The Dissertation Committee for Seong-Kyu Song

certifies that this is the approved version of the following dissertation:

Applying Active Network Adaptability to Wireless Networks

Committee:

Scott M. Nettles, Supervisor

Gustavo de Veciana

Edward J. Powers

Theodore S. Rappaport

Jonathan M. Smith

Applying Active Network Adaptability to Wireless Networks

by

Seong-Kyu Song, B.S.E., M.S.E.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2004

To my family

for their love and prayers

Acknowledgments

First of all, I would like to thank my supervisor, Dr. Scott M. Nettles for his support and guidance. He guided me through the maze of research with the remarkable insights and the great encouragements. He also showed me a good role model as a teacher, a researcher, and a human being.

I am deeply indebted to Dr. Jonathan M. Smith for all his advice and support. I am very grateful to Dr. Edward J. Powers for his supportive guidance, to Dr. Gustavo de Veciana for his kind encouragement, and to Dr. Theodore S. Rappaport for his helpful direction.

My gratitude to the members of the NetLab; Stephen Shannon, Pisai Setthawong, Chandresh Kumar Jain, Hari Shankar, Sudipta Das, Minyoung Park, Yihong Zhou, Gibeom Kim, Sangwoo Lee, and Soon Hyuk Choi for all their enjoyable friendship and helpful discussions.

I would also like to thank my fabulous friends I have met here in Austin for their friendship. Special thanks to Kyoil Kim, Yongju Jeon, Joonhyuk Kang, Sangkyu Park, Dong-Ho Kim, Kiwoon Kim, Seokjin Lee, Hak-Soo Yu, Jaeki Yoo, Youngmoon Choi and Jangwon Lee.

Finally, I thank God for everything. Glory to God for everything! He allowed me many great things, one of which is my wonderful family. I cannot fully express my gratitude to my extraordinary parents and parents-in-law for their love, prayers, patience, and support. Their love has raised me up and touched my life. Their prayers have guided and protected me. This work is dedicated to my lovely wife, Ye-Rang Kim, who has shared everything of life and always supported me with incomparable love and encouragement, and to my precious son, Andrew Junsuh Song, who is always giving me great smiles and invaluable happiness.

Seong-Kyu Song

The University of Texas at Austin

December 2004

Applying Active Network Adaptability to Wireless Networks

Publication No. _____

Seong-Kyu Song, Ph.D.

The University of Texas at Austin, 2004

Supervisor: Scott M. Nettles

The IP-based Internet, although wildly successful, is limited in its ability to evolve and adapt, in particular at the network layer. Mobile/wireless networking is an important emerging area in which adaptivity and evolvability is likely to be especially important due in part to the widely varying nature of the underlying communication channels themselves.

We believe that active networking (AN) enables valuable adaptivity that existing technologies currently lack. This is because AN enables highly flexible packet functionality, on-the-fly protocol deployment, even on a packet-by-packet granularity, and cost-effective network expansion. Because adaptivity and evolvability is at a premium, we have chosen to test our belief in the

mobile/wireless networking domain using three case studies: Mobile IP, ad hoc routing, and TCP over wireless. In our work, we demonstrate AN's adaptivity by developing a series of designs, simulation studies, and working prototypes.

Mobile IP is a protocol that supports mobility within the existing IP architecture by separating naming and addressing. While its design fits the conventional architecture and is feasible in current networks, Mobile IP exemplifies the inability of current networks to evolve effectively. Using Active Packet evolution and Update evolution techniques, we show how to deploy the new protocol and to evolve networks to support Mobile IP.

Ad hoc networks are infrastructureless networks in which hosts are typically mobile and must act as routers. Mobility makes routing hard because the state of links changes frequently and routing heterogeneity is likely. We show how AN can provide useful routing adaptation to host mobility, in addition to routing evolution.

In the last case study, we address the performance degradation of TCP over lossy links. TCP's congestion control may cause under-utilization of bandwidth in wireless networks. We demonstrate AN's adaptation to changing link conditions. Furthermore, taking advantage of flexible cross-layer interactions, we show AN's ability to adapt to changes in TCP flow information. We show that active packets are especially useful in this context because they are extremely agile and allow adaptation on a packet-by-packet basis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern network architectures, in particular the IETF's IP-based Internet, have been very successful. With success, however, has come the recognition of limitations, in particular with respect to customization and evolution. As the network's size grows larger, it becomes harder to add new services or to modify the currently deployed protocols. For example, it took seven years for the initial deployment of Random Early Detection (RED) into the real network [1].

We recognize several reasons for the limitations of current network architectures. Firstly, there are limitations in the fundamental architecture. In the current 'hourglass' architecture of the IETF protocol stack, IP plays a role as a simple unifier between upper and lower layers. Although the simplicity and openness of IP are advantageous for scalability, this architecture allows no specialization of the network layer by either the application layer or the link

layer. Further, since the network layer works as a standard unifier, it is hard to add general services at this level.

In addition, there are structural limitations in the standardization, implementation, and deployment of the network infrastructure. Through the slow process of consensus-based standardization, protocols must be defined and engineered before the system is widely deployed. Standardization requires a long time to engineer the protocols thoroughly before implementation and deployment. One of the reasons for the long time frame of standardization is that once the protocols are deployed, it is difficult to change or evolve them. Further, even after standardization, there are generally "reserved" fields in packet header formats for future use. The reservation of packet header fields is a limited way of supporting network evolvability; this space is wasted until and unless such evolution occurs. Once a protocol is built into the infrastructure, it is very difficult to make modifications or updates that take effect quickly. If a new requirement is discovered and added into the standard (which itself takes significant time), some or all of network elements' software will need to be changed. This requires significant time and effort especially with increasing network size. Thus, there are problems in deploying new functionality.

Lastly, the architecture is based on abstractions that are often violated. With the help of the unifying network layer, upper layers should be independent of the link layer. As a matter of fact, however, there are implicit assumptions built into the upper layers about the lower layers and vice versa. For example, the link layer handles packets individually on the assumption that

all the payloads from applications are of the same importance. The transport layer also has assumptions about the link layer. In TCP congestion control, for instance, packet drops are interpreted as a symptom of congestion on the assumption that links are highly reliable. Another example is that a TCP connection is identified by a four tuple <`source IP address`, `destination IP address`, `source port number`, `destination port number`>, and renumbering of TCP connections is not allowed. The transport layer assumes that end hosts use static addresses and do not change addresses during a connection. These assumptions not only violate abstraction boundaries, but also are no longer justified, especially with the advent of link layers with new characteristics, such as wireless links.

1.1 Mobility and Adaptability

Wireless links are being rapidly deployed and as a result mobile networks are an important emerging technology[1]. However, there has been a slow deployment of the supporting protocols in the Internet; and link characteristics, such as high bit-error rate, long delay, rapidly changing links, host mobility, and ad hoc networks, have created problems due to the limitations of current network architecture.

Since wireless links change their status quickly, in some cases faster than a Round-Trip Time (RTT), protocols and systems need to adapt to link

[1]In this document, we will use "mobile networks" as a shorthand for networks with mobile nodes typically communicating over wireless links.

fluctuations rapidly. The current architecture allows layer interactions only through the unifying network layer; thus, it is hard to specialize link layers or application layers for wireless links. In addition, mobility implies new services, such as location services; however, it is not easy to add these new services to the current architecture.

The current structure of standardization, implementation, and deployment of network infrastructure is also a problem in mobile networking. Current mobile network deployment is happening simultaneously with standardization, but we can expect our understanding of mobile networking protocols to improve rapidly, creating a need to change out-of-date protocols. Therefore, mobile networks require more flexible and timely protocol deployment and evolution.

Abstraction breaking has become apparent in mobile networking as well. The new characteristics of wireless links, such as lossy links and location changes during connections, conflict with the assumptions of the current network; and thus mobile networks suffer from problems like performance degradation due to dropped packets or limited functionality.

Due to the problems described above, the existing architecture's limitations have proven significant, and mobile networking is an area in which adaptivity and evolvability is likely to be especially important. As further evidence, Marjory Blumenthal and David Clark [2] support our arguments that the current network architecture might be ineffective due to the difficulty of adapting the layering to new circumstances in mobile networks.

1.2 Active Networking

Active Networking (AN) has been developed to address the limitations of current networks with the ability to create, deploy, and manage services promptly [3]. By introducing programmability into the network infrastructure, AN can provide application-specific and link-specific customization, as well as flexibility and extensibility in designing and deploying protocols [4]. To be specific, extensible routers support dynamic protocol extension and customization; and third party extensions help to allow new services to be deployed easily and promptly. Further, programmable packets actualize services and protocols on the fly. Packet-by-packet adaptivity enables the network to adjust very rapidly to changing environments. We provide all necessary AN background in Chapter 2.

1.3 Thesis

Our thesis is:

> Active Networking can provide useful *adaptivity and evolvability* for mobile networks, especially when faced with rapidly changing network conditions.

1.4 Goals and Approach

Our goal is to demonstrate how AN can be applied to the problems of mobile networks, where network environments are dynamically changing and the traditional architecture has been shown to be limited. To support this claim, we have performed a series of case studies based on three important problems in mobile networking: Mobile IP deployment, ad hoc routing, and TCP performance over wireless links.

To explore the issues raised by our case studies, we have both developed a series of working prototypes that embody our techniques and performed simulations to test these techniques on larger scales than we can practically experiment with directly. For simulating and measuring network performance, we use the ns-2 simulator [5]. *Ns* is a discrete event simulator developed by the VINT project [6]. For prototyping, we use our AN testbed system, the Mobile Active Network Environment (MANE) [7]. MANE is discussed in some detail in Chapter 2.

For each case study, we discuss the current state of the art for the problem at hand in the existing network architecture. We consider both the standardization and deployment of the protocols and technologies relevant to each study. At the conclusion of each case study we compare adaptivity and evolvability with both qualitative and quantitative discussion based on our experiments and implementation to the state of the art in the existing network.

One issue our thesis and this dissertation does not address is the raw performance of our AN solutions. This is because our implementation technology, discussed in Chapter 2, emulates wireless transmission and in general has been designed to facilitate experiments in flexibility rather than having been optimized for speed. However, these issues have been extensively considered in other AN research [8, 9, 10]. We discuss these results and their bearing on our work in Chapter 2.

1.4.1 Case I: Mobile IP

Our first case study applies AN-based adaptivity and evolution to implementing and deploying the mobile IP protocol [11, 12]. To demonstrate network evolution for mobility, we have chosen to add support to MANE for what is essentially Mobile IP. Several reasons motivated our choice of mobility from which to draw our examples. First, mobility is an area in which new protocols and improvements to existing protocols are being developed rapidly. Thus it is an area where better evolutionary capabilities could be a real benefit, since then protocols could be deployed and later upgraded and replaced as new techniques develop. Second, in the particular area of Mobile IP, current protocols are constrained in their design to require only local changes to the network infrastructure. Practical evolution capability would allow other (preferable) protocols to be developed. Finally, mobility is an interesting domain in its own right, and the current work allows us to begin to understand the issues there in the context of our design and implementation environment.

In this case study, we show how AN can be used to upgrade a network's services on the fly, without centralized coordination and without halting network service. By doing so, we are making a strong claim that AN *can* be used to quicken the pace of network service evolution. For our demonstration, we present how to augment an active network that provides standard, IP-like service to support routing for mobile hosts, in the spirit of Mobile IP.

Through Active Packet Evolution and Update Evolution [7], we show how to deploy a new protocol to support mobility easily. The chief advantage of Active Packet evolution is that it is lightweight and allows third parties to enhance the functionality of the network without changing the nodes themselves. Further, using Update Evolution, application-transparent evolution can be achieved even when the system design has not anticipated the need for a particular kind of change. In some sense, this embodies the entire goal of AN.

1.4.2 Case II: Ad Hoc Routing

In mobile ad hoc networks (MANETs), there is no fixed infrastructure (routers) and all the nodes may move with any frequency [13]. Since path changes and link failures may happen frequently, routing is one of the most difficult issues in MANETs. The dynamic characteristics of MANETs require ad hoc routing protocols to adapt to rapidly changing conditions.

Furthermore, since MANETs can occur without prior planning, it is entirely possible that not all the nodes are equipped with the same routing

protocol. However, many ad hoc routing protocols in the literature have been based on the assumption that one specific routing protocol can be pre-deployed and used on all the nodes in the network. To the best of our knowledge, there has been no work on routing problems in heterogeneous ad hoc networks. We expect that AN can provide solutions to the requirements for adaptive and heterogeneous routing.

Since AN can actualize a routing protocol on the fly by evaluating mobile code carried in lightweight active packets, AN provides mechanisms to implement adaptivity in ad hoc routing. In our preliminary experiments, we demonstrate evolutionary adaptivity in which active packets help to upgrade a routing protocol easily and without further modifications to the infrastructure. We can extend this approach to create an adaptive routing protocol or *multi-mode routing protocol* by injecting active packets for the optimal routing protocol based on current network conditions. By taking advantage of AN's dynamic linking and loading of router extensions, we also expect to deploy new protocols easily and to overcome the routing heterogeneity problem.

1.4.3 Case III: TCP over wireless

The Transmission Control Protocol (TCP) is a connection-oriented, byte-stream transport layer protocol responsible for end-to-end reliable communications [14, 15]. For reliable connections, TCP supports in-order delivery and retransmissions. Further, in order to utilize network resources efficiently, TCP employs several adaptive mechanisms, such as flow control and conges-

tion control.

TCP's congestion control is based on the assumption that links are so reliable that packet drops occur only due to congestion on routers; and TCP recognizes packet drops as the symptoms of node congestion. The problem is that the error recovery and congestion control mechanisms are closely coupled [16]. When packets are dropped, in addition to retransmitting the dropped packets, TCP launches congestion control to decrease the bandwidth usage of the sender. This mechanism is not effective over the wireless links, because wireless links are lossy and packet drops can be due to either *node congestion* or *link errors*. If the drops are due to bit-errors over the links, congestion control causes the sending host to under-utilize the network bandwidth and TCP suffers from overall performance degradation [17].

Using AN technology, we will address this problem of TCP over wireless links. Our approaches fall into two classes: horizontal and vertical. In the horizontal approach, link layer protocols attempt to cope with the channel variations adaptively and transparently to the end-to-end connections. The vertical approach complements the horizontal approach to control the TCP flow adaptively by allowing information sharing between layers and structured layer crossing violations.

1.5 Road Map

The remainder of this document is organized as follows. Chapter 2 presents the background relevant to our study. We discuss the difficulty of deploying mobile networks; and we describe the basic concepts of Active Networking (AN) and our AN testbed MANE. The following three chapters present three case studies of AN applied to mobile/wireless networking. Chapter 3 shows how AN facilitates to newly deploy the mobile IP protocol. Chapter 4 presents the case study of ad hoc routing and AN adaptivity issues on ad hoc networks. Chapter 5 deals with applying AN techniques to TCP performance issues over wireless links. Chapter 6 summarizes the contributions of our work. Finally, Chapter 7 concludes the dissertation.

Chapter 2

Background

In this chapter, we present the background that is needed to understand the remainder of the dissertation. First, we discuss the difficulty of deploying wireless networks into the Internet. Next, we describe the basic concepts and the general architecture of Active Networking systems. We then present the details of our AN system, MANE, and its packet programming language, PLAN. We conclude with a brief discussion of the performance implications of our AN approach.

2.1 TCP/IP Architecture and Wireless Links

In the current 'hourglass' architecture of the IETF protocol stack shown in Figure 2.1, the IP layer protocol plays a role as a simple unifier between upper layers and heterogeneous lower layers. Upper layers do not need to care

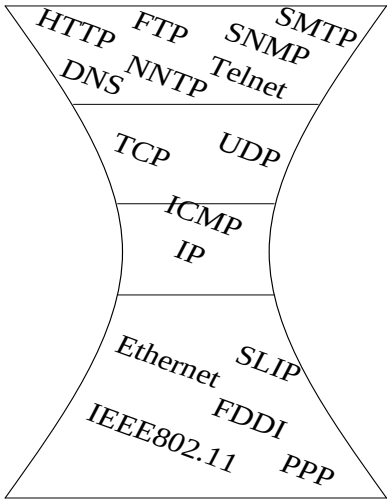

Figure 2.1: TCP/IP Hourglass Architecture

about different link layers, and vice versa. Through layering, complex tasks are broken into more manageable, smaller pieces of functionality and the implementation details of each layer are hidden from other layers. The logical separation of tasks and information hiding make it easy to change or modify parts of a layer later without affecting other layers. However, one of the drawbacks of layering is that the layers need to be defined and engineered before the system is widely deployed. After the system is deployed, modifications are limited within the layered architecture. Therefore, the layering paradigm might be ineffective due to the difficulty of adapting the layering to new circumstances [2].

