.

$distdb find-node <hint>
Returns the replication group that is responsible for the subspace in which the given hint falls.

$distdb central-update <hint>
Returns nothing. Distributes the given updated hint sent by the central database to the appropriate replication group.