University of Nebraska at Omaha
DigitalCommons@UNO

Student Work

12-1-1999

MIPPS: A Mobile IP Protocol Simulator.

Efren Serra

Follow this and additional works at: https://digitalcommons.unomaha.edu/studentwork

Recommended Citation
Serra, Efren, "MIPPS: A Mobile IP Protocol Simulator." (1999). *Student Work*. 3581.
https://digitalcommons.unomaha.edu/studentwork/3581

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.

MIPPS

A Mobile IP Protocol Simulator

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment

of the Requirements for the Degree

Masters of Science

University of Nebraska at Omaha

by

Efren Serra

December 1, 1999

UMI Number: EP74779

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI EP74779

Published by ProQuest LLC (2015). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

THESIS ACCEPTANCE

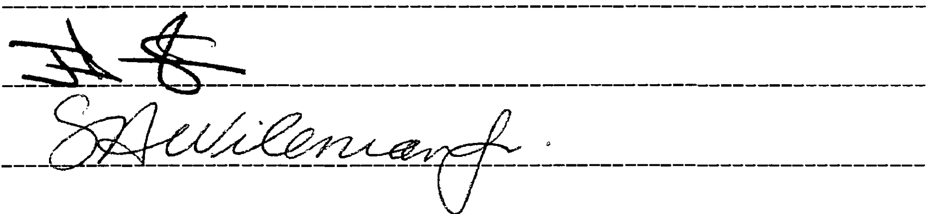
Acceptance for the faculty of the Graduate College,

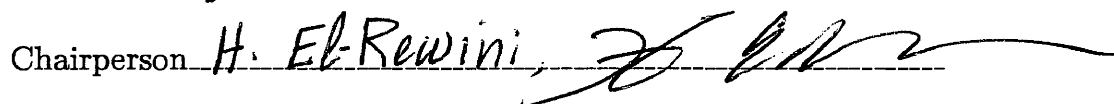
University of Nebraska, in partial fulfillment of the

requirements for the degree Masters of Science,

University of Nebraska at Omaha.

Committee



Chairperson 

Date Dec. 2, 1999

Abstract

We consider the problem of examining the performance requirement for mobility support on the Internet as specified by the Internet Engineering Task Force (IETF) Mobile-IP protocol.

In this thesis, we have designed, implemented, and utilized a discrete-event graphical simulation tool that provides a fundamental collection of building blocks and runnables, as well as a general environment for time-oriented simulations of communication networks that employ IPv4 or Mobile IP as the network layer protocol. The major goal of this graphical tool is to help communications network researchers answer "what if" questions, which may also help them improve any potential shortcomings of the Mobile-IP protocol.

Using this tool, we have conducted a number of experiments to study several interactions amongst the entities that comprise the Mobile-IP protocol. In particular, we studied the mobile hosts-home agent interaction, the fixed host-home agent interaction, and the foreign agent-mobile hosts interaction.

Acknowledgements

I would like to take this opportunity to express my gratitude to my advisor, Professor Hesham El-Rewini, for allowing me to purse my own path on this thesis, providing me with more than the necessary tools to finish this thesis, helping me stay on track, and to grow as an individual while expanding my technical knowledge. His patience, mentorship, and optimism throughout our research, course work, and this thesis, has made my experience as a graduate student at the University of Nebraska at Omaha enjoyable and informative.

Last, but not least, I would like to thank the members of my thesis committee, Professors Zhengxin Chen, Justin Stollen, and Stanley Wileman, for constructive criticism on earlier thesis drafts.

Contents

1 Introduction **1**

1.1 The IETF Mobile-IP Protocol 3

1.1.1 Infrastructure 4

1.1.2 Agent Discovery 6

1.1.3 Registration 8

1.1.4 Tunneling 10

1.2 The Issues 11

1.2.1 Registration overhead at a home agent 12

1.2.2 Rate of foreign agent's agent advertisements 14

1.2.3 Size of the input queue at a mobility agent 15

1.3 Their Importance 16

1.4 Thesis Organization 17

2 Previous and Related Work 20

2.1 Sony Mobile Host Protocol 21

2.1.1 Virtual Network 21

2.1.2 Design of VIP 24

2.2 Columbia Mobile Host Protocol 31

2.2.1 The Setup 31

2.2.2 Componets of Mobile*IP 34

2.3 Stanford Mobile Host Protocol 39

2.3.1 MosquitoNet Mobile IP Design 40

2.4 Mobile-IP Performance Studies 45

2.4.1 Wireless CSMA Cell 45

2.4.2 Congestion in a Cell 46

2.4.3 The Effect of Noise 50

2.5 Summary 52

3 An Overview of Ptolemy 55

3.1 Introduction 55

3.2 Ptolemy Kernel 57

3.3 Models of Computation 60

3.4 Dataflow Models of Computation 61

3.5 Discrete-Event Models of Computation 63

3.6 Synchronous Reactive Modeling 63

3.7 Finite State Machines 64

3.8 Mixing Models of Computation 64

3.9 Code Generation 65

3.10 Conclusion 66

4 An Overview of Mipps **72**

4.1 Introduction 72

4.2 Stars in Mipps 73

4.2.1 Network stars 73

4.2.2 Network interface stars 75

4.2.3 IP stars 77

4.2.4 Mobile IP stars 79

4.3 Galaxies in Mipps 81

4.3.1 Fixed host galaxies 82

4.3.2 Fixed router galaxies 84

4.3.3 Mobile host galaxies 85

4.3.4 Mobility agent galaxies 87

4.4 Some Sample Universes in Mipps 88

4.4.1 The MHs-HA Interaction Example 88
4.4.2 The FA-MHs Interaction Example 89
4.4.3 The FHs-HA Interaction Example 89

5 Experimental Results **94**

5.1 The MHs-HA interaction 94
5.1.1 Introduction 94
5.1.2 The experimental plan 95
5.1.3 Observations 96
5.1.4 Data analysis 99
5.2 The FHs-HA interaction 101
5.2.1 Introduction 101
5.2.2 The experimental plan 101
5.2.3 Observations 103
5.2.4 Data analysis 103
5.3 The FA-MHs interaction 104
5.3.1 Introduction 104
5.3.2 The experimental plan 105
5.3.3 Observations 106
5.3.4 Data analysis 106

6 Summary and Conclusions **115**

6.1 Summary 116

6.2 Future Work 118

6.3 Value of the Work 119

A Definition of classes in Mipps **121**

A.1 The network classes definition 121

A.1.1 The Net class definition 121

A.2 The network interface classes definition 125

A.2.1 The if_input class definition 125

A.2.2 The if_ouput class definition 131

A.3 The IP classes definition 138

A.3.1 The ip_input class definition 138

A.3.2 The ip_output class definition 147

A.4 The Mobile IP classes definition 158

A.4.1 The ipmobile_input class definition 158

A.4.2 The ipmobile_output class definition 171

List of Tables

1.1 Binding updates at home agent 13

5.1 The MHs-HA interaction: FHSnd routing tables 97

5.2 The MHs-HA interaction: PktRouter routing tables 97

5.3 The MHs-HA interaction: FA routing tables 97

5.4 The MHs-HA interaction: HA routing tables 97

5.5 The MHs-HA interaction: MHSnd routing tables 97

5.6 The FHs-HA interaction: FHSnd routing tables 108

5.7 The FHs-HA interaction: HA routing tables 108

5.8 The FHs-HA interaction: FA routing tables 109

5.9 The FHs-HA interaction: MHRcv routing tables 109

5.10 The FA-MHs interaction: FHSnd routing tables 110

5.11 The FA-MHs interaction: HA routing tables 111

5.12 The FA-MHs interaction: FA routing tables 111

5.13 The FA-MHs interaction: MHRcv routing tables 112

List of Figures

1.1 Mobile IP tunneling using "IP in IP" encapsulation 18

1.2 Mobile IP tunneling using the minimal tunneling protocol . . 19

2.1 VIP header format 26

2.2 Basic mobile host scenario 41

2.3 Triangle route optimization 43

2.4 The time to send a packet T_{send} 48

2.5 Throughput bound in a cell vs noise 51

3.1 The overall organization of Ptolemy version 0.7 58

3.2 Domains available with Ptolemy 0.7 68

3.3 Block objects in Ptolemy 69

3.4 A complete Ptolemy application (a Universe) 70

3.5 A Target in Ptolemy 70

3.6 A Domain in Ptolemy 71

3.7 A universal EventHorizon in Ptolemy 71

4.1 The top level pallete of building blocks and universes 73

4.2 The user pallete of stars and galaxies in Mipps 74

4.3 The network stars in Mipps 75

4.4 The network interface stars in Mipps 77

4.5 The IP stars in Mipps 79

4.6 The Mobile IP stars and galaxies in Mipps 81

4.7 The fixed host galaxies in Mipps 82

4.8 The FHsnd galaxy internal components. 83

4.9 The FHRcv galaxy internal components. 84

4.10 The fixed router galaxies in Mipps 84

4.11 The PktRouter galaxy internal components. 85

4.12 The mobile host galaxies in Mipps 86

4.13 The MHsnd galaxy internal components. 86

4.14 The MHRcv galaxy internal components. 90

4.15 The mobility agent galaxies in Mipps 90

4.16 The HA galaxy internal components. 90

4.17 The FA galaxy internal components. 90

4.18 The demo universes in Mipps 91

4.19 The MHs-HA interaction demo in Mipps 91

4.20 The FA-MHs interaction demo in Mipps 92

4.21 The FHs-HA interaction demo in Mipps 93

5.1 The MHs-HA interaction: one MH 98

5.2 The MHs-HA interaction: two MHs 108

5.3 The MHs-HA interaction: three MHs 109

5.4 The FHs-HA interaction: mean time of 1.0 sec 110

5.5 The FHs-HA interaction: mean time of 3.0 sec 111

5.6 The FHs-HA interaction: mean time of 5.0 sec 112

5.7 The FA-MHs interaction: one MHRcv 113

5.8 The FA-MHs interaction: two MHRcv 114

To my Wife, Mary Patricia (Crowley) Serra, who has stood by me all these years.

Chapter 1

Introduction

The global Internet is growing at a tremendous rate. There are now 29.7 million hosts connected to the Internet, and this number is doubling approximately every year. The average time between new networks connecting to the Internet is about 10 minutes. Initiatives such as the National Information Infrastructure and the increasing commercial uses of the Internet are likely to create even faster growth in the future [1].

At the same time, portable computing devices such as laptop and palmtop computers are becoming widely available at very affordable prices, and many new wireless networking products and services are becoming available based on technologies such as spread-spectrum radio, infrared, cellular and satellite.

Mobile computers today often are as capable as many home or office desktop computers and workstations, featuring powerful CPUs, large main memories, and hundreds of megabytes of disk space, multimedia sound capabilities, and color displays. High-speed local area wireless networks are commonly available that provide metropolitan or even nationwide service [1].

With these dramatic increases in portability and ease of network access, it becomes natural for users to expect to be able to access the Internet at any time and from anywhere, and to transparently remain connected and continue to use the network as they move about. However, internetworking protocols such as IP [2] used in the Internet do not currently support host mobility. A mobile user, today, must generally change IP addresses when connecting to the Internet at a different point or through a different network; the user must modify a number of configuration files and restart all network connections, making host movement difficult, time consuming, and error prone.

To address this need in the Internet, the Internet Engineering Task Force (IETF) has put forth the 20th draft on how the Internet Protocol (IP) can support mobile computing [3]. Because the Internet started out small and took several years for the infrastructure to grow from its Arpanet origin, there was enough time to refine the protocols and tune their parameters

while traffic grew year by year. The invasion of mobile users, however, will not take such a gradual progression. Once the relevant protocols are installed and propagated, mobile users will join the Internet in large numbers and from far-flung gateways (cf. the recent example of the World Wide Web). This will cause a rush to produce consumer hardware and software to support ubiquitous access (cf. the commercial success of Netscape) [4].

Since the lead time is short, there is an urgent need to examine the requirements mobile users impose and the impact they have on the performance of the IETF Mobile-IP protocol.

1.1 The IETF Mobile-IP Protocol

This section provides an overview of the current state of the basic IETF Mobile-IP protocol [1]. The protocol provides transparent routing of packets to a mobile host in the Internet and requires no modification to existing routers or correspondents hosts. No support is provided, however, for caching a mobile host's location at correspondent hosts or for allowing correspondent hosts to tunnel packets directly to a mobile host's current location. These features are being developed within the IETF as a separate set of extensions

to this basic protocol, and are not discussed here.

1.1.1 Infrastructure

Each mobile host is assigned a unique *home address* in the same way as any other Internet host, within its *home network*. Hosts communicating with a mobile host are known as *correspondents hosts* and may, themselves , be either mobile or stationary. In sending an IP packet to a mobile host, a correspondent host always addresses the packet to the mobile host's home address, regardless of the mobile host's current location.

Each mobile host must have a *home agent* on its home network that maintains a registry of the mobile host's current location. This location is identified as a *care-of-address*, and the association between a mobile host's home address and its current care-of-address is called a *mobility-binding*, or simply a *binding*. Each time the mobile host establishes a new care-of-address, it must *register* the new binding with its home agent so that the home agent always knows the current binding of each mobile host that it serves. A home agent may handle any number of mobile hosts that share a common home network.

A mobile host, when connecting to a network away from its home network,

may be assigned a care-of-address in one of two ways. Normally, the mobile host will attempt to discover a *foreign agent* within the network being visited, using an *agent discovery* protocol. The mobile host then *registers* with the foreign agent, and the IP address of the foreign agent is used as the mobile host's care-of-address. The foreign agent acts as a local forwarder for packets arriving for the mobile host and for all other locally visiting mobile hosts registered with this foreign agent. Alternatively, if the mobile host can obtain a temporary local address within the network being visited (such as through DHCP [5]), the mobile host may use this temporary address as its care-of-address.

While a mobile host is away from its home network, a mobile host's home agent acts to forward all packets for the mobile host to its current location for delivery locally to the mobile host. Packets addressed to the mobile host that appear on the mobile host's home network must be intercepted by the mobile host's home agent, for example by using "proxy" ARP [6] or through cooperation with the local routing protocol in use on the home network.

For each such packet intercepted, the home agent *tunnels* the packet to the mobile host's current care-of-address. If the care-of-address is provided by a foreign agent, the foreign agent removes any tunneling headers from the

packet and delivers the packet locally to the mobile host by transmitting it over the local network on which the mobile host is registered. If the mobile host is using a locally obtained temporary address as a care-of-address, the tunneled packet is delivered directly to the mobile host.

Home agents and foreign agents may be provided by separate nodes on a network, or a single node may implement the functionality of both a home agent (for its own mobile hosts) and a foreign agent (for other visiting mobile hosts). Similarly, either function or both may be provided by any existing IP routers on a network, or they may be provided by separate hosts on the network.

1.1.2 Agent Discovery

The *agent discovery* protocol operates as a compatible extension of the existing *ICMP router discovery* protocol [7]. It provides a means for a mobile host to detect when it has moved from one network to another, and for it to detect when it has returned home. When moving into a new foreign network, the agent discovery protocol also provides a means for a mobile host to discover a suitable foreign agent in this new network with which to register.

On some networks, depending on the particular type of network, addi-

tional link-layer support may be available to assist in some or all of the purposes of the agent discovery protocol. A standard protocol must be defined for agent discovery, however, at least for use on networks for which no link-layer support is available. By defining a standard protocol, mobile hosts are also provided with a common method for agent discovery that can operate in the same way over all types of networks. If additional link-layer support is available, it can optionally be used by mobile hosts that support it to assist in agent discovery.

Home agents and foreign agents periodically advertise their presence by multicasting an *agent advertisement* message on each network to which they are connected and for which they are configured to provide service. Mobile hosts listen for agent advertisement messages to determine which home agents or foreign agents are on the network to which they are currently connected. If a mobile host receives an advertisement from its own home agent, it deduces that it has returned home and registers directly with its home agent. Otherwise, the mobile host chooses whether to retain its current registration or to register with a new foreign agent from among those it knows of.

While at home or registered with a foreign agent, a mobile host expects

to continue to receive periodic advertisements from its home agent or from its current foreign agent, respectively. If it fails to receive a number of consecutive expected advertisements, the mobile host may deduce either that it has moved or that its home agent or current foreign agent has failed. If the mobile host has recently received other advertisements, it may attempt registration with one of those foreign agents. Otherwise, the mobile host may multicast an *agent solicitation* message onto its current network, which should be answered by an agent advertisement message from each home agent or foreign agent on this network that receives the solicitation message.

1.1.3 Registration

Much of the basic IETF Mobile-IP protocol deals with the issue of registration with a foreign agent and with a mobile host's home agent. When establishing service with a new foreign agent, a mobile host must register with that foreign agent, and must also register with its home agent to inform it of its new care-of-address. When instead establishing a new temporarily assigned local IP address as a care-of-address, a mobile host must likewise register with its home agent to inform it of its new address. Finally, when a mobile host returns to its home network, it must register with its home

agent to inform it that it is no longer using a care-of-address.

To register with a foreign agent, a mobile host sends a *registration request* message to the foreign agent. The registration request includes the address of the mobile host and the address of its home agent. The foreign agent forwards the request to the home agent, which returns a *registration reply* message to the foreign agent. Finally, the foreign agent forwards the registration reply message to the mobile host. When registering directly with its home agent, either when the mobile host has returned home or when using a temporarily assigned local IP address as its care-of-address, the mobile host exchanges the registration request and reply messages directly to its home agent.

Each registration with a home agent or foreign agent has associated with it a *lifetime* period, negotiated during registration. After this lifetime period expires, the mobile host's registration is deleted. In order to maintain continued service from its home agent or foreign agent, the mobile host must re-register within this period. The lifetime period may be set to infinity, in which case no re-registration is necessary. When registering with its home agent on returning to its home network, a mobile host registers with a zero lifetime and deletes its current binding, since a mobile host needs no services of its home agent while at home.

1.1.4 Tunneling

The Mobile-IP protocol allows the use of any tunneling method shared between a mobile host's home agent and its current foreign agent (or the mobile host itself when a temporary local IP address is being used as a care-of-address). During registration with its home agent, a list of supported tunneling methods is communicated to the home agent. For each packet later tunneled to the mobile host, the home agent may use any of these supported methods.

The protocol requires support for "IP in IP" encapsulation for tunneling, as illustrated in Figure 1.1. In this method, to tunnel an IP packet , a new IP header is wrapped around the existing packet; the source address in the new IP header is set to the address of the node tunneling the packet (the home agent), and the destination address is set to the mobile host's care-of-address. The new header added to the packet is shaded in gray in Figure 1.1. This type of encapsulation may be used for tunneling any packet, but the overhead for this method is the addition of an entire new IP header (60 bytes with options) to the packet.

Support is also recommended for a more efficient "minimal" tunneling protocol [8, 9], which adds only 8 or 12 bytes to each packet. This tunneling

protocol is illustrated in Figure 1.2, with the new header added to the packet shaded in gray. Here, only the modified fields of the original IP header are copied into a new forwarding header added to the packet between the original IP header and any transport-level header such as TCP or UDP. The fields in the original IP header are then replaced such that the source address is set to the address of the node tunneling the packet (only if the packet is being tunneled by a node other than the original sender), and the destination address is set to the mobile host's care-of-address. This type of encapsulation adds less overhead to each packet, but it cannot be used with packets that have already been fragmented by IP, since the small forwarding header does not include the fields needed to represent that the original packet is a fragment rather than a whole IP packet.

1.2 The Issues

The issues that we examine here are specific to the IETF proposal. This proposal does not specify the physical link for supporting mobility; for that, we assume a wired link using a Carrier Sensing Multiple Access (CSMA) protocol. The issues are summarized in the following sections:

1.2.1 What is the registration overhead at a home agent?

A mobile host registers whenever it detects that its point-of-attachment to the network has changed from one link to another. Also, because these registrations are valid only for a specified lifetime, a mobile host reregisters when it has not moved but when its existing registration is due to expire.

Upon receipt of a registration request, a home agent performs a set of validity checks. If the registration request is invalid for any reason – most notably, because of an authentication failure – the home agent sends a registration reply to the mobile host with a suitable *Code* field indicating the reason for the failure. In such a case, the home agent does not modify the mobile host's binding entry(ies) in any way.

If the registration request is valid, then the home agent updates the mobile host's binding entry(ies) according to the specified care-of-address, mobile host's home address, lifetime, and S fields, as specified in Table 1.1.

Finally, the home agent sends a registration reply to the mobile host indicating that the registration was successful. The IP source and destination address and the UDP source and destination port in the registration reply

Registration Request Fields:			Result
Care-of-Address	**Lifetime**	**S Bit**	
any address $\neq$ home address	> 0	0	Replace all of the mobile host's existing bindings (if any) with a new binding for the specified care-of-address.
any address $\neq$ home address	> 0	1	Create a binding for the specified care-of-address, leaving any other existing bindings for the mobile host unmodified.
any address $\neq$ home address	0	1	Delete the mobile host's binding for the specified care-of-address, leaving any other existing bindings unmodified.
mobile host's home address	0	any	Delete all of the mobile host's bindings
mobile host's home address	> 0	any	Call the manufacturer of the mobile host because it is broken.

Table 1.1: Binding updates at home agent

are simply reversed from the respective fields in the registration request.

The receipt of a registration request from a mobile host, the set of validity checks performed on behalf of a registration request, and the forwarding of a registration reply to a mobile host that is away from home is an overhead due to registration at a home agent. To study the nature of this overhead, we consider the traffic in the fixed net that is generated by or destined for a user on the home network, and measure its throughput as a function of the mobility of a mobile host that belongs to this home agent.

1.2.2 What is the rate of foreign agent's agent advertisements?

As mobility agents, home agents, foreign agents, or both, must periodically multicast or broadcast ICMP router advertisements to each link on which a host is configured to perform as a home agent, foreign agent, or both. These advertisements are used by a mobile host that is connected to such a link to determine whether any mobility agents are present and, if so, their respective identities (IP addresses) and capabilities.

Let T_{advert} be the time (in seconds) between two consecutive foreign agent advertisements. The IETF proposal recommends that the rate for foreign agent advertisements be slower than once per second, i.e., $T_{advert} > 1$. This constraint prevents the downlink bandwidth – from foreign agent to mobile hosts – from being dominated by agent advertisements.

We examine the downlink traffic to see what an appropriate value for T_{advert} should be. (The IETF proposal recommends that T_{advert} be 1/3 of the lifetime specified by a foreign agent in its advertisement; this lifetime is the maximun it will accept for each mobile host registration on its visitor list, so the choice of T_{advert} indirectly affects the lifetime of an advertisement.)

The lifetime is the time remaining before the registration expires, and the mobility binding erased from the forwarding list. The mobile node should therefore register again before the lifetime of its registration expires. The default lifetime in the IETF proposal is 1800 seconds.

Various factors combine to determine the appropriate choice for the registration lifetime. In this work we show how one such factor – number of nodes to which a mobility agent multicasts foreign agent advertisements – can affect a foreign agent's specification for lifetime in its advertisements.

1.2.3 What is the size of the IP input queue at a mobility agent?

The Mobile IP working group has specified the use of encapsulation as a way to deliver datagrams from a mobile host's "home network" to an agent that can deliver datagrams locally by conventional means to the mobile host at its current location away from home [10]. The use of encapsulation may also be desirable whenever the source (or an intermediate router) of an IP datagram must influence the route by which a datagram is to be delivered to its ultimate destination.

The required method of encapsulation is "IP in IP", as illustrated in

Figure 1.1. In this method, to tunnel an IP packet, a new IP header is wrapped around the existing packet; the source address in the new IP header is set to the address of the node tunneling the packet (the home agent), and the destination address is set to the mobile host's care-of-address. This type of encapsulation may be used for tunneling any packet, but the overhead for this method is the addition of an entire new IP header (60 bytes with options) to the packet. We examine the number of packets dropped at the foreign agent, as we vary the size of the buffer for incoming IP datagrams, to determine what an appropriate value for the latter should be.

1.3 Their Importance

To support mobility, some extra traffic on the fixed net is necessary. A study of Issues 1 and 2 can help us understand how this extra traffic contributes to the growth of the Internet traffic, as well as help network administrators anticipate the impact of mobile users.

The proposed mobile Internet Protocol has several parameters, and Issues 2 and 3 concern the choice of values for two of them. These choices are important because the protocol's performance depends on the values of its

parameters.

1.4 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 will cover previous and related work on mobile host protocols. Chapter 3 will summarize *Ptolemy*, the object-oriented framework upon which we develop our IETF Mobile-IP protocol simulator, and the *DE domain*, a generic discrete-event modeling environment useful for the simulation of queueing systems, communication networks, and hardware systems. In chapter 4 we shall go into greater detail on the design of Mipps; we shall categorize and describe the extensions we made to Ptolemy to facilitate the actual implementation of such a system. In Chapter 5 we shall present the experimental results. In Chapter 6 we shall summarize our work and present our conclusions.