Wireless link layers exemplify the existing network's inability to adapt to new environments. Wireless links were of limited concern in the initial

design and deployment of the Internet protocol, but have emerged as an important technology. The characteristics of wireless links, such as high bit-error rate, long delay, and intermittent connection status, have significant impacts on the overall performance on the network [18, 19]. According to the layering principle, these characteristics should be handled within link layer protocols without affecting other layers. However, simple insertion of wireless link layers into the current Internet protocol stack has not been effective because of the implicit assumptions about link characteristics in upper layer protocols. For instance, the routing protocols suppose that link connections change very slowly. TCP also makes an assumption that link layers are so reliable that packet drops only occur due to router congestion. These assumptions conflict with the properties of wireless links. Because the currently deployed protocols were designed and implemented based on these assumptions, the Internet protocols have limits in adopting wireless links.

2.2 Active Networking

The Internet can be viewed as a programmed network in that the end hosts and routers operate by protocols or *stored programs*. However, the Internet is not fundamentally *reprogrammable* and can be re-programmed only by the vendors by means of the slow process of consensus-based standardizations. By introducing programmability, Active Networking (AN) aims at application and link specific customization and speedy deployment of protocols and ser-

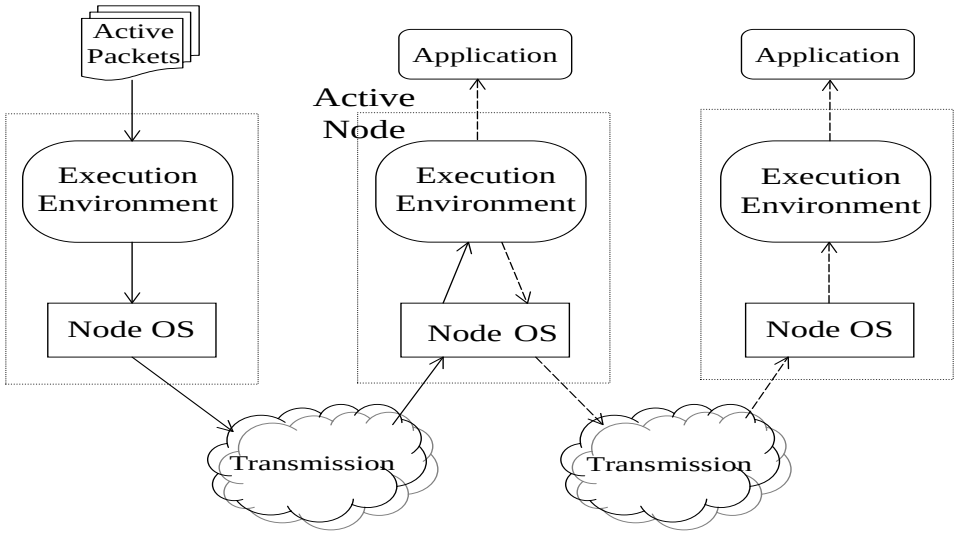

Figure 2.2: Architecture of Active Node and Execution of Active Packets

vices [3]. Based upon remote evaluation [20] and mobile code technologies [21], AN inserts an *evaluate* phase into the conventional *store-and-forward* delivery mechanism of data networks. Using these techniques, an AN's functionality can be evolved in an incremental manner. Figure 2.2 illustrates the general architecture of active nodes and execution of active packets.

The realization of this AN architecture can be characterized by two extreme approaches: the *active extension (AE)* approach, based on programmable switches, and the *active packet (AP)* approach, based on programmable packets [3]. In general, active packets contain programs that execute as they pass through the nodes of the network. Their execution can perform management actions on the nodes, affect their own routing, or form the basic distributed

computational framework of larger protocols. Through their computations on remote nodes, active packets can affect hop-by-hop operations, which will also affect end-to-end performance. Complementary to active packets are downloadable node extensions that form the basis of the programmable network infrastructure. They provide the services that active packets can use while they are executing on a node. These extensions can be downloaded and dynamically linked.

The flexibility of these two technologies together makes AN a good choice for environments that require a high degree of adaptivity. Further, because AN can accomplish protocol implementation on the fly, it is easy and quick to accommodate new services in the existing networks. Also there is no need to define packet header formats in advance since packet programs serve this function. Therefore, AN saves time in standardizing and deploying new protocols.

For the similar objectives of dynamic protocol customization and rapid protocol evolution, protocol boosters were suggested [22]. Protocol boosters attempt to improve performance or add additional functionality transparently to the existing system. The methodology facilitates dynamic behavior changes and optimistic protocol realization on an as-needed basis. Since then, there have been several prototype systems implementing an AN environment: UPenn's SwitchWare [23], MIT's ANTS [24], and BBN's SmartPackets [25], among others. Even though their architectures and applications are slightly different, their common objective is to show how the concept of active net-

working can be implemented and to study the benefits of this new approach.

2.2.1 Network Evolution through Active Networking

Broadly speaking, by "network evolution" we mean any incremental change to a network that modifies or enhances existing functionality or adds new functionality. In the context of Active Networking a somewhat more ambitious goal is appropriate: evolution should be able to occur at remote nodes while the network is operational with only minimal disruption to existing services.

AN achieves evolution by changing the programs that operate the network. Thus the ways in which we can evolve the network are dictated by the programmability mechanisms that are available to make such changes. In some cases, these mechanisms are AN specific, but generally they are drawn from general programming language technology. Thus, although later we will choose instances of these mechanisms that are specific to our platform, this discussion is general and applies broadly to AN systems.

In this section, we describe three mechanisms for achieving AN evolution. In each case, we discuss what type of evolution is supported by the mechanism. We also consider how the mechanism might support application-aware or transparent evolutions.

Active Packets

Active packets (AP) are perhaps the most radical AN technology for evolution and they are the only mechanism that, at a high level, are specific to AN. Such

packets carry (or literally are) programs that execute as they pass through the nodes of network. A packet can perform management actions on the nodes, effect its own routing, or form the basis of larger protocols between distributed nodes, e.g. routing protocols. Such packets can form the glue of the network, much like conventional packets, but with qualitatively more power and flexibility.

The AN community has explored a number of AP systems. The early systems include Smart Packets [25], ANTS [24], and PLAN [26], while more recent systems include PAN [10], SafetyNet [27], StreamCode [28], and SNAP [9]. Although these systems differ on many details of their design and implementation, they all support the basic AP model and thus the same general styles of evolution.

Active packets support the first and simplest type of network evolution we identify, *Active Packet evolution*, which does not require changes to the nodes of the network. Instead, it functions solely by the execution of APs utilizing standard or existing services. The disadvantage of this approach is that taking advantage of new functionality requires the use of new packet programs. This means that at some level the applications using the functionality must be aware that the new functionality exists. This is the kind of evolution facilitated by pure AP systems, such as ANTS [24], and in essence it embodies the AN goal of application-level customization.

Plug-In Extensions

The programmability mechanism that is broadly familiar outside the AN community is the *plug-in extension*. Plug-in extensions can be downloaded and dynamically linked into a node to add new node-level functionality. For this new functionality to be used, it must be callable from some prebuilt and known interface. For example, a packet program will have a standard way of calling node resident services. If it is possible to add a plug-in extension to the set of callable services (typically by extending the service name space) then such an extension "plugs in" to the service call interface.

Plug-in extensions are commonly used outside of AN. For example, the Linux kernel enables plug-in extensions for network-level protocol handlers, drivers, and more. Java-enabled web browsers support applets, which are a form of plug-in extension. Plug-ins are also common to AN. In CANES [29], nodes execute programs that consist of a fixed underlying program and a variable part, called the injected program. The fixed program contains *slots* that are filled in plug-in extensions. In Netscript [30], programming takes place by composing components into a custom dataflow. In this case, each element in the composed program is a plug-in, and the abstract description of such an element forms the extension interface. Plug-ins are used in hardware-based approaches as well, including the VERA extensible router at Princeton [31], and Active Network Nodes (ANN) at Washington University and ETH [32, 33].

Plug-in extensions support the second type of evolution we identify,

plug-in extension evolution. When used in conjunction with APs, packet programs can use new node resident services specialized to their needs rather than just standard services. Such evolution is particularly important if standard services are not sufficient to express a needed application. The combination of Active Packet and plug-in extension evolution is facilitated by systems such as PLANet [34], SANE [35], and SENCOMM [36].

Plug-in extensions that must be referenced by new AP programs are obviously not application transparent. However, as long as a plug-in simply replaces an existing interface, whether that interface is accessed from an AP or even in a more conventional system that does not support APs at all, then the evolution can be application transparent. This situation occurs in CANES, for example. However, the system still must have been designed to allow the required change (e.g. in CANES, this is made possible by the slots in the fixed program).

Update Extensions

The final programmability mechanism we consider is the *update extension*. Update extensions may also be downloaded, but they go beyond plug-in extensions in that they can update or modify existing code and can do so even while the node remains operational. Thus, such extension can add to or modify a system's functionality even when there does not exist an interface for it to hook into.

There is significant research literature on such extensions (e.g. [37, 38,

39], to name a few) although in general the focus has been on code maintenance rather than evolving distributed system functionality. We are using Dynamic Software Updating [8] which was initially inspired by the difficulties of crafting a plug-in interface for the packet scheduler in PLANet [34, 40]. However, the system itself is not specific or specialized to AN.

Update extensions support the final type of network evolution we identify, *update extension evolution*, which occurs when network nodes are updated in more or less arbitrary ways. This means that the evolution can affect the operation of existing functionality, even if such functionality was not explicitly designed to be extended (as was required for plug-in extensions). This means that in general evolutions that are transparent to the clients of a service are feasible. To our knowledge, only our current system, MANE, supports this type of network evolution.

2.3 MANE: An Active Networking Testbed for Mobile Networks

Our AN testbed, the Mobile Active Network Environment (MANE), implements the UPenn SwitchWare architecture [23] and is an evolution of the UPenn testbed, PLANet [34]. To balance flexibility, performance, and safety, the SwitchWare architecture provides users with a two-level network programming interface; lightweight packet programming and general-purpose node programming, thus unifying the two main AN approaches. Based on this archi-

tecture, PLANet implements network layer services for active networking. In PLANet, PLAN (Packet Language for Active Networks) [26] is used as the packet programming language, while Caml provides loading and linking of active extension's written in the Caml language [41].

PLAN is a special-purpose functional language for packets of a programmable network. PLAN defines a special construct called a *chunk*, which is used to describe the remote execution of PLAN programs on other nodes [42]. Chunks consist of some code, a function name to execute, and arguments for the function. When a chunk is evaluated, the named function is invoked with the arguments. Remote evaluation is achieved by injecting and evaluating a chunk on remote hosts. Chunks provide flexibility that cannot be obtained by traditional packet headers. Active packets are used as 'glue' for general-purpose node-resident *services*, and therefore do not themselves require general-purpose functionality. As a result, PLAN can be (and has been) restricted with no loss in overall functionality, but with a gain in the safety guarantees for packet programs. We will discuss PLAN in detail with an example in a later section.

In MANE, active packet programs are also written in PLAN [26]. MANE routers and their extensions (both plug-in and update) are implemented in software based on Typed Assembly Language (TAL) [43]. TAL is a cousin of proof-carrying code (PCC) [44], a framework in which native machine code is coupled with annotations such that the code can be automatically proved to satisfy certain safety conditions. A well-formed TAL program is memory safe

popcorn	Compiler from Popcorn to TALx86
talc	Type-checker for TALx86
link-verifier	Verifier for safety of a linked set of TALx86 files
assembler	Assembler for TALx86

Table 2.1: TALx86 Components

(i.e. no pointer forging), control-flow safe (i.e. no jumping to arbitrary memory locations), and stack-safe (i.e. no modifying of non-local stack frames) among other desirable properties. TAL has been implemented for the Intel IA32 instruction set; this implementation, called TALx86 [45], includes a TAL verifier and a prototype compiler from a type-safe C-like language called Popcorn, to TAL. The Table 2.1 lists the TALx86 components. To be specific, MANE is written in Popcorn, which is then compiled to TAL.

The reason for our choice of TAL, as opposed to, for example, Java (popular among AN researchers), is two-fold. First, TAL is in essence native assembly code, and therefore has a high upper-bound on performance. Second, our confidence in TAL (and PCC in general) is improved due to its relatively small *trusted computing base* [46]: only the typechecker and the runtime system must be trusted to ensure that loaded extensions are safe; the compiler of those extensions does not have to be trusted. This characteristic contrasts the Java Virtual Machine (JVM)'s that employs just-in-time (JIT) compilers [47]: not only does the Java verifier have to be trusted, but the JIT compiler (which internally converts the verified JVM code to native code) has to be trusted as

well. So while JIT-compiled Java systems are approaching the performance of native code systems, they do so at greater security risk.

MANE improves upon PLANet in a number of ways, such as hierarchical addressing, mobility emulation, and service update extensions [7]. For hierarchical addressing, we separately implemented network layer and link layer protocols. By changing link interfaces based on the addressing hierarchy, mobility emulation is possible. Enhancing a node with a plug-in or update extension is achieved through type-safe dynamic linking [48].

MANE presents a two-level namespace architecture. References in the packet to services are resolved by the service plug-in namespace, while references between plug-ins and/or the rest of the program are resolved by the program namespace. In both cases, these namespaces may be changed at runtime to refer to new entities. A benefit of this separation is that the presentation of each namespace can be parameterized by policy, for example, to include security criteria. This is useful because the division between the Active Packet, plug-in extension and update extension layers constitutes a likely division of privilege. APs are quite limited in what they can do, so we allow arbitrary users to execute those packets. However, when a packet calls a service, implemented as a plug-in extension, the privilege of the packet can be checked before allowing the call to take place [49]. Similarly, when an update extension is loaded, the privilege of those extensions that would relink against the update extension can be checked before allowing the relinking to take place. A frequent use of plug-in extensions is to extend the services available to PLAN

packets. To do this, extensions are loaded and plugged into the service symbol table. When future active packets are processed, they will reference this table, and thus have access to the new functionality. Update extensions are dynamically linked as well, but differ from plug-in extensions in that the existing node code and any existing extensions are *relinked* following the update [8]. In this way, they may 'notice' that a new version of a particular module has been loaded. This process allows us to make fundamental changes that were not foreseen by the original system implementors.

2.3.1 Network Service

From the point of view of network architecture, MANE is much like IP in key ways. MANE addresses are globally unique and hierarchical. The hierarchy is based on sub-nets of nodes and individual nodes on a sub-net can broadcast to each other, while communication with nodes on other networks must be mediated by routers. MANE routers run a conventional link-state routing protocol and although there is no support currently for multicast, it could be added using the same techniques used for IP-multicast. MANE supports a form of DHCP which can dynamically assign both an address and a default router to a node connected to a given network. MANE uses an ARP style protocol to resolve the link-level address corresponding to a network level address and there is a provision for proxy-ARP as well.

There are, of course, key differences between the IP Internet and MANE. MANE communicates using only active packets and nodes can be extended

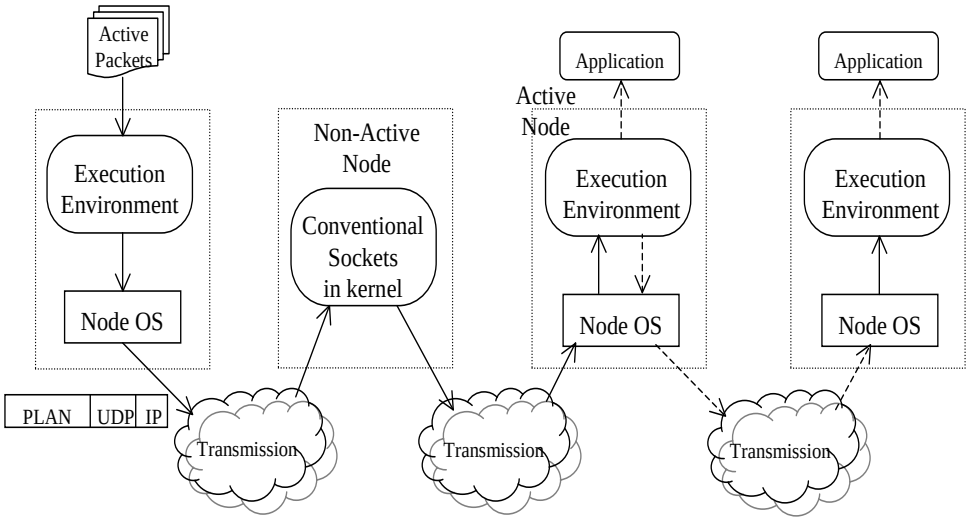

Figure 2.3: Transmission of Active Packets

and updated. To support packet programming, MANE provides certain basic services, such as a means to identify a node and to store and retrieve *soft state* based on a key. Such a soft-store is an essential service for Active Packets and is provided by many AN systems [24, 34].

At its lowest level, MANE communicates by using UDP as a point-to-point channel. PLAN programs are encapsulated inside UDP packets as payloads, as shown in Figure 2.3. On top of UDP, MANE then provides an emulation of broadcast networks. This level also provides support to emulate physical node mobility, allowing a node to leave a sub-net and to join new sub-nets. To the higher-level software, this emulation is transparent and the high level protocols assume they can broadcast to all the other nodes on their

sub-net. When a node leaves a sub-net, it no longer can directly send to or receive from nodes on the old sub-net; when it travels to a new sub-net, it can both send and receive to node on the new sub-net. This support for mobility is adequate for experimenting with mobile-IP style mobility [11, 12], but will need to be augmented to support more general mobile networks.

2.3.2 PLAN

PLAN is a special-purpose functional language for packets of a programmable network. PLAN has lightweight and limited functionality based on a restricted set of primitives and data types. Functional limitations of PLAN are supplemented with active extensions that provide service routines to PLAN programs. Therefore, PLAN is restricted with no loss in overall functionality, but with a gain in the safety guarantees for packet programs. For safety, all PLAN programs are guaranteed to terminate. Furthermore, the number of PLAN packets that can be generated from an initial packet is limited by a global resource bound, thus limiting the resources consumed in the network.

Before we describe the details of PLAN, we present how PLAN works using a simple example that performs `ping`. In the following sections, we will explain some of PLAN's primitive operations and core services, and how exception handling works. These sections may be skipped (or returned to), as they are only needed to gain a full understanding of a small number of examples later in the dissertation.

```
1: fun ping (source:host, destination:host) : unit =
2:   if (thisHostIs(destination)) then
3:     OnRemote(|ack|(), source, getRB(), defaultRoute)
4:   else
5:     OnRemote(|ping|(source, destination), destination,
6:              getRB(), defaultRoute)
7:
8: fun ack () : unit = print ("Ping Success")
```

Figure 2.4: PLAN for `ping`

A PLAN example: `ping`

Figure 2.4 presents PLAN code to test if the destination host is active. The main function, `ping()` (Line 1), has two arguments of the source address and the destination address. The other function, `ack()` (Line 8), is a simple acknowledgment function to print out the `ping` result on the source.

`Ping` works as follows: when this program is injected into the network, it executes at the source. It tests to see if it is at the destination (Line 2). Since it is not, it will execute the `else` clause (Lines 4 – 6). `OnRemote()` spawns a new active packet and provides multi-hop transmission of the packet without execution until it reaches a remote host. The first argument of `OnRemote()` is an expression of a function call. The function to be called, `ping`, is enclosed by |'s followed by the arguments, `(source, destination)`. The second argument is the remote host's address. `GetRB()` returns the *resource bound* available to the new packet. Resource bound acts much like a hop-count, and restricts packet generation. With the help of the default routing function, `defaultRoute`, this

packet is sent to the next hop on a route to the destination.

When it reaches the destination, the function call, `ping()`, evaluates the `if` statement (Lines 2 – 3). In this case, `OnRemote()` generates a new active packet to be sent to the source. This active packet performs the function call, `ack()`, and prints out "Ping Success" on the source.

Although simple, this example shows that protocols can be implemented in PLAN without defining new packet types, as well as the basic idea of using PLAN for conditional execution and how remote evaluation works.

Primitive Operations

The most important network primitives are `OnRemote()` and `OnNeighbor()`. By spawning new active packets, they enable remote computation. The list iterators, `foldl` and `foldr`, are provided as PLAN primitives rather than as service functions because PLAN does not support language-level parametric polymorphism or higher-order functions [50].

- `OnRemote`

The syntax of this primitive is:

`OnRemote(`*E, H, Rb, Routing*`)`.

The meaning of this primitive is to evaluate E on node H. Rb is the resource bound of the new active packet generated by this primitive. It acts much like a hop-count. Finally, *Routing* is a function used to decide how to route the packet to H.

E must be an expression of a function call. Syntactically, the function to be called, f, is enclosed by |'s followed by the arguments, $(a_1, ..., a_n)$. H is an expression of type `host`. Rb is an integer indicating the resource bound transferred from that of the current packet. *Routing* should be a service function of type `host` $\rightarrow$ `host` $\times$ `dev`; *Routing(H)* must return a neighbor node that is the next hop on a route to H, and the device that the packet should be sent through.

- `OnNeighbor`

This network primitive is similar to `OnRemote`, except that the child packet must be evaluated on a neighbor node. The syntax is:

`OnNeighbor(`*E, H, Rb, D*`)`.

The meaning is to evaluate E on neighbor node H. Rb is the resource bound transferred from that of the current packet. D indicates the interface name of type `dev` that the packet should be sent through. Devices are normally obtained by calling a routing function for the neighbor node.

- `foldl`

`Foldl` is the left-associative list iterator. The meaning of

$$foldl(f, a, [b_1; ...; b_n])$$

is

$$f(f...f(f(a, b_1), b_2)..., b_n)$$

where f is the name of either PLAN function or service function.

- `foldr`

`Foldr` is the right-associative list iterator. The meaning of

$$foldr(f, [a_1; ...; a_n], b)$$

is

$$f(a_1, f(a_2, ...f(a_n, b)...)).$$

Together these two iterators allow us to perform many iterative operations on lists, despite the fact that PLAN, by design, has no way to express loops.

Core Services

Services are node resident functions that can be called from PLAN. Services may be divided into two categories: the core services and additional service packages. The core services are expected to be resident on all active nodes. Each service package consists of one or more service functions that are callable from PLAN programs. Like all PLAN functions, services always return a value; a service will return `unit` if the output has no meaning. Some of the core services are as follows:

- `getRB`: () → `int`

  Returns the current amount of resource bound.

- `getSrc`: () → `host`

  Returns the originator of the packet.

- `thisHost`: () → `host list`

  Returns a list of all network addresses for devices on the current node.

- `thisHostOf`: `dev` → `host`

  Returns the network address corresponding to the given device.

- `thisHostIs`: `host` → `bool`

  Returns true if the given address matches any of the addresses of interfaces on the current node.

- `getSrcDev`: () → `dev`

  Returns the interface on which the packet arrived.

- `getDevs`: () → `dev list`

  Returns a list of all interfaces on the current node.

- `defaultRoute`: `host` → `host` × `dev`

  Returns the next hop address and the device that the packet should be sent through. By the default, this uses the RIP routing function.

- `getNeighbors`: () → `(host` × `dev) list`

  Returns the list of the neighbors attached to the same physical network.

- `length`: `a list` → `int`

  Returns the length of the given list.

- `member`: `(a, a list)` → `bool`

  Checks whether the given element is in the list or not.

- `remove`: `(a, a list)` $\rightarrow$ `a list`

  Removes the specified element from the given list.

Exception Handling

The syntax for exception handling is:

$$\texttt{try}\ exp_{try}\ \texttt{handle}\ e \Rightarrow exp_{handle}$$

where e can be either an exception literal or a variable name. If the exception e is raised and caught during the execution of exp_{try}, then expression exp_{handle} is evaluated. If e is not a literal exception, it is regarded as a variable name. Any exception raised during evaluating exp_{try} will be bound to e, and thus any PLAN exception can be caught.

Some of the language level exceptions are as follows:

- `ServiceNotPresent` is raised when a service is called, which is not resident on the active node.
- `DivByZero` is raised upon an attempted division by zero.
- `NotEnoughRB` is raised when the current packet runs out of resource bound, or if it attempts to allocate to the child packet more than available resource bound.
- `HostNotLocal` is raised when a router is asked to forward a packet to a node that is not connected.

- `NoRouteEntry` is raised when the invoked routing function does not know of a route for the specified destination.

2.4 The Performance of Active Networking

A variety of existing AN research has focused on the question of how AN systems perform. Since these issues are well understood, we have designed and implemented MANE with an eye toward flexibility and limited hardware demands, rather than performance evaluation. In particular, as described, MANE emulates wired and wireless networks at the user level using UDP communication. This greatly aids experimentation, especially on a limited hardware base not equipped with wireless network interfaces, but it also severely limits the raw performance of the system. Nevertheless, it is useful to understand the performance impacts of these technologies and so here we summarize the results already obtained by the AN community.

Three aspects of the SwitchWare architecture may compromise systems performance as compared to conventional network architectures. First, the use of Active Packets may result in a space overhead in the packets as compared to conventional packet headers. Second, there may be a execution time overhead for processing Active Packets. Third, there may be an execution time overhead for the use of Active Extensions. Each of these issues has been addressed by the AN community and we discuss each one in turn here. The bottom line is that these results strongly suggest that the overhead of using AN is small.

Space Overhead for Active Packets

In MANE, APs carry the code for the packet programs as text strings in the actual packets. This is the simplest, most direct approach, but also the one that is the least space efficient. In general, AP systems have chosen to either carry packet programs *by value*, where some representation of the code is carried in the packet itself, or, *by reference*, where only a reference that allows the actual code to be looked up is carried in the packet. The former approach has an advantage when packet programs are not used repeatedly, while the later has an advantage when the same programs are reused many times. The two approaches are compatible and we would expect mature AN systems to support both modes.

In PLAN and its followup SNAP (discussed in more detail below) packets carry programs by value. This still allows a more compact representation than carrying the actual program text. In particular, in the CAML version of PLAN [26], packet programs are carried as parse trees, while SNAP carries packets in a compact byte code representation. Still, neither of these systems made any serious attempt to optimize space use in AP's.

The most compact representation in wide use is that of the Active Network Transport System, ANTS [24]. ANTS uses Java for its packet programs and because most Java programs have very large representations, especially compared to PLAN, it was important for ANTS not to carry packet programs by value. Instead, ANTS carries programs by reference using a 64 bit crypto-

graphic hash of the actual Java code. This hash is used to lookup code in a node local cache. If the cache does not contain the program, ANTS has mechanisms to fetch the code from other nodes. This overhead of 64 bits compares quite favorably to the overhead used in conventional network architectures to identify the protocol and protocol version for the packet. Thus, in general, APs do not need to use more space than conventional packets.

Interestingly, APs actually offer the opportunity to save space compared to conventional packets. This is because of two features of conventional packets. First, because they may need to have something added to them in the future due to some initially unanticipated need, conventional packets often have "reserved" fields defined. Until they are used, these fields are simply wasted space. Since APs accommodate new features using new packet programs, they have no such wasted space. Second, another source of space waste in conventional packets are fields that are rarely used. An example is IP's fragmentation fields, which are used by only a few percent of packets in the Internet. In AN systems, packets that do not need a feature can carry programs that do not require the data used by that feature, thus resulting in lower overhead.

Execution Time Overhead for Active Packets

In most early AN systems there was a significant overhead for AP execution. There were three principle reasons for this, all of which also apply to MANE. First, these systems were implemented outside the OS kernel, requiring expen-

sive kernel crossings to process packets. Second, the representations used by these languages for packets was not optimized for rapid execution, resulting in overhead for marshalling and unmarshalling. Finally, the languages and their implementations were not optimized for execution performance. In general, these inefficiencies arose because early systems were designed and implemented to explore the basic ideas behind AN and not for high performance. In fact, it was necessary to build these initial systems to even be able to understand what factors were key in the design of more efficient AP systems.

Fortunately, two second generation AP systems have shown that all of the above overheads can be eliminated and that AP systems can be quite competitive with conventional architectures. The first of these efforts, a High-Performance Active Network (PAN) [10], showed that by implementing AP processing in the kernel and by using C as the AP language good performance could be achieved. Unfortunately, this approach compromised system safety.

A more comprehensive effort was Jonathan Moore's dissertation on Safe and Nimble Active Packets (SNAP) [51]. SNAP built on the experience gained from early PLAN implementations and most PLAN programs, including those used in this dissertation, can be compiled into SNAP programs [52]. SNAP uses a compact and efficient byte-code representation for AP programs and it transmits these packets using a layout that can be executed "in place." This eliminates marshalling and unmarshalling overheads and also reduces other transmission overheads. SNAP itself is processed by a high performance byte-code interpreter coded in C. The design of SNAP is such that it can be safely

executed in the kernel as part of the normal OS processing of packets and SNAP has been added to the Linux kernel. The result is that executing SNAP packets adds only a few percent overhead as compared to IP in Linux [9]. Thus, in general, we expect the overhead of using APs can be made small.

Execution Time Overhead for Active Extensions

The most ambitious and sophisticated AE system to date is the Dynamic Updating system [8]. As discussed, dynamic updating supports both plug-in and update extensions and it was chosen as the implementation environment for MANE because of this flexibility. Dynamic updating in turn is based on Typed Assembly Language (TAL) [43]. Since TAL provides a safe low level assembly language target for compilers there is no inherent performance penalty in its use. Thus the key question is whether Dynamic Updating introduces significant overheads. The question is addressed in detail in Michael Hicks' dissertation [53] and in the Dynamic Updating papers [54, 8]. The conclusion is that Dynamic Updating adds only a small overhead, well within the variance that different compilers introduce.

Chapter 3

Case Study I: Mobile IP

Our first case study applies AN-based adaptivity and evolution to implementing and deploying the Mobile IP protocol [11, 12]. We show how AN can be used to upgrade a network's services on the fly, without centralized coordination and without halting network service. By doing so, we are making a strong claim that AN *can* be used to quicken the pace of network service evolution. For our demonstration, we describe how to augment an active network that provides standard, IP-like service to support routing for mobile hosts, in the spirit of Mobile IP. While our primary goal is to demonstrate the capabilities of AN technology in evolving a network, a secondary goal is to explore the suitability and usefulness of AN techniques within the mobile networking domain.

3.1 Mobile IP Background

The goal of Mobile IP is simply to allow an end-node to move from one physical network (subnet in IETF terminology) to another while still communicating using its original IP address. The difficulty arises because IP addresses are used both to name a node and to identify the location of the node. This is done by embedding the subnet where the node is located in its IP address (which is to say in its name). Routers deliver a packet destined for an IP address to the subnet embedded in that address. Unfortunately, if that node has moved to another subnet, then its original subnet found in its IP address is the wrong place to deliver it. Mobile IP addresses this problem by separating the naming and location issue by using a forwarding mechanism as described below [11, 12].

Our motivation for choosing Mobile IP as an example comes because its design was severely constrained by concerns about Internet evolution. In particular, the designers of Mobile IP were concerned that it would be virtually impossible to deploy if it required changes to a significant part of the network. Thus the design was constrained so that only the mobile node itself, a router on the original subnet, and a router on the new subnet needed to be changed to support the protocol. The belief was that a user had control over the mobile node and at least some potential influence over the routers in their home networks and perhaps on the routers on their new network. It was deemed infeasible to change the internal nodes of the Internet, despite the fact that

not doing so would lead to suboptimal performance [55, 56, 57]. We will see that AN allows both much easier evolution of the original Mobile IP design, but also a relaxation of these fundamental design constraints.

3.1.1 Basic Mobile IP

Figure 3.1 shows a basic example of Mobile IP. Only end nodes can be mobile. A mobile node (MN) has a "home" network, which is implicit in its address. Even when a node is mobile, packets for it are sent to its home network for delivery. If a host is not mobile, packets are delivered conventionally. If a host is mobile, when it connects to a remote or "foreign" network, it acquires a local address from the FA. The MN then sends a registration packet to the HA on its home network with the information that it can be contacted at its newly acquired address (*care-of-address*). When a packet from the correspondent node (CN) arrives at the HA destined for the MN, the HA tunnels the packet to the FA using its address on the foreign network. There, the packet is de-tunneled and delivered to the MN.