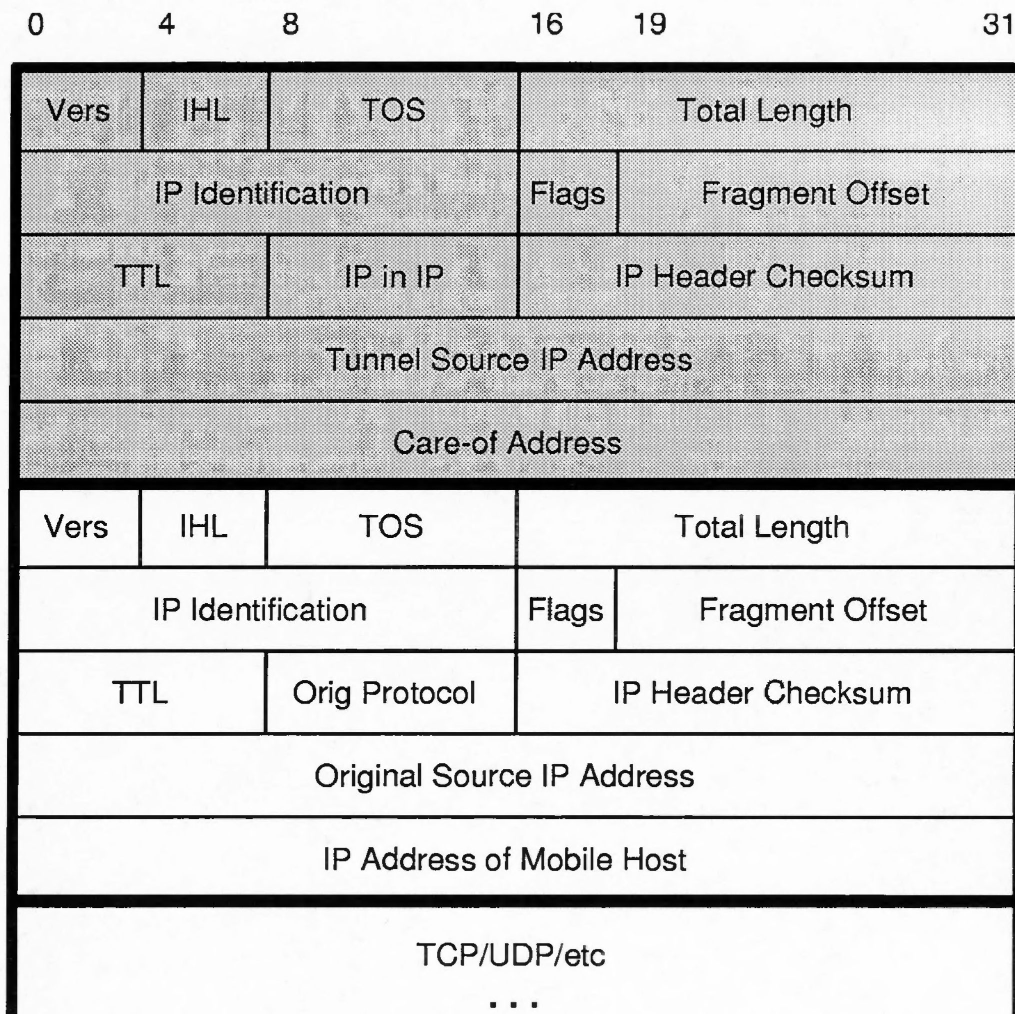


Figure 1.1: Mobile IP tunneling using "IP in IP" encapsulation

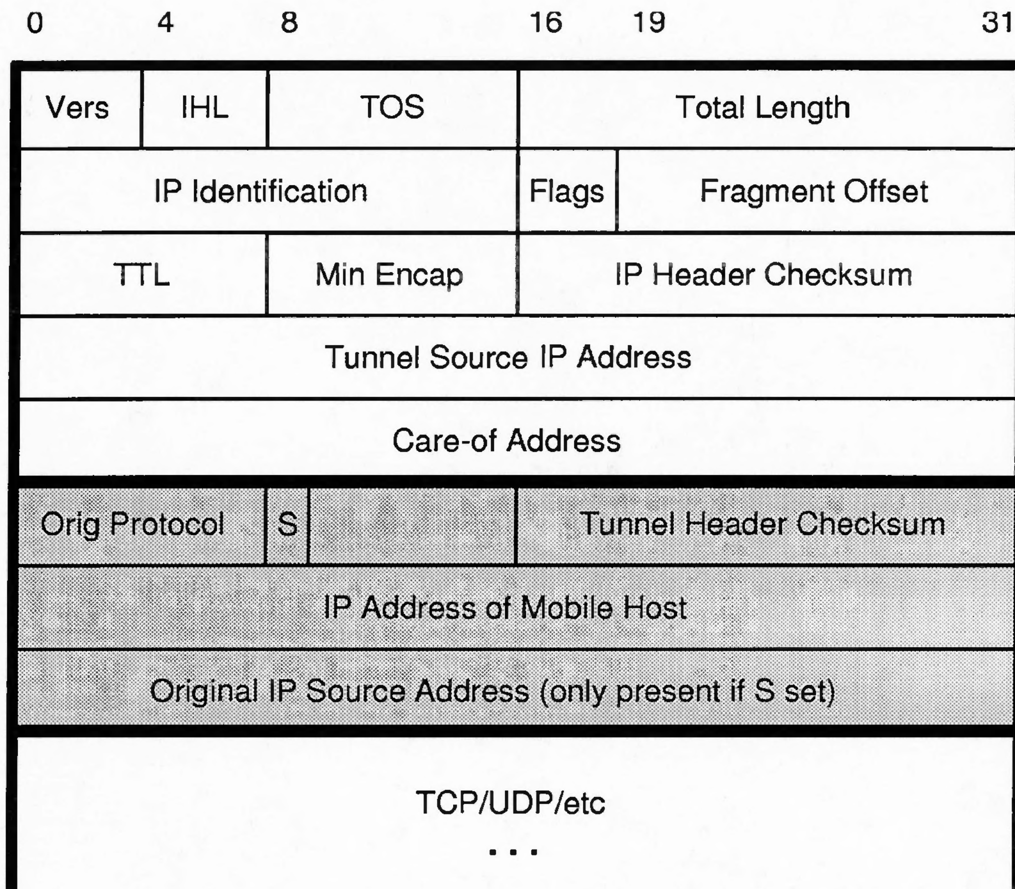


Figure 1.2: Mobile IP tunneling using the minimal tunneling protocol

Chapter 2

Previous and Related Work

In this chapter, we shall review three proposals from *Sony*, *Columbia*, and *Stanford* for mobile host protocols (MHP) that are compatible with the TCP/IP protocol suite. The purpose of this chapter is to give the reader a feeling of where the state of the art was when the IETF Mobile IP proposal was started, and why and how they fall short of providing acceptable mobile internetworking solutions.

Excerpted by permission of Fumio Teraoka, Sony Computer Science Laboratory Inc. "Design, Implementation, and Evaluation of Virtual Internet Protocol". Copyright by Sony Computer Science Laboratory Inc. All rights reserved.

2.1 Sony Mobile Host Protocol

In [11], Teraoka *et al.* of Sony Computer Science Laboratory Inc., claimed that **host migration transparency should be provided by the network layer**. They divided the conventional network layer into two sublayers: the *virtual network* sublayer and the *physical network* sublayer. This division provides the foundation for their concept of *virtual network* and they proposed the *propagating cache method* to efficiently implement their virtual network [12]. Virtual Internet Protocol (VIP) was derived from IP by applying the concept of virtual network and the propagating cache method, as an example of a virtual network protocol.

2.1.1 Virtual Network

Concept

In order to provide the transport layer with host migration transparency, [11] introduced two network layer identifiers of a host: one is migration independent and the other is migration dependent. The transport layer can specify the target host by the migration independent identifier so that the transport layer is not aware of host migration. To incorporate these two identifiers into

the network layer, [11] introduced the concept of virtual network. A virtual network is a logical network. Each host is considered to be permanently connected to a particular virtual network, called the *home network* of the host. A host never migrates among virtual networks even if it migrates among the actual, or physical, networks.

In order to incorporate the concept of virtual network into the network architecture, [11] divided the conventional network layer into two sublayers: the *virtual network sublayer* (VN-sublayer) and *physical network sublayer* (PN-sublayer). A *virtual network address* (VN-address) is assigned to a host in the VN-sublayer and a *physical network address* (PN-address) is assigned to a host in the PN-sublayer. The VN-address of a host never changes no matter where the host migrates while the PN-address changes if the host migrates. The transport layer specifies the target host by its VN-address so that the transport layer is not aware of host migration.

Propagating Cache Method

Since the transport layer specifies the target host by its VN-address, the main function of the VN-sublayer is to convert a VN-address into its corresponding PN-address. In [11], due to limitations in name servers or network directories

for managing dynamic data such as the location of migrating hosts, Teraoka *et al.* proposed the *propagating cache method* to provide efficient address conversion.

In the propagating cache method, every host and gateway has a cache for address conversion, called an Address Mapping Table (AMT). If the source host has an AMT entry for the destination host, the source host executes address conversion before sending a packet; the PN-sublayer can then correctly deliver this packet to the destination host. Otherwise, the source host assumes that the destination host is connected to its home network and sends the packet accordingly. As the packet traverses the network in transict to the destination network of the destination host, if an intermediate gateway has the AMT entry for the destination host, the gateway executes address conversion and forwards the packet to the current physical location of the destination host. When a host or gateway receives a packet, the VN-sublayer creates or updates the AMT entry for the source host of the received packet. Thus, AMT information propagates across the interconnected networks as communication progresses.

2.1.2 Design of VIP

IP address vs. VIP address

According to the concept of virtual network, a host is assigned two addresses in the network layer: a VN-address and a PN-address. In the TCP/IP suite of protocols, an IP address maps directly to a PN-address since it specifies the location of a host in the Internet. In [11], Teraoka *et al.* introduce the VIP address as a VN-address. The conventional IP layer is divided into two sublayers: the VIP sublayer and the IP sublayer. A host has an IP address in the IP sublayer and a VIP address in the VIP sublayer. Since the VIP address of a host never changes even if the host migrates to another network, the TCP/UDP layer and application programs over the TCP/UDP layer can uniquely specify a host by the VIP address. Basically, the IP address becomes invisible to the TCP/UDP layer.

Packet and AMT Formats

VIP is implemented as an IP option. The header format of VIP is shown in Figure 2.1. The upper half of the figure shows the normal IP header and the lower half shows the VIP header as an option of IP. The VIP header consists of eight fields:

- option type ($VIP_{OptType}$) and option length (VIP_{OptLen}): these two fields are defined in IP for options. The value of the VIP_{OptLen} field, the length of the VIP header, is 20 bytes.
- VIP type (VIP_{Type}): There are six types:
  - *VipData*: a normal data packet.
  - *VipConn*: announces that the sources host has just been attached to a subnetwork.
  - *VipConnAck*: an acknowledgement packet of the *VipConn* packet.
  - *VipDisc*: announces that the source host is about to disconnect from the subnetwork.
  - *VipDelAmt*: announces that an AMT entry becomes obsolete.
  - *VipErrObs*: error notification packet announcing that an AMT entry is obsolete.
- hold time (VIP_{hold}): specifies the time limit to hold the AMT entry for the source host of this packet.
- source VIP address ($VIP_{SrcAddr}$) and destination VIP address ($VIP_{DstAddr}$).

<table>
<tr><td>ver</td><td>ihl</td><td colspan="2">tos</td><td colspan="4">total length</td></tr>
<tr><td colspan="4">id</td><td>f</td><td colspan="3">fragment offset</td></tr>
<tr><td colspan="2">ttl</td><td colspan="2">proto</td><td colspan="4">header checksum</td></tr>
<tr><td colspan="8">source IP address</td></tr>
<tr><td colspan="8">destination IP address</td></tr>
<tr><td colspan="2">opt type</td><td colspan="2">opt len</td><td colspan="2">type</td><td colspan="2">hold</td></tr>
<tr><td colspan="8">source VIP address</td></tr>
<tr><td colspan="8">destination VIP address</td></tr>
<tr><td colspan="8">source address timestamp</td></tr>
<tr><td colspan="8">destination address timestamp</td></tr>
</table>

Figure 2.1: VIP header format

- source address timestamp (VIP_{SrcTS}): specifies the version number of the source (VIP, IP) address pair.

- destination address timestamp (VIP_{DstTS}): specifies the version number of the destination (VIP, IP) address pair.

The Address Mapping Table (AMT) entry for VIP consits of four data fields each of which is 4 bytes, and one extra field (4 bytes) included for management purposes. The size of the AMT entry is 20 bytes. The four data fields are as follows:

- VIP address (AMT_{VIP}): acts as a key.

- IP address (AMT_{IP}): the requested value.
- address timestamp (AMT_{AddrTS}): specifies the version number of the VIP/IP address pair of this entry.
- idle timer (AMT_{idle}): used for aging of the AMT entry. An entry which has not been accessed for a certain time is deleted.

Communication Procedures

In packet transmission, the TCP/UDP layer issues a transmission request to the VIP sublayer with the VIP address of the destination host. The VIP sublayer then generates the VIP header and sets each field. Next, the VIP sublayer searches for the AMT entry for the destination host. If the entry is found, the VIP sublayer converts the VIP address of the destination host into the corresponding IP address. Otherwise, the VIP sublayer assumes that the VIP address and the IP address of the destination host are the same; that is, the VIP sublayer assumes that the destination host exists in its home network. The procedures in the IP sublayer are the same as that of the conventional IP layer. The IP sublayer then issues a transmission request to the network interface layer.

In packet reception, some modifications to the existing IP procedures are

necessary. The IP sublayer must call two functions in the VIP sublayer:

- (f-1) A function which creates or updates the AMT entry for the source host of the received packet. If the AMT entry for the source host does not exists and the VIP address and the IP address of the source host are different, an entry is created. If the entry already exists, one of two conditions will incur an entry update: if either the IP address of the source host has changed or the value of the VIP_{SrcTS} field of the received packet is newer than the value of the AMT_{AddrTS} field in the AMT entry for the source host.

- (f-2) A function which converts the VIP address of the destination host into the corresponding IP address, if necessary. If the AMT entry for the destination host is found and the value of the VIP_{DstTS} field of the received packet is older than the value of the AMT_{AddrTS} field in the AMT entry for the destination host, this function returns the values of the AMT_{IP} field and the AMT_{AddrTS} field.

When the data link layer notifies the IP sublayer of packet reception, the IP sublayer calls *f-1* and compares the destination IP address field of the received packet with its own IP address. If the two IP addresses match,

the IP sublayer passes the packet to the VIP sublayer. Otherwise, the IP sublayer calls *f-2*. If address conversion occurs, the IP sublayer modifies the destination IP address field in the IP header and the VIP sublayer modifies the VIP_{DstTS} field in the VIP header. The packet is then forwarded to the next gateway or to the destination host.

Migration Procedures

When a migrating host is connected to a subnetwork, the host sends a *VipConn* packet to its home network and waits for a *VipConnAck* packet. The *VipConn* packet is relayed by gateways. Since the *VipConn* packet includes the VIP and the IP addresses of the source host, intermediate gateways can learn the relation of these two addresses and create or update the AMT entry for the source host. The *home gateway*, which is the gateway in the home network of the migrating host, receives the *VipConn* packet, creates the AMT entry for the migrating host, and returns the *VipConnAck* packet to it. Note the the home gateway also broadcasts the *VipConn* packet within the home network of the migrating host if the home network is a broadcast type network such as Ethernet. In addition, the home gateway creates an entry in the ARP table for the migrating host to answer the ARP request

for that host.

Upon packet transmission to a migrating host, if the source host does not have the AMT entry for the migrating host, the host assumes that the IP address of the migrating host is equal to its VIP address. The packet then heads for the home network of the migrating host. If a gateway on the path has the AMT entry for the migrating host, the gateway converts the VIP address into the corresponding IP address and forwards the packet to the actual location of the migrating host. If the migrating host returns a packet to the source host, the source host can create the AMT entry for the migrating host so that the host can thereafter directly specify the IP address of the migrating host.

When a migrating host is about to disconnect from a subnetwork, it sends a *VipDisc* packet to its home gateway. When the home gateway receives the *VipDisc* packet, it broadcasts a *VipDelAmt* packet to all subnetworks to which it is connected. Gateways in the subnetwork receive the *VipDelAmt* packet. If a gateway has the AMT entry for the migrating host , it deletes the entry and also broadcasts the packet to any other connected subnetworks. If a gateway does not have an AMT entry for the migrating host, the gateway does nothing. Thus, most AMT entries of the migrating host are deleted

when the host is about to disconnect from a subnetwork.

2.2 Columbia Mobile Host Protocol

In [14], Ioannidis *et al.* of Columbia University, claimed that **mobility should be handled at the Network Layer**. To solve this problem they have designed and implemented a system where the mobile hosts require support from cooperating infrastructure machines to maintain its addressability. In the following sections we present the design of their mobile IP system.

2.2.1 The Setup

The entities involved in a Mobile*IP systems are the *Mobile Hosts (MHs)* and the *Mobile Support Routers (MSRs)*. An MH may be moving only within its "home" network, in which case it will be called a *local* MH, or just an MH; or it may be moving to other networks outside its administrative domain, in which case it will be called a *popup* MH, or simply a *Popup*.

All mobile hosts of an organization have IP addresses from the same subnet or set of subnets. As far as the rest of the network is concerned,

Excerpted by permission of John Ioannidis, Department of Computer Science, Columbia University. "IP-based Protocols for Mobile Internetworking" . Copyright by John Ioannidis. All rights reserved.

they appear to be on the same subnet, or same connected network. This *Virtual Subnet* is composed of the union of the Cells (defined bellow), and its connectivity is provided by the MSRs. In other words, the Virtual Subnet that the mobiles are in is in fact a partioned network, and the MSRs act (by exchanging MH location information and tunneling IP datagrams between each other) to heal this partition.

MSRs are last-hop routers; they route between mobiles, or mobiles and the rest of the network. An MSR defines one or more "cells"; one cell per logical network interface. Cells are the geographical, and/or logical regions where an MH can directly communicate with its MSR. Since MSRs are all entry-routers to the virtual subnet, they may be thought of as a distributed router.

Mobile Hosts (MHs) send and receive packets via the MSRs, except when the target is in the same cell and the hardware allows peer-to-peer communications, in which case an MH may talk directly to another MH.

Conceptually, each MH has two IP addresses. One which serves the 'traditional' IP address purpose of being an EID (Endpoint Identifier) and at the same time routing to a last-hop router. This is called the Home Address of the mobile. In addition, the mobile adquires a nonce address which is

local to the "cell" in which it moves. This nonce address is a real, secondary address when the host moves arbitrarily. When the host is on the same direct link as an MSR, the nonce address is not really part of the MH; rather, all the MHs in that cell have the address of the MSR as their collective nonce address. The nonce address has been "factored out" of the MHs and into the MSR. **This is the optimization that makes intra-campus mobility very efficient**. This factoring makes sense because once the packet has reached the last-hop router, its destination IP address only serves to derive the MAC layer address of the destination. Hence, as far as the rest of the routing system is concerned, all the (local) mobiles in a cell can have the same nonce address, that of the MSR, as that is the address used to bring the packet to the last-hop router (the MSR). Once the packet is at the MSR, its real IP address is used by the MSR to find the mobile's MAC address and deliver the packet.

Home addresses are assigned from a subnet from the organization's network. As far as the rest of that network is concerned, all mobiles appear to be on the same subnet(s). The MSRs advertise routes to this Mobile Subnet, using any suitable interior routing protocol (e.g., RIP [16], IGRP [17], etc.). A collection of MSRs that directly exchange MH location information, and is

practically always under common administrative control, is called a *campus*.

As this setup suggests, MSRs are the routers to the MHs; symmetrically, the MSR an MH is being served by is its gateway to the world. Datagrams originating at MHs and destined to non-mobile hosts are routed in the traditional manner; based on its routing tables, the MSR serving an MH just routes the outgoing IP datagrams to the next hop router, or to the destination if the destination is on a directly connected subnet of the MSR. IP datagrams destined to MHs are, through normal routing mechanisms, eventually routed to an MSR; if the target MH happens to be in that MSR's cell, the datagram is simply delivered. Otherwise, the datagram is **tunneled** to an appropriate MSR, and then delivered from there.

2.2.2 Componets of Mobile*IP

There are three components to this model:

1. Locating and tracking an MH.
2. Propagating routing information.
3. IP datagram delivery.

How to locate and track mobiles has implications on how to adquire and propagate routes, which, in turn, determines how to deliver datagrams to their targeted hosts.

Locating and Tracking Mobiles

There are three issues:

1. Initial contact with the MSR.
2. Maintaining status.
3. Handoff/cell-switch.

- Initial MH-MSR Interaction

  In [14], Ioannidis *et al.* decided to have **the mobiles identify themselves to their respective MSRs** when they wish to be available for others to contact.

  When the MH greets the MSR, it provides its IP address, a timestamp, and authentication information, if necessary. The IP address is used by the MSR as a key to its special routing tables. The timestamp is a strictly-increasing value, guaranteed to change faster than a mobile can switch cells.

The MSR can either acknowledge or reject the Greeting. In its acknowledgement, it will say what the expiration timeout for the mobile is. In the rejection, it will tell the mobile *why* it was rejected, so that the mobile may take corrective action.

- Maintaining Status

  Once the mobile has identified itself to its MSR, it can send and receive packets. Knowledge of this mobile's location is present in the MSR it just greeted, and in any MSRs that might have queried this MSR and found its location. To guard against mobiles migrating to other cells without notifying their previous MSR, MSRs expire entries concerning MHs that they have not communicated with over a predefined timeout interval. To avoid being purged from the MSR tables, MHs periodically "ping" their MSR to renew their expiration interval.

- Changing Cells

  Now, when changing cells, either on its own or because it was told so, an MH must handshake with its new MSR and inform the MSR that it has moved. If the hand-off is initiated by an MSR, the mobile need not inform the new potential MSR(s) that it is coming their way,

even if the mobile knew it was going to migrate and had told its MSR so. Once the mobile has moved and handshaked with its new MSR, it informs the previous MSR that it has moved. This is done by presenting the current MSR with a *previous-MSRs* list, and then expecting its new MSR notify its previous MSRs. As acknowledgements of these notifications come back, the MSR forwards them to the mobile, which subsequently removes these old MSRs from its list. The burden of fault-tolerance rests solely with the MH; the MSRs need not maintain state information of each of its MH's previous MSRs.

Propagating Routing Information

When an MSR needs the location of an MH, it queries all the other MSRs, and the one who actually carries it, responds. This is reminiscent of the way ARP [15] works; ask for the information when it is needed, and cache it while it is being used. Hence, the distribution of location information is traffic-driven; only the MSRs with the *need-to-know* are informed of an MH's location.

There are five ways an MSR receives MH location information:

1. When an MH Greets.

2. When an MH moves out of an MSR's cell, it notifies its old MSR of the new MSR's address, using the Forwarding Pointer.

3. When a packet for an unknown MH arrives at an MSR, the MSR queries the other MSRs; the one currently serving the mobile replies.

4. When an MSR forwards a packet to another MSR which it believes is currently serving the MH, but the MH has moved in the meanwhile, the target MSR sends back a redirect informing the sending MSR of the new destination.

5. In the previous case, if the other MSR no longer has any information about the MH, it sends back a message saying its information had expired, and the original MSR proceeds as in case 3.

Delivering Packets

In [14], Ioannidis*et al.* chose tunneling as the mechanism for delivering a datagram originating in an arbitrary part of the network to a mobile host. Tunneling may be accomplished in two ways in IP-like protocols: source routing and encapsulation. Source routing is adding a list of routers that a packet must first go through before reaching its final destination. Encapsulation is

including the packet in another packet whose source and destination address are the endpoints of the tunnel.

For Mobile*IP, Ioannidis *et al.* chose to do encapsulation rather than source-routing. The MSRs are the encapsulators/decapsulators, and they are the routers that pick up traffic destined to mobiles from the static network.

2.3 Stanford Mobile Host Protocol

In [13], Baker *et al.* of Stanford University, claimed that **mobile hosts should not assume any explicit mobility support from the networks they visit, aside from basic Internet connectivity**. They address two problems that mobile hosts must overcome to make full use of the physical infrastructure for ubiquitous network connectivity. The first is that mobile hosts must be able to switch seamlessly between different types of network devices, and the second is that mobile hosts must be able to visit foreign networks that do not provide any support for mobility.

To solve these problems and gain more experience with mobility, they have designed and implemented a system that requires support only in the

---

Excerpted by permission of Mary Baker, Department of Computer Science, Stanford University. "Supporting Mobility in MosquitoNet". Copyright by the Regents of Stanford University. All rights reserved.

home domain of the mobile host and on the mobile host itself. Next, we present the design of their mobile IP system.

2.3.1 MosquitoNet Mobile IP Design

System Components

Of the three basic entities in their mobile IP system, the mobile host, home agent, and correspondent hosts, only the mobile host and home agent require mobility support. The mobile hosts require somewhat more support in this system than in implementations with foreign agents, since mobile hosts in this system must be able to encapsulate and decapsulate packets on their own.

The mobile host must be able to receive packets from correspondent hosts wherever it moves. To remain reachable, it must receive packets addressed to it at its home network. When at home, it directly receives these packets. When it leaves and connects to another network, these packets must be forwarded to it. To accomplish this, the mobile host needs to adquire a temporary care-of-address from the new network (perhaps dynamically via DHCP). Since this approach does not assume the existence of a separate foreign agent in the new network, the mobile host serves as its own foreign

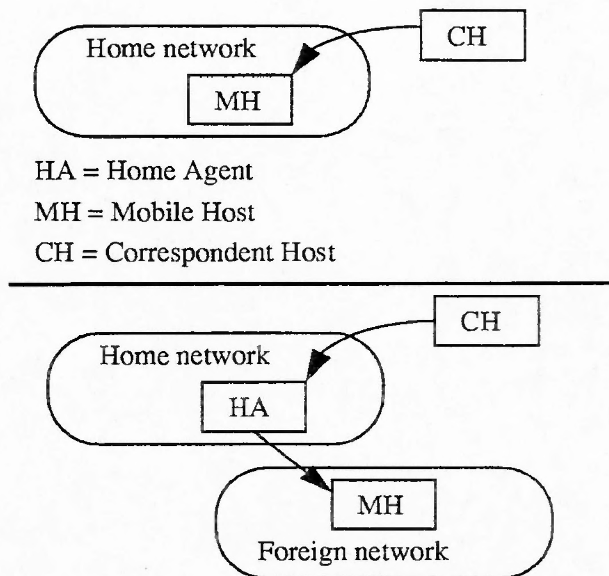


Figure 2.2: **Basic mobile host scenario:** The top half of the figure shows a correspondent host communicating with a mobile host that is still on its home network. The bottom half of the figure shows the path packets take when the mobile host moves to a foreign network. The correspondent host continues sending packets to the mobile host's home IP address. An agent in that network (the home agent) takes responsibility for forwarding these packets to the mobile host's new location on the foreign network.

agent and sends a registration message to its home agent to notify it of the new care-of-address. At this point the home agent is prepared to tunnel any packets it receives from a mobile host's correspondent hosts to the mobile host's current care-of-address, as illustrated in Figure 2.2.