3.1.2 Extension of Mobile IP: Route Optimization

Because the packets from the CN are routed based on the MN's home address, the base Mobile IP protocol forces all the packets for the MN to be routed through the HA. As shown in Figure 3.1, there are indirect connections from the CN to the MN through the HA. This is referred to as *triangle routing*;

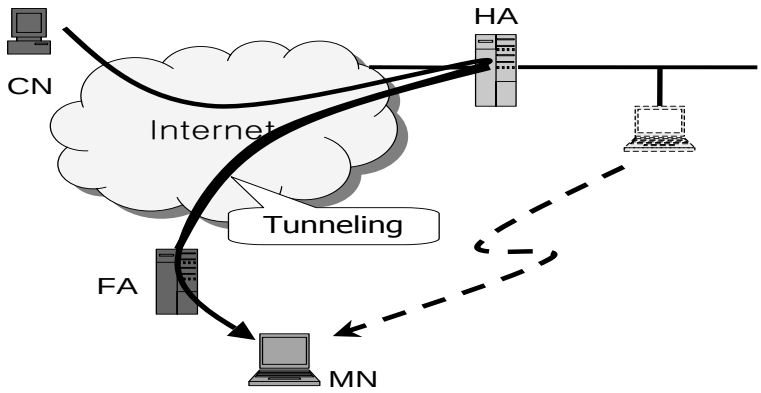

Figure 3.1: Mobile IP

packets may often be routed along paths that are longer than optimal. This indirect routing is a direct consequence the constraints in the Mobile IP design. The IP address is the key to packet routing. Meanwhile, the IP address works for both identification and location. Mobile IP supports mobility by separating the roles of the IP address and adding mobility agents on the edges of the network. Within the current Mobile IP architecture, triangle routing is inevitable; this exemplifies the evolutionary limitations of current networks.

Because triangle routing may cause significant delay and unnecessary overhead on the home network and the Internet, *route optimization* was proposed [58]. It requires changes to the CN, placing it outside the original architectural constraints of Mobile IP. By allowing the CN to cache a binding of the MN's home address and care-of-address, route optimization avoids triangle routing. If the CN has a binding entry for the MN, it can tunnel

the packets directly to the care-of-address of the MN and deliver the packets without any assistance from the HA. Route optimization is likely to improve Mobile IP performance, however, the CN must be aware of the possibility of the MN's mobility and be able to maintain a binding cache containing the care-of-address of the MN. It is quite difficult to evolve the Internet to support route optimization, because route optimization requires all possible CNs to be changed for full deployment.

3.1.3 The State of The Art: Standardization and Deployment

We discuss the current state of Mobile IP in the current network both in terms of standardization and deployment. The key point is that despite its being a well understood technology for some thirteen years, it is neither fully standardized nor widely deployed. Thus far, we have failed to evolve the Internet to support Mobile IP.

The idea of supporting mobility in the IP layer was first proposed in 1991 [11]. The first Request For Comments (RFC) for IP mobility support, RFC 2002, was published in 1996 by The Internet Engineering Task Force (IETF) [59]. RFC 2002 was obsoleted by RFC 3220 in January, 2002 [60], which was obsoleted by RFC 3344 in August, 2002 [12]. It has been thirteen years since the idea was first published; the standardization for Mobile IP is still an ongoing process [61].

Meanwhile, the Internet draft for route optimization was expired without standardization in March, 2002 [62]. Because route optimization requires all IPv4 nodes to be changed with the addition of a binding cache, it was deemed not realistic to standardize route optimization. Since the standardization of Mobile IPv6 is at an early stage, the IETF included route optimization as a "fundamental part of the protocol, rather than a nonstandard set of extensions" [63]. Unfortunately, deployment of IPv6 has also been very limited.

Even though the cellular industry has considered using Mobile IP to support wireless data service [64, 65], Mobile IP has not been deployed throughout the Internet. In order to deploy the base Mobile IP protocol in the current network, the required modifications are as follows:

- Architecture
  - New functional entities on the edges: Home Agents, Foreign Agents, and Mobile Nodes
  - MNs should maintain a constant IP address when it changes its location.
- Functionality
  - Mobility Agents (HAs and FAs) should be equipped with the following functionality:
    * Location Management

        - HAs should maintain a Location Directory for MN's care-of-address.
        - FAs should maintain a visitor list.
    * Registration: there should be a defined protocol and message formats for MN's registration.
    * Tunneling
        - HAs should be able to intercept and tunnel the packets destined for MNs.
        - There should be defined protocols, such as IP-in-IP encapsulation, for tunneling and de-tunneling between the HAs and FAs.

- Extended Functionality: Route Optimization
    - Correspondent nodes should be aware of mobility and maintain a binding cache for MNs.
    - There should be a defined protocol and message formats for maintaining that binding cache.

As we will see in Section 3.5 implementing and deploying this level of functionality is a non-trivial task.

3.2 MANE Modifications for Mobile IP

We can implement and deploy a new protocol using AN's various mechanisms discussed in Section 2.2.1. A key differentiating factor among these mechanisms concerns whether a new service is application-aware or application-transparent. Application-aware network services require that an application must be aware that it is doing so before it can use a new service. For example, using IP-style multicast requires the sending application (or perhaps the middle-ware used by the application) to send to a special multicast address. In contrast, application-transparent services are those that act without the application's knowledge. For example, in IP-style mobility, packets destined for a host's home network are transparently forwarded to that host's current remote network; the sending application does not need to be aware of mobile IP services for them to work. In making this distinction, we have realized that APs and, in many cases, plug-in extensions cannot solely provide transparent service; they require the aid of update extensions. However, the added power of plug-in and update extensions makes them a greater security risk; services would benefit from using a combination of mechanisms to balance the needs of the application and of the network.

Consider what must be added to MANE to support this protocol:

1. The home agent must be identified.

2. There must be a way to send a registration packet.

3. There must be a way to recognize when a packet arrives at the home agent.

4. There must be a way to create a tunnel.

5. There must be a way to remove the original packet from the tunnel.

The application-aware and transparent evolutions will share the same implementation for many of these functions. The shared implementation is the inherently non-transparent part of Mobile IP, including basically all but Point 3.

3.3 Active Packet Evolution

If it is acceptable for the node trying to communicate with the mobile host to be aware that the host might be mobile, then it is possible to implement the basic Mobile IP protocol discussed in Section 3.1 using only APs. Some of this implementation can be shared with the update extension evolution example (in Section 3.4). We discuss the common elements first, followed by a discussion of the aspects unique to APs.

3.3.1 Setting Up the Forwarding Path

The application-aware and transparent versions share the same infrastructure for setting up the forwarding path to a mobile host (while they differ in how packets are actually forwarded). Before a mobile node leaves its home network,

```
1: fun addMe(source:host, HAgent:host, FAgent:host) : unit =
2: (  storeTuple(source,(HAgent,FAgent,100));
3:    proxyARP(source) )
4:
5: fun register():unit =
6:   OnRemote(|addMe|(homeName(), homeAgent(), localName()),
7:           homeAgent(), getRB(), defaultRoute)
```

Figure 3.2: PLAN for Setting Up Forwarding Path

it must identify the router that serves as its home agent. For simplicity, we assume that its default router serves this purpose. Once the node has attached itself to a new network and has a unique address, it sends an AP containing a control program to register itself to its home agent.

Figure 3.2 shows the PLAN code for setting up the forwarding path. The main function, `register()` (Lines 5 – 7), generates a new packet to execute on the home agent. When executed, this program calls the function `addMe()`, and simply adds the information to the home agent's soft state keyed by the mobile node's home network address (Line 2). In addition, the function makes the home agent work as a ARP proxy for the mobile node (Line 3). Both the application aware and transparent versions share the same soft-state entries, allowing them to use the same control program and to coexist.

3.3.2 Forwarding: The Application-aware Protocol

The key questions remaining are how do we detect that a packet is at the home agent of a mobile host and how is the packet then tunneled to the

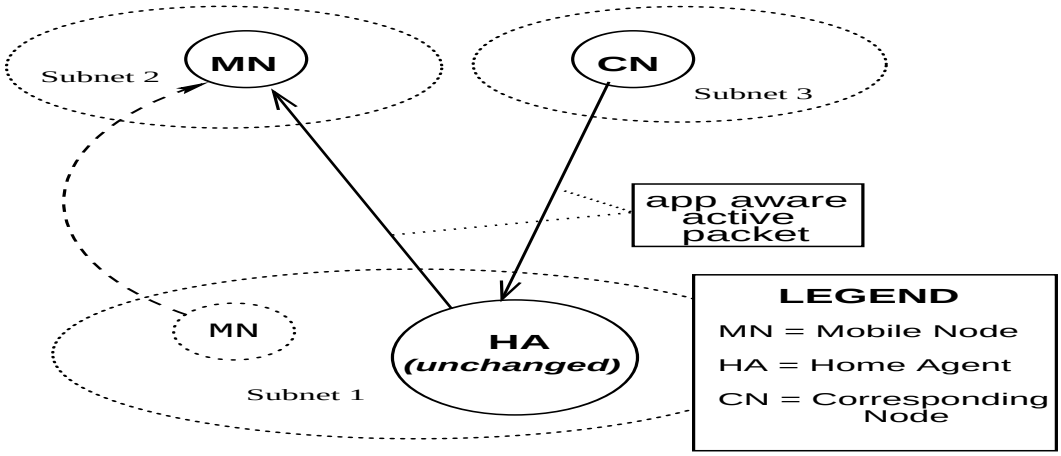

Figure 3.3: Active Packets for Mobile-IP

unique address. Because this version is application aware, both of these steps can be done by having the application use a special AP as shown in Figure 3.3.

The PLAN code for the packet that must be sent by the application is shown in Figure 3.4. `GetToAgent` is the main function and when it executes, it first looks up `dest` in the soft-store using `lookupTuple` (Line 3). If that succeeds, it has found the home agent and it uses `OnRemote` to send a new packet, the tunnel, that will execute `FoundFA` at the foreign agent with the same arguments as `getToAgent`. `OnRemote` provides multi-hop transmission of the packet without execution until it reaches the foreign agent. If the lookup fails the handle will execute. If we have actually reached the host then we deliver the packet. Otherwise, it looks up the next hop toward `dest`. It then uses `OnNeighbor`, which only transmits a packet one hop, to send the packet. Thus the packet travels hop-by-hop looking for the home agent.

```
1: fun getToAgent(dest, payload, port) =
2:  try
3:   let val agents = lookupTuple(dest) in (* (HA, FA) tuple *)
4:    OnRemote(|FoundFA|(dest, payload, port), #2 agents, getRB(), defRoute)
5:   end
6:  handle NotFound =>
7:   if (thisHostIs(dest)) then deliver(payload, port)
8:   else
9:    let val next = defaultRoute(dest) in
10:    OnNeighbor(|getToAgent|(dest, payload, port), #1 next,
11:                getRB(), #2 next))
12:   end
13:
14: fun FoundFA(dest, payload, port) =
15:  let val hop = defaultRoute(dest) in
16:   OnNeighbor(|deliver|(payload, port), #1 hop, getRB(), #2 hop))
17:  end
```

Figure 3.4: PLAN packet for Mobile-IP

Now consider the `FoundFA` function. It executes on the foreign agent, which in our case is the mobile host, but might be some other node on the same sub-net. It sends a packet to the `dest` by executing the `deliver` function. This is where the original packet is removed from the tunnel. Notice that all of that functionality is encoded in the tunnel packet program itself. The foreign agent does not need to have any knowledge of its role as a tunnel endpoint; it just has to support PLAN.

3.3.3 Route Optimization: Binding Update on CN

Although avoiding the triangle routing problem is clearly a desirable goal, efforts to achieve this goal in the existing network have been abandoned due to the difficulty of changing the CN. In this section, we show that with an

```
1:  fun getToAgent(dest, payload, port) =
2:   try
3:    let val agents = lookupTuple(dest) in (
4:     OnRemote(|FoundFA|(dest, payload, port), #2 agents, getRB()/2,
defRoute);
5:     if(!thisHostIs(getSrc()))
6:       OnRemote(|storeTuple|(dest, (#1 agents, #2 agents, 100)),
7:                      getSrc(), getRB(), defRoute)
8:    )  end
9:   handle NotFound =>
10:    if (thisHostIs(dest)) then deliver(payload, port)
11:    else
12:     let val next = defaultRoute(dest) in
13:     OnNeighbor(|getToAgent|(dest, payload, port), #1 next,
14:           getRB(), #2 next))
15:     end
16:
17:  fun FoundFA(dest, payload, port) =
18:   let val hop = defaultRoute(dest) in
19:    OnNeighbor(|deliver|(payload, port), #1 hop, getRB(), #2 hop))
20:   end
```

Figure 3.5: PLAN packet for Mobile IP with Route Optimization

almost trivial modification of our application aware implementation we can support route optimization.

In order to avoid the triangle routing by route optimization, the CN needs to maintain a binding cache for the MN; the binding cache needs to be updated with the MN's current location by the HA. In the application-aware approach, since the CN is aware of mobility and able to send special packets, we evolved our existing implementation to support route optimization using APs. Figure 3.5 presents PLAN packet supporting route optimization.

The three underlined lines of code show the changes made to our original version (see Fig 3.4). The change is simple, when a packet reaches the HA,

not only does it tunnel the packet to the FA, it also sends an update to the CN (lines 6, 7), which installs the binding in the CN's soft state. After that when the application sends a PLAN packet, it will find the binding on the CN and will tunnel it directly to the FA. The `if` (line 5) merely avoids updating the CN when it already has a binding. The rest of the code works exactly as it did in the original version. We also note that none of the other active packets need to change to include this support.

Needless to say, this change is so trivial that it was accomplished in a few minutes. However, it makes a big difference in optimizing packet routing.

3.3.4 Route Optimization: Binding Update with a Proxy Agent

With AN we can go beyond the form of route optimization unsuccessfully envisioned for the existing architecture. The version we have already shown will optimize routing between a particular CN and a MH. However, every time a new CN communicates it will have to go through the HA and then have its route optimized.

Because AN allows us to actually change the interior of the network we can do better. The idea is simple, not only do we install a binding on a CN, we also install bindings in any active routers along a path from the HA to the CN, making these routers *proxy agents* for the HA. Proxy agents are another kind of mobility agent that maintain soft state of the MN's location.

By spreading the MN's binding information through the proxy agents, we can improve the routing behavior of packets sent from CNs that have not yet sent a packet to the MN. As soon as such a packet encounters a proxy agent, it is tunneled to the FA.

Even though proxy agents are likely to enhance route optimization, it was thought infeasible to include them in the Mobile IP architecture due to the difficulty of evolving the internals of the current network. However, the AN architecture allows flexible network evolution, even in the internals of the network. We designed and implemented proxy agents that are spread throughout the network.

Figure 3.6 is the PLAN code for route optimization with proxy agents. The underlined parts were added to the route optimization packet as shown in Figure 3.5. The change is simple. As with our other packets, when a binding cache is found (line 3) the packet is tunneled to the FA. If this happens before reaching the HA then the route has been optimized. Next, instead of just updating the CN, a new packet is spawned (lines 5-7) which will single-hop through the network back to the CN, trying to install a new binding where possible. This new packet (lines 22-30) uses much the same logic as the original one for its single hopping and so was very easy to code. It took about 30 minutes to program the packet; three lines of the original packet were modified and nine lines were added.

```
1:  fun getToAgent(dest, payload, port) =
2:   try
3:    let val agents = lookupTuple(dest) in (
4:     OnRemote(|FoundFA|(dest, payload, port), #2 agents, getRB()/2,
defRoute);
5:     let val next = defaultRoute(getSrc()) in (
6:       OnNeighbor(|updateProxy|(dest, #1 agents, #2 agents, getSrc()),
7:                      #1 next, getRB(), #2 next)
8:    )  end
9:   handle NotFound =>
10:    if (thisHostIs(dest)) then deliver(payload, port)
11:    else
12:     let val next = defaultRoute(dest) in
13:     OnNeighbor(|getToAgent|(dest, payload, port), #1 next,
14:           getRB(), #2 next))
15:     end
16:
17:  fun FoundFA(dest, payload, port) =
18:   let val hop = defaultRoute(dest) in
19:    OnNeighbor(|deliver|(payload, port), #1 hop, getRB(), #2 hop))
20:   end
21:
22:  fun updateProxy(mn, ha, fa, cn) =
23:   ( try
24:    storeTuple(mn, (ha, fa, 100))
25:   handle ServiceNotPresent => ();
26:   if(thisHostIs(cn)) then ()
27:   else
28:    let val next = defaultRoute(cn) in
29:     OnNeighbor(|updateProxy|(mn, ha, fa, cn), #1 next,
30:           getRB(), #2 next) end )
```

Figure 3.6: PLAN packet for Mobile IP with Proxy Route Optimization

3.4 Update Extension Evolution

A potential problem with the evolutions shown so far is that some part of the system must send special packets to take advantage of the new service. In some applications, being aware of new packets or services is acceptable, while in others it is not. The latter is true for Mobile IP: it would be unreasonable to change all possible senders on the network to use our special packets from Section 3.3 to send to a potentially mobile host. In this section, we demonstrate how using update extensions, we are able to evolve the network so that forwarding is transparent to the sender and does not require using a special packet.