The mobile host must also be able to send packets as well as receive them. At home, it sends packets in the normal fashion. While away from home, in this basic protocol, outgoing packets from the mobile host are also tunneled through the home agent to the correspondent hosts. With no foreign agent

on the foreign network, the mobile host must encapsulate these outgoing packets itself. This system can sometimes optimize the route for outgoing packets by sending them directly to the correspondent hosts, as is described in 2.3.1.

The basic role of a home agent is two-fold. It must decapsulate packets sent from the mobile host for delivery to correspondent hosts, and it must encapsulate packtes sent from correspondent hosts for delivery to the mobile host's care-of-address. To encapsulate packets sent to the mobile host, the home agent must be able to intercept them when they arrive in the home network. To intercept these packets, the home agent must function as the ARP proxy for the mobile host upon receiving its registration request. This is done by adding an ARP entry in the home agent's own ARP cache. The home agent must then broadcast gratuitous ARP on behalf of the mobile host to avoid any stale ARP cache entries on hosts in the same subnet as the mobile host's home. The home agent also adds an entry to its route table specifying that all packets for the mobile host's home IP address must be encapsulated. It adds a *mobility binding* to an internal table to record the mobile host's care-of-address and other information such as the lifetime of the registration and any authentication information.

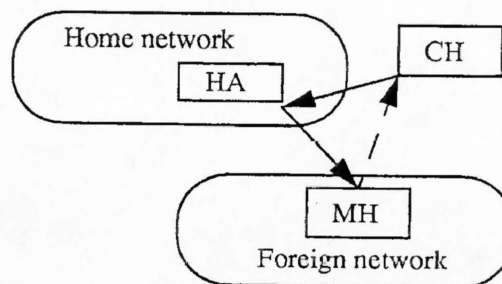


Figure 2.3: **Triangle route optimization:** This figure shows a simple route optimization called a tringle route. The optimization allows the mobile host to send a packet to the correspondent host without tunneling it through the home agent. This path is shown with a dotted line. The path for packets from the correspondent host to the mobile host remains unoptimized and passes through the home agent.

When the mobile host returns home, it de-registers with the home agent, which then removes the mobility binding and the special routing table entry. The home agent should also stop functioning as the ARP proxy for the mobile host.

Routing Optimizations

The first form of route optimization available in MosquitoNet is where a mobile host can send packets directly to its correspondent host. This forms a *triangle route*, as illustrated in Figure 2.3. For this simple optimization, the source IP address of the packets from the mobile host are set to the mobile host's home IP address rather than its current care-of-address. In this way, the mobile host can move as many times as it desires without the correspondent host noticing. As far as the correspondent host knows,

the mobile host is always at its home address. If the source address were allowed to reflect the current care-of-address, then packets with a changed address would not be recognized as coming from the mobile host without modifications to the correspondent host's software.

The problem with this optimization is that it does not work with some security- concious routers that forbid transit traffic. Transit traffic is traffic with a source address not local to the network, as is the case with a packet using the mobile host's home IP address as source address. If the foreign network has been set up to forbid transit traffic, then routers will drop outgoing packets from a mobile host using the triangle route optimization.

A variant of the triangle route optimization, suitable for use on networks that forbid transit traffic, still sends the packet directly to the correspondent host but encapsulates the packet using the mobile host's local source IP address. Now the packet will not be dropped by the filters in the local router, because it has a valid local source address. This optimization eliminates the sub-optimal routing for outgoing packets but not the encapsulation overhead. It is appropriate when the mobile host knows that the destination host has transparent IP-in-IP decapsulation capability such as is found in recent Linux development kernels.

2.4 Mobile-IP Performance Studies

In [4], Y. C. Tay *et al.* examine the performance requirement for and impact of mobility support. This study looks at two such performance issues, on the congestion, and throughput in a wireless cell. The approach throughout is to concentrate on drawing conclusions that can enhance the intuition and help the work of designers, engineers, and administrators.

2.4.1 Wireless CSMA Cell

In here, its assumed that the uplink communication in a wireless cell is by a non-persistent CSMA protocol. Altough this is similar to satellite communication from earthbound transmitters, an important difference is that the small size for a cell makes carrier sensing possible.

Briefly, the CSMA protocol is as follows: When a mobile host wants to send a packet, it checks whether the wireless channel is busy; if so, the node *backs off* (i.e. waits a while) before trying again. When the host finds the channel silent, it proceeds to send its packet *in its entirety.* It is possible that this transmission is corrupted by *collision* (with transmissions from other hosts) and *noise* (e.g. from electronic gadgets, or obstruction by objects that appear between the base station and the sender as the latter

moves). Since the sender cannot detect packet corruption (and so cannot abort a transmission) the job is left to the base station, which will broadcast an acknowledgement if there is no error. A sender that does not hear an acknowledgement within some time-out period must schedule a retransmission.

2.4.2 Congestion in a Cell

A new issue for a wireless CSMA cell is mobility, which changes M, the number of mobile hosts in a cell. To understand the effect M has on the transmission from each host, suppose the uplink traffic from each host to the base station is λ_{per_host} in packets per unit time. The total traffic (i.e. including transmissions corrupted by noise) is $M\lambda_{per_host}/(1-p_{noise})$, where p_{noise} is the probability that a packet is corrupted. Therefore

$$p_{busy} = \frac{M\lambda_{per_host}T_{packet}}{1-p_{noise}} . \tag{2.1}$$

A mobile host is likely to encounter wireless cells with varying frequencies [47], and must adjust its rate of transmission attempts accordingly. One popular mechanism for doing so is the *exponential backoff* [48], for which there is a constant $T_{backoff}$ so that, if a host finds the channel busy, it remains

idle for a random period with mean $2^{i-1}T_{backoff}$ after the ith attempt ($i =$ 1, 2, ...). The waiting time for the channel to be idle is therefore

$$T_{wait} = \frac{p_{busy}}{1 - 2p_{busy}} T_{backoff} \ . \tag{2.2}$$

This is an approximation since transmission attempts are, in fact, not statistically independent.

In practice, the mean idling period stops doubling after some N_1 attempts, and a message is discarded if it is not transmitted after some N_2 attempts (N_2 depending on a time-out) so the above approximation is good only if N_1 and N_2 are sufficiently large (say, greater than 7). Note here that exponential backoff requires $1 - 2p_{busy} > 0$ (i.e. $p_{busy} < 0.5$).

Substituting 2.1 into 2.2 gives

$$T_{wait} = \frac{T_{backoff}}{\frac{1-p_{noise}}{M\lambda_{per_host}T_{packet}} - 2} \ . \tag{2.3}$$

The time taken to send a packet (excluding retransmissions because of corruption by noise) is

$$T_{send} = T_{wait} + T_{packet} = \frac{T_{backoff}}{\frac{1-p_{noise}}{M\lambda_{per_host}T_{packet}} - 2} + T_{packet} \ . \tag{2.4}$$

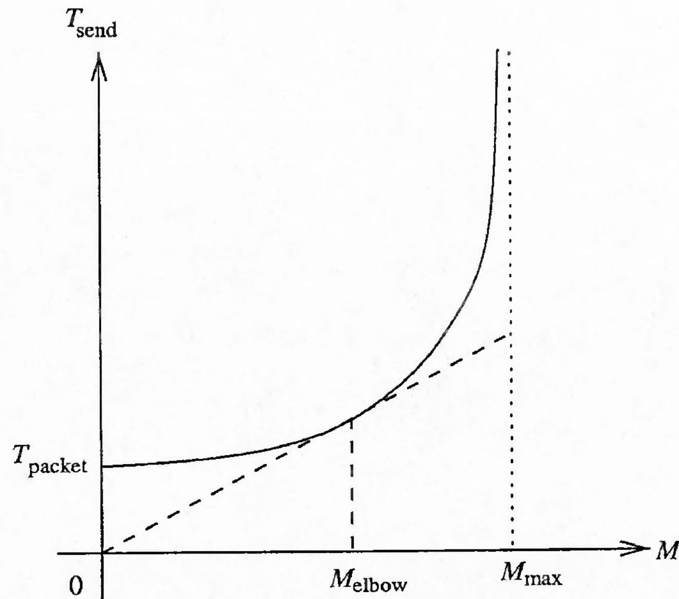


Figure 2.4: How packet sending time varies with the number of mobile hosts $M_{max} = (1 - p_{noise})/2\lambda_{per_host}T_{packet}$

This relationship is illustrated in Figure 2.4. Note that T_{send} increases rapidly as M increases – a sign of congestion. To get a guide for how many mobile hosts can be supported in a cell, the idea of an "elbow" is used, where M_{elbow} is defined by the point on the graph of T_{send} vs M at which the tangent passes through the origin. Treating M as a continuous variable the "elbow" is defined by

$$\frac{dT_{send}}{dM} = \frac{T_{send}}{M} at M = M_{elbow}.$$

Differentiating 2.4 now gives

$$M_{elbow} = \frac{1}{2 + \sqrt{2\frac{T_{backoff}}{T_{packet}}}} \frac{1 - p_{noise}}{\lambda_{per_host} T_{packet}} . \tag{2.5}$$

Note that M_{elbow} is smaller if $T_{backoff}$ is larger. Intuitively, a larger $T_{backoff}$ implies a longer waiting time to retransmit (see 2.3), so fewer nodes can be accomodated.

Let $G_{per_node} = \lambda_{per_host} T_{packet} / (1 - p_{noise})$, which is called the *offered load* (per node) in CSMA terminology. This gives us the following rule of thumb:

Congestion Point in a Cell

The time for sending a packet increases rapidly if the number of mobile hosts exceeds

$$\frac{1}{(2 + \sqrt{2\frac{T_{backoff}}{T_{packet}}}) G_{per_node}} .$$

One may consider this rule of thumb as defining the congestion point in a cell. A foreign agent concerned about quality of service in its cell may conceivably use this as a guide for limiting the number of mobile hosts on its visitor list, since the IETF proposal provides for registration denial by the foreign agent if there are insufficient resources (Code=66).

2.4.3 The Effect of Noise

One effect of noise is already evident from 2.5, where M_{elbow} is linear in p_{noise}; informaly, one could say that congestion increases linearly with noise. [4] show that this linearity holds as well when one considers the throughput sustainable in a cell.

Suppose the uplink traffic from M mobile hosts are $\lambda_1, \lambda_2, \ldots, \lambda_M$, so the total traffic is $\lambda_{to_base} = \lambda_1 + \lambda_2 + \ldots + \lambda_M$. In CSMA terminology, $S = \lambda_{to_base} T_{packet}$ is called the *throughput.* The analog of 2.3 is

$$T_{wait} = \frac{T_{backoff}}{\frac{1-p_{noise}}{S} - 2} \ . \tag{2.6}$$

Stability of the transmission queue at each mobile host requires that

$$\lambda_i (T_{wait} + T_{packet}) < 1, so \lambda_{to_base} T_{wait} < (\lambda_1 + \ldots + \lambda_M)(T_{wait} + T_{packet}) < M.$$

From 2.6 , we get

$$\frac{S}{\frac{1-p_{noise}}{S} - 2} \frac{T_{backoff}}{T_{packet}} < M \ .$$

This gives a quadratic in S with a negative and a positive root, and the

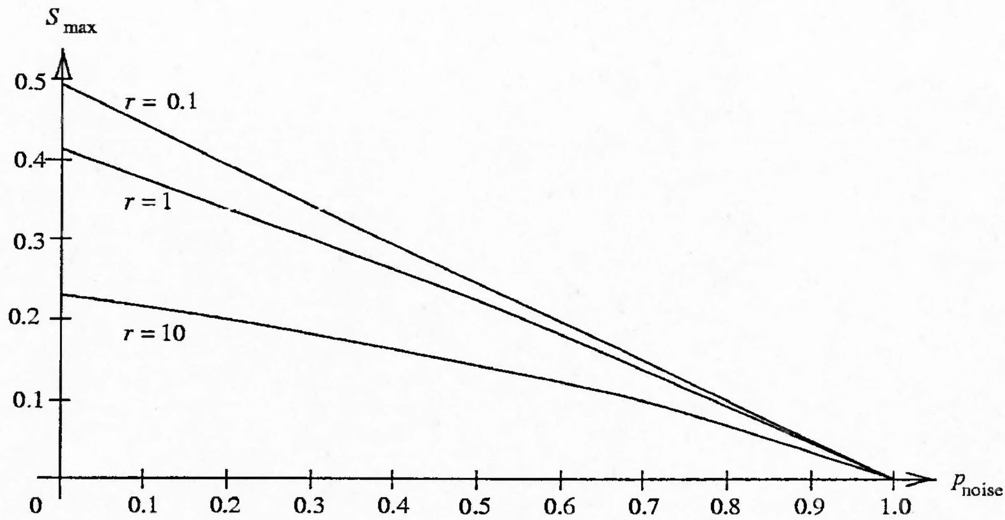


Figure 2.5: How the throughput bound varies with noise ($r = T_{backoff}/MT_{packet}$)

latter imposes an upper bound on S:

Throughput Bound

$$S < S_{max} = \frac{MT_{packet}}{T_{backoff}}\left(\sqrt{1 + (1 - p_{noise}\frac{T_{backoff}}{MT_{packet}})} - 1\right) . \qquad (2.7)$$

This relationship is illustrated in Figure 2.5. It shows that, for a realistically acceptable level of noise (say, $p_{noise} < 0.2$), S_{max} decreases linearly (almost) with p_{noise}.

2.5 Summary

This chapter examined a number of mobile internetworking schemes, which varied in terms of the locus of mobile host location information, network infrastructure, and requirements. It started by presenting a design that takes a network-layer centric approach to solving the mobile host location information problem; the Sony VIP approach, followed by two schemes that take a more agent centric approach to solving the same problem. These two later schemes leave the IP network layer intact and instead rely on software implementable agents that work in unison to hide host mobility from the network and application layer protocols. In what follows we indicate why none of these designs are really suited for a high-volume mobile internetworking system for today's data networks.

These are the problems with the Sony approach:

1. It doubles the address space requirements for mobile hosts, since they are all required to have two IP addresses when they move away from their home networks.

2. All participating non-mobile hosts and routers have to be modified for this scheme to work with any degree of efficiency.

3. AMTs can grow without bounds.

4. Options processing is required at every router along the path, potentially slowing down the routing process.

These are the problems with the Columbia approach:

1. The approach is not fully backwards compatible, as it has no multicast support.

2. It requires a large pool of temporary addresses in *popup* mode, and each MSR must advertise connectivity, leading to inefficient use of network resources.

3. The existing infrastructure is not adequate for this scheme to work with any degree of efficiency since there must be an MSR wherever an MH may connect.

These are the problems with the Stanford approach:

1. The mobile host needs to adquire a temporary IP address in the foreign network. There may be networks that refuse to do this.

2. If packets for a mobile host arrive at a foreign network the mobile host has just left, those packets might be erroneously delivered to a newly

arrived host that has been assigned the same temporary IP address the recently departed host used.

3. When a mobile host leaves a network, it must inform its home agent of its new care-of-address. However, any packets already sent by the home agent before it receives the new registration will arrive at the old network and will be lost.

4. The lack of foregin agent support complicates the design of the mobile host. This is beacuse mobile hosts effectively contain a simplified foreign agent. The result is that some networking software on mobile hosts must be aware of its actual physical location and must handle routing operations and changes to the physical network interfaces.

Chapter 3

An Overview of Ptolemy

3.1 Introduction

The core of Ptolemy is a compact software infrastructure upon which specialized design environments (called *domains*) can be built. The software infrastructure called *the Ptolemy kernel*, is made up of a family of C++ class definitions. Domains are defined by creating new C++ classes derived from the base classes in the kernel.

Domains can operate in eihter of two modes:

Excerpted by permission of Eduard A. Lee, Department of Electrical Engineering and Computer Science, University of California-Berkeley. "The Almagest: Vol. 1 - Ptolemy User's Manual". Copyright by the Regents of the University of California. All rights rcserved.

- Simulation – A scheduler invokes code segments in an order appropriate to the model of computation.
- Code generation – Code segments in an arbitraty language are stitched together to produce one or more programs that implement the specified function.

The use of an object-oriented software technology permits a domain to interact with one another without knowledge of the features or semantics of the other domain. Thus, using a variety of domains, a team of designers can model each subsystem of a complex, heterogeneous system in a natural and efficient manner. These different subsystems can be nested to form a tree of subsystems. This hierarchical composition is key in specifying, simulating, and synthesizing complex, heterogeneous systems.

By supporting heterogeneity, Ptolemy provides a research laboratory to test and explore design methodologies that support multiple design styles and implementation technologies. A simple example is simulating the effects of transmitting compressed video and audio over an asynchronous transfer mode (ATM) network. The network will delay, drop, and reorder packets based on the congestion. Compression and decompression, however, work on the video and audio data, and the time associated with the data is not relevant to the

signal processing. The simulation in this case is heterogeneous: the network processes discrete events (packets) with a notion of time, whereas the signal processing processes data independent of time. Other examples of heterogeneous systems include integrated control and signal processing architectures, mixed analog/digital simulation, and hardware/software codesign.

In short, Ptolemy is a flexible foundation upon which to build prototyping environments. The Ptolemy 0.7 release contains, for example, dataflow-oriented graphical programming for signal processing [18] [19] [20] , a multi-threaded process networks modeling environment [21], a synchronous/reactive programming framework [22], discrete-event modeling of communication networks [23] [24] [25], and synthesis environments for embedded software [26] [27] [28]. The Ptolemy system is fundamentally extensible, as all of the source code is released. Users can create new component models, new design process managers, and even entirely new programming environments.

3.2 Ptolemy Kernel

The overall organization of the latest release of the Ptolemy system is shown in Figure 3.1. A typical use of Ptolemy involves starting two Unix processes,

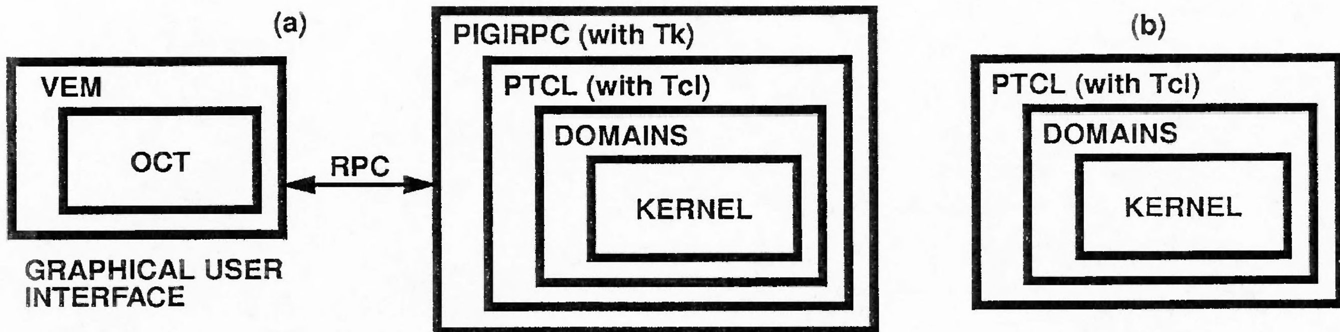


Figure 3.1: The overall organization of Ptolemy version 0.7, showing two possible execution styles: (a) graphical interface and (b) textual interface.

as shown in Figure 3.1(a), by running `pigi` (Ptolemy interactive graphical interface). The first process contains the `vem` user interface and the `oct` design database [33] , and the other process contains the Ptolemy kernel. An alternative is to run Ptolemy without the graphical user interface, as a single process, as shown in Figure 3.1(b). In this case, the textual interpreter is based on the Tool Command Language, Tcl [34] [35], and is called `ptcl` for Ptolemy Tcl. It is possible to design other user interfaces for the system. We are releasing a preliminary version of a third interface called Tycho. In its current from, Tycho is best suited for language-sensitive editing and consoles for tools such as Matlab and Mathematica.

The executable programs `pigiRpc` or `ptcl` can be configured to include any subset of the available domains. The most recent picture of the domains that Berkeley has developed is shown in Figure 3.2. Many different styles of

design are represented by these domains. More are constantly being developed both at U.C. Berkeley and elsewhere, to experiment with or support alternative styles.

The Ptolemy kernel provides the most extensive support for doamins where a design is represented as a network of blocks, as shown in Figure 3.3. A base class in the kernel, called `Block`, represents an object in this network. Base classes are also provided for interconnecting blocks (`PortHoles`) as well as for carrying data between blocks (`Geodesic`) and managing the garbage collection efficiently (`Plasma`). Not all domains use these classes, but most current ones do, and hence can very effectively use this infrastructure.

Figure 3.3 shows some of the representative methods defined in these base classes. For example, note the ***initialize***, ***run***, and ***wrapup*** methods in the class `Block`. These provide an interface to whatever functionality the block provides, representing for example functions performed before, during, and after (respectively) the execution of the system.

Blocks can be hierarchical, as shown in Figure 3.4. The lowest level of the hierarchy, as far as Ptolemy is concerned, is derived from a kernel base class called `Star`. A hierarchical block is a `Galaxy`, and a top-level system representation is a `Universe`.

3.3 Models of Computation

The Ptolemy kernel does not define any model of computation. In particular, although the Berkeley team has done quite a bit of work with dataflow domains in Ptolemy, every effort has been made to keep dataflow semantics out of the kernel. Thus, for example, a network of blocks could just as easily represent a finite-state machine, where each block represents a state. It is up to a particular domain to define the semantics of a computational model.

Suppose we wish to define a new domain, called `XXX`. We would define a set of C++ classes derived from kernel base classes to support this domain. These classes might be called `XXXStar`, `XXXUniverse`, etc., as shown in Figure 3.4.

The semantics of a domain are defined by classes that manage the execution of a specification. These classes could invoke a simulator, or could generate code, or could invoke a sophisticated compiler. The base class mechanisms to support this are shown in Figure 3.5. A `Target` is the top-level manager of the execution. Similar to a `Block`, it has methods called `setup`, `run`, and `wrapup`. To define a simulation domain called `XXX`, for example, one would define at least one object derived from Target that runs the simulation. As suggested by Figure 3.5, a Target can be quite sophisticated. It

can, for example, partition a simulation for parallel execution, handling off the partitions to other Targets compatible with the domain.

A Target will typically perform its function via a Scheduler. The Scheduler defines the operational semantics of a domain by controlling the order of execution of functional modules. Sometimes, schedulers can be specialized. For instance, a subset of the dataflow model of computation called synchronous dataflow (SDF) allows all scheduling to be done at compile time. The Ptolemy kernel supports such specialization by allowing nested domains, as shown in Figure 3.6. For example, the SDF domain (see Figure 3.2) is a subdomain of the BDF domain. Thus, a scheduler in the BDF domain can handle all stars in the SDF domain, but a scheduler in the SDF domain may not be able to handle stars in the BDF domain. A domain may have more than one scheduler and more than one target.

3.4 Dataflow Models of Computation

One of the most mature domains included in the current system is the synchronous dataflow (SDF) domain [18], which is similar to that used in Gabriel. This domain is used for signal processing and communications al-

gorithm development, and has particularly good support for multirate algorithms [19]. It has been used at Berkeley for instruction, at both the graduate and undergraduate level [36]. A dynamic dataflow (DDF) domain extends SDF by allowing data-dependent flow of control, as in Blosin. Boolean dataflow (BDF) [20] has a compile-time scheduler for dynamic dataflow graphs [37].

Several code-generation domains use dataflow semantics [38]. These domains are capable of synthesis of C code, assembly code for certain programmable DSPs [39], VHDL, and Silage [29]. A significant part of the research that led to the development of these domains has been concerned with synthesizing code that is efficient enough for embedded systems [26] [27] [20]. A large amount of effort has also been put into the automatic parallelization of the code [40] [41] [42] , and on parallel architectures that take advantage of it [37] [44].

A generalization of dataflow, called Kahn process networks [43], has been realized by Tom Parks in the PN domain [21].

3.5 Discrete-Event Models of Computation

A number of simulation domains with discrete-event semantics have been developed for Ptolemy, but only the DE domain is released with Ptolemy 0.7. The DE domain is a generic discrete-event modeling environment, useful for simulating queueing systems, communication networks, and hardware systems. The discrete-event domains no longer released with Ptolemy 0.7 are Thor [45] for modeling circuits at the register-transfer level [29], communicating processes (CP) for modeling large-scale systems at a high level of abstraction, and message queue (MQ) for modeling a centralized network controller in a large-scale cell-relay network simulations [46].

3.6 Synchronous Reactive Modeling

The software analogy of synchronous digital circuits has been realized by Stephen Edwards in the SR domain [22]. This model of computation is better suited than dataflow to control-intensive applications, and is more efficient than DE.

3.7 Finite State Machines

Another approach to designing control-intensive applications is to mix the new FSM domain with dataflow, DE, or (future releases) SR. The FSM domain is still very new and has many limitations, but we believe that for the long term, it provides one of the most exciting developments in the Ptolemy software.

3.8 Mixing Models of Computation

Large systems often mix hardware, software, and communication subsystems. The hardware subsystems may include pre-fabricated components, such as custom logic, processors with varying degrees of programmability, systolic arrays, and multiprocessor subsystems. Tools supporting each of these components are different, possibly using dataflow principles, regular iterative algorithms, communicating sequential processes, control/dataflow hybrids, functional languages, finite-state machines, and discrete-event system theory and simulation.

In Ptolemy, domains can be mixed and even nested. Thus, a system-level description can contain multiple subsystems that are designed or specified

using different styles. The kernel support for this is shown in Figure 3.7. An object called `XXXWormhole` in the `XXX` domain is derived from `XXXStar`, so that from the outside it looks just like a primitive in the `XXX` doamin. Thus, the schedulers and targets of the `XXX` domain can handle it just as they would handle any other primitive block. However, inside, hidden from the `XXX` domain, is another complete subsystem defined in another domain, say `YYY`. That domain gets invoked through the `setup`, `run`, and `wrapup` methods of `XXXWormhole`. Thus, in a broad sense, the wormhole is polymorphic. The wormhole mechanism allows domains to be nested many levels deep, e.g., one could have a DE domain within an SDF domain within a BDF domain. The FSM domain is designed to always be used in combination with other domains.

3.9 Code Generation

Domains in Figure 3.2 are divided into two classes: simulation and code generation. In simulation domains, a scheduler invokes the run methods of the blocks in a system specification, and those methods perform a function associated with the design. In code generation domains, the scheduler also

invokes the run methods of the blocks, but these run methods synthesize code in some language. That is, they generate code to perform some function, rather than performing the function directly. The Target then is responsible for generating the connecting code between the blocks (if any is needed). This mechanism is very simple, and language independent. We have released code generators for C, Motorola 56000 assembly, and VHDL languages, as shown in Figure 3.2.

An alternative mechanism that is supported but less exploited in current Ptolemy is for the target to analyze the network of blocks in a system specification and generate a single monolithic implementation. This is what we call compilation. In this case, the primitive blocks (`Stars`) must have functionality that is recognized by the target. In the previous code generation mechanisms, the functionality of the blocks is arbitrary and can be defined by the end user.

3.10 Conclusion

In summary, the key idea in the Ptolemy project is to mix models of computation, implementation languages, and design styles, rather than trying to

develop one, all-encompassing technique. The rationale is that specialized design techniques are (1) more useful to the system-level designer, and (2) more amenable to high-quality high-level synthesis of hardware and software. The Ptolemy kernel demonstrates one way to mix tools that have fundamentally different semantics, and provides a laboratory for experimenting with such mixtures.

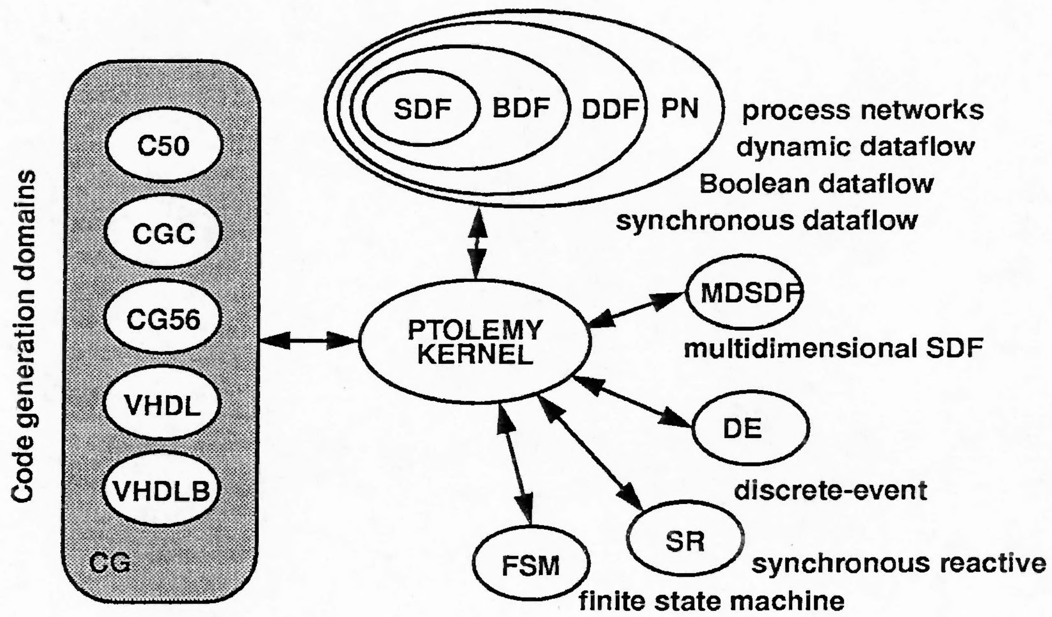


Figure 3.2: Domains available with Ptolemy 0.7

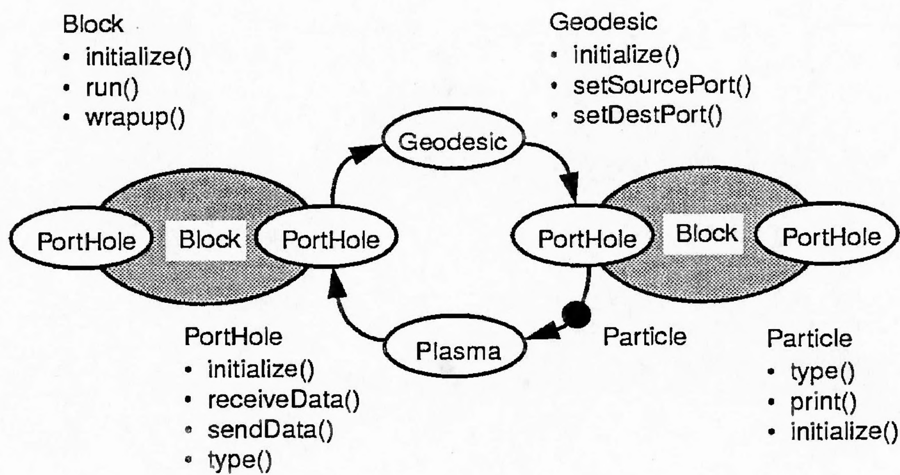


Figure 3.3: Block objects in Ptolemy can send and receive data encapsulated in Particles through Portholes. Buffering and transport is handled by the Geodesic and garbage collection by the Plasma. Some methods are shown.

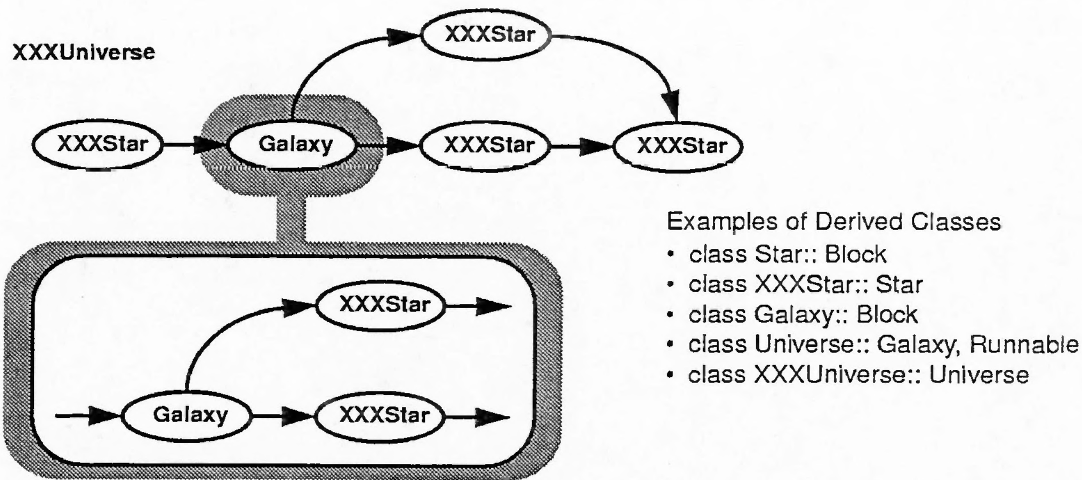


Figure 3.4: A complete Ptolemy application (a Universe) consists of a network of Blocks. Blocks may be Stars (atomic) or Galaxies (composite). The "XXX" prefix symbolizes a particular domain (or model of computation).

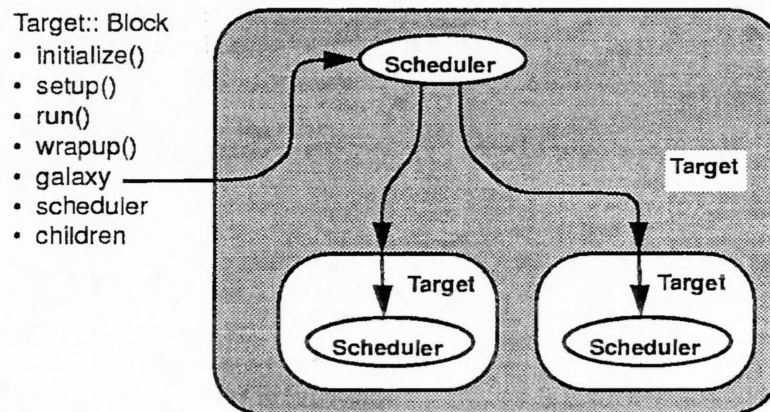


Figure 3.5: A Target, derived from Block, manages a simulation or synthesis execution. It can invoke it's own Scheduler on a Galaxy, which can in turn invoke Schedulers sub-Targets.

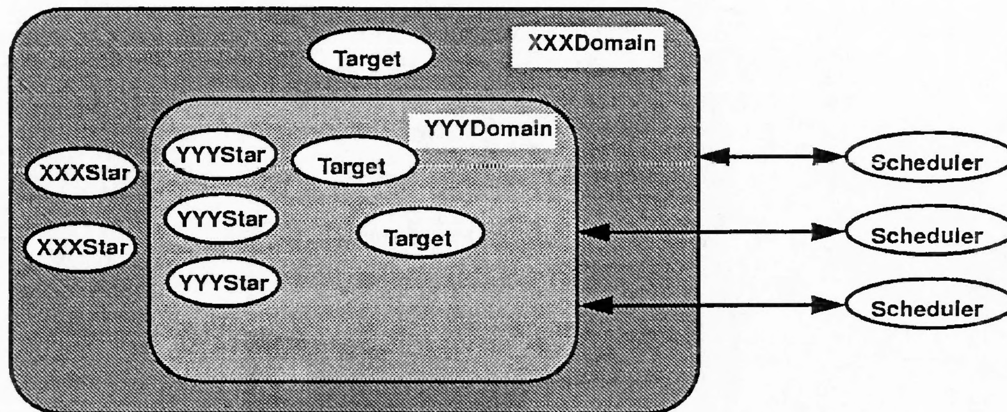


Figure 3.6: A Domain (XXX) consists of a set of Stars, Targets, and Schedulers that support a particular model of computation. A sub-Domain (YYY) may support a more specialized model of computation.

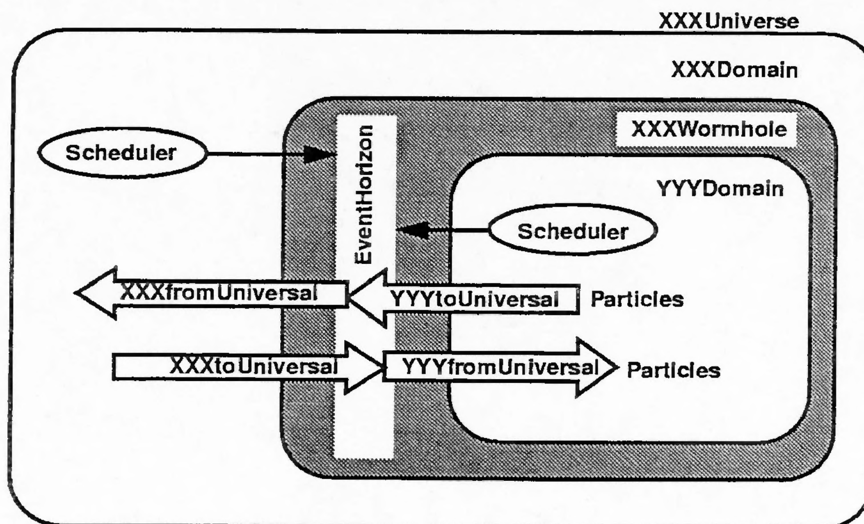


Figure 3.7: The universal EventHorizon provides an interface between the external and internal domains.

Chapter 4

An Overview of Mipps

4.1 Introduction

The Mobile IP protocol simulator (Mipps) in our work provides a fundamental collection of *buidling blocks* (`Stars` and `Galaxies`) and *runnables* (`Universes`), as well as a general environment for time-oriented simulations of communication networks that employ IPv4 or Mobile IP as the network layer protocols.

The top level pallete is shown in figure 4.1.

Our idea is to provide the communications network researcher with a set of object-oriented components that could allow him or her rapid development

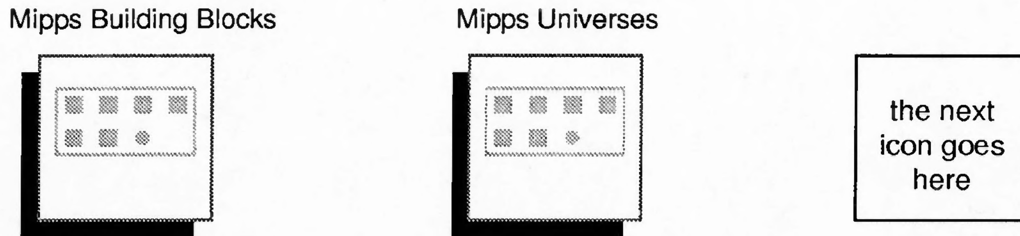


Figure 4.1: The top level pallete of building blocks and universes

of experiments and collection of performance measures.

The user pallete is shown in figure 4.2.

4.2 Stars in Mipps

The lowest level (atomic) objects in Ptolemy are of type Star. Stars in Mipps were designed to perform a given computation in support of the overall function of the system. Such a computation can be considered the *service* provided by the star. A user of a star need only be concerned with the inputs, the outputs, and, most important, the internal processing and latency of the star.

4.2.1 Network stars

The network stars provided by Mipps have multiple inputs and multiple outputs. These stars read in `NetworkCell` packets from their input `PortHoles`,

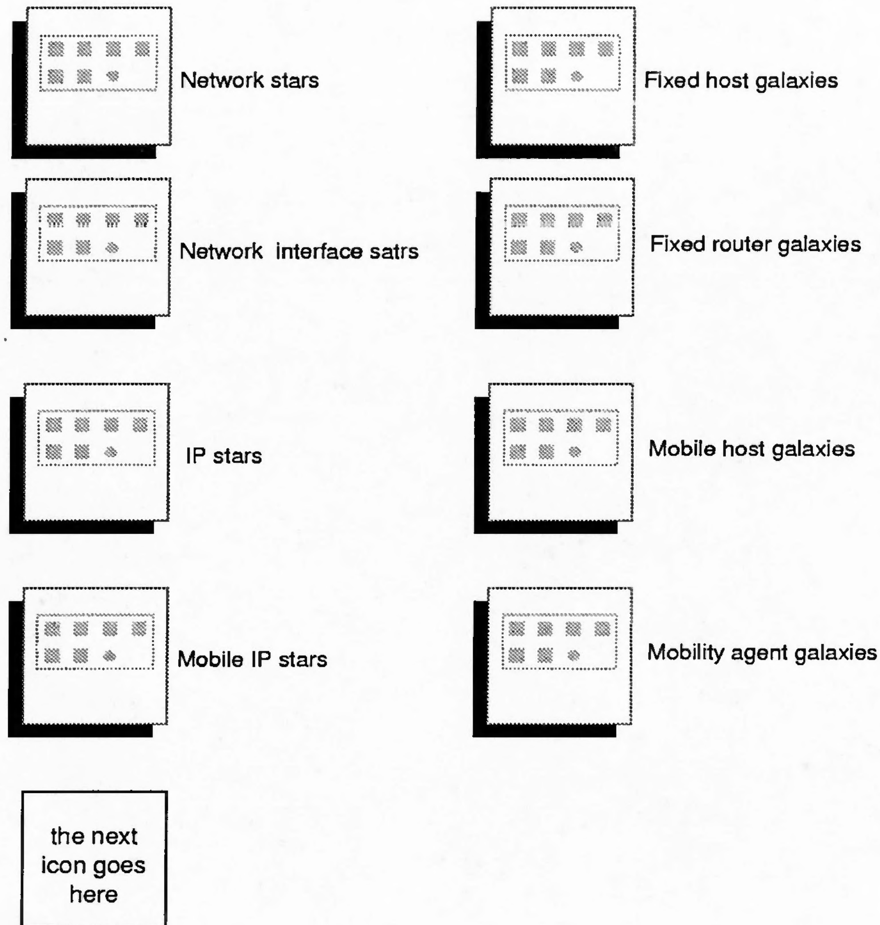


Figure 4.2: The user pallete of stars and galaxies in Mipps

use the destination field in their physical-layer header as an index into the output `PortHoles` mapping table and pass these `NetworkCells` to the next hop or final destination. They implement an ideal LAN without communication delays or collisions. The role of the *medium access control* (MAC) sublayer is played by the DE target and its schedulers. The definition of the `Net` class may be found in appendix A.

The network pallete is shown in figure 4.3.

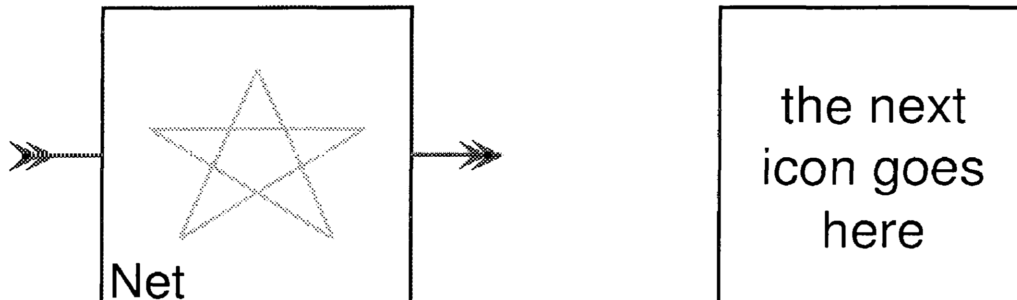


Figure 4.3: The network stars in Mipps

4.2.2 Network interface stars

The *network interface layer* controls the network hardware, performs mappings from IP addresses to hardware addresses, encapsulates and transmits outgoing packets, and accepts and demultiplexes incoming packets. Strictly speaking, the network interface stars provided by Mipps decapsulate or encapsulate **NetworkCell** packets delivered to or send from the network layer protocols. The **if_input** star strips datagrams of their physical headers and passes these to the network layer; the **if_output** star receives packets from the network layer and appends to them a physical header with the physical address of the next destination.

The **if_input** star has one input **PortHole** labeled `if_rcv`, one output **PortHole** labeled **ipintrq**, and it reads **NetworkCell** packets from a single

source. The processessing of a packet is as follows: first, its arrival is recorded by incrementing the state variable `if_ipackets`; second, its physical header is stripped off; and third, its size its measured and recorded in the state variable `if_ibytes`. Finally, the packet is put on the IP input queue. The definition of the `if_input` class may be found in appendix A; it maintains the following statistics:

- if_ipackets #packets received by this interface
- if_ibytes #bytes received by this interface

The `if_output` star has one input `PortHole` labeled `ip_output`, one output `PortHole` labeled `if_snd`, and it reads `NetworkCell` packets from a single source. This star performs several functions before sending a packet on its output `PortHole`. First, it translates the IP address of the next destination to a physical address. Whenever this transalation yields a physical address, the state variables `if_opackets` and `if_obytes` are updated so as to reflect the current number of packets and number of bytes send on this interface. Second, it appends a physical header to the outgoing packet. And finally, it puts the outgoing packet on the `if_snd` port. The definition of the `if_output` class may be found in appendix A ; it maintains the following statistics:

- if_opackets #packets sent by this interface

- if_obytes #bytes sent by this interface

The network interface pallete is shown in figure 4.4.

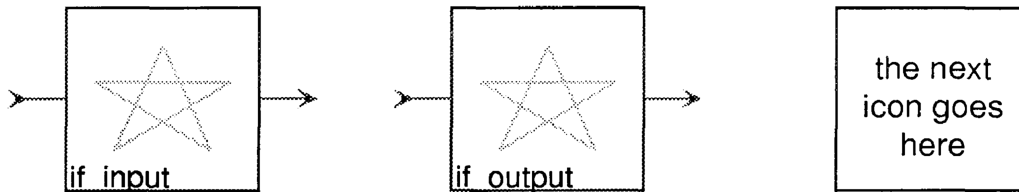


Figure 4.4: The network interface stars in Mipps

4.2.3 IP stars

Conceptually, IP is a central switching point in the *network layer* protocol software. It accepts incoming datagrams from the network interface software as well as outgoing datagrams that higher-level protocols generate. After routing a datagram, IP either sends it to one of the network interfaces or to a higher-level protocol on the local machine.

In Mipps, we simplistically think of IP software in two distinct parts: one that handles input and one that handles output. The input part `ip_input` passes incoming datagrams to higher-level protocols or to the output part. The output part `ip_output` uses the local routing table to choose a next hop for outgoing datagrams.

The `ip_input` star has one input `PortHole` labeled `ipintrq`, two output `PortHoles` labeled `pr_input` and `ip_output`, respectively, and it reads `NetworkCell` packets from multiple sources. For each packet read, it searches its list of network interfaces. Whenever a match is found, the packet is passed to higher-level protocols. Otherwise, assuming `ipforwarding` is enabled, the packet is forwarded to the output part on its `ip_output` port or silently discarded. The definition of the `ip_input` class may be found in appendix A; it maintains the following statistics:

- ips_delivered #packets delivered to upper level
- ips_forward #packets forwarded
- ips_total total #packets received
- ips_odropped #packets dropped

The `ip_output` star has two input `PortHoles`, one labeled `pr_output` and the other labeled `ip_input`, one output `PortHole` labeled `if_output`, and it reads `NetworkCell` packets from either the input part or higher-level protocol software. This is where the routing of packets takes place. The routing software in this star performs the following steps when it searches its routing table:

1. Search for a matching host address.

2. Search for a matching network address.

3. Search for a default entry. (The default entry is normally specified in the routing table as a network entry with a network ID of 0.)

The definition of the **ip_output** class may be found in appendix A; it maintains the following statistics:

- ips_noroute #packets discarded – no route to destination

The IP pallete is shown in figure 4.5.

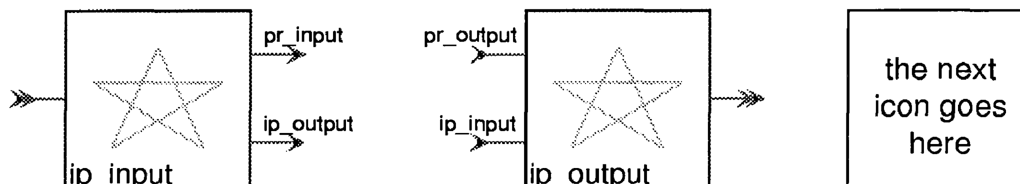


Figure 4.5: The IP stars in Mipps

4.2.4 Mobile IP stars

In essence, Mobile IP is IP plus a mechanism for routing IP packets to mobile nodes which may be connected to any subnet while using their permanent IP address. In Mipps, we took the same approach to Mobile IP software as we did to IP software; we think of Mobile IP software in two distinct parts:

one that handles input plus decapsulation and one that handles output plus IP-in-IP encapsulation. The input part `ipmobile_input` passes decapsulated incoming datagrams to higher-level protocols or encapsulated incoming datagrams to the output part. The output part `ipmobile_output` uses the local routing table to choose a next hop or tunnel end-point for outgoing datagrams.

The `ipmobile_input` star has one input `PortHole` labeled `ipmobileintrq` and two output `PortHoles` labeled `pr_input` and `ipmobile_output`, respectively, and it reads `NetworkCell` packets from multiple sources. Besides performing the standard function of passing incoming datagrams to higher-level protocol software, this star will decapsulate incoming datagrams whenever its state variable *ipdecapsulating* is set to true. The definition of the `ipmobile_input` class may be found in appendix A; it maintains the following additional statistics:

- ipms_drpackets #packets decapsulated
- ipms_rpackets #packets requesting registration

The `ipmobile_output` star has two input `PortHoles` labeled `pr_output` and `ipmobile_input`, two output `PortHoles` labeled `ifp` and `vifp`, respec-

tively, and it reads `NetworkCell` packets from either the input part or higher-level protocol software. In addittion to searching its routing table for a next hop, this star will encapsulate datagrams destined to mobile hosts present in its mobile host table. The definition of the `ipmobile_output` class may be found in appendix A; it maintains the following additional statistics:

- ipms_epackets #packets encapsulated
- ipms_mforward #packets sent as agent advertisements

The Mobile IP pallete is shown in figure 4.6.

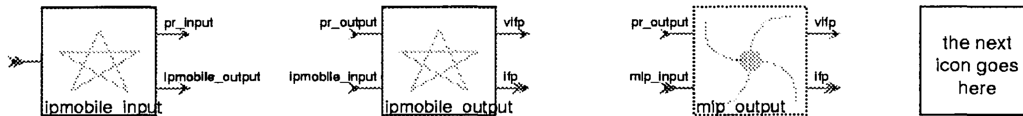


Figure 4.6: The Mobile IP stars and galaxies in Mipps

4.3 Galaxies in Mipps

A Galaxy may contain internally both galaxies and stars. A Galaxy may exist only as a descriptive tool, in that a `Scheduler` may ignore the hierarchy, viewing the entire network of blocks as flat. Galaxies in Mipps were designed to map to entities such as hosts or routers; these entities possess structure

and perform multiple computations in support of the overall function of the system.

4.3.1 Fixed host galaxies

The fixed host galaxies in Mipps abstract most of the common features of applications being hosted on PCs, workstations, mainframes, file servers, and other types of computers whose point of attachement to a network or sub-network will seldomly change. At a high-level of abstraction, these common features are: application layer software, network layer software, and network interface layer software.

The fixed host pallete is shown in figure 4.7.