The basic strategy is shown Figure 3.7. Here a packet that is not aware of mobility is destined for a mobile node. However, because we have updated the Home Agent to support transparent forwarding, it is able to intercept the packet and tunnel it to the mobile host. Thus, although we use Active Packets and plug-in extensions to help perform our evolution in a convenient way, the key to transparent evolution is really the use of update extensions.

Because mobility is inherently not transparent to the mobile host itself, we can reasonably have it set up the forwarding path to the remote agent as described in Section 3.3.1, with the added benefit that the nontransparent and transparent techniques can coexist.

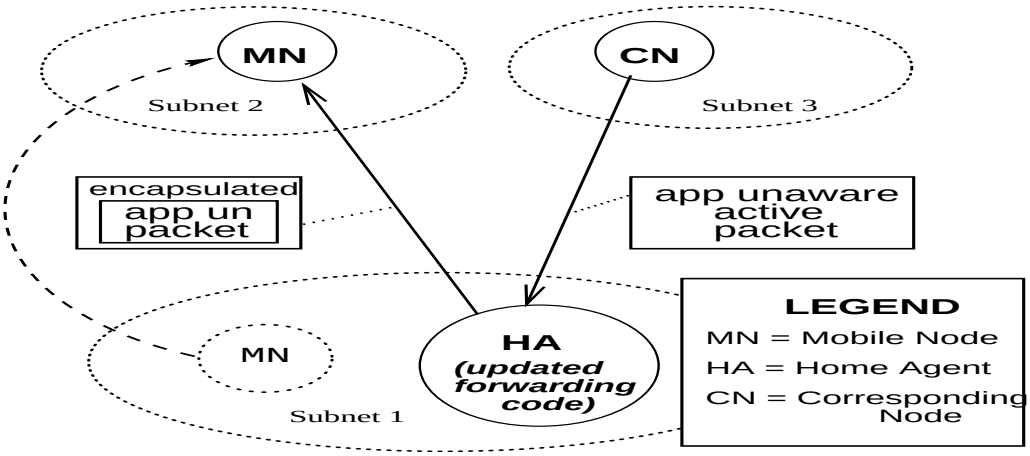

Figure 3.7: Update Extension for Mobile-IP Evolution

3.4.1 Forwarding: The Application-transparent Protocol

To make the forwarding transparent to the sender requires a way to detect that a packet has arrived at the home agent and to forward it to the foreign agent *without having to rewrite the sender's packet code.* The most straightforward way of doing this is to modify the router's forwarding logic: whenever a packet arrives, look up its destination address in the soft state that is used to record which hosts are mobile, and if present, forward the packet to the foreign agent. Pseudocode for router forwarding logic is shown below (in a C-like syntax), with the additional part shown in italics:

```
void sendToNextHop(pkt_t packet, host_t dest){
  host_t nextHop = Route(dest);
```

```
  if is_mobile_host(dest) then
    tunnel_to_foreign_agent(packet, dest);
  else
    send_on_link(packet, dest, next_hop);
}
```

Note that the code implementing `is_mobile_host` and `tunnel_to_foreign_agent` would be implemented elsewhere and loaded separately.

While the addition of an if-statement to the forwarding loop is conceptually simple, it is impossible to realize *on the fly* without the support of update extensions. This is because the forwarding loop in MANE was not designed for change; that is, it did not provide a plug-in interface for performing new operations in the loop.[1] As a result, effecting this change would require changing the code statically and recompiling, and then bringing down the node and restarting it with the new code. This is practical when only a few nodes need to be updated, but much less so if an evolution needs to be widespread.

On the other hand, the power of update extensions makes them more dangerous. For example, the added conditional in the forwarding loop above will be invoked for *all* packets, even those not interested in mobility. In an active packet-only system, the needs of one packet will not interfere with another's in this way. Similarly, allowing multiple users update arbitrary parts

[1] We could imagine designing the forwarding loop to allow for extensibility, as is the case in CANES [29]. However, there will always be parts of the system that were not coded to anticipate future change, and therefore will lack a plug-in interface. As a result, these parts of the system can only be updated if with update extensions.

of the router's code could result in incompatible changes, and/or an unintelligible code base. As such, update extensions will likely be limited to privileged users, limiting their applicability.

Implementing this change as an update extension in MANE requires two actions. First, dynamically load and link a mobility module that implements the test of whether a packet needs to be forwarded as well as providing the tunneling code. Second, dynamically load a new version of the forwarding code that does the required test and then update the old running code with the new version.

One interesting point remains. How is the tunnel itself created? Essentially, it is created in the same way as in the non-transparent case, although obviously done by the node resident mobility code. A new AP is created which when executed on the foreign agent unpacks the original packet and delivers it. Note that tunneling this way works even when the packet being tunneled is not active and thus again avoids the need for the foreign agent to act explicitly as the tunnel end-point.

3.4.2 Transparent Proxy Agents

Here we discuss how we added the proxy agents from Section 3.3.4 so that even when the CN is not aware of mobility (and even may not be active) we can still achieve route optimization. The more limited form of route optimization in which only the CN has a binding cache is just a special case of what we describe here.

We implemented a new AE which not only transparently forwards packets to the FA, but which also sends an AP single hopping back toward the CN. As this active packet is executed on active routers along its path, may encounter two cases. The first case is that the router already has the proxy agent AE installed. In this case, the AP simply adds the information about the MH and FA to the binding cache and moves to the next active router closer to the CH. The second case is that the router does not already have the proxy agent AE installed. In that case, assuming that security concerns allow it to, the AP installs the proxy agent AE using dynamic updating. Note that this is done on the fly without taking the router down. The eventual effect of this is that the proxy agent AE will be gradually spread throughout the network on an on-demand basis. If the only node that allows the AE to be installed is the CH then we have the more restricted version of route optimization envisioned for the current architecture, except that now we actually have a way of deploying the new functionality and in an incremental way.

Our original AE for application-transparent Mobile IP was 84 lines of Popcorn. Our new extension that supports proxy agents and dynamically deploying itself required an additional 46 lines of Popcorn to implement. The additional coding took less than a day. Also notice that by using dynamic updating, not only can we evolve nodes that have neither extension loaded, but we can also evolve nodes that had our original extension loaded so that they have the proxy agent functionality.

3.5 Advantages of AN for Evolution

In this section, we support our thesis by directly comparing the evolvability of the traditional network to that of a network with support for AN. We make this comparison in terms of standardization, implementation, deployment, and the ability to evolve the base Mobile IP protocol to support route optimization. In each of these areas AN makes evolution easier than in the current architecture.

3.5.1 Standardization

Traditional Arch. Since the idea of supporting mobility in the current network was first published in 1991 [11], there have been many efforts to standardize the Mobile IP protocol [59, 60, 12, 61]. After the first Request For Comments (RFC) for Mobile IP, RFC 2002, was published as a *standards track* RFC in 1996, there have been changes in the standard document [60, 12, 61]. The current one, RFC 3344, was published in August, 2002 [12] and, in fact, the standardization for Mobile IP is still in progress [61]. Further, the effort to evolve the base Mobile IP to support route optimization was abandoned without standardization in March, 2002 [62]. Because route optimization requires all IPv4 nodes to be changed, it was deemed not realistic to standardize route optimization.

AN Arch. In a basic way, the highly dynamic nature of AN side-steps the need for standardization. At its heart, the need for standardization is moti-

vated by the need to interoperate; all participants in a protocol agree on the standard and so they may interoperate. For the application-aware implementations, all that is needed to make the protocols interoperate is that the soft state stored at the HA by the MN be of the form expected by the packet sent by the CN. This is just a matter of sending compatible APs. For the application-transparent version, it is even easier. All that is needed is for the packets sent by the MN and the HA to be compatible. Since the MN has the same home network as the HA, this requires coordination only at that network. It is quite possible to imagine that different home networks might have different versions of Mobile IP, for example to handle issues of scale differently.

Another advantage that AN has is that in the AN architecture, we can implement new protocols in a lightweight way using packet programming and it is not hard to change network infrastructure by dynamically linking and updating active extensions. Therefore, without the fear for the difficulty of network evolution, we can deploy new protocols and evolve them on an as-needed basis. This means that the need to get the protocol "right" is greatly reduced. This need is one of the issues that makes the current architecture hard to standardize; the fear of deploying a broken protocol.

Conclusion Because AN makes much simpler demands on the standardization process, with respect to standardization using AN makes evolving the network significantly easier.

3.5.2 Implementation

To implement Mobile IP, we need mobility agents (HAs and FAs) with the following functionality;

- Registration

  Traditional Arch. There should be a defined protocol and message format for the registration process. The protocol is embodied by software that understands the message format and follows the associated actions. It takes quite a long time to implement static software containing all the message formats and associated actions altogether.

  AN Arch. Mobility agents use the existing soft state facilities to maintaining the MN's mobility binding. Further, the packets that perform registration are actualized by packet programming, thus it is not necessary to define the message formats in advance. Furthermore, it is easy to add message types on an as-needed basis.

- Tunneling

  Traditional Arch. HAs should be able to intercept and tunnel the packets destined for MNs. The routing program at HAs needs to be changed to look into the location directory before forwarding the packets to the home network. HAs must implement one or more protocols for

tunneling, such as IP-in-IP encapsulation [66]. FAs should be able to support location management and packet relaying for the visiting MNs. All of above functionalities need to be implemented and included in the software stacks of HAs and FAs.

AN Arch. In both approaches, all that is needed for the FA is that it support packet execution, a basic assumption of the approach. In the application-aware approach, no special node resident functionality is necessary. Since the APs contain the program for tunneling, home routers and foreign routers are just required to be able to execute packet programs.

In the application-transparent approach, recognition that a packet needs to be tunneled is done by node resident code. But this code can be dynamically added to existing routers. The tunneling and detunneling itself is implemented by packet programming using chunks. No special predefined formats or software for processing these formats are needed.

We can also consider the complexity of the implementations. As the standardization of Mobile IP proceeds, there have been several implementations of the Mobile IP protocol [67]. One of the most widely available implementations under a BSD-style license is the Monarch Project's Mobile IPv4 implementation [68]. Table 3.1 presents comparison of Monarch Project's implementation and our MANE implementation.

Criteria	Monarch Project's	MANE
Line of Code for the base Mobile IP	12815 in C	App.-aware: 0 in Popcorn + 40 in PLAN App.-trans.: 84 in Popcorn + 14 in PLAN
Line of Code for the Extension of Route Optimization	1483 in C	App.-aware: 0 in Popcorn + 3 additional in PLAN (43 in PLAN) App.-trans.: 46 additional in Popcorn (130 in Popcorn)
Line of Code for Proxy Agent	N/A	App.-aware: 0 in Popcorn + 3 modified in PLAN + 9 additional in PLAN (52 in PLAN) App.-trans.: 130 in Popcorn

Table 3.1: Comparison of Mobile IP Implementations

Clearly a great deal more code was needed to implement Mobile IP in the traditional manner. This is not surprising given the significantly greater number of things that needed to be implemented as discussed above. We do not know how long it took for the traditional implementation. However, it took less than a week to implement both versions of the base AN implementation and about a day to implement all three versions of route optimization. It seems highly unlikely that 76 times more C code was completed in this amount of time.

Conclusion It is hard to draw strict conclusions about the implementation of these systems because there are many variables. However, it would appear

that AN admits significantly shorter and simpler implementations. Simpler implementations are easier to create and change and thus it is easier to evolve the network from an implementation point of view.

3.5.3 Deployment

In the current network architecture, deploying a protocol before standardization is neither realistic nor efficient. Since the standardization of Mobile IP is still in progress [61], there is no real deployment throughout the network. In order to deploy the Mobile IP protocol, the current network needs the modifications described in Section 3.1.3. The following shows how the needed modifications might be deployed in both the traditional architecture and the AN architecture.

- Deploying New Functional Entities for Mobile IP

  Traditional Arch. There is a need to deploy special routers, in particular Home Agents and Foreign Agents, throughout the network. This would take significant time in the current Internet.

  AN Arch. In the application-aware case, the new functionality is "deployed" just by sending the appropriate packets. In the application-transparent case, using dynamic updating the HAs can be deployed on the fly without disrupting the network. There is no need for special

FA functionality. This only takes as long as the time to transmit the extension and apply the update.

We can get a crude bound on how fast this deployment might take for each architecture in the following way. For the traditional architecture to work at all on a minimal set of nodes requires changing the network implementations on the HA, FA, and MH. At a minimum, this requires downloading new kernel updates, applying them, and rebooting for each node. This will certainly take tens of minutes. Further, it will result in an interruption of service on all of these nodes. And it will only allow mobility to the network with the updated FA.

We measured the time need to deploy our base AE. It took 20 msec to transmit it one hop and 2.2 secs to apply the update. Neither of these operations disrupted the functioning of the node and when we were done, we had support for mobility from the updated home network to any foreign network.

Conclusion Thus far deploying Mobile IP in the existing network has not been feasible. Even for a minimal deployment, the current network will take 10's of minutes to deploy at best, as compared to seconds for AN. From the deployment point of view AN provides superior evolution ability.

3.5.4 Evolution of Route Optimization

Finally, adding route optimization is an example of evolution in and of itself. This section compares the traditional architecture and AN architecture for how easy this evolution is.

Traditional Architecture

For route optimization, the CNs must be aware of mobility and maintain a binding cache of the MNs' location. In other words, since all nodes are potentially CNs, all IPv4 nodes must be changed to keep a binding cache and support tunneling according to the binding information. Furthermore, there needs to be a protocol and message formats for registration of the MN's binding information at CNs. In addition to changes in all IPv4 nodes, additions to the protocol and message types at mobility agents need to be implemented. Because route optimization requires all IPv4 nodes to be changed, the IETF abandoned the standardization of route optimization for Mobile IPv4 in March, 2002 [62].

Due to architectural limitations, the only way of achieving route optimization is through the CNs' participation. Without further architectural changes, it is impossible to support route optimization transparently to the CNs. If these changes were to be made, they might take a form much like our proxy agents. Their implementation would require another process of standardizing and engineering protocol extensions and message types. From the

experiences with HAs and FAs, it is a reasonable guess that it will take significant time to deploy them into the network. Further, doing so would require making changes to the internal nodes of the network, which was considered to be difficult enough that it was ruled out in the initial architecture.

Active Networking Architecture

In order to claim the ease of AN's evolvability as applied to Mobile IP, we demonstrated three evolutionary implementations for route optimization. It took about a day to implement all three evolutions.

- **Application-Aware Route Optimization** This is an evolution of the base application-aware approach. While, it is hard to evolve Mobile IP in the traditional architecture, we could achieve this evolution by modifying the packet programs with a small number of additional lines. Active packets from the CN contain additional operations for updating the binding cache at the CN. As shown in Section 3.3.3, route optimization is achieved by only 3 additional lines in PLAN. It took only 5 minutes to change the packet program.

- **Application-Aware Route Optimization with Proxy Agent** In order to further reduce routing inefficiency, we demonstrated an additional optimization by introducing proxy agents. In the application-aware version this required no modification to node resident code. As describe in Section 3.3.4, we implemented an active extension for the new function-

ality in less than 30 minutes. The application-aware active packets can accomplish the necessary evolution with 9 additional lines and 3 modified lines in PLAN.