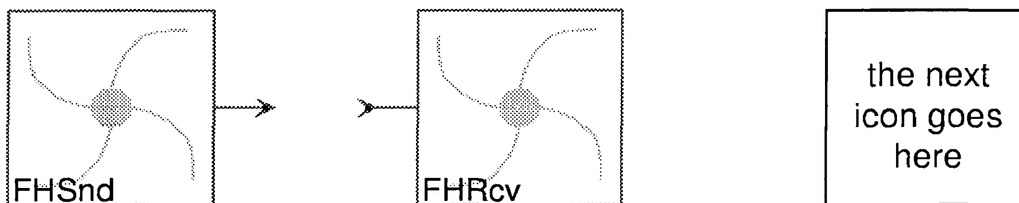


Figure 4.7: The fixed host galaxies in Mipps

The **FHSnd** galaxy of Mipps is used for simulating the send capability of an application hosted on a computer fixed to a network or subnetwork. As shown below, the application layer software is implemented by connecting a **Poisson** star to a **PktSrc** galaxy; both of these components are provided

by the DE domain in Ptolemy. The result of this combination is a source of `NetworkCell` packets whose `meanTime` between packets can be varied. The network layer software and the network interface layer software are implemented in the `ip_output` star and the `if_output` star, respectively, that are provided by Mipps.

The FHSnd galaxy internal components is shown in figure 4.8.

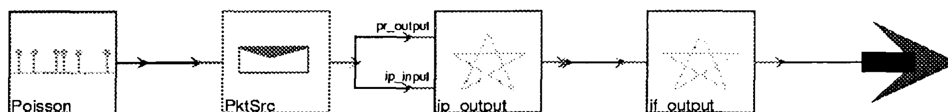


Figure 4.8: The FHsnd galaxy internal components.

The `FHRcv` galaxy of Mipps is used for simulating the receive capability of an application hosted on a computer fixed to a network or subnetwork. As shown below, the application layer software is implemented by connecting the stars `CellUnload`, `UnPacketize`, `Ramp`, and `XGraph` together. These stars are provided by the DE domain in Ptolemy. The network layer software is implemented by the `ip_input` star and the network interface layer software is implemented by the `if_input` star; both of these stars are provided by Mipps.

The FHRcv galaxy internal components is shown in figure 4.9.

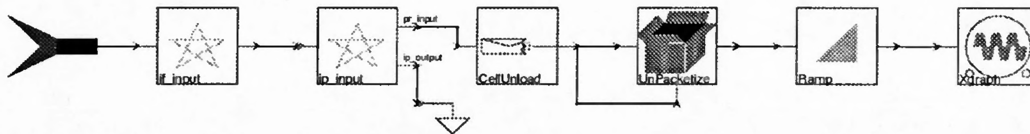


Figure 4.9: The FHRcv galaxy internal components.

4.3.2 Fixed router galaxies

The fixed router galaxies in Mipps abstract most of the common features of IP routers whose points of attachement to networks or subnetworks will seldomly change. These common features are: network layer software and network interface layer software.

The fixed router pallete is shown in figure 4.10.

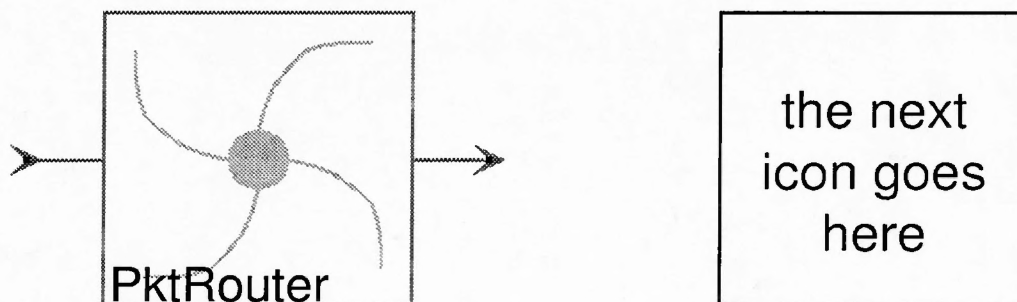


Figure 4.10: The fixed router galaxies in Mipps

The **PktRouter** galaxy of Mipps is used for simulating a device which operates at the network layer. That is, a device which forwards **NetworkCell** packets based upon information contained in their respective network layer headers. The network layer software is implemented by connecting an ip_input

star to an **ip_output** star. The interface layer software is implemented by connecting an **if_input** star to the IP input part and an **if_output** star to the IP output part. All of these stars are provided by Mipps.

The PktRouter galaxy internal components is shown in figure 4.11.

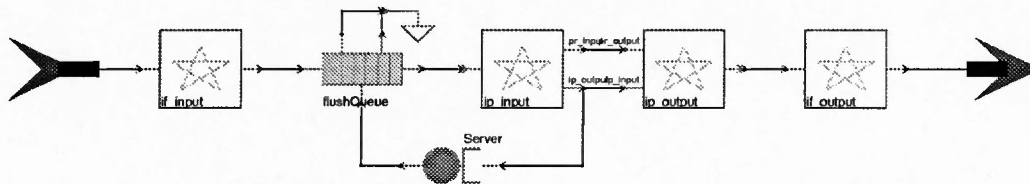


Figure 4.11: The PktRouter galaxy internal components.

4.3.3 Mobile host galaxies

The mobile host galaxies in Mipps abstract most of the common features of applications being hosted on a mobile node; that is, a node which can change its point of attachement to a network or subnetwork while maintaining any ongoing communications and using only its (pertmanent) IP home address. On most mobile nodes of interest, these common features are: application layer software, network layer software, and network interface layer software.

The mobile host pallete is shown in figure 4.12.

The `MHSnd` galaxy of Mipps is used for simulating the send capability of an application hosted on a computer that changes its point of attachement

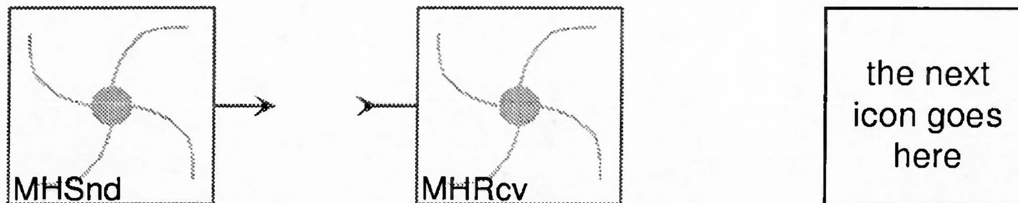


Figure 4.12: The mobile host galaxies in Mipps

to a network or subnetwork. The application layer software is implemented by connecting the stars **Poisson**, **Ramp**, **Packetize**, and **CellLoad** together. The result of this combination is a source of **NetworkCell** packets whose **meanTime** between packets can be varied. These stars are provided by the DE domain in Ptolemy. The network layer software is implemented by **ip_output** and the network interface layer software by **if_output**; both of these stars are provided by Mipps.

The MHSnd galaxy internal components is shown in figure 4.13.

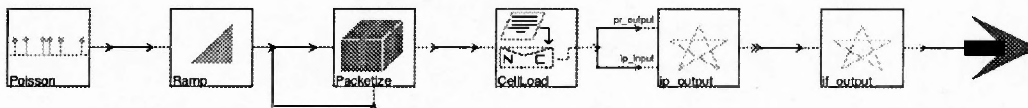


Figure 4.13: The MHsnd galaxy internal components.

The **MHRcv** galaxy of Mipps is used for simulating the receive capability of an application hosted on a computer that changes its point of attachement to a network or subnetwork. The application layer software is implemented by connection the stars **CellUnload**, **UnPacketize**, **Ramp**, and **XGraph** together.

These stars are provided by the DE domain in Ptolemy. The network layer software is implemented by `ipmobile_input` and the network interface layer software by `if_input` ; both of these stars are provided by Mipps.

The MHRcv galaxy internal components is shown in figure 4.14.

4.3.4 Mobility agent galaxies

The mobility agent galaxies in Mipps abstract the common features of Mobile IP routers whose points of attachement to networks or subnetworks will seldomly change. These common features are: network layer software and network interface layer software.

The mobility agent pallete is shown in figure 4.15.

The `HA` galaxy of Mipps is used for simulating a device which operates at the network layer. That is, a device which forwards `NetworkCell` packets based upon information contained in their respective network layer headers, tunnels datagrams for delivery to the mobile node when it is away from home, and maintains current location information for the mobile node.

The HA galaxy internal components is shown in figure 4.16.

The `FA` galaxy of Mipps is used for simulating a device which operates at the network layer. That is, a device which forwards `NetworkCell` packets

based upon information contained in their respective network layer headers, detunnels and delivers datagrams to the mobile node that were tunneled by the mobile node's home agent.

The FA galaxy internal components is shown in figure 4.17.

4.4 Some Sample Universes in Mipps

Mipps comes with three representative sample universes. The first universe focuses on the MHs-HA interaction, the second universe focuses on the FA-MHs interaction, and the third universe focuses on the FHs-HA interaction. How to setup these universes is described in the following subsections.

The demos pallete is shown in figure 4.18.

4.4.1 The MHs-HA Interaction Example

To setup this universe one most perform these steps:

1. initialize the `FHSnd` galaxies,

2. initialize the `MHSnd` galaxies,

3. initialize the `FA` and the `HA` galaxies, and

4. initialize the `FHRcv` galaxies

The home agent demo pallete is shown in figure 4.19.

4.4.2 The FA-MHs Interaction Example

To setup this universe one most perform these steps:

1. initialize the `FHSnd` galaxies,
2. initialize the `HA` and the `FA` galaxies, and
3. initialize the `FHRcv` galaxies

The foreign agent demo pallete is shown in figure 4.20.

4.4.3 The FHs-HA Interaction Example

To setup this universe one most perform these steps:

1. initialize the `FHSnd` galaxies,
2. initialize the `HA` and the `FA` galaxies, and
3. initialize the `MHRcv` galaxies

The mobility agent demo pallete is shown in figure 4.21.

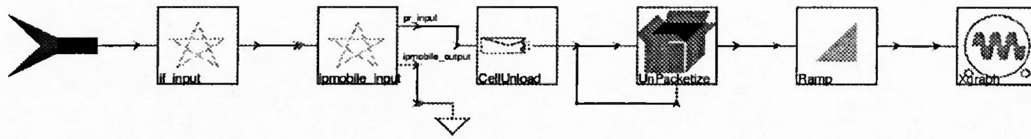


Figure 4.14: The MHRcv galaxy internal components.

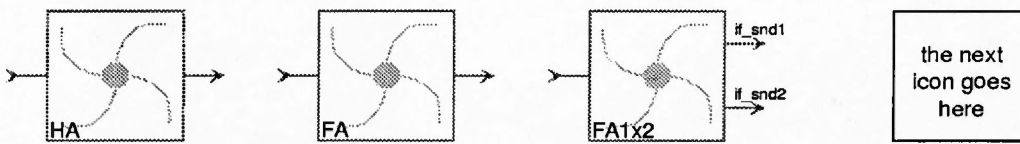


Figure 4.15: The mobility agent galaxies in Mipps

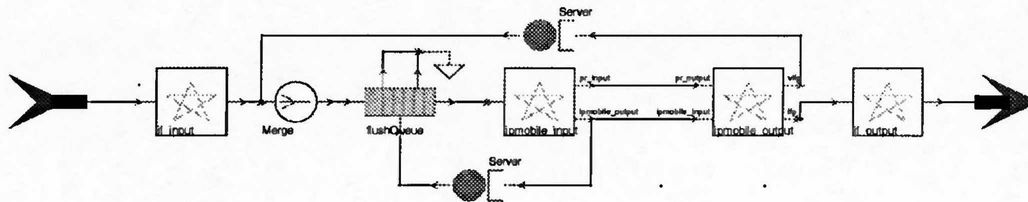


Figure 4.16: The HA galaxy internal components.

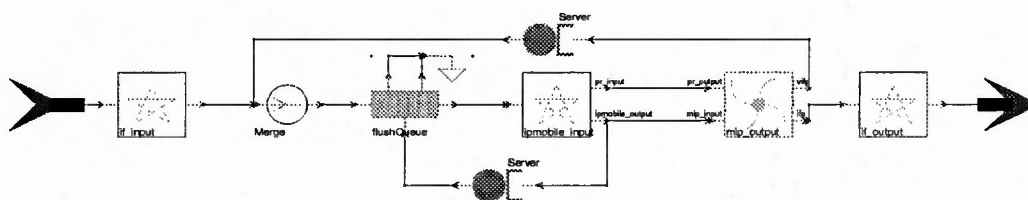


Figure 4.17: The FA galaxy internal components.

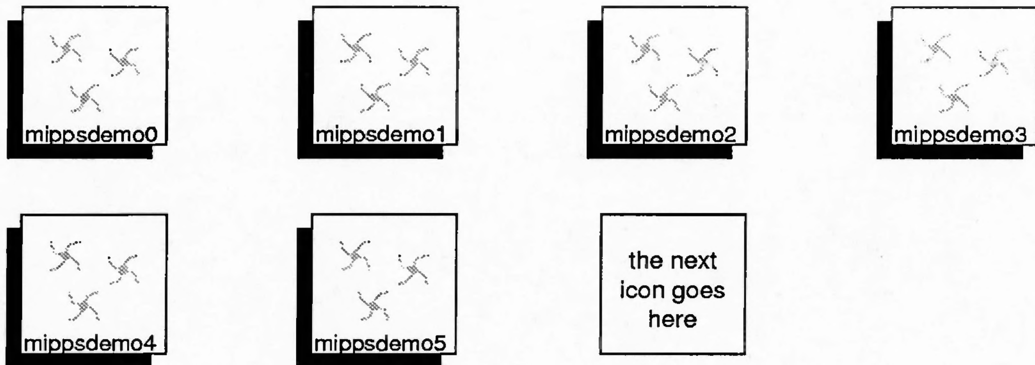


Figure 4.18: The demo universes in Mipps

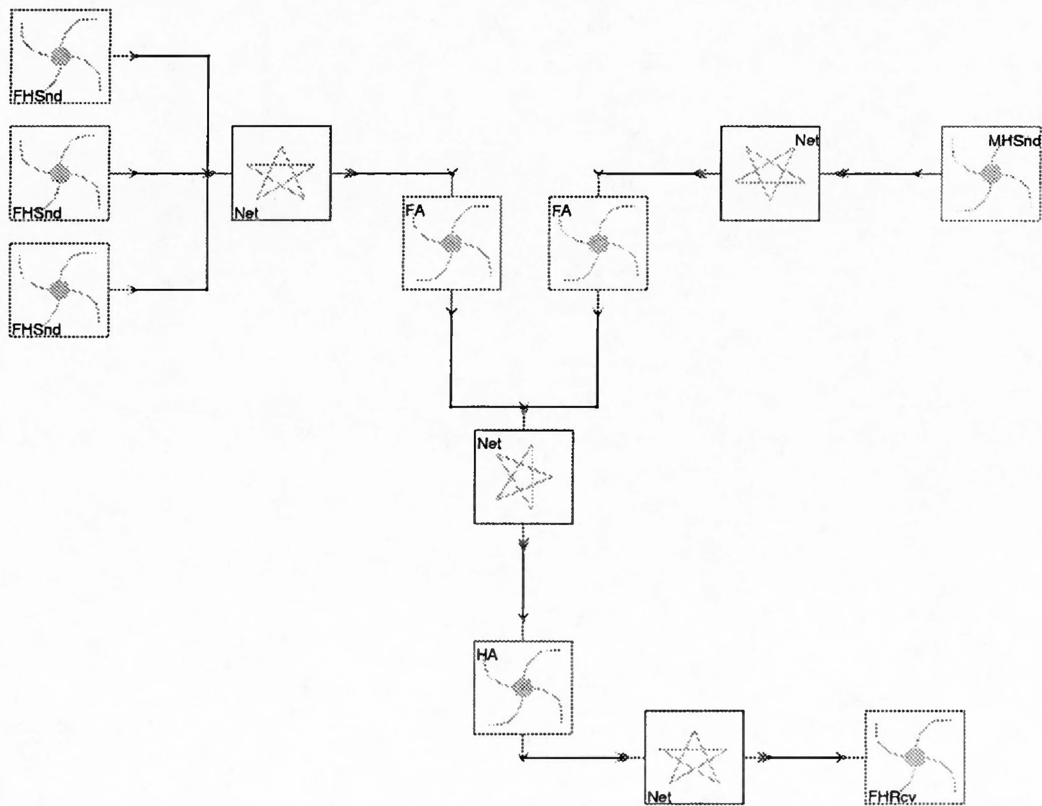


Figure 4.19: The MHs-HA interaction demo in Mipps

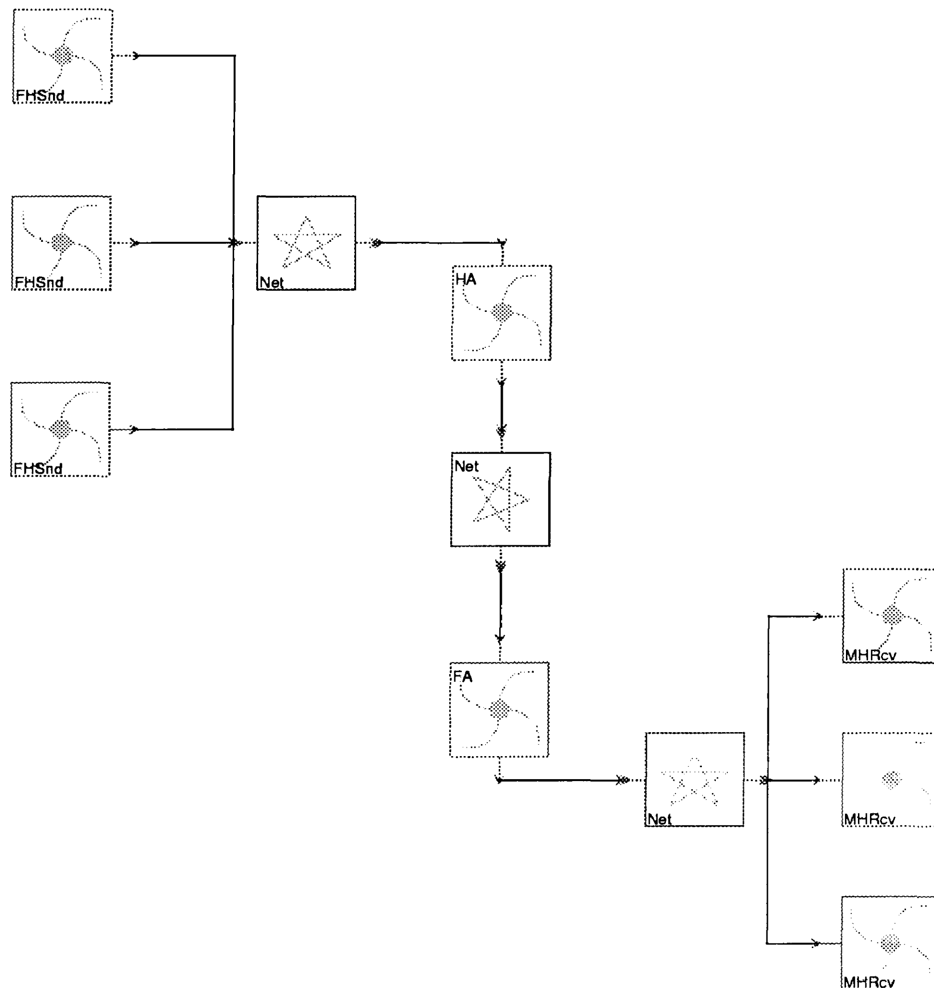


Figure 4.20: The FA-MHs interaction demo in Mipps

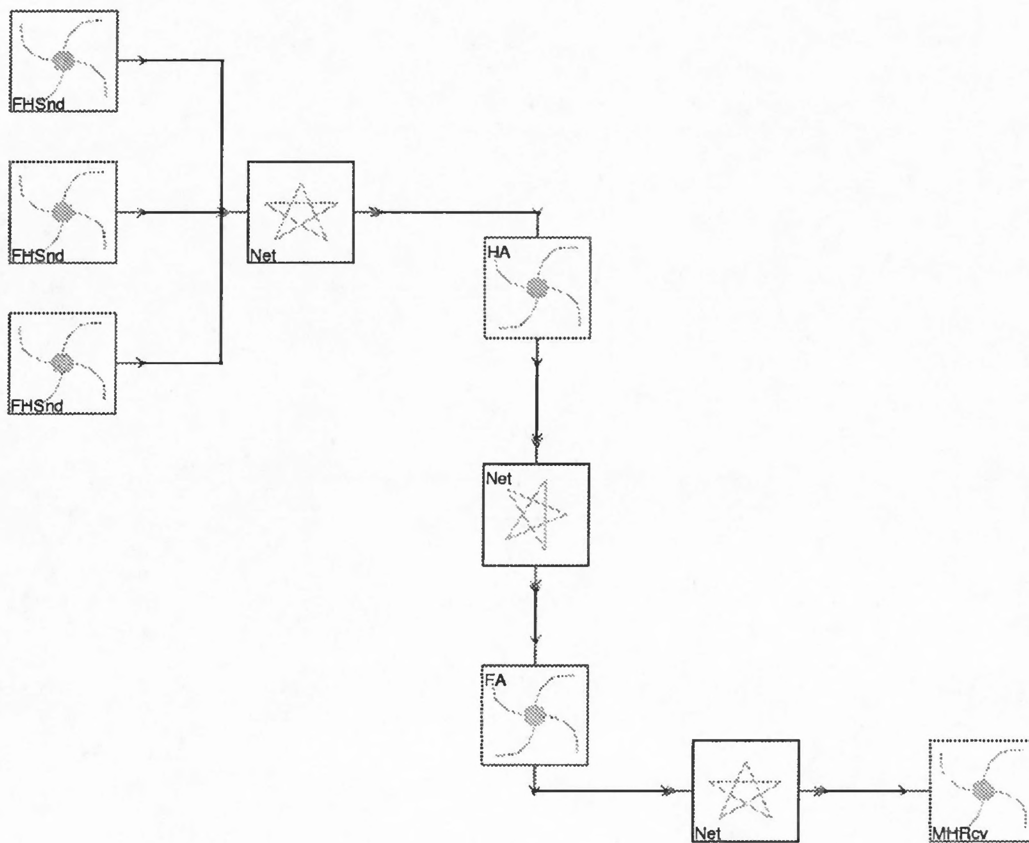


Figure 4.21: The FHs-HA interaction demo in Mipps

Chapter 5

Experimental Results

5.1 The MHs-HA interaction

5.1.1 Introduction

A mobile host registers whenever it detects that its point-of-attachement to the network has changed from one link to another or when its existing registration is due to expire. From the point of view of a home agent, the receipt of a registration request from a mobile host, the set of validity checks performed on behalf of a registration request, and the forwarding of a registration reply to a mobile host that is away from home are overheads at a home agent.

In this experiment, we consider how the time between consecutive registration requests and the number of mobile hosts affect the percentage of data packets, the percentage of registration request packets, and the percentage of dropped packets at a home agent.

5.1.2 The experimental plan

The experimental setup is as shown in figure 4.19. In here there are three **FHSnd** galaxies, one **PktRouter** galaxy, one **FA** galaxy, one **HA** galaxy, and one **FHRcv** galaxy. Also, there are four **Net** stars. Note that this figure represents a snapshot of this universe; as data collection progresses, the number of mobile hosts is changed from one to three.

Our data collection scheme consist of changing the rate at which a mobile host sends registration requests to its home agent, or the mean time between consecutive registration requests **meanTime**, per number of mobile hosts. As the experiment progresses, we collect the following statistics at the home agent:

- **ifi_ipackets** #packets received by the network interface
- **ifi_opackets** #packets sent by the network interface

- `ips_forward` #packets forwarded by the IP input module
- `ips_total` total #packets received by the IP input module
- `ipms_rpackets` #regsistration requests packets received by the IP input module

Knowing these statistics, obtaining the percentage of data packets, the percentage of registration requests packets, and the percentage of packets that are dropped is straightforward.

Shown below, are the routing and arp tables of all the galaxies in this experiment:

5.1.3 Observations

The following figures show the combined percentages of data, dropped, and registration request packets plotted against the time between consecutive mobile host registration requests for one, two, and three mobile hosts.

destinations	gateways	flags
25.0.0.0	10.0.0.4	UGH
20.0.0.0	10.0.0.4	UGH
10.0.0.4	10.0.0.x	UH

ARP entry
10.0.0.4

Table 5.1: The MHs-HA interaction: FHSnd routing and arp tables, where x = 1, 2, or 3. The rate at which these hosts send data or the mean time between packet sends is 1.0 sec. Also, the number of hops to the home agent is two.

destinations	gateways	flags
25.0.0.0	20.0.0.3	UGH
20.0.0.3	20.0.0.1	UH

ARP entry
20.0.0.3

Table 5.2: The MHs-HA interaction: PktRouter routing and arp tables. The size of the IP input queue is 25 packets. Also, the number of hops to the home agent is one.

destinations	gateways	flags
25.0.0.0	20.0.0.3	UGH
20.0.0.3	20.0.0.2	UH

ARP entry
20.0.0.3

Table 5.3: The MHs-HA interaction: FA routing and arp tables. The size of the IP input queue is 25 packets. Also, the number of hops to the home agent is one.

destinations	gateways	flags
25.0.0.2	25.0.0.1	UH

ARP entry
25.0.0.2

Table 5.4: The MHs-HA interaction: HA routing and arp tables. The size of the IP input queue is 25 packets.

destinations	gateways	flags
25.0.0.0	15.0.0.2	UGH
20.0.0.0	15.0.0.2	UGH
15.0.0.2	15.0.0.y	UH

ARP entry
15.0.0.2

Table 5.5: The MHs-HA interaction: MHSnd routing and arp tables, where y = 1, 2, or 3. The rate at which these hosts send registration requests or the mean time between registration requests was varied from one to twelve. Also, the number of hops to the home agent is two.

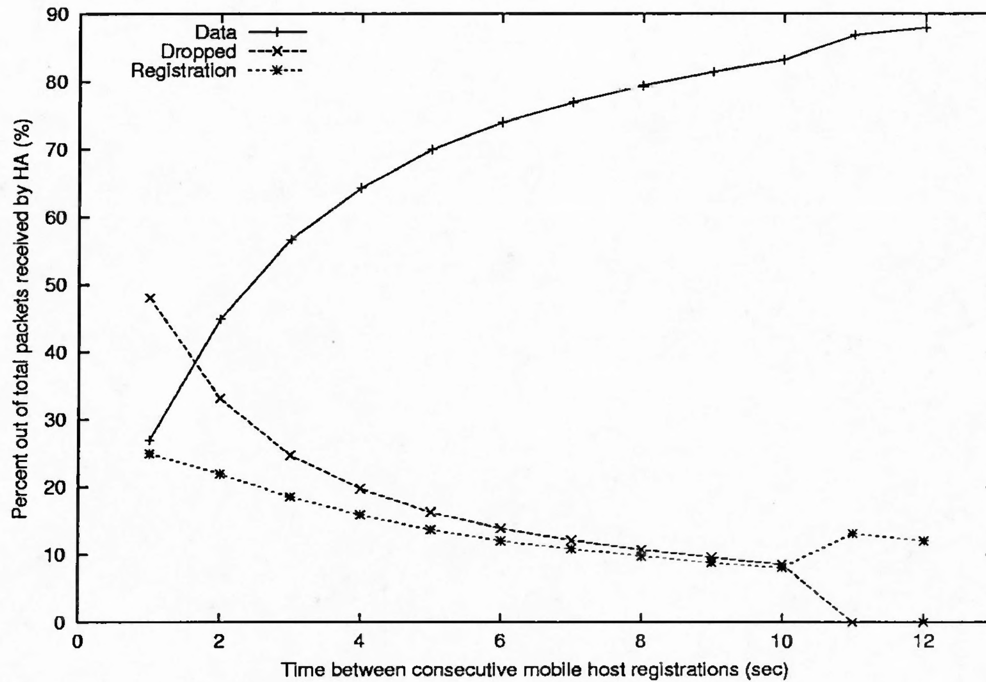


Figure 5.1: The MHs-HA interaction: one MH

The above figure shows the data, dropped, and registration packets plotted as percentages of the total number of packets received by the home agent versus the time between consecutive mobile host registration requests. For this data set, the number of mobile host sending registration requests is one.

The above figure shows the data, dropped, and registration packets plotted as percentages of the total number of packets received by the home agent versus the time between consecutive mobile host registration requests. For this data set, the number of mobile host sending registration requests is two.

The above figure shows the data, dropped, and registration packets plotted as percentages of the total number of packets received by the home agent versus the time between consecutive mobile host registration requests. For this data set, the number of mobile host sending registration requests is three.

5.1.4 Data analysis

Figure 5.1 shows the dependency of the throughput, the number of packets dropped, and the number of registration requests at a HA - for the case of one MH that is away from home - on the rate of registration request of this MH. We can make several conclusions from this graph. If we are trying to minimize the overhead at a HA due to registration requests, then an appropriate rate of registration requests by a MH that is away from home is near one registration request every 10 sec. On the other hand, if we are trying to maximize the throughput at a HA, then an appropriate rate of registration requests by a MH that is away from home is near one registration request every 12 sec.

Figure 5.2 shows that the number of registration requests at a HA, relative to the total number of packets received by the latter, reaches a maximum at a mean time between registration requests of 3 sec. This is due to a decrease in the rate of registration requests from one registration every 1 or 2 sec to one registration every 3 sec, and to an increase in the number of MHs from one to two. This change in the rate (number of MHs) increases the probability that a registration request is handled by the HA instead of being dropped by the latter due to overflow of its IP input queue. Also, for the case of two MHs that are away from home, minimizing the overhead at a HA due to registration requests and maximizing the throughput at a HA occur near a rate of one registration every 12 sec.

Similarly, this figure 5.3 shows that the number of registration requests at a HA, relative to the total number of packets received by the latter, reaches a maximum at the rates of one registration request every 4 and 5 sec. The conclusion for this behavior is the same as that for the case of two mobile hosts. Also, a rate of approximately one registration request every 12 sec will minimize the registration overhead and will maximize the throughput at a HA.

5.2 The FHs-HA interaction

5.2.1 Introduction

The required method of encapsulation in Mobile IP is "IP in IP" encapsulation . In this method, to tunnel an IP packet, a new IP header is wrapped around the existing packet; the source address in the new IP header is set to the address of the node tunneling the packet (the home agent), and the destination address is set to the mobile host's care of address. This type of encapsulation may be used for tunneling any packet, but the overhead for this method is the addition of an entire new IP header (20 bytes and 60 bytes with options) to the packet.

In this experiment, we consider how the size of the IP input queue and the rate at which fixed hosts send data affect the percentage of dropped packets, the percentage of encapsulated packets, and the percentage throughput at a home agent.

5.2.2 The experimental plan

The experimental setup is as shown in figure 4.21. In here there are three `FHSnd` galaxies, one `HA` galaxy, one `FA` galaxy, and one `MHRcv` galaxy. Also,

there are three **Net** stars; the first separates the fixed host and the home agent; the second separates the home agent and foreign agent; and the third separates the foreign agent and the mobile host.

Our data collection scheme consist of sending data packets to the home agent as we vary the size of the IP input queue. The rate of the data sources, or the mean time between packet sends **meanTime**, is kept constant as the size of the IP queue input takes values in the set {1, ..., 10, 15, 20, 25}. After collecting all the necesary data values, this rate is modified and the process is repeated. We collect data for a mean time between packet sends of 1.0, 3.0, and 5.0 sec. As the experiment progresses, we record the following statistics at the home agent:

- **ifi_ipackets** #packets received by the network interface
- **ifi_opackets** #packets sent by the network interface
- **ips_forward** #packets forwarded by the IP input module
- **ips_total** total #packets received by the IP input module
- **ipms_epackets** #packets encapsulated by the IP output module

Knowing these statistics, obtaining the percentage of dropped packets,

the percentage of encapsulated packets, and the percentage of throughput packets is straightforward.

Shown below, are the routing and arp tables of all the galaxies in this experiment:

5.2.3 Observations

The following figures show the combined percentages of dropped, encapsulated, and throughput packets plotted against the size of the IP input queue at a home agent for a mean time between packet sends of 1.0, 3.0, and 5.0 sec.

5.2.4 Data analysis

Figures 5.4 and 5.5 show that the overhead due to packet encapsulation at a HA for a rate of packet sends of one packet every 1 or 3 sec is too high for the HA to handle; increasing the IP input queue at the HA does not reduce the time spent by the HA encapsulating packets destined for mobile hosts that are away from home. To the contrary, increasing this buffer actually increases the chances that several packets from the network interface are placed ahead of the last encapsulated packet, since encapsulated packets have

to be rehandled by the routing software. This in turn, increases the chances of this encapsulated packet from being dropped due to buffer overflow.

On the other hand, figure 5.6 shows that a rate of packet sends of one packet every 5 sec does not cause such an overhead due to packet encapsulation at the HA as in the previous two cases. Also, to minimize the overhead due to packet encapsulation and to maximize the throughput at a HA, the latter should have approximately an IP input queue of size 25 and the rate of incoming packets from the network interface should not be faster than one packet every 5 sec.

5.3 The FA-MHs interaction

5.3.1 Introduction

As mobility agents, home agents, foreign agents, or both, must periodically multicast or broadcast ICMP router advertisements to each link on which a host is configured to perform as a home agent, foreign agent, or both. These advertisements are used by a mobile host that is connected to such a link to determine whether any mobility agents are present and, if so, there respective identities (IP addresses) and capabilities.

In this experiment, we consider how the rate at which a foreign agent sends foreign agent advertisements affects the downlink traffic at the foreign network and the percentage of data packets, to see what an appropriate value for the rate of foreign agent's advertisements should be.

5.3.2 The experimental plan

The experimental setup is as shown in figure 4.20. In here there are three **FHSnd** galaxies, one **PktRouter** galaxy, one **FA** galaxy, and one MHRcv galaxy. Also, there are three **Net** stars. Note that this figure represents a snapshot of this universe; as data collection progresses, the number of mobile hosts is changed from one to three.

Our data collection scheme consist of changing the rate at which the foreign agent sends agents advertisements on its network, or the mean time between consecutive agent advertisements **meanTime**, per number of mobile hosts. As the experiment progresses, we collect the following statistics at the foreign agent:

- **ifi_ipackets** #packets received by the network interface
- **ifi_opackets** #packets sent by the network interface

- `ips_forward` #packets forwarded by the IP input module
- `ips_total` total #packets received by the IP input module
- `ipms_mpackets` #agent advertisement packets multicast by the foreign agent

Knowing these statistics, obtaining the percentage of data packets and the percentage of foreign agent advertisements packets that are sent on the downlink is straightforward.

Shown below, are the routing and arp tables of all the galaxies in this experiment:

5.3.3 Observations

The following figures show the combined percentages of data or foreign agent advertsiment packets sent on the downlink, plotted against the time between consecutive agent advertisements for one, two, and three mobile hosts.

5.3.4 Data analysis

Figures 5.7 and 5.8 show the dependency of the number of foreign agent advertisements and the number of data packets on the downlink (from the

foreign agent to the hosts on its network) on the rate of foreign agent advertisements and the number of mobile hosts to which these agent advertisements are multicast.

As one would expect, the number of agent advertisements (data packets) increases (decreases) as the number of MHs to which a FA multicasts these agent advertisements increases irregardless of the rate at which these advertisements are sent on the downlink. Obviously, this increase (decrease) is worse at higher rates (one agent advertisement every 1 sec) than at lower rates (one agent advertisement every 13 sec).

Therefore, if maximizing the throughput and minimizing the number of agent advertisements on the donwlink are the primary objectives of a network or system administrator, then a foreign agent should be configured to send agent advertisements near a rate of one agent advertisement every 13 sec. If on the other hand, maximizing both the throughput and the number of agent advertisments on the downlink are equally important, then a foreign agent should be configured to send agent advertisements near a rate of one agent advertisement every 4 sec.

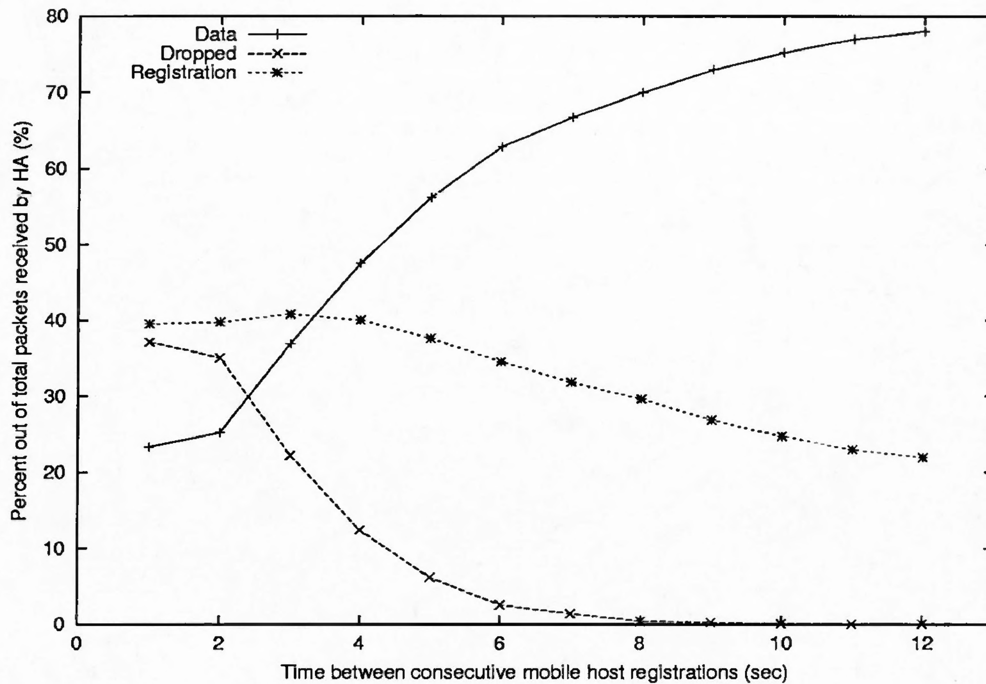


Figure 5.2: The MHs-HA interaction: two MHs

destinations	gateways	flags
10.0.0.4	10.0.0.x	UH
15.0.0.0	10.0.0.4	UGH
20.0.0.0	10.0.0.4	UGH

ARP entry
10.0.0.4

Table 5.6: The FHs-HA interaction: FHSnd routing and arp tables, where x = 1, 2, or 3. The rate at which these hosts send data or the mean time between packet sends is varied between 1.0, 3.0, and 5.0 sec. Also, the number of hops to the home agent is one.

destinations	gateways	flags
15.0.0.2	15.0.0.1	UH
20.0.0.0	15.0.0.2	UGH

ARP entry
15.0.0.2

Table 5.7: The FHs-HA interaction: HA routing and arp tables. The size of the IP input queue takes values in the set {1, ..., 10, 15, 20, 25}

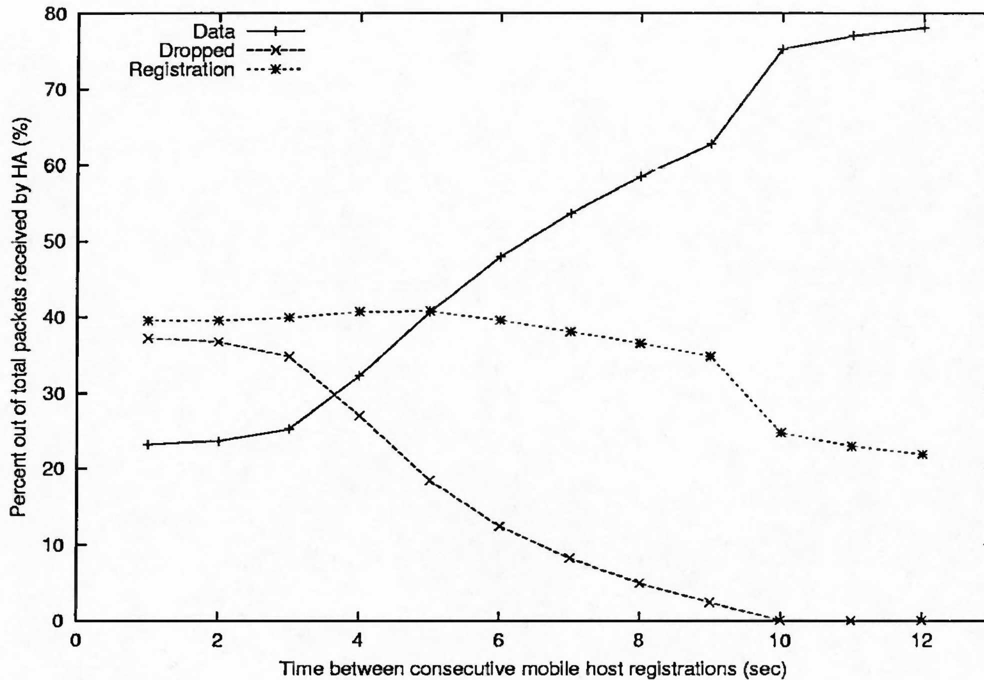


Figure 5.3: The MHs-HA interaction: three MHs

destinations	gateways	flags
20.0.0.2	20.0.0.1	UH
20.0.0.0	20.0.0.1	UH

ARP entry
20.0.0.2

Table 5.8: The FHs-HA interaction: FA routing and arp tables. The size of the IP input queue is 25 packets. Also, the number of hops to the home agent is one.

destinations	gateways	flags
20.0.0.1	20.0.0.2	UH
0.0.0.0	20.0.0.1	UGH

ARP entry
20.0.0.1

Table 5.9: The FHs-HA interaction: MHRcv routing tables

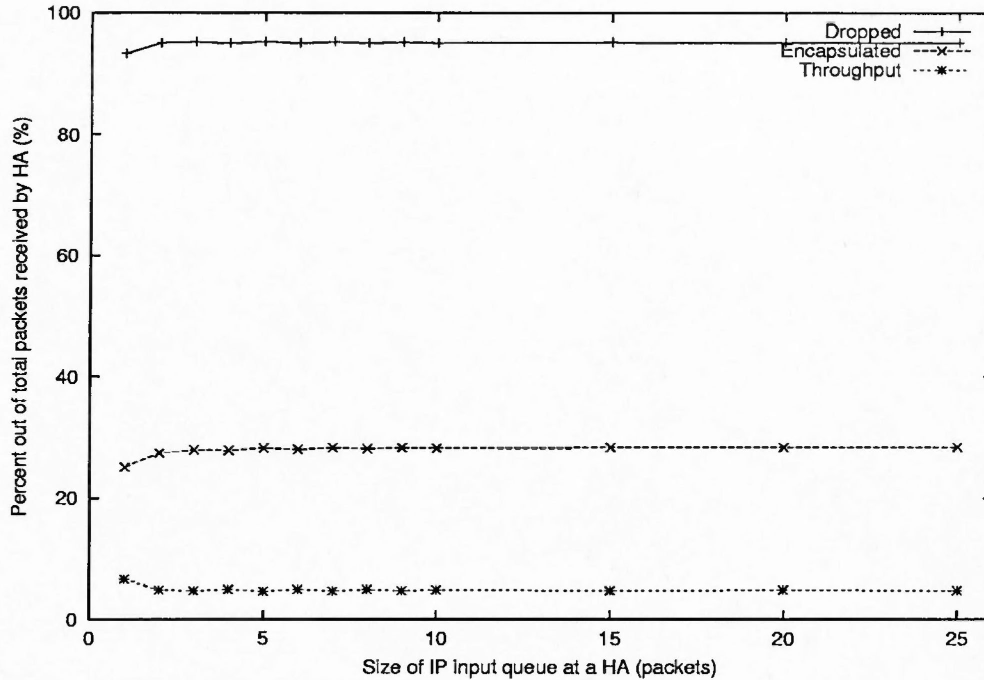


Figure 5.4: This figure shows the dropped, encapsulated, and throughput packets plotted as percentages of the total number of packets received by the home agent versus the size of the IP input queue. The mean time between packet sends is 1.0 sec.

destinations	gateways	flags
10.0.0.4	10.0.0.x	UH
15.0.0.0	10.0.0.4	UGH
20.0.0.0	10.0.0.4	UGH

ARP entry
10.0.0.4

Table 5.10: The FA-MHs interaction: FHSnd routing and arp tables, where x = 1, 2, or 3. The rate at which these hosts send data or the mean time between packet sends is 1.0 sec. Also, the number of hops to the foreing agent is two.

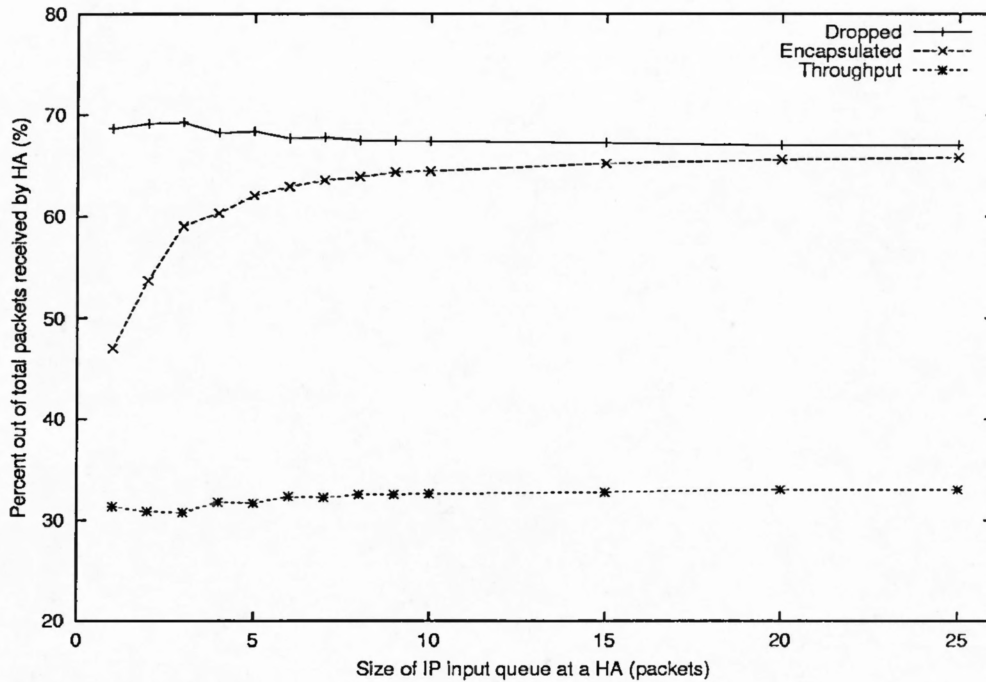


Figure 5.5: This figure shows the dropped, encapsulated, and throughput packets plotted as percentages of the total number of packets received by the home agent versus the size of the IP input queue. The mean time between packet sends is 3.0 sec.

destinations	gateways	flags
15.0.0.0	15.0.0.1	UH
20.0.0.0	15.0.0.2	UGH

ARP entry
15.0.0.2

Table 5.11: The FA-MHs interaction: PktRouter routing and arp tables. The size of the IP input queue is 25 packets. Also, the number of hops to the home agent is one.

destinations	gateways	flags
20.0.0.0	20.0.0.4	UH

ARP entry
20.0.0.y

Table 5.12: The FA-MHs interaction: FA routing and arp tables, where y = 1, 2, or 3. The size of the IP input queue is 25 packets. Also, the number of hops to the foreign agent is one.

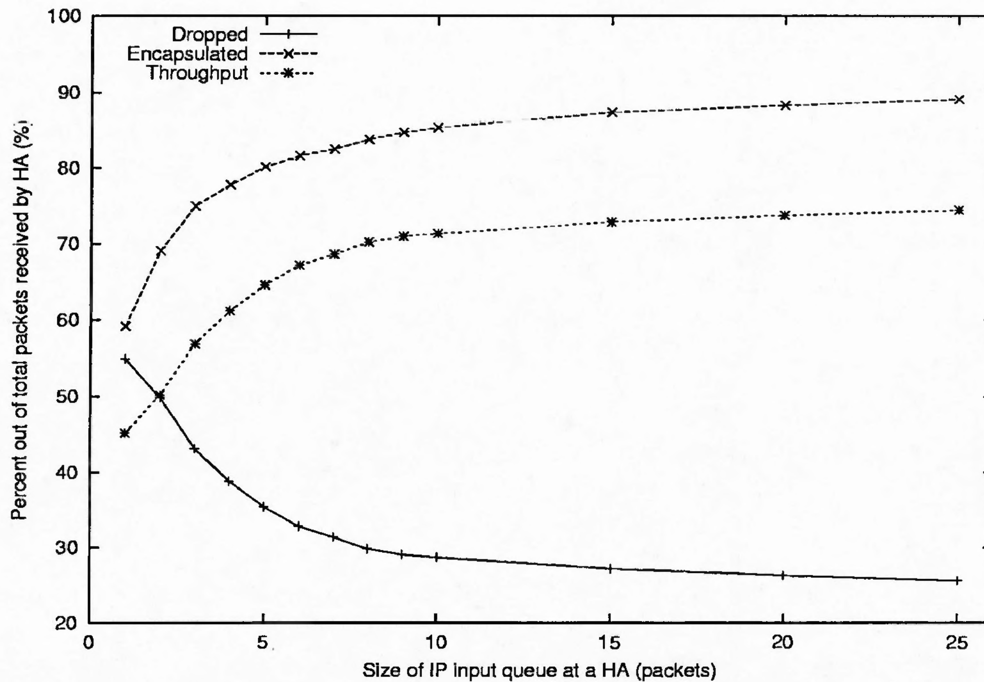


Figure 5.6: This figure shows the dropped, encapsulated, and throughput packets plotted as percentages of the total number of packets received by the home agent versus the size of the IP input queue. The mean time between packet sends is 5.0 sec.

destinations	gateways	flags
20.0.0.0	20.0.0.z	UH
0.0.0.0	20.0.0.4	UGH

ARP entry
20.0.0.z

Table 5.13: The FA-MHs interaction: MHRcv routing tables, where z = 1, 2, or 3.

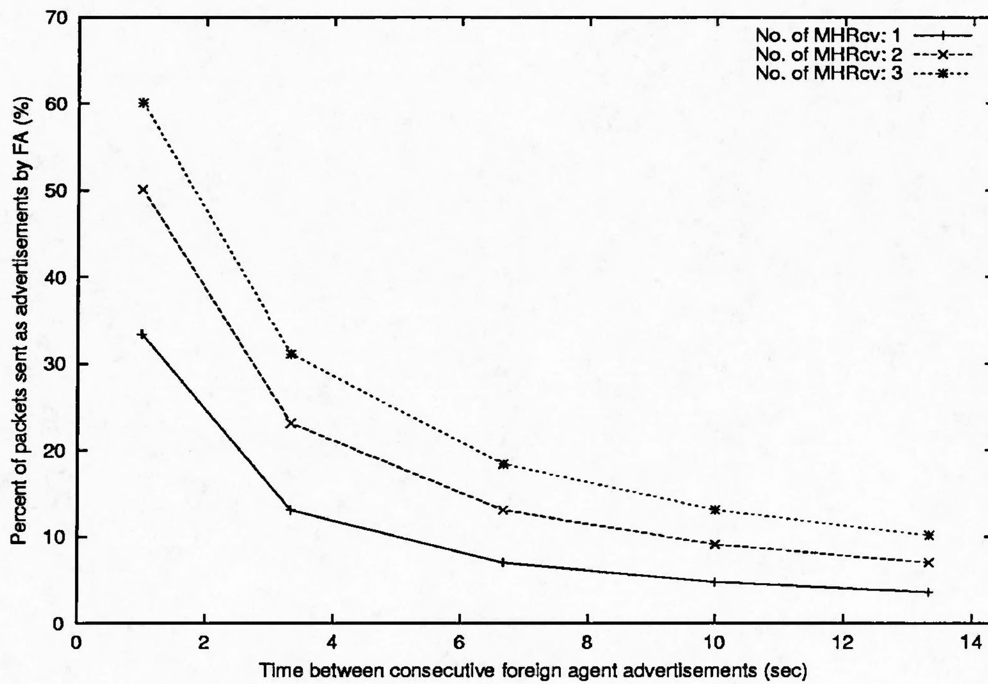


Figure 5.7: This figure shows the percentage of packets on the downlink that are agent advertisements versus the time between consecutive foreign agent advertisements.