- **Application-Transparent Route Optimization with Proxy Agent** In this evolution, we demonstrated an evolution from the base transparent case. Even in cases where we cannot update CNs, proxy agents can reduce routing inefficiency by getting a MN's binding information closer to the CN. It is feasible to deploy proxy agents into the network on an as-needed basis using dynamic updating. Active packets conveying both the modified extension and MN's binding information update either the active extensions or the binding cache at proxy agents. For transparent route optimization with proxy agent functionality, we implemented the new MANE extensions with an additional 46 lines in Popcorn in about a day.

Conclusion

The IETF has abandoned evolving Mobile IP to support route optimization as infeasible. Using AN, not only is it feasible, but methods like the use of proxies, which the IETF consider infeasible, are feasible. Clearly AN facilitates evolution of route optimization.

3.6 Lessons for AN

Although we have focused our discussion of evolution on comparing AN to the currently deployed architecture, it is also important to consider what lessons we have learned that inform the AN community. In particular, we have explored two different styles of evolution: Active Packet Evolution and Update Evolution.

Active Packet Evolution The chief advantage of Active Packet evolution is that it is lightweight and allows third parties to enhance the functionality of the network without changing the nodes themselves. Thus, from the point of view of security, this style of evolution is the most desirable and gives the widest variety of users the ability to evolve the network. One disadvantage is that it is inherently not application transparent. Another key disadvantage is that if the existing node interface does not support some critical piece of functionality, it may be impossible to achieve the desired result. Despite this, our example (and others, e.g. [69, 70]) shows that even with only very basic services, non-trivial applications are feasible. An interesting challenge to the AN community is to design a set of node-resident services that maximizes the range of evolutions that can be achieved with just APs. Since, as in our example, the APs can often embody a substantial part of the control aspect of a protocol, this effort would be quite different from typical protocol design and would need to focus on providing the generic components that support the aspects of a variety of protocols that can not be expressed in the packets.

Interestingly, just the simple soft state provided by ANTS [24] and our system, is already a significant step in that direction.

Update Evolution The example here is the first example of update evolution in AN, chiefly because MANE is the first AN system to support such evolution. The advantage of this approach is clear: application-transparent evolution can be achieved even when the system design has not anticipated the need for a particular kind of change. In some sense, this embodies the entire goal of AN. There are two disadvantages. One, dynamic updating is not a widespread technology like dynamic loading, though it can be conceptually simple to implement [8]. Second, and more importantly, the power of update extensions implies the need for greater security and reliability considerations than for plug-in extensions or active packets. In the short term, this means that only privileged users with access to the entire router code base should make use of this technology. In the long term, more research is needed to understand how to manage multiple updaters of the same code and ways to limit their system-wide effects.

Based on the taxonomy in [7], we presented two kinds of Mobile IP examples that illustrate what expressibility gains are possible as successively more powerful techniques are used. However, these gains in expressibility are balanced by the greater security risks of more powerful techniques. Greater security risks imply that fewer users may deploy a system. Thus a basic design principle for AN systems should be to use the least powerful evolutionary mechanisms possible so as to maximize the range of users that may deploy a system.

Chapter 4

Case Study II: Ad Hoc Routing

Our second case study applies AN-based adaptivity and evolution to routing in ad hoc networks. In mobile ad hoc networks (MANETs), there is no fixed infrastructure (or routers) and all the nodes may move with any frequency; thus path changes and link failures can happen frequently [13]. Perhaps the most interesting case of such networks is to support mobile nodes communicating wirelessly. As the nodes move around, link conditions between them may change frequently and routing needs to cope with those variations nimbly. Moreover, adhocness can lead to routing heterogeneity where different parts of the network should use different routing algorithms; thus the capability for adding a new protocol or evolving old protocols promptly is required.

In this case study, we show how these issues may be better addressed by using the adaptability available in an AN framework. Because active packets can be used to deploy a routing protocol on the fly, AN-capable nodes can

agilely switch between routing protocols. By selecting an optimal protocol depending on node mobility and traffic activity, ad hoc networks can achieve better performance and reduce routing overhead. Also, active extensions allow rapid and easy expansion of the node resident parts of routing protocols. Therefore, not every node needs to be equipped with multiple protocol stacks in order to overcome discrepancies in routing protocols.The work in this chapter appears in the proceedings of International Working Conference on Active Networking (IWAN) 2004 [71].

4.1 Background

Unlike infrastructure-based wireless networks (e.g., cellular networks), mobile ad hoc networks (MANETs) require neither specific infrastructure nor fixed network configurations. MANETs are self-organizing without prior planning or deployment. When it is too costly or impossible to construct infrastructure (e.g., in battlefields) or there is a need to make or tear down a network promptly (e.g., in conventions or rescue operations), MANETs are a cost-effective alternative. While MANETs are efficient in low-cost and rapid deployment, they are unstable – partly because of node movement, partly because of the inherent characteristics of the wireless medium, such as intermittent connections and high bit-error rates. They may suffer from significant changes in topology, link status or capacity. In order to deal with these variations in MANET environments, there is a need for adaptability and flexibility,

especially in routing protocols.

In a MANET, since the range of wireless transmission is limited, there is likely to be a need for multi-hop paths between a source and a destination. Thus every node is expected to participate in forwarding packets and to act as a router [72]. Depending on the readiness of route information, ad hoc routing protocols are classified as either proactive (*table-driven*) or reactive (*on demand*) [73]. While the proactive protocols maintain routing tables through the periodic exchanges of routing information, the reactive protocols acquire route information on demand. Proactive protocols are thought to be inefficient due to excess routing overhead in frequently changing environments. Reactive protocols have an inherent delay for route discovery and require buffer space in the network layer for data packets waiting to be transmitted [74]. In general, there is a trade-off between delay and routing overhead. Some hybrid protocols combine the two approaches, but they require special preconditions, such as network hierarchy or Global Positioning System (GPS) capability. Some of the proposed protocols are listed in Table 4.1.

Each of the proposed protocols has its own unique advantages and disadvantages [73, 81]. These differences suggest that each protocol has a MANET environment for which it is the optimal routing choice. Unfortunately, it is difficult to pre-determine the optimal routing protocol for the whole feasible operational range of an ad hoc network. There is no "best" routing protocol for all network configurations. Ideally ad hoc routing protocols would adapt to changing network conditions.

Proactive	- DSDV (Destination-Sequenced Distance Vector Routing) [75] - OLSR (Optimized Link State Routing Protocol [76] - FSR (Fisheye State Routing Protocol) [77]
Reactive	- DSR (Dynamic Source Routing) [78] - AODV (Ad hod On-Demand Distance Vector Routing) [79]
Hybrid	- ZRP (Zone Routing Protocol) [80]

Table 4.1: Ad Hoc Routing Protocols

Furthermore, since MANETs are created without prior planning, it is entirely possible that the nodes comprising the MANET are heterogeneous in terms of routing. That is, all the nodes may not be equipped with the same routing protocol, and yet it may be necessary for the nodes to conform to a unified routing protocol. Without prior knowledge about various routing protocols, this would compromise the desired ability for any node to be able to discover a route to any other reachable node in the network.

In addition to sub-optimal routing and heterogeneity problems, routing in ad hoc networks requires more systematic interactions across several layers, not just the network layer. Performance of a routing protocol depends on various network conditions, such as host mobility, connection activity, or traffic patterns, etc. It seems likely that cross-layer interactions are important in mobile ad hoc networks. This view is supported by results showing that simulation factors in physical layer modelling affect the performance of higher-level protocols such as routing [82].

4.1.1 Reactive Ad Hoc Routing

In this section, we will describe two typical protocols of reactive ad hoc routing: Dynamic Source Routing (DSR) and Ad hoc On-Demand Distance Vector (AODV).

DSR The Dynamic Source Routing (DSR) protocol [78] is an on-demand routing protocol that searches for a *source route* (the sequence of nodes that the packet should visit) to a destination by *flooding* ROUTE REQUEST packets on the network only when the initiating node has data packets to send. Source route information is gathered in the packet header as the ROUTE REQUEST packet is forwarded. In order to reduce the routing overhead and make the best possible use of route information, each node maintains a *route cache*. Since there is no need for periodic 'hello' packets or neighbor detection packets, DSR is simple and has low overhead under light traffic load conditions. It is also possible to find multiple routes for a destination with one ROUTE REQUEST packet flooding and to work over unidirectional or asymmetric links.

DSR is composed of two operations: *Route Discovery* and *Route Maintenance*:

Basic Route Discovery When a source node, S, has data to send, it first searches for a valid, previously discovered source route to a destination node, D, in its route cache. If a valid route is not found in the route cache, S broadcasts a ROUTE REQUEST packet to its directly connected neighbors. All intermediate nodes will re-broadcast the first instance

of a ROUTE REQUEST that is seen, after appending its own address to the *Route Record* field of the ROUTE REQUEST packet. Eventually the request packet will reach D, and the Route Record field of the ROUTE REQUEST packet will contain a source route from S to D. Assuming that the links are symmetric, D obtains the source route to S by reversing the Route Record; and returns the Route Record to S in a ROUTE REPLY packet, which follows this source route. This mechanism also works when there are asymmetric links. D can find reverse source route to S by either flooding its own ROUTE REQUEST or piggybacking the ROUTE REPLY on its own ROUTE REQUEST packet to S.

Optimized Route Discovery Intermediate nodes can learn of local topology simply by peeking in the ROUTE REQUEST and ROUTE REPLY packets that they are asked to forward. Since the Route Record field of the ROUTE REQUEST packet contains a source route from S, every forwarding node can get a reversed source route to S. Also, a source route between S and D can be found in ROUTE REPLY packets. Moreover, neighbor nodes can learn source route information by overhearing routing packets sent by other nodes. Now, intermediate nodes are allowed to generate ROUTE REPLY packets. They search for a source route in their own route caches before they rebroadcast a request; if there is a valid route in the cache, they reply but do not rebroadcast the request. It is also possible for the nodes to update the route caches for S or D during

forwarding of ROUTE REQUEST and ROUTE REPLY, respectively.

Route Maintenance It is possible for links in a source route to be cut off at any time due to node movements or changing link conditions. When using a source route, S is notified of the link failure by a ROUTE ERROR packet generated by the node adjacent to the broken link. S deletes routes containing the broken links from its route cache, and initiates a new route discovery process unless there is another valid route to D in the route cache. S spreads the stale information by piggybacking the ROUTE ERROR packet in its next ROUTE REQUEST packet to prevent the neighbors from generating ROUTE REPLY packets containing the same invalid link.

AODV The Ad hoc On-Demand Distance Vector (AODV) routing protocol [79] is an on-demand version of the Destination Sequenced Distance Vector (DSDV) routing protocol [75]. As in DSR, routes are discovered on an as-needed basis and routing information is maintained in the routing table only as long as they are necessary. However, while DSR's route caches keep whole paths to a destination, AODV's routing tables maintain next hop addresses along with other information, such as destination sequence number, hop count, a precursor list, and expiration time. Sequence numbers are used to discern stale routes and maintain route freshness. Hop count indicates a distance to the destination, which is used to calculate the shortest path. Precursors are the neighboring nodes that use the entry in order to forward data packets.

Unlike DSR, AODV specifies the lifetime of route entries with an algorithm for estimating the expiration time.

Though it is not specified this way in the RFC [83], AODV can also be divided into Route Discovery and Route Maintenance. Message formats defined for AODV are similar to those of DSR: Route Request (RREQ), Route Reply (RREP), and Route Error (RERR).

Route Discovery When a source, S, needs route information to a destination, D, and there is no valid route entry for D in its routing table, S broadcasts a RREQ to the neighbors. To reduce the overhead of RREQ broadcasting, AODV may use an *expanding ring search*, in which increasingly larger neighborhoods are searched through control of a Time-To-Live (TTL) field in the RREQ packets. When a node receives a RREQ, it first updates or creates reverse route information for S in its routing table. If the receiving node is either the destination or a node whose routing table has valid information for D with a larger or the same destination sequence number as in the RREQ, it sends a RREP to S without re-broadcasting the RREQ. Otherwise, it re-broadcasts the RREQ increasing the hop count by one and decreasing the TTL by one. As the RREP passes back to S, the intermediate nodes set up the forward route to D by updating their routing table with the valid destination sequence number, hop count, precursor list and lifetime for D.

Route Maintenance As in DSR, a node detecting a link break generates

and sends an RERR message to the precursors. Precursors can be regarded as next hops for delivering an RERR. Depending on the number of precursors, the RERR is either unicast or broadcast. A node receiving an RERR updates entries of the routing table and forwards the RERR message to the precursors of the entry. An entry for the unreachable destination is marked invalid with an updated lifetime, after which the entry will be deleted from the routing table. The lifetime field of the routing table functions as both the expiration time for a valid route and the deletion time for an invalid route.

Comparison of DSR and AODV

Though DSR and AODV are both on-demand routing protocols and share similar behaviors [84], they have different mechanisms each with its own merits [81]. Table 4.2 summarizes the differences between DSR and AODV.

Since DSR may allow both multiple replies and multiple route entries in the cache, a source can acquire significant knowledge about topology with one ROUTE REQUEST packet. DSR's aggressiveness in route discovery and maintenance has beneficial effects on throughput and delay when mobility is low. On the other hand, high mobility may cause DSR cache entries to become stale rapidly, and aggressiveness can turn into a liability.

While DSR is simple and does not specify requirements for timer use, AODV attempts to be conservative in maintaining route table by utilizing various timer values. However, in calculating the timer values, AODV requires

	DSR	AODV
Route Discovery	On-Demand	On-Demand
Route Management	Aggressive Route Cache	Conservative Route Table with Lifetime
Route Entry	Multiple, Source route	One, Next hop
Dominant Message	RREP	RREQ
Merits	Asymmetric link Support	QoS Support
Requirements for optimization	Promiscuous mode	Network parameters

Table 4.2: Comparison of DSR and AODV

network parameters, such as the network diameter and node traversal time [83].

4.1.2 The State of The Art: Standardization and Deployment

The IETF Working Group (WG) for Mobile Ad-hoc Networks (MANET) [85] had focused on various MANET problems, performance issues and related candidate protocols; it published an *informational*[1] RFC 2501 in 1999 [72]. After that, it has dealt with standardization of a number of routing protocol specifications. Currently, there are three *experimental* documents to be considered standard ad hoc routing protocols [83, 87, 88].

Since the DSR protocol was first published in 1996 [78], several Internet-

[1]In the Internet standardization process, each RFC has a status: Informational, Experimental, or Standards Track (Proposed Standard, Draft Standard, Internet Standard), or Historic [86]. RFCs with Informational or Experimental status can be regarded as a guidance rather than official standards.

Drafts for DSR have been proposed [89]. However, standardization is still in progress and currently the 10th draft is awaiting standardization [90].

The AODV protocol was proposed in 1999 [79]. After several Internet-Drafts were renewed [91], RFC 3561 was published as an *experimental* document for AODV in 2003 [83]. From the viewpoint of the Internet standardization, the work on AODV is still in progress [92] and it is expected that a *standards track* document for AODV will be published eventually.

There have been several implementations of DSR [93, 94, 95] and AODV [96]. Even though there have been several experimental ad hoc networks [97, 98], neither DSR nor AODV have been deployed in real networks.

Generally, in order to support reactive ad hoc routing protocols, the required modifications are as follows:

- Architecture
  - Every node should work as a router and cooperate in multi-hop routing.
  - The network layer must maintain a routing buffer to keep packets during route discovery.
  - The link layer should be in close cooperation with the network layer; for example, during connection, the link layer monitors link status for route maintenance.
- Functionality

- The network layer executes routing protocols on demand.
  - A source node initiates route discovery by broadcasting a route request packet.
  - Intermediate nodes participate in route discovery by either propagating the request or replying to it.
  - During connection, if a node in the route detects link breakage, it sends a route error packet to the source.
- The link layer informs network layer of link status during connection.
  - By using either existing link-level acknowledgements or passive acknowledgements, the link layer monitors link status.
  - When a link is broken, the link layer protocol notifies the network layer so that the network layer performs properly for route maintenance.

In summary, even though the need for ad hoc networks and multi-hop routing protocols is widely acknowledged, standardization for ad hoc routing protocols is only slowly progressing. Further, there has not been real deployment of any ad hoc routing protocols.

4.2 MANE Modifications

Here, we describe the modifications of MANE needed to support ad hoc networking in general and in particular to support on-demand routing protocols.