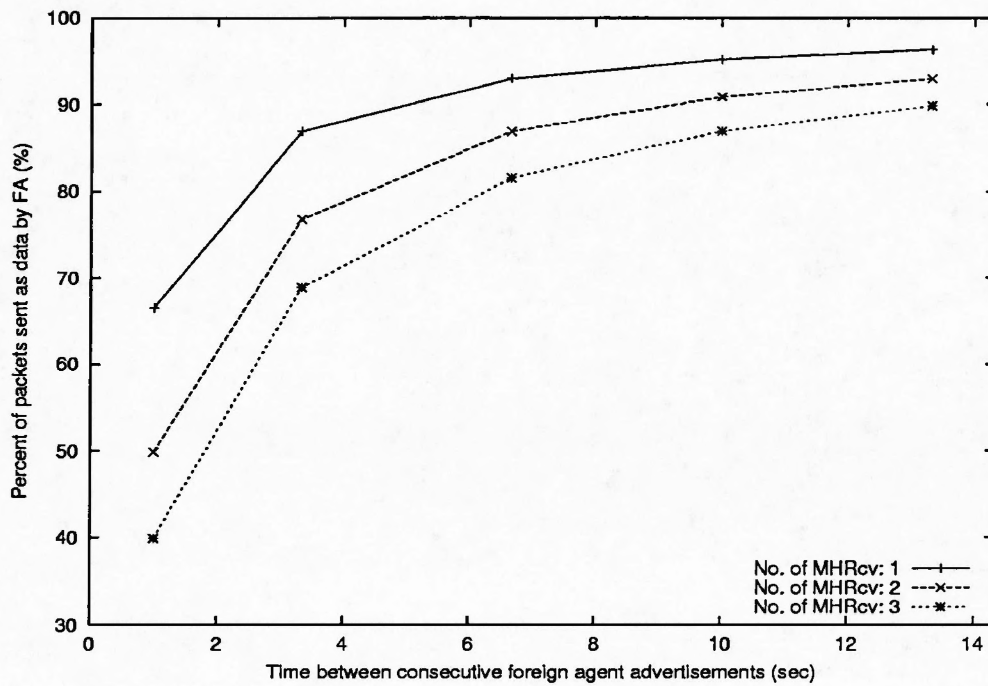


Figure 5.8: This figure shows the percentage of packets on the downlink that are data packets versus the time between consecutive foreign agent advertisements.

Chapter 6

Summary and Conclusions

In this thesis, we studied the IETF Mobile IP protocol, examined three different protocols for mobile internetworking, presented an overview of an object-oriented framework for simulating and prototyping heterogeneous systems, and then designed, built, and utilized a system that allowed us to make some performance measures on the IETF Mobile IP protocol. In this last chapter, we shall summarize how the IETF Mobile IP protocol works, and provide some performance measures on some of the functions of this protocol. Following that, we shall briefly discuss future directions, both in terms of extensions and additions to this work, and in terms of complementary research that needs to be done. Finally, we shall summarize the contributions

of this thesis.

6.1 Summary

The IETF Mobile IP protocol defines three functional entities where its mobility protocols must be implemented; these are, the mobile node, the home agent, and the foreign agent. Without an annoying number of forward and backward references, here is, at a very high level, how Mobile IP works:

1. Home agents and foreign agents advertise their presence on any attached links by periodically multicasting or broadcasting special Mobile IP messages called *Agent Advertisements.*

2. Mobile hosts listen to these *Agent Advertisements* and examine their contents to determine whether they are connected to their home network or a foreign network.

3. A mobile host connected to a foreign network acquires a care-of-address. A foreign agent care-of-address can be read from one of the fields within the foreign agent's *Agent Advertisement.* A collocated care-of-address must be adquired by some assignment procedure, such as the *Dynamic*

Host Configuration Protocol, the *Point-to-Point Protocol's IP Control Protocol* , or manual configuration.

4. The mobile host *Registers* the care-of-address acquired in step 3 with its home agent, using a message-exchange defined by Mobile IP.

5. The home agent or some other router on the home network advertises reachability to the network-prefix of the mobile host's home address, thus attracting packets that are destined to the mobile host's home address. The home agent intercepts these packets and tunnels them to the care-of-address that the mobile host registered in step 4.

6. At the care-of-address the original packet is extracted from the tunnel and then delivered to the mobile host.

7. Packets sent by the mobile host are routed directly to their destination, without any need for tunneling.

The conclusions to draw from our measurements of the performance measures of some of the functions of this protocol are:

- A rate of approximately one registration request every 12 sec will minimize the registration overhead and will maximize the throughput at a HA.

- To maximize the throughput and minimize the number of agent advertisements on the donwlink a FA should be configured to send agent advertisements near a rate of one agent advertisement every 13 sec.

- To maximize both the throughput and the number of agent advertisments on the downlink a FA should be configured to send agent advertisements near a rate of one agent advertisement every 4 sec.

- The Mobile-IP protocol should have its own input queue for routing encapsulated packets.

- The the Mobile-IP protocol input queue should be approximately 25 packets in size.

6.2 Future Work

The work presented in this thesis offers some new results on the performance of some of the functions of the IETF Mobile IP protocol. Still, much work remains to be done to truly understand the implications of this new protocol standard within the context of the Internet. Further work on this protocol could be centered around the following issues:

- What is the overhead – in terms of traffic – for supporting mobility?
- What is the overall added traffic in the fixed net for supporting mobility?
- How long should the registration of a mobile host's location last?
- How does noise on the transport medium affect the protocol?

6.3 Value of the Work

The value of the work presented in this thesis to the networking community is both new and practical.

- It provided upper bounds on the time between consecutive mobile host's reregistration request.
- It provided upper bounds on the time between consecutive foreign agent's agent advertisements, when these advertisements are multicast to three or more mobile hosts.
- It pointed out that the Mobile-IP protocol should have its own input queue for routing encapsulated packets.

- It showed why the Mobile-IP protocol should have its own input queue for routing encapsulated packets.

- It provided a lower bound on the size of the Mobile-IP protocol input queue.

Finally,

- It produced a working system, Mipps, that can be used for obtaining performance measures of the other functions of the IETF Mobile IP protocol.

Appendix A

Definition of classes in Mipps

A.1 The network classes definition

A.1.1 The Net class definition

```
defstar {

    name { Net }

    domain { DE }

    version {%W% %G%}

    author { Efren A. Serra }

    location { $PTOLEMY/mipps/src/stars }

    desc {
```

```
This star reads in input NetworkCells from multiple input
sources and routes them to the appropriate output.
    }

    defstate {
        name { linktable }
        type { intarray }
        default { "0" }
        desc { frame physical address to output port number map }
    }

    inmulti {
        name { input }
        type { message }
    }

    outmulti {
        name { output }
        type { message }
```

```
}

hinclude { "NetworkCell.h" }

setup {
    for (int i = 0; i < linktable.size(); i++) {
        if ((linktable[i] < 0) ||
            (linktable[i] >= output.numberPorts())) {
            Error::abortRun(*this,
                            "Routing table entry out of legal range.");
            return;
        }
    }

}

go {
    /* Pass timestamp without change */
    completionTime = arrivalTime;
```

```
//INPUT
        InDEMPHIter inIter(input);
        InDEPort* inPtr;
        while ((inPtr = inIter++) != 0) {
            while (inPtr->dataNew) {
                /* Get the frame on the _input_ port */
                Envelope inEnvlp;
                inPtr->get().getMessage(inEnvlp);
                TYPE_CHECK(inEnvlp, "NetworkCell");
                const NetworkCell* cellPtr =
                    (const NetworkCell*) inEnvlp.myData();
                const int dest = cellPtr->dest();
                if ((dest < 0) || (dest >= linktable.size())) {
                    Error::abortRun(*this, "Cell with illegal destination.");
                    return;
                }
                const int port = linktable[dest];
//OUTPUT
                OutDEMPHIter outIter(output);
```

```
                OutDEPort* outPtr = outIter++;
                for (int i = port; i > 0; i--) { outPtr = outIter++; }
                /* Put it on the _output_ port */
                outPtr->put(completionTime) << inEnvlp;
                /* Get next simulation event */
                inPtr->getSimulEvent();
            }
        }
    }
} //end defstar { Net }
```

A.2 The network interface classes definition

A.2.1 The if_input class definition

```
defstar {
    name { if_input }
    domain { DE }
    version {%W% %G%}
    author { Efren A. Serra }
```

```
    location { $PTOLEMY/mipps/src/stars }
    desc {
The network interface abstraction defines the interface between
protocol software in the operating system and the underlying
hardware.  It hides hardware details and allows protocol software
to interact with a variety of network hardware using the same
data structures.
    }

    defstate {
        name { ifa_addr }
        type { string }
        default { "127.0.0.1" }
        desc { address of interface }
    }

    defstate {
        name { ofilename }
        type { string }
```

```
    default { "" }
    desc { name of file to save output }
}

input {
    name { if_rcv }
    type { message }
    desc { interface input queue }
}

output {
    name { ipintrq }
    type { message }
    desc { IP input queue }
}

hinclude {
    "NetworkCell.h",
    <iostream.h>,
```

```
        <fstream.h>
    }

    protected {
        char* if_name;
        unsigned long int if_ipackets;
        unsigned long int if_ibytes;
    }

    inline method {
        name { net_input }
        access { protected }
        arglist { "(InDEPort& ifq, OutDEPort& ipq)" }
        type { void }
        code {
//INPUT
            /* Get packet from _if_rcv_ queue */
            if_ipackets++;
            Envelope inEnvlp;
```

```
            ifq.get().getMessage(inEnvlp);

            TYPE_CHECK(inEnvlp, "NetworkCell");

            /* Strip the physical header off */

            NetworkCell* cellPtr = (NetworkCell*) inEnvlp.writableCopy();

            Message* outMssg = cellPtr->writableData();

            if_ibytes += cellPtr->size();

            /* Free memory resources */

            LOG_DEL; delete cellPtr;

//OUTPUT

            /* Put it on the _ipintrq_ queue */

            Envelope outEnvlp(*outMssg);

            ipq.put(completionTime) << outEnvlp;

        }

    }


    setup {

        if_name = "le";

        if_ipackets = if_ibytes = 0;

    }
```

```
go {
    /* Pass time stamp without change */
    completionTime = arrivalTime;
    /* Process new data on the _if_rcv_ port */
    if (if_rcv.dataNew) {
        net_input(if_rcv, ipintrq);
    }
}

wrapup {
    ofstream ofile(ofilename);
    if (!ofile) {
        Error::abortRun(*this, "cannot open output file");
    }
    ofile << if_name << '\n'
          << "#packets received on interface "
          << if_ipackets << '\n'
          << "#bytes received "
```

```
            << if_ibytes << endl;
        ofile.close();
    }
} //end defstar { if_input }
```

A.2.2 The if_ouput class definition

```
defstar {
    name { if_output }
    domain { DE }
    version {%W% %G%}
    author { Efren A. Serra }
    location { $PTOLEMY/mipps/src/stars }
    desc {
The network interface abstraction defines the interface between
protocol software in the operating system and the underlying
hardware.  It hides hardware details and allows protocol software
to interact with a variety of network hardware using the same
data structures.
    }
```

```
defstate {
    name { ifa_addr }
    type { string }
    default { "127.0.0.1" }
    desc { address of interface }
}

defstate {
    name { arptable }
    type { stringarray }
    default { "127.0.0.1" }
    desc { IP address to physical address map }
}

defstate {
    name { ofilename }
    type { string }
    default { "" }
```

```
    desc { name of file to save output }
}

input {
    name { ip_output }
    type { message }
    desc { IP output function }
}

output {
    name { if_snd }
    type { message }
    desc { interface output queue }
}

hinclude {
    "NetworkCell.h",
    <fstream.h>,
    <iostream.h>,
```

```
    <string.h>,
    <sys/types.h>,
    <sys/socket.h>,
    <netinet/in.h>,
    <arpa/inet.h>
}

protected {
    char* if_name;
    unsigned long int if_opackets;
    unsigned long int if_obytes;
}

inline method {
    name { net_output }
    access { protected }
    arglist { "(OutDEPort& ifq)" }
    type { void }
    code {
```

```
            /* Get the datagram on the _ip_output_ port */
            Envelope inEnvlp;
            ip_output.get().getMessage(inEnvlp);
            TYPE_CHECK(inEnvlp, "NetworkCell");
            NetworkCell* cellPtr = (NetworkCell*) inEnvlp.writableCopy();
            /* Get IP address of next destination */
            struct in_addr ip_dst;
            ip_dst.s_addr = cellPtr->dest();
            char* ip_dest = inet_ntoa(ip_dst);
            /* Translate IP address to physical address */
            for (int i = 0; i < arptable.size(); i++) {
                if (strcmp(ip_dest, arptable[i]) == 0) {
                    /* Matching IP address found */
#ifdef DEBUG
cout << "if_output: " << ip_dest << " ---> " << i << endl;
#endif
                    if_opackets++;
                    int mssgSize = cellPtr->size();
                    if_obytes += mssgSize;
```

```
                    Message* toLoad = cellPtr->writableData();
                    LOG_NEW; NetworkCell* shipItOut =
                        new NetworkCell(*toLoad,
                                        int(1),
                                        int(i),
                                        int(mssgSize),
                                        0,
                                        float(completionTime),
                                        0);
//OUTPUT
                    Envelope outEnvlp(*shipItOut);
                    /* Send frame on the _if_snd_ port */
                    ifq.put(completionTime) << outEnvlp;
                    break;
                }
            }
            /* Free memory resources */
            LOG_DEL; delete cellPtr;
        }
```

```
}


setup {

    if_name = "le";

    if_opackets = if_obytes = 0;

}


go {

    /* Pass time stamp without change */

    completionTime = arrivalTime;

    /* Process new data on _ip_output_ port */

    if (ip_output.dataNew) {

        net_output(if_snd);

    }

}


wrapup {

    ofstream ofile(ofilename);

    if (!ofile) {
```

```
            Error::abortRun(*this, "cannot open output file");
        }
        ofile << if_name << '\n'
              << "#packets sent on interface "
              << if_opackets << '\n'
              << "#bytes sent "
              << if_obytes << endl;
        ofile.close();
    }
} //end defstar { if_output }
```

A.3 The IP classes definition

A.3.1 The ip_input class definition

```
defstar {
    name { ip_input }
    domain { DE }
    version {%W% %G%}
    author { Efren A. Serra }
```

```
location { $PTOLEMY/mipps/src/stars }

desc {

}


defstate {

    name { in_ifaddrs }

    type { stringarray }

    default { "127.0.0.1" }

    desc { IP address list }

}


defstate {

    name { ipforwarding }

    type { int }

    default { 1 }

    desc { should the system forward IP packets? }

}


defstate {
```

```
    name { ofilename }
    type { string }
    default { "" }
    desc { name of file to save output }
}

inmulti {
    name { ipintrq }
    type { message }
    desc { IP input queue }
}

output {
    name { pr_input }
    type { message }
    desc { protocol input queue }
}

output {
```

```
    name { ip_output }
    type { message }
    desc { IP output queue }
}

hinclude {
    "NetworkCell.h",
    <fstream.h>,
    <iostream.h>,
    <string.h>,
    <sys/types.h>,
    <sys/socket.h>,
    <netinet/in.h>,
    <arpa/inet.h>
}

protected {
    unsigned long int ips_cantforward;
    unsigned long int ips_delivered;
```

```
    unsigned long int ips_forward;
    unsigned long int ips_total;
    unsigned long int ips_odropped;
}

inline method {
    name { ip_forward }
    access { protected }
    arglist { "(Envelope& envlp, int srcrt)" }
    type { void }
    code {
        ip_output.put(completionTime) << envlp;
        ips_forward++;
    }
}

inline method {
    name { ipintr }
    access { protected }
```

```
        arglist { "(void)" }

        type { void }

        code {

            /* Pass timestamp without change */

            completionTime = arrivalTime;

//INPUT

            InDEMPHIter inIter(ipintrq);

            InDEPort* inPtr;

            while ((inPtr = inIter++) != 0) {

                while (inPtr->dataNew) {

                    /* Get the packet on the _ipintrq_ port */

                    ips_total++;

                    Envelope inEnvlp;

                    inPtr->get().getMessage(inEnvlp);

                    TYPE_CHECK(inEnvlp, "NetworkCell");

                    const NetworkCell* cellPtr =

                        (const NetworkCell*) inEnvlp.myData();

                    /* Get the packet final destination */

                    struct in_addr ip_dst;
```

```
                ip_dst.s_addr = cellPtr->dest();
                char* ip_dest = inet_ntoa(ip_dst);
//OUTPUT
                /* Check own IP addresses */
                for (int i = 0; i < in_ifaddrs.size(); i++) {
                    if (strcmp(ip_dest, in_ifaddrs[i]) == 0) {
                        /* Put packet on the _pr_input_ port */
                        ips_delivered++;
                        pr_input.put(completionTime) << inEnvlp;
                        goto next;
                    }
                }
//OUTPUT
                /* Not for us; forward if possible */
                if (ipforwarding == 0) {
                    ips_cantforward++;
                } else {
                    /* Forward packet on the _ip_output_ port */
#ifdef DEBUG
```

```
cout << "ip_input: forwarding to: " << ip_dest << endl;
#endif
                        ip_forward(inEnvlp, 0);
                    }
                    /* Get next simulation event */
        next:
                    inPtr->getSimulEvent();
                }
            }
        }
    }

    setup {
        ips_cantforward = 0;
        ips_delivered = 0;
        ips_forward = 0;
        ips_total = 0;
        ips_odropped = 0;
    }
```

```
go {
    ipintr();
}

wrapup {
    ofstream ofile(ofilename);
    if (!ofile) {
        Error::abortRun(*this, "cannot open output file");
    }
    ofile << "#datagrams delivered to upper level "
          << ips_delivered << '\n'
          << "#packets forwarded "
          << ips_forward << '\n'
          << "total #packtes received "
          << ips_total << '\n'
          << "#packtes dropped because of resource shortages "
          << ips_odropped << endl;
    ofile.close();
```

```
    }
} //end defstar { ip_input }
```

A.3.2 The ip_output class definition

```
defstar {
    name { ip_output }
    domain { DE }
    version {%W% %G%}
    author { Efren A. Serra }
    location { $PTOLEMY/mipps/src/stars }
    desc {
    }

    defstate {
        name { destination }
        type { stringarray }
        default { "127.0.0.1" }
        desc {}
    }
```

```
defstate {
    name { gateway }
    type { stringarray }
    default { "127.0.0.1" }
    desc {}
}

defstate {
    name { flag }
    type { stringarray }
    default { "UGH" }
    desc {}
}

defstate {
    name { out_ifaddrs }
    type { intarray }
    default { "0" }
```

```
    desc { list of interface addresses }
}

defstate {
    name { ofilename }
    type { string }
    default { "" }
    desc { name of file to save output }
}

input {
    name { pr_output }
    type { message }
}

input {
    name { ip_input }
    type { message }
    desc {}
```

```
}

outmulti {
    name { ifp }
    type { message }
    desc {}
}

hinclude {
    "NetworkCell.h",
    <fstream.h>,
    <iostream.h>,
    <string.h>,
    <sys/types.h>,
    <sys/socket.h>,
    <netinet/in.h>,
    <arpa/inet.h>
}
```

```
header {
    const int NET_MASK = 0xffffff00;
}

protected {
    unsigned long int ips_noroute;
    char* ro_dst;
}

inline method {
    name { if_output }
    access { protected }
    arglist { "(MultiOutDEPort& ifPtr, Envelope& envlp)" }
    type { void }
    code {
        const NetworkCell* cellPtr =
            (const NetworkCell*) envlp.myData();
        /* Get the packet final destination */
        struct in_addr ip_dst;
```

```
            struct in_addr net;
            ip_dst.s_addr = cellPtr->dest();
            net.s_addr = cellPtr->dest() & NET_MASK;
            char* ip_dest = strdup(inet_ntoa(ip_dst));
            char* ia_net = strdup(inet_ntoa(net));
#ifdef DEBUG
cout << "ip_output: dest: " << ip_dest << endl;
cout << "ip_output: net: " << ia_net << endl;
#endif
            /* Search the routing tables */
            for (int i = 0; i < destination.size(); i++) {
                if ((strcmp(ip_dest, destination[i]) == 0) &&
                    (strcmp("UH", flag[i]) == 0)) {
                    /* Direct route */
#ifdef DEBUG
cout << "ip_output: direct route; dest: " << ip_dest << endl;
#endif
                    ro_dst = destination[i];
                    int hostAddr = cellPtr->dest();
```

```
                        int mssgSize = cellPtr->size();

                        Message* toLoad = envlp.writableCopy();

                        LOG_NEW; NetworkCell* shipItOut =

                            new NetworkCell(*toLoad,

                                            int(1),

                                            int(hostAddr),

                                            int(mssgSize),

                                            0,

                                            float(completionTime),

                                            0);

                        Envelope outEnvlp(*shipItOut);

//OUTPUT

                        const int port = out_ifaddrs[i];

                        OutDEMPHIter outIter(ifPtr);

                        OutDEPort* outPtr = outIter++;

                        for (int j = port; j > 0; j--) { outPtr = outIter++; }

                        /* Put the packet on the _ifp_ port */

                        outPtr->put(completionTime) << outEnvlp;

                        break;
```

```
                }
                if ((strcmp(ia_net, destination[i]) == 0) &&
                    (strcmp("UGH", flag[i]) == 0)) {
                    /* Indirect route */
#ifdef DEBUG
cout << "ip_output: indirect route; next hop: " << gateway[i] << endl;
#endif
                    ro_dst = gateway[i];
                    int nxtHop = inet_addr(gateway[i]);
                    int mssgSize = cellPtr->size();
                    Message* toLoad = envlp.writableCopy();
                    LOG_NEW; NetworkCell* shipItOut =
                        new NetworkCell(*toLoad,
                                        int(1),
                                        int(nxtHop),
                                        int(mssgSize),
                                        0,
                                        float(completionTime),
                                        0);
```

```
                    Envelope outEnvlp(*shipItOut);
//OUTPUT
                    const int port = out_ifaddrs[i];
                    OutDEMPHIter outIter(ifPtr);
                    OutDEPort* outPtr = outIter++;
                    for (int j = port; j > 0; j--) { outPtr = outIter++; }
                    /* Put the packet on the _ifp_ port */
                    outPtr->put(completionTime) << outEnvlp;
                    break;
                }
            }
            if (ro_dst == 0) ips_noroute++;
            /* Free memory resources */
            free(ip_dest); free(ia_net);
        }
    }

    setup {
        ips_noroute = 0;
```

```
        ro_dst = 0;
    }

    go {
        Envelope inEnvlp;
        /* Pass time stamp without change */
        completionTime = arrivalTime;
//INPUT
        if (pr_output.dataNew) {
#ifdef DEBUG
cout << "ip_output: There is data from pr_output.\n";
#endif
            /* Get packet on the _pr_output_ port */
            pr_output.get().getMessage(inEnvlp);
            /* Check type of input packet */
            TYPE_CHECK(inEnvlp, "NetworkCell");
//OUTPUT
            if_output(ifp, inEnvlp);
        }
```

```
//INPUT
        if (ip_input.dataNew) {
#ifdef DEBUG
cout << "ip_output: There is data from ip_input.\n";
#endif
            /* Get packet on the _ip_input_ port */
            ip_input.get().getMessage(inEnvlp);
            /* Check type of input packet */
            TYPE_CHECK(inEnvlp, "NetworkCell");
//OUTPUT
            if_output(ifp, inEnvlp);
        }
    }

    wrapup {
        ofstream ofile(ofilename);
        if (!ofile) {
            Error::abortRun(*this, "cannot open output file");
        }
```

```
        ofile << "#packets discarded -- no route destination "
              << ips_noroute << endl;
        ofile.close();
    }
} //end defstar { ip_output }
```