4.2.1 Addressing

Like IP addresses, MANE addresses are globally unique and hierarchical. A node is identified by a network number and a host number. The hierarchy is based on sub-nets of nodes and each node on a sub-net can broadcast to all other nodes. Communication with nodes on other networks must be mediated by routers. Based on this hierarchy, we have already made MANE support Mobile-IP-like mobility by utilizing AN's evolution techniques as discussed in Chapter 3. For ad hoc networks, where each node works as a router, MANE uses a flat addressing scheme. All the nodes in an ad hoc network have the same network number, and host numbers are used as a unique address.

4.2.2 Mobility Emulation

MANE emulates broadcast networks by keeping track of which nodes are on a particular sub-net and using UDP to communicate between neighbors. Broadcast is achieved by repeatedly unicasting to every neighbor. This mechanism also supports emulation of physical node mobility, allowing a node to leave a sub-net and to join new sub-nets. Even though this emulation is transparent to a higher level, MANE needed to inject special APs to disconnect and

connect a node [7].

For ad hoc networks, we need a more scalable and better distributed way of emulating physical mobility. Therefore, we adopted a method similar to that used by `ns` for wireless network simulation [5]. There is a pre-generated mobility file emulating node movements. Also, there is a virtual master node with a global "god's eye" view, whose role is to update neighbor lists by sending neighbor information packets periodically to every node. The virtual master obtains neighbor information from the mobility file. Neighbor information is used only in emulating physical mobility and wireless link broadcasting, not in network-layer routing.

4.2.3 Routing Buffer in the Network Layer

Since we are experimenting with reactive routing protocols, there needs to be a buffer space - *the routing buffer* - to save the data packets during route discovery. When routing information is available, the corresponding packets are released from the routing buffer and pushed into the lower layer queue for transmission. To support reactive routing protocols, we implemented the routing buffer in the network layer. If there is no route information for a packet, a sender saves the packet in the routing buffer and initiates route discovery. Route reply packets cause the sender to free the packet from the routing buffer and resume the transmission of the packet.

4.2.4 Link Layer Acknowledgements

Since any links can be broken while in connection due to either node movements or channel deterioration, ad hoc routing protocols need to monitor route breakdown. For route maintenance and link breakage detection, we can use either MAC protocol in use (such as the link level acknowledgments in IEEE 802.11 [99]) or passive acknowledgment [100]. In MANE, we utilize link-level acknowledgments in detecting link breakdown. After transmitting a packet, the link layer protocol saves the packet in the *interface queue* and waits for the corresponding acknowledgement (ACK). If there is no ACK during a certain period of time or if a negative acknowledgement (NACK) is received, the link layer protocol retransmits the packet. When a certain number of trials fail, the node sends a route error packet to the source.

4.3 A Simple Version of DSR

We first present a simple version of the DSR protocol, which we will later show how to deploy and evolve. In our simple version, no use of the route cache is made at the intermediate nodes. All intermediate nodes simply re-broadcast the first instance of a route request received after appending their own address, and ROUTE REPLY packets are generated only by the destination.

In MANE, a protocol is implemented in two levels: active extensions and active packets. AEs are node-resident and implement the service functions needed for the protocol, while APs serve to glue together the AE functionality

Functions	Types
Get_ID()	`null` $\Longrightarrow$ `int`
LookUp_RouteCache (dest)	`host` $\Longrightarrow$ `host list`
SaveIn_RouteCache (dest, srcRoute)	`host*(host list)` $\Longrightarrow$ `null`
Mark_Dup_Request (source, ID)	`host*int` $\Longrightarrow$ `null`
Check_Dup_Request (source, ID)	`host*int` $\Longrightarrow$ `bool`

Table 4.3: Service Functions for DSR

and actualize the protocol. We first present the services needed for DSR, followed by the APs that are used by the protocol.

4.3.1 An Active Extension for DSR

Table 4.3 shows node resident services needed by DSR. `Get_ID()` generates a unique identification number for a new route request. There are two functions, `LookUp_RouteCache()` and `SaveIn_RouteCache()`, for managing the *Route Cache*. To filter out duplicate requests, `Mark_Dup_Request()` and `Check_Dup_Request()` are used to manage the *Duplicate Request Check List*.

4.3.2 Active Packets for Basic Route Discovery

Figure 4.1 shows the pseudocode for route discovery, while Figure 4.2 shows

```
1: INPUT: destination address D, list of hosts R
2: if this is a duplicate request then
3:    discard this packet
4: else
5:    if arrived at D then
6:       send Route Reply with R
7:       save R in route cache
8:    else
9:       append my address to R
10:      flood this request to all neighbors
11:   end if
12: end if
```

Figure 4.1: Pseudocode for Basic DSR Route Discovery

```
1: fun routeDiscovery(src, dst, id, srtRecord) =
2: if(not Check_Dup_Request(src, id)) then (
3:   Mark_Dup_Request(src, id);
4:   if(thisHostIs(dst)) then (
5:     SaveIn_RouteCache(src, srtRecord);
6:     routeReply(src, dst, srtRecord, reverse(srtRecord))
7:   )
8:   else (  (*intermediate nodes *)
9:     let val myAddr = thisHostOf(getSrcDev())
10:    in
11:      OnNeighbor(|routeDiscovery|(src, dst, id, myAddr::srtRecord),
12:                 broadcast, getRB(), getSrcDev())
13:    end
14:  )
15: else () (* dup req. discard *)
```

Figure 4.2: PLAN for Basic DSR Route Discovery

the PLAN implementation. The pseudocode shows that as the packet executes at each node duplicates are discarded (Line 2 and 3). Then, if the packet is at the destination a route reply is sent and the route is saved (Lines 5 – 7), anticipating the possibility of data being sent back to the source. If the packet is not at the destination, the current address is simply added to the route and the packet reflooded (Lines 8 – 10).

In addition to the service functions in Table 4.3, the PLAN code in Figure 4.2 uses a number of PLAN core services and language constructs, which are discussed in detail in Chapter 2. `thisHostIs()` returns a boolean value for whether the given network address matches the address of the current node. `getSrcDev()` returns the interface on which the packet arrived, and `thisHostOf()` returns the network address corresponding to the given device. Using these functions and the list operator for concatenation, `::`, the route request packet can obtain the source route as it is propagated through the network (Lines 9 – 13). `OnNeighbor()` is a network primitive that generates a child AP executing on a neighbor of the current node. `getRB()` returns the amount of resource bound available in the packet.

The actual algorithm corresponds closely to the pseudocode. In Line 2 route discovery starts by checking for duplicate requests. If the request has been already seen, this packet is discarded (Line 15). If not, it will save the tuple <source address, request id> in the Duplicate Request Check List (Line 3). If the request has arrived at the destination, D saves the source route to S and generates a route reply packet (Lines 4 – 7). Note that the

```
1: INPUT: source address S, list of hosts R
2: if arrived at S then
3:    save R in cache
4:    exit route discovery
5:    send data using R
6: else
7:    forward this packet to S
8: end if
```

Figure 4.3: Pseudocode for Basic DSR Route Reply

```
1: fun routeReply(src, dst, srcRoute, routing) =
2: if(thisHostIs(src)) then (
3:   SaveIn_RouteCache(dst, srcRoute);
4:   exitRouteDiscovery()
5: )
6: else (
7:   let val nexthop = hd(routing)
8:       val routing = tl(routing)
9:   in  OnNeighbor(|routeReply|(src, dst, srcRoute, routing),
10:                  nexthop, getRB(), getSrcDev())
11:  end
12: )
```

Figure 4.4: PLAN for Basic DSR Route Reply

source route is reversed to be used as a route for the route reply. If this is an intermediate node, the node's address is prepended to the current source route and `OnNeighbor` is used to broadcast the request to all the 1-hop neighbors (Lines 8 – 14).

Figure 4.3 shows the pseudocode for route reply, while Figure 4.4 shows the PLAN implementation. The pseudocode shows that a packet is simply forwarded at intermediate nodes, while at the source the route is saved in the

cache and then any data destined for the destination is sent.

Again, the PLAN code corresponds closely to the pseudocode. If the reply has arrived at the source, the route is saved and route discovery exits, triggering the data packets to be sent (implicitly). Lines 7–11 show how the reverse source route is used at an intermediate node. In Line 7 the `nextHop` is read from the front of the list and in Line 8 it is removed from the list. In Lines 9 – 10 `OnNeighbor` is used to send the reply to the next hop, along with the truncated route.

4.4 Deploying DSR

Given the varied environments faced by MANETs, it is quite possible that the most appropriate routing algorithm will not already be deployed on all the nodes. In fact, given that MANETs are a new technology, it is possible that no routing algorithm of any kind is deployed. This is exactly the sort of problem that AN was designed to solve. In particular, let us consider how we could deploy our simple version of DSR.

Our DSR implementation has two components, the AE making up the service routines and the APs that use these routines. Since the APs carry their own code with them, deploying them is trivial; we simply inject the required APs into the network. Deploying the AE is only slightly more complex.

In MANE, the code for an AE can be dynamically linked into a running node. During this linking process, the AE can define new services that can be

```
1: INPUT: destination address D, list of hosts R, Extension E
2: if DSR Service Not Present then
3:    Load DSR Extension From This Packet
4: end if
5: DSR Route Discovery
```

Figure 4.5: Dynamic DSR Deployment

called from PLAN. Once this has been done the APs that use those services will be able to function. Now the only question is how to discover which nodes need to have the AE installed and how to transport the code to those nodes. There are many possible approaches. For example, we could imagine an ANTS-like [24] system where APs implicitly discover whether the needed code is node-resident and then download it from predecessor nodes or perhaps from some global repository.

In our implementation, we used a simpler approach. The route request packet carries the extension in the packet itself and tests to see if it needs to be loaded as it floods the network. Figure 4.5 shows the pseudocode for this simple solution. In Line 2, the packet checks if the extension it needs is present. If not, it will dynamically load and install the extension on the node before executing route discovery. This simple use of *plug-in evolution* [7] allows us to deploy the DSR protocol dynamically and in a timely manner.

However, we have failed to consider one potentially important point. Most of the changes we made to MANE that were described in Section 4.2 were really concerned with improving our emulation of mobility and would not be needed for a real network. However, some of the changes would actually need

to be made to support DSR or AODV. In particular, the proactive routing algorithms typically used in wired networks have no need to potentially queue packets when a route does not exist; they simply drop those packets. Adding this queue is not simply a matter of plugging in a new PLAN callable service function, it requires more fundamental changes to the node implementation.

This is an excellent example of where MANE's support for "update extensions" comes into play. Using Michael Hicks dynamic updating technology [8], we can load an extension that makes significant changes to the node implementation, including inserting the new queuing mechanism.

4.5 Evolving DSR

The ability to deploy a new protocol on the fly using AEs is a powerful mechanism for evolving the network. However, it is also a heavyweight mechanism, requiring that code be dynamically linked into a running node. Using update evolution is even heavier weight, since it enables almost arbitrary changes to be made to a node. It seems likely that only a few network users will be trusted to make these kinds of heavyweight changes to running network nodes. Does this mean that only those privileged users will be able to evolve or customize the network?

In this section, we show that significant protocol evolution can be achieved without resorting to making permanent changes to the node. The key mechanism is, of course, packet programmability. If there is a need to

evolve or customize a routing protocol, APs can implement the needed one without modifying the services of the nodes in the network. This way of *Active Packet evolution* [7] enables the network to promptly evolve with the help of common and reusable AEs. PLAN plays an important role here because its strong safety and security guarantees allow the unprivileged, third-party user to program the network safely.

4.5.1 Active Packets for Optimized DSR

Our initial DSR implementation is quite simple and does not take advantage of many of the optimizations that are possible. In particular, intermediate nodes simply implement flooding, despite having route caches that might contain the route that we are searching for. In order to utilize route control packets efficiently and reduce routing overhead, the protocol needs to be optimized by allowing intermediate nodes to participate in routing aggressively. Specifically, request-broadcasting nodes can obtain a source route to S, and reply-forwarding nodes can acquire a source route to D. These nodes save route information for efficient use of the route cache. Before re-broadcasting the request, intermediate nodes can search their route cache. If there is a valid entry, they can reply without re-broadcasting the request further. Most importantly, we can implement this optimized DSR by only re-programming APs, and we do not need to modify the DSR services in a node-resident AE.

Figures 4.6, 4.7, 4.8 and 4.9 show the pseudocode and PLAN code for optimized DSR route discovery, respectively. The underlined portions indicate

```
 1: INPUT: destination address D, list of hosts R
 2: if this is a duplicate request then
 3:    discard this packet
 4: else
 5:    save R in cache
 6:    if arrived at D then
 7:       send Route Reply with R
 8:    else
 9:       if route found in my cache then
10:          send Route Reply with R and found route
11:       else
12:          append my address to R
13:          flood this request to all neighbors
14:       end if
15:    end if
16: end if
```

Figure 4.6: Pseudocode for Optimized DSR Route Request

the parts that have been added to our initial simple implementation.

At intermediate nodes route discovery changes in two basic ways. First, in addition to flooding the route discovery packet, the packet also saves the partial route in its cache (Line 4), thus increasing its knowledge of possible routes at essentially no cost. Second, the packet looks in the intermediate node's cache for a route to the destination (Line 9). If it exists, then it returns its current route concatenated with the cached route (Lines 10 – 13), thus expediting the route discovery process. Route reply adds a single optimization, replies also add routes to the route caches on intermediate nodes (Figure 4.9, Line 3).

Although in this example, new APs are used to perform a general opti-

```
1: fun routeDiscovery(src, dst, id, srtRecord) =
2: if(not Check_Dup_Request(src, id)) then (
3:   Mark_Dup_Request(src, id);
4:   SaveIn_RouteCache(src, srtRecord);
5:   if(thisHostIs(dst)) then (
6:     routeReply(src, dst, srtRecord, reverse(srtRecord)) )
7:   else (  (*intermediate nodes *)
8:     let val myAddr = thisHostOf(getSrcDev()) in (
9:      try ( let val cachedRouteRec = LookUp_RouteCache(dst)
10:               val fRouteRec = myAddr::cachedRouteRec
11:               fun listconcat(elem1, list1) = elem1::list1
12:               val newSrtRecord = foldr(listconcat,
13:                      reverseHostList(srtRecord), fRouteRec) in (
14:                  routeReply(src, dst, id, newSrtRecord,
15:                             reverseHostList(newSrtRecord)) ) )
16:     handle NotFound => (
17:        OnNeighbor(|routeDiscovery|(src, dst, id, myAddr::srtRecord),
18:                   broadcast, getRB(), getSrcDev()) ) )
19:      end ) )
20: else () (* dup req. discard *)
```

Figure 4.7: PLAN for Optimized DSR Route Request

mization, they can also be used to perform application-specific customizations as well. For example, in the current protocol, if no route reply shortcutting occurs, the route that is chosen is the one taken by the first route request packet to arrive at the destination. An application might desire to use a different metric, such as the route that has the largest bottleneck bandwidth. Assuming we had service routines that could tell us link bandwidths, then we could easily program a route request packet that would measure the bottleneck bandwidth and return a route reply for any route request that arrived at the destination with a better value than previous route requests.

1: INPUT: source address S, list of hosts R
2: save R in cache
3: **if** arrived at S **then**
4: exit route discovery
5: send data using R
6: **else**
7: forward this packet to S
8: **end if**

Figure 4.8: Pseudocode for Optimized DSR Route Reply

```
1: fun routeReply(src, dst, srcRoute, routing) =
2: if(thisHostIs(src)) then (
3:     SaveIn_RouteCache(dst, srcRoute);
4:   exitRouteDiscovery()  )
5: else (
6:   let val srcRoute2dst = subHostList(srcRoute, routing)
7:        val nexthop = hd(routing)
8:       val routing = tl(routing)
9:   in  (
10:    SaveIn_RouteCache(dst, srcRoute2dst);
11:    OnNeighbor(|routeReply|(src, dst, srcRoute, routing),
12:                  nexthop, getRB(), getSrcDev())
13:  )
14:  end
15: )
```

Figure 4.9: PLAN for Optimized DSR Route Reply

4.6 Transitionary Adaptivity

Another way to take advantage of AN is to use it to allow routing protocols to adapt dynamically to changing network conditions. From the discussion in the previous section and Table 4.2, it appears that DSR may be more sensitive to mobility than AODV. Under lower mobility, since there are relatively few link changes, DSR's aggressive caching strategy should be effective in achieving better performance than AODV. However, in high-mobility cases, AODV seems likely to defeat DSR because of more conservative routing management. Superiority between them switches according to node movement frequency. The key point is that AODV appears to work better when levels of mobility are high, while DSR appears to work best when mobility is low. Thus, even if the preferred protocol is in use, it is entirely possible that the level of mobility may shift, making it desirable to change protocols.