A.4 The Mobile IP classes definition

A.4.1 The ipmobile_input class definition

```
defstar {
    name { ipmobile_input }
    domain { DE }
    version {%W% %G%}
    author { Efren A. Serra }
    location { $PTOLEMY/mipps/src/stars }
    desc {
    }

    defstate {
```

```
    name { in_ifaddrs }
    type { stringarray }
    default { "127.0.0.1" }
    desc { IP address of local interfaces }
}

defstate {
    name { ipforwarding }
    type { int }
    default { 1 }
    desc { specifies whether IP layer can forward datagrams }
}

defstate {
    name { ipdecapsulating }
    type { int }
    default { 0 }
    desc {}
}
```

```
defstate {
    name { visitortable }
    type { stringarray }
    default { "" }
    desc { IP addresses of mobile hosts visiting this foreign agent's
           home network
    }
}

defstate {
    name { caddrtable }
    type { stringarray }
    default { "" }
    desc { care of addresses advertized by foreign agent }
}

defstate {
    name { ofilename }
```

```
    type { string }
    default { "" }
    desc { name of file to save output }
}

inmulti {
    name { ipmobileintrq }
    type { message }
    desc {}
}

output {
    name { pr_input }
    type { message }
    desc {}
}

output {
    name { ipmobile_output }
```

```
    type { message }
    desc {}
}

protected {
    unsigned long int ips_cantforward;
    unsigned long int ips_delivered;
    unsigned long int ips_forward;
    unsigned long int ips_total;
    unsigned long int ips_odropped;
    unsigned long int ipmobile_dpackets;
    unsigned long int ipmobile_rpackets;
}

hinclude {
    "NetworkCell.h",
    <fstream.h>,
    <iostream.h>,
    <string.h>,
```

```
        <sys/types.h>,
        <sys/socket.h>,
        <netinet/in.h>,
        <arpa/inet.h>
    }

    inline method {
        name { ipmobileintr }
        access { protected }
        arglist { "(void)" }
        type { void }
        code {
        /* Pass time stamp w/o change */
        completionTime = arrivalTime;
//INPUT
        InDEMPHIter inIter(ipmobileintrq);
        InDEPort* inPtr;
        while ((inPtr = inIter++) != 0) {
            while (inPtr->dataNew) {
```

```
        ips_total++;

        /* Get the datagram on the _ipmobileintrq_ port */

        Envelope inEnvlp;

        inPtr->get().getMessage(inEnvlp);

        TYPE_CHECK(inEnvlp, "NetworkCell");

        const NetworkCell* cellPtr =

            (const NetworkCell*) inEnvlp.myData();

        /* Get the datagram final destination */

        struct in_addr ip_dst;

        ip_dst.s_addr = cellPtr->dest();

        char* ip_dest = inet_ntoa(ip_dst);

//OUTPUT

        /* Check own IP addresses */

        int size = in_ifaddrs.size();

        for (int i = 0; i < size; i++) {

            if ((strcmp(ip_dest, in_ifaddrs[i]) == 0)

                && (int(*cellPtr) == 1)) {

                if (ipdecapsulating) {

                    ipmobile_dpackets++;
```

```
/* Decapsulate datagram */
NetworkCell* outrPtr =
    (NetworkCell*) inEnvlp.writableCopy();
NetworkCell* inrPtr =
    (NetworkCell*) outrPtr->writableData();
/* Free memory resources */
LOG_DEL; delete outrPtr;
/* Search 'visitor list' */
struct in_addr visitor;
visitor.s_addr = inrPtr->dest();
char* visAddr = inet_ntoa(visitor);
int size = visitortable.size();
for (int j = 0; j < size; j++) {
    if (strcmp(visAddr, visitortable[j]) == 0) {
        int careofAddr = inet_addr(caddrtable[j]);
        int mssgSize = inrPtr->size();
        Message* toLoad = inrPtr->writableData();
        LOG_NEW; NetworkCell* shipItOut =
            new NetworkCell(*toLoad,
```

```
                                                  int(1),
                                                  int(careofAddr),
                                                  int(mssgSize),
                                                  0,
                                                  float(completionTime),
                                                  0);
                Envelope outEnvlp(*shipItOut);
                /* Forward datagram */
                ipmobile_forward(outEnvlp, 0);
                /* Free memory resources */
                LOG_DEL; delete inrPtr;
                goto next;
            }
        }
    }
    /* Pass datagram to _pr_input_ port */
    ips_delivered++;
    pr_input.put(completionTime) << inEnvlp;
    goto next;
```

```
                    }
                    if ((strcmp(ip_dest, in_ifaddrs[i]) == 0)
                        && (int(*cellPtr) == 0)) {
#ifdef DEBUG
cout << "ipmobile_input: dropping datagram destined for: " << ip_dest << endl;
#endif
                        ipmobile_rpackets++;
                        ipmobile_output.put(completionTime) << inEnvlp;
                        goto next;
                    }
                }
//OUTPUT
                /* Not for us; forward if possible */
                if (ipforwarding == 0) {
                    ips_cantforward++;
                } else {
                    /* Forward datagram */
                    ipmobile_forward(inEnvlp, 0);
#ifdef DEBUG
```

```
cout << "ipmobile_input: forwarding to: " << ip_dest << endl;
#endif
                }
                /* Get next simulation event */
        next:
                inPtr->getSimulEvent();
            }
        }
        }
    }

    inline method {
        name {ipmobile_forward}
        access { protected }
        arglist { "(Envelope& envlp, int srcrt)" }
        type { void }
        code {
            ipmobile_output.put(completionTime) << envlp;
            ips_forward++;
```

```
    }
}

setup {
    ips_cantforward = 0;
    ips_delivered = 0;
    ips_forward = 0;
    ips_total = 0;
    ips_odropped = 0;
    ipmobile_dpackets = 0;
    ipmobile_rpackets = 0;
}

go {
    ipmobileintr();
}

wrapup {
    ofstream ofile(ofilename);
```

```
    if (!ofile) {
        Error::abortRun(*this, "cannot open output file");
    }
    ofile << "#packets received for unreachable destination "
          << ips_cantforward << '\n'
          << "#packets decapsulated  "
          << ipmobile_dpackets << '\n'
          << "#packets received requesting registration "
          << ipmobile_rpackets << '\n'
          << "#packets delivered to upper level "
          << ips_delivered << '\n'
          << "#packets forwarded "
          << ips_forward << '\n'
          << "total #packets received "
          << ips_total << '\n'
          << "#packets dropped because of resource shortages "
          << ips_odropped << endl;
    ofile.close();
}
```

```
} //end defstar { ipmobile_input }
```

A.4.2 The ipmobile_output class definition

```
defstar {
    name { ipmobile_output }
    domain { DE }
    version {%W% %G%}
    author { Efren A. Serra }
    location { $PTOLEMY/mipps/src/stars }
    desc {
    }

    defstate {
        name { destination }
        type { stringarray }
        default { "127.0.0.1" }
        desc {}
    }
```

```
defstate {
    name { gateway }
    type { stringarray }
    default { "127.0.0.1" }
    desc {}
}

defstate {
    name { flag }
    type { stringarray }
    default { "UGH" }
    desc {}
}

defstate {
    name { mhtable }
    type { stringarray }
    default { "" }
    desc { mobile host table }
```

```
}

defstate {
    name { fatable }
    type { stringarray }
    default { "" }
    desc { foreign agent table }
}

defstate {
    name { out_ifaddrs }
    type { intarray }
    default { "0" }
    desc {}
}

defstate {
    name { ofilename }
```

```
    type { string }
    default { "" }
    desc { name of file to save output }
}

input {
    name { pr_output }
    type { message }
    desc {}
}

input {
    name { ipmobile_input }
    type { message }
    desc {}
}

output {
    name { vifp }
```

```
    type { message }
    desc { virtual interface pointer }
}

outmulti {
    name { ifp }
    type { message }
    desc { interface pointer }
}

hinclude {
    "NetworkCell.h",
    <fstream.h>,
    <iostream.h>,
    <string.h>,
    <sys/types.h>,
    <sys/socket.h>,
    <netinet/in.h>,
    <arpa/inet.h>
```

```
}

protected {
    unsigned long int ipmobile_epackets;
    unsigned long int ips_noroute;
    unsigned long int ips_mforward;
    const char* ro_dst;
}

header {
    const int NET_MASK = 0xffffff00;
}

inline method {
    name { if_output }
    access { protected }
    arglist { "(MultiOutDEPort& _ifp, Envelope& envlp)" }
    type { void }
    code {
```

```
        int size = 0;
        const NetworkCell* cellPtr =
            (const NetworkCell*) envlp.myData();
        /* Get the datagram final destination */
        struct in_addr ip_dst;
        struct in_addr ip_net;
        ip_dst.s_addr = cellPtr->dest();
        ip_net.s_addr = cellPtr->dest() & NET_MASK;
        char* ip_dest = strdup(inet_ntoa(ip_dst));
        char* ia_net = strdup(inet_ntoa(ip_net));
#ifdef DEBUG
cout << "ipmobile_output: dest: " << ip_dest << endl;
cout << "ipmobile_output: net: " << ia_net << endl;
#endif
        /* Search the routing tables */
        size = destination.size();
        for (int i = 0; i < size; i++) {
            if ((strcmp(ip_dest, destination[i]) == 0) &&
                (strcmp("UH", flag[i]) == 0)) {
```

```
                /* Direct route */
#ifdef DEBUG
cout << "ipmobile_output: direct route; dest: " << ip_dest << endl;
#endif
                ro_dst = destination[i];
                int hostAddr = cellPtr->dest();
                int mssgSize = cellPtr->size();
                Message* toLoad = envlp.writableCopy();
                LOG_NEW; NetworkCell* shipItOut =
                    new NetworkCell(*toLoad,
                                    int(1),
                                    int(hostAddr),
                                    int(mssgSize),
                                    0,
                                    float(completionTime),
                                    0);
                Envelope outEnvlp(*shipItOut);
//OUTPUT
                const int port = out_ifaddrs[i];
```

```
            OutDEMPHIter outIter(_ifp);
            OutDEPort* outPtr = outIter++;
            for (int j = 0; j > port; j--) { outPtr = outIter++; }
            /* Put the datagram on the _ifp_ port */
            outPtr->put(completionTime) << outEnvlp;
            goto free;
        }
        if ((strcmp(ia_net, destination[i]) == 0) &&
            (strcmp("UGH", flag[i]) == 0)) {
            /* Indirect route */
#ifdef DEBUG
cout << "ipmobile_output: indirect route; next hop: " << gateway[i] << endl;
#endif
            ro_dst = gateway[i];
            int nxtHop = inet_addr(gateway[i]);
            int mssgSize = cellPtr->size();
            Message* toLoad = envlp.writableCopy();
            LOG_NEW; NetworkCell* shipItOut =
                new NetworkCell(*toLoad,
```

```
                                        int(1),
                                        int(nxtHop),
                                        int(mssgSize),
                                        0,
                                        float(completionTime),
                                        0);
                Envelope outEnvlp(*shipItOut);
//OUTPUT
                const int port = out_ifaddrs[i];
                OutDEMPHIter outIter(_ifp);
                OutDEPort* outPtr = outIter++;
                for (int j = 0; j > port; j--) { outPtr = outIter++; }
                /* Put the datagram on the _ifp_ port */
                outPtr->put(completionTime) << outEnvlp;
                goto free;
            }
        }
free:
        if (ro_dst == 0) ips_noroute++;
```

```
    /* Free memory resources */
    free(ip_dest); free(ia_net);
    }
}

inline method {
    name { ip_mforward }
    access { protected }
    arglist { "(MultiOutDEPort& _ifp, Envelope& envlp)" }
    type { void }
    code {
        int size = destination.size();
        NetworkCell* cellPtr = (NetworkCell*) envlp.writableCopy();
        Message* toLoad = cellPtr->clone();
        for (int i = 0; i < size; i++) {
            if (strcmp(flag[i], "UH") == 0) {
                ips_mforward++;
                int hostAddr = inet_addr(destination[i]);
                int mssgSize = cellPtr->size();
```

```
                LOG_NEW; NetworkCell* shipItOut =
                    new NetworkCell(*toLoad,
                                    int(1),
                                    int(hostAddr),
                                    int(mssgSize),
                                    0,
                                    float(completionTime),
                                    0);
                Envelope outEnvlp(*shipItOut);
//OUTPUT
                const int port = out_ifaddrs[i];
                OutDEMPHIter outIter(_ifp);
                OutDEPort* outPtr = outIter++;
                for (int j = 0; j > port; j--) { outPtr = outIter++; }
                /* Put the datagram on the _ifp_ port */
                outPtr->put(completionTime) << outEnvlp;
            }
        }
        /* Free memory resources */
```

```
        LOG_DEL; delete cellPtr;
    }
}

inline method {
    name { vif_output }
    access { protected }
    arglist { "(MultiOutDEPort& _ifp, Envelope& envlp)" }
    type { void }
    code {
    int size = 0;
    const NetworkCell* cellPtr =
        (const NetworkCell*) envlp.myData();
    /* Get the datagram final destination */
    struct in_addr ip_dst;
    struct in_addr ip_net;
    ip_dst.s_addr = cellPtr->dest();
    ip_net.s_addr = cellPtr->dest() & NET_MASK;
    char* ip_dest = strdup(inet_ntoa(ip_dst));
```

```
        char* ia_net = strdup(inet_ntoa(ip_net));
#ifdef DEBUG
cout << "ipmobile_output: dest: " << ip_dest << endl;
cout << "ipmobile_output: net: " << ia_net << endl;
#endif
        /* If this is a multicast message... */
        if (strcmp(ip_dest, "127.255.255.255") == 0) {
            ip_mforward(_ifp, envlp);
            goto free;
        }
        /* Search the routing tables */
        size = destination.size();
        for (int i = 0; i < size; i++) {
            if ((strcmp(ip_dest, destination[i]) == 0) &&
                (strcmp("UH", flag[i]) == 0)) {
                /* Direct route */
#ifdef DEBUG
cout << "ipmobile_output: direct route; dest: " << ip_dest << endl;
#endif
```

```
                ro_dst = destination[i];

                int hostAddr = cellPtr->dest();

                int mssgSize = cellPtr->size();

                Message* toLoad = envlp.writableCopy();

                LOG_NEW; NetworkCell* shipItOut =

                    new NetworkCell(*toLoad,

                                    int(1),

                                    int(hostAddr),

                                    int(mssgSize),

                                    0,

                                    float(completionTime),

                                    0);

                Envelope outEnvlp(*shipItOut);

//OUTPUT

                const int port = out_ifaddrs[i];

                OutDEMPHIter outIter(_ifp);

                OutDEPort* outPtr = outIter++;

                for (int j = 0; j > port; j--) { outPtr = outIter++; }

                /* Put the datagram on the _ifp_ port */
```

```
            outPtr->put(completionTime) << outEnvlp;
            goto free;
        }
        if ((strcmp(ia_net, destination[i]) == 0) &&
            (strcmp("UGH", flag[i]) == 0)) {
            /* Indirect route */
#ifdef DEBUG
cout << "ipmobile_output: indirect route; next hop: " << gateway[i] << endl;
#endif
            ro_dst = gateway[i];
            int nxtHop = inet_addr(gateway[i]);
            int mssgSize = cellPtr->size();
            Message* toLoad = envlp.writableCopy();
            LOG_NEW; NetworkCell* shipItOut =
                new NetworkCell(*toLoad,
                                int(1),
                                int(nxtHop),
                                int(mssgSize),
                                0,
```

```
                                    float(completionTime),
                                    0);
                Envelope outEnvlp(*shipItOut);
//OUTPUT
                const int port = out_ifaddrs[i];
                OutDEMPHIter outIter(_ifp);
                OutDEPort* outPtr = outIter++;
                for (int j = 0; j > port; j--) { outPtr = outIter++; }
                /* Put the datagram on the _ifp_ port */
                outPtr->put(completionTime) << outEnvlp;
                goto free;
            }
        }
        /* Search the mobile host table */
        size = mhtable.size();
        for (int i = 0; i < size; i++) {
            if ((strcmp(ip_dest, mhtable[i]) == 0)) {
                ipmobile_epackets++;
                /* Tunnel encapsulate */
```

```
#ifdef DEBUG
cout << "ipmobile_output: tunnel-encapsulate; FA: " << fatable[i] << endl;
#endif
                ro_dst = fatable[i];
                int foreignAgent = inet_addr(fatable[i]);
                int mssgSize = cellPtr->size();
                Message* toLoad = envlp.writableCopy();
                LOG_NEW; NetworkCell* shipItOut =
                    new NetworkCell(*toLoad,
                                    int(1),
                                    int(foreignAgent),
                                    int(mssgSize),
                                    0,
                                    float(completionTime),
                                    0);
                Envelope outEnvlp(*shipItOut);
//OUTPUT
                /* Put the datagram on the _vifp_ port */
                vifp.put(completionTime) << outEnvlp;
```

```
                goto free;
            }
        }
free:
        if (ro_dst == 0) ips_noroute++;
        /* Free memory resources */
        free(ip_dest); free(ia_net);
        }
    }

    setup {
        ipmobile_epackets = 0;
        ips_noroute = 0;
        ips_mforward = 0;
        ro_dst = 0;
    }

    go {
        Envelope inEnvlp;
```

```
        /* Pass time stamp without change */
        completionTime = arrivalTime;
//INPUT
        if (pr_output.dataNew) {
            /* Get the datagram on the _pr_output_ port */
            pr_output.get().getMessage(inEnvlp);
            /* Check type input packet */
            TYPE_CHECK(inEnvlp, "NetworkCell");
//OUTPUT
            if_output(ifp, inEnvlp);
        }
//INPUT
        if (ipmobile_input.dataNew) {
            /* Get datagram on _ipmobile_input_ port */
            ipmobile_input.get().getMessage(inEnvlp);
            /* Check type of input packet */
            TYPE_CHECK(inEnvlp, "NetworkCell");
//OUTPUT
            vif_output(ifp, inEnvlp);
```

```
        }
    }

    wrapup {
        ofstream ofile(ofilename);
        if (!ofile) {
            Error::abortRun(*this, "cannot open output file");
        }
        ofile << "#packets encapsulated "
              << ipmobile_epackets << '\n'
              << "#packets discarded -- no route to destination "
              << ips_noroute << '\n'
              << "#packets sent as agent advertisements "
              << ips_mforward << endl;
        ofile.close();
    }
} //end defstar { ipmobile_output }
```

Bibliography

[1] David B. Johnson, "Scalable Support for Transparent Mobile Host Internetworking," *Mobile Computing*, pp. 103-128, 1996.

[2] J. B. Postel, editor. Internet Protocol. Internet Request for Comments RFC 791, September 1981.

[3] C. Perkins, editor. IP Mobility Support. Internet Request for Comments RFC 2002, October 1996.

[4] Y. C. Tay, K. C. Chua, "On the Performance of a Mobile Internet Protocol and Wireless CSMA Cell," EE Dept., National University of Singapore, 1997. (http://mip.ee.nus.edu.sg).

[5] R. Droms. Dynamic Host Configuration Protocol. Internet Request for Comments RFC 1541, October 1993.

[6] J. B. Postel. Multi-LAN Address Resolution. Internet Request for Comments RFC 925, October 1984.

[7] S. E. Deering. ICMP Router Discovery Messages. Internet Request for Comments RFC 1256, September 1991.

[8] D. B. Johnson. Transparent Internet Routing for IP Mobile Hosts. Internet Draft, July 1993. Work in progress.

[9] D. B. Johnson, "Scalable and Robust Internetwork Routing for Mobile Hosts," *Proceedings of the 14th International Conference on Distributed Computing Systems*, pp. 2-11, June 1994.

[10] C. Perkins. IP in IP Encapsulation. Internet Request for Comments RFC 2003, October 1996.

[11] F. Teraoka, *et al.*, "Design, Implementation, and Evaluation of Virtual Internet Protocol," *Proceedings of the 12th International Conference on Distributed Computing Systems*, June 1992.

[12] F. Teraoka, *et al.*, "A Network Architecture Providing Host Migration Transparency," *Proceedings of ACM SIGCOMM 91*, September 1991.

[13] M. Baker, *et al.*, “Supporting Mobility in MosquitoNet,” *1995 Winter USENIX*, January 1996.

[14] J. Ioannidis, *ct al.*, “Protocols for Mobilc Host Intcrnctworking,” *Proceedings of ACM SIGCOMM 91*, June 1991.

[15] D. C. Plummer. Ethernet Address Resolution Protocol: Or Converting network protocol addresses to 48-bit Ethernet addresses for transmission on Ethernet hardware. RFC 826, November 1982.

[16] C. L. Hedrick. Routing Information Protocol. Internet Request For Comments RFC 1058, June 1988.

[17] C. L. Hedrick. An Introduction to IGRP, August 1991. Available with anonymous FTP from ftp.cisco.com, file igrp.doc.

[18] E. A. Lee and D. G. Messerschmitt, “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing,” *IEEE Trans. on Computers*, vol. 36, no. 1, pp. 24-35, January 1987.

[19] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschimitt, "Multirate Signal Processing in Ptolemy," *Proc. of Int. Conf. on Acousticts, Speech, and Signal Processing*, Toronto, Canada, April 1991, vol. 2, pp. 1245-1248.

[20] J. Buck and E. A. Lee, "The Token Flow Model," *Advanced Topics in Dataflow Computing and Multithreading*, ed. L. Bic, G. Gao, and J. Gaudiot, IEEE Computer Society Press, 1993.

[21] T. M. Parks, *Bounded Scheduling of Process Networks*, Technical Report UCB/ERL 95/105, Ph.D. Dissertation, EECS Department, University of California, Berkeley, CA, 94720-1770, December 1995. (http://ptolemy.eecs.berkeley.edu/papers/parksThesis).

[22] S. A. Edwards, *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*, Ph.D. Dissertation, ERL Technical Report UCB/ERL M97/31, EECS Dept., University of California, Berkeley, 1997. (http://ptolemy.eecs.berkeley.edu/papers/97/sedwardsThesis).

[23] G. Walter, *ATM, Speech Coding, and Cell Recovery*, M.S. Report, EECS Dept., University of California, Berkeley, CA 94720, December 1992.

[24] P. Haskell, *Flexibility in the Interactions Between High-Speed Networks and Communications Applications*, Ph.D. Dissertation, EECS Dept., UC Berkeley, Berkeley CA 94720, October 1993.

[25] W. T. Chang, S. H. Ha, and E. A. Lee, "Heterogeneous Simulation – Mixing Discrete-Event Models with Dataflow," invited paper, RASSP special issue of the Journal on VLSI Signal Processing, January 1997. (http://ptolemy.eecs.berkeley.edu/papers/96/heterogeneity).

[26] S. S. Bhattacharyya and E. A. Lee, "Looped Schedules for Dataflow Descriptions of Multirate Signal Processing Algorithms," *Formal Methods in Systems Design*, No. 5, No. 3, December 1994 (Updated from Technical Report UCB/ERL M93/37, EECS Dept., UC Berkeley, May 21, 1993).

[27] S. S. Bhattacharyya and E. A. Lee, "Scheduling Synchronous Dataflow Graphs for Efficient Looping," *J. of VLSI Signal Processing*, vol. 6, December 1993.

[28] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, "Hierarchical Static Scheduling of Dataflow Graphs onto Multiple Processors," *Proc. of Int. Conf. on Acoustics, Speech, and Signal Processing*, May 1995, pp. 2643-2646, Detroit, MI.

[29] A. Kalavade, and E. A. Lee, "A Hardware/Software Codesign Methodology for DSP Applications," *IEEE Design and Test of Computers*, vol. 10, no. 3, pp. 16-28, September 1993.

[30] D. G. Messerschmitt, "A Tool for Structured Functional Simulation," *IEEE J. on Selected Areas in Communications*, vol. 2, no. 1, pp. 137-147, January, 1984.

[31] E. A. Lee, W. H. Ho, E. Goei, J. Bier, and S. Bhattacharyya, "Gabriel: A Design Environment for DSP," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. 37, no. 11, pp. 1751-1752, November 1989.

[32] J. Bier, E. Goei, W. Ho, P. Lapsley, M. O'Reilly, G. Sih and E. A. Lee, "Gabriel: A Design Environment for DSP," *IEEE Micro Magazine*, October 1990, vol. 10, no. 5, pp. 28-45.

[33] D. Harrison, P. Moore, R. Spickelmier, and A. R. Newton, "Data Management and Graphics Editing in the Berkeley Design Environment," *Proc. of IEEE Int. Conf. on Computer-Aided Design*, November 1986, pp. 24-27.

[34] J. K. Ousterhout, "Tcl: An Embeddable Command Language," *Proc. of USENIX Conference*, January 1991, pp. 105-115.

[35] J. K. Ousterhout, *An Introduction to Tcl and Tk*, Addison-Wesley Publishing, Redwood City, CA, 1994, ISBN 0-201-63337-X.

[36] E. A. Lee, "A Design Lab for Statistical Signal Processing," *Proc. of Int. Conf. on Acoustics, Speech, and Signal Processing*, March 1992, vol. 4, pp. 81-84, San Francisco, CA.

[37] E. A. Lee, "Consistency in Dataflow Graphs," *IEEE Trans. on Parallel and Distributed Systems*, vol. 2, no. 2, pp. 223-235, April 1991.

[38] P. K. Murthy, *Multiprocessor DSP Code Synthesis in Ptolemy*, Memorandum No. UCB/ERL M93/66, Electronics Research Laboratory, University of California, Berkeley CA 94720, 1993.

[39] Anthony Wong, *A Library of DSP Blocks and Applications for the Motorola DSP96000 Family*, M.S. Report, Plan II, EECS Dept., UC Berkeley, CA 94720, May 1992.

[40] Soonhoi Ha and E. A. Lee, "Compile-Time Scheduling and Assignment of Dataflow Program Graphs with Data-Dependent Iteration," *IEEE Trans. on Computers*, vol. 40, no. 11, pp. 1225-1238, November 1991.

[41] S. Ha, *Compile-Time Scheduling of Dataflow Program Graphs with Dynamic Constructs*, Ph.D. Dissertation, EECS Dept., University of California, Berkeley, CA 94720, April 1992.

[42] G. C. Shi and E. A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 2, February 1993.

[43] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Info. Proc.*, Stockholm, Sweden, August 1974, pp. 471-475.

[44] S. Sriram and E. A. Lee, "Design and Implementation of an Ordered Memory Access Architecture," *Proc. of Int. Conf. on Acoustics, Speech, and Signal Processing*, April 1993, vol. I, pp. 345-348, Minneapolis, MN.

[45] R. Alverson, *et al.*, *THOR user's manual: Tutorial and commands*, Technical Report CSL-TR-88-348, Stanford University, January 1988.

[46] A. Lao, *Heterogeneous Cell-Relay Network Simulation and Performance Analysis with Ptolemy*, M.S. Report, Electronics Research Laboratory, University of California, Berkeley, CA 94720, February 1994.

[47] G.H. Forman and J. Zahorjan, *The challenges of mobile computing*, IEEE Computer (April 1994), 38-47.

[48] R.M. Metcalfe and D.R. Boggs, *Ethernet: Distributed packet switching for local computer networks*, Communications of the ACM 19, 7 (July 1976), 395-404.