Our approach is to build a hybrid protocol that can easily switch between AODV and DSR as mobility levels change. The possible design space for such hybrid protocols is immense and it is important to keep in mind that our goal is to demonstrate that AN has achieved its goals with respect to adaptability, not to explore this design space or to propose the "best" protocol. By showing a fairly simple example, it should be clear that AN techniques will facilitate the implementation, development, and exploration of a wide variety of such protocols.

Functions	Types
LookUp_RIB (routing protocol, dest)	`string*host` $\Rightarrow$ `host*int*int` *or* $\Rightarrow$ `host list`
SaveIn_RIB(dest, destSeq, hopCount, nextHop) *or* (dest, source_route)	`host*int*int*host` *or* `host*(host list)` $\Longrightarrow$ null
Get_RREQ_ID()	`null` $\Longrightarrow$ `int`
Mark_Dup_Request (source, RREQ_ID)	`host*int` $\Longrightarrow$ `null`
Check_Dup_Request (source, RREQ_ID)	`host*int` $\Longrightarrow$ `bool`
Get_SrcSeq()	`null` $\Longrightarrow$ `int`
Get_DestSeq(dest)	`host` $\Longrightarrow$ `int`

Table 4.4: Service Functions for Hybrid Protocol

4.6.1 An Active Extension for the Hybrid Protocol

The key to creating a hybrid protocol that can switch rapidly between differing algorithms is to create a set of generic AE services that can be used by all algorithms. Once this is done, we can then accomplish the actual switching between protocols quite easily using APs. This general idea is an important aspect of AN; by providing generic, reusable, composable node resident components, we can then use packet programs to create many different protocols and enable easy switching between protocols.

Here, we take this idea only to a point by creating generic services com-

mon to both DSR and AODV as shown in Table 4.4. The most important of these, `LookUp_RIB()` and `SaveIn_RIB()`, manipulate a generic *Route Information Base (RIB)*, which is a combined form of the DSR route cache and the AODV route table. Notice that we have used parametric polymorphism so that these functions can take arguments and return values that are appropriate to either DSR or AODV. The next three services, `Get_RREQ_ID()`, `Mark_Dup_Request()`, and `Check_Dup_Request()`, are concerned with duplicate elimination during flooding. These are good examples of general services that we might expect to see reused by many different protocols and, in fact, they have already appeared in our simple DSR implementation. The final two services, `Get_SrcSeq()` and `Get_DestSeq()`, are concerned with manipulating sequence numbers. Although here they are specific to the AODV aspect of our protocol, we can certainly imagine that with more experience, we could define a general set of sequence number manipulation services that would be reusable across a variety of protocols.

4.6.2 An Active Packet for the Hybrid Protocol

Using the services above, we can now program an AP that can adapt to changing conditions. If we actually wished to deploy an adaptable protocol, a key question would be when to adapt. However, our goals are really to show that adaptation is feasible, not to research how best to do it. Thus we assume there exists some global policy module that monitors mobility and informs us as to when to adapt.

Figure 4.10 shows the PLAN program for the hybrid routing request. The AP for the hybrid route request contains three functions: `routeRequestAtSrc()`, `dsrRREQ()`, and `aodvRREQ()`. The source, S, evaluates `routeRequestAtSrc()` and decides which protocol to use. At low mobility, S injects a DSR route request packet by calling an `OnNeighbor()` that evaluates `dsrRREQ()` on all the neighbor nodes (Lines 2 – 4). At high mobility, S spawns a child AP that executes `aodvRREQ()` with the appropriate sequence numbers and a hop counter (Lines 5 – 7). The two functions, `dsrRREQ()` and `aodvRREQ()`, contain the algorithm for the route request of the corresponding routing protocol.

When there is valid information for the request (on intermediate nodes or the destination node), a reply packet is generated by the function call, `dsrRREP()` (Lines 14 & 19) or `aodvRREP()` (Lines 31 & 38). The optimized DSR protocol allows intermediate nodes to reply to the request (Lines 18 – 20). In replying with cached information, the reply-generating node needs to concatenate the route record and cached information (Lines 19 – 20). In AODV, the destination sequence number is compared to validate freshness of the cached information (Line 34).

Figure 4.11 shows PLAN program for the route reply. Similarly to the route request, the hybrid reply packet contains two different function calls for either DSR or AODV, respectively. `DsrRREP()` activates the optimized DSR route reply, which updates the RIB (Lines 3 & 8), handling the source route (Line 5), and getting the reply forwarded to the source (Lines 6, 7, and 9 –

```
1: fun routeRequestAtSrc(src, dst) =
2: if(mobility = 0) then
3:   OnNeighbor(|dsrRREQ|(src, dst, Get_RREQ_ID(), [ ]),
4:              broadcast, getRB(), getSrcDev())
5: else
6:   OnNeighbor(|aodvRREQ|(src, dst, Get_RREQ_ID(), Get_SrcSeq(),
7:        Get_DestSeq(dst), 0), broadcast, getRB(), getSrcDev())
8:
9:  fun dsrRREQ(src, dst, id, srtRecord) =
10: if(not Check_Dup_Request(src, id)) then (
11:   Mark_Dup_Request(src, id);
12:   SaveIn_RIB(src, srtRecord);
13:   if(thisHostIs(dst)) then
14:     dsrRREP(src, dst, srtRecord, reverse(srtRecord))
15:   else (  (* intermediate nodes *)
16:    let val myAddr = thisHostOf(getSrcDev())
17:        val newSrtRecord = myAddr::srtRecord
18:    in ( try ( let val srcRt:(host) list = LookUp_RIB("DSR", dst)
19:                in dsrRREP(src, dst, listcon(reverse(srcRt),
20:                           newSrtRecord), reverse(srtRecord)) end )
21:         handle NotFound => (
22:           OnNeighbor(|dsrRREQ|(src, dst, id, newSrtRecord),
23:                broadcast, getRB(), getSrcDev()) ) ) end ) )
24: else () (* dup req. discard *)
25:
26: fun aodvRREQ(src, dst, id, srcSeq, dstSeq, hopCount) =
27: if(not Check_Dup_Request(src, id)) then (
28:   Mark_Dup_Request(src, id);
29:   SaveIn_RIB(src, srcSeq, hopCount+1, getSrc());
30:   if(thisHostIs(dst)) then
31:      aodvRREP(src, dst, dstSeq, 0)
32:   else ( try (  (* intermediate nodes *)
33:             let val rt_entry:(host*dev*int*int) = LookUp_RIB("AODV", dst)
34:             in ( if(dstSeq > #3 rt_entry) then (
35:                    OnNeighbor(|aodvRREQ|(src, dst, id, srcSeq, dstSeq,
36:                        hopCount+1), broadcast, getRB(), getSrcDev()))
37:                  else
38:                    aodvRREP(src, dst, #3 rt_entry, #4 rt_entry) ) end )
39:          handle NotFound => (
40:          OnNeighbor(|aodvRREQ|(src, dst, id, srcSeq, dstSeq, hopCount+1),
41:                        broadcast, getRB(), getSrcDev()) ) ) )
42: else () (* dup req. discard *)
```

Figure 4.10: PLAN for Hybrid Route Request

```
1: fun dsrRREP(src, dst, srcRoute, routing) =
2: if(thisHostIs(src)) then (
3:   SaveIn_RIB(dst, srcRoute);
4:   exitRouteDiscovery()  )
5: else ( let val srcRoute2dst = subHostList(srcRoute, routing)
6:            val nexthop = hd(routing)
7:            val routing = tl(routing) in  (
8:        SaveIn_RIB(dst, srcRoute2dst);
9:        OnNeighbor(|dsrRREP|(src, dst, srcRoute, routing),
10:                            nexthop, getRB(), getSrcDev()) )
11:       end )
12:
13: fun aodvRREP(src, dst, dstSeq, hopCount) =
14:   SaveIn_RIB(dst, dstSeq, hopCount, getSrc());
15:   if(thisHostIs(src)) then
16:     exitRouteDiscovery()
17:   else (
18:     let val nexthop = LookUp_RIB("AODV", src) in
19:       OnNeighbor(|aodvRREP|(src, dst, dstSeq, hopCount+1),
20:                  #1 nexthop, getRB(), #2 nexthop)
21:     end )
```

Figure 4.11: PLAN for Hybrid Route Reply

10), while `aodvRREP` executes the AODV route reply by forwarding the reply to the source through the reverse path (Lines 19 – 20).

4.6.3 Simulation of the Hybrid Protocol

Although our goal was not primarily to explore the design of hybrid routing algorithms per se, we still wanted to see if we could show that such an algorithm could indeed result in improved performance when faced with changing

mobility. In order to explore this question we simulated our algorithm as well as DSR and AODV.

Experimental Setup

As a simulator, we used `ns-2`, which is a discrete event simulator widely used in networking research [5]. As a measure of performance, we used the Packet Delivery Ratio (PDR). PDR is the ratio of the number of the transmitted packets to the number of received packets and larger numbers are better. For a direct comparison, we used CBR traffic rather than TCP traffic because congestion control and flow control offer different loads according to network conditions for TCP. Each node moves according to the "random waypoint" model [78], in which the nodes repeatedly move and then pause. In this model, the pause time and the movement speed characterize the mobility of the network. In each simulation, the same scenarios of movements and traffic are used for DSR, AODV, and the hybrid protocol. The reported values are averages taken from ten simulations under different movements and traffic scenarios.

The packet size is 512 bytes, and 4 packets are generated per second. The number of CBR sources is 25 out of 50 total nodes. For each simulation, 50 nodes move around in a 1000 m $\times$ 1000 m square space for 1500 seconds. To simulate changing mobility, we divided the simulation time into 3 parts of 500 seconds each. In the first part (0–500 seconds), there is no movement and the network is stationary. In the second part (500–1000 seconds), all the nodes move at a maximum speed of 10 m/s with a pause time randomly selected

between 0 and 250 seconds. In the last 500 seconds, the maximum speed is 20 m/s and the pause time is 0 seconds. For the hybrid protocol, initially DSR is used and as the mobility increases the nodes switch to AODV. Specifically, during the first half of the simulation, route control packets follow DSR semantics and data packets are routed using DSR. After 750 sec., the interface for the routing protocol is changed to AODV and route control packets follow AODV semantics. For the simulation of DSR and AODV, we used the existing `ns` versions developed by the Monarch project [97].

Results

The simulation results are shown in Figure 4.12. The x-axis is simulation time and the y-axis is the PDR. We observe that in general as mobility increases, the PDR decreases because of more frequent link failures or changes. However, DSR and AODV have different rates of decrease and there is a crossing point where dominance changes. In particular, while DSR's PDR is better than that of AODV under low mobility, DSR shows more degradation as mobility increases. On the other hand, AODV is relatively robust to changes in mobility.

Not surprisingly, since the hybrid protocol switches between DSR and AODV, its performance basically follows the better protocol in the whole range of mobility. At low mobility, the hybrid protocol adopts DSR's aggressive route discovery and caching scheme and it performs similarly to DSR. However, as mobility increases, it works like AODV and becomes robust to increased mobility. The region from 500 to 750 seconds is the only exception, because

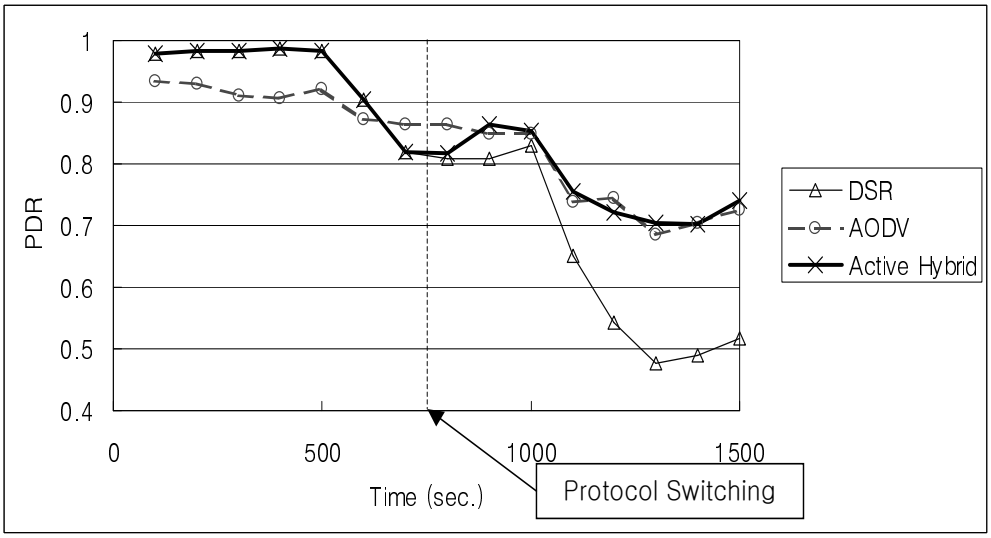

Figure 4.12: PDR over time for DSR, AODV, and Hybrid

during that period, we have not switched away from DSR. Since it is not possible to change routing packets once they are injected into the network, DSR packets that are already injected are handled while network mobility is high. Also, after switching, AODV initiates route discovery to build the route table. These explain why the hybrid protocol lags in following the performance of AODV between 800 sec. and 900 sec. If we allow DSR and AODV to share route information by combining the DSR route cache and the AODV route table as described in Section 4.6.1, we could improve the performance during the switching period. From the simulation results, we see that the hybrid protocol is adaptive to network mobility and suitable for networks under varying mobility environments.

4.7 Advantages of AN for Evolution and Adaptation

In this section, we compare the evolvability and adaptivity of the traditional network and the AN-enabled network. We make this comparison in terms of standardization, implementation and deployment. In each of these areas AN makes evolution and adaptation easier than in the current architecture.

4.7.1 Standardization

Traditional Arch. It has been 8 years since the DSR protocol was published; standardization is still in progress [90]. The AODV protocol was published in 1999 [79]; an *experimental* standard document for AODV was issued in 2003 [83]. It took 4 years to achieve limited standardization of the AODV protocol. However, the standardization of AODV is still in progress [92] and a *standards track* document is expected to be published some time in the future.

AN Arch. There are no substantial demands on standardization for deploying and conforming a routing protocol. AN can also evolve routing protocols on an as-needed basis without elaborate standardization.

Conclusion Because AN makes much simpler demands on the standardization process, with respect to standardization using AN makes evolving the network significantly easier.

4.7.2 Implementation

To implement reactive ad hoc routing protocols, the system should meet the following specifications:

- Working as a router in multi-hop routing,
- Running a routing protocol on demand,
- Maintaining a routing buffer,
- Detecting link breakage, and
- Recovering from route failure.

Traditional Arch. In the traditional architecture, ad hoc routing protocols must be agreed upon before network configuration. Based on the standards, ad hoc routing protocols are implemented in the protocol stack. In order to overcome routing heterogeneity, all the potential routing protocols must be implemented in the protocol stack on the nodes. However, this is a heavyweight and inextensible approach; the protocol stack must be changed as new routing protocols emerge. It is also hard to evolve a widely-deployed routing protocol. Evolutionary features must be implemented and deployed as a form of new software. In the traditional architecture, an evolution means the burden of changes in some parts of the network infrastructure.

Table 4.5 compares the lines of code of various DSR implementations: Monarch Project [93], picoNet [94], DSR router project by University of Col-

Criteria	Monarch	Piconet	UC Boulder	MANE
Line of Code for Basic DSR	N/A	N/A	N/A	661 in Popcorn + 33 in PLAN
Line of Code for Optimized DSR	9089 in C	2681 in C	6200 in C	0 in Popcorn + additional 8 in PLAN

Table 4.5: Comparison of DSR Implementations

Criteria	UCSB	UIUC	MANE
Line of Code	8346 in C	5513 in C	683 in Popcorn + 43 in PLAN

Table 4.6: Comparison of AODV Implementations

orado, Boulder [95], and MANE. Due to the limitations of the current network architecture, there is no comparable implementation supporting only basic DSR. Table 4.6 presents the lines of code of various AODV implementations [96].

AN Arch. The AN architecture requires two programming interfaces to implement a routing protocol: supporting extensions and actualizing packets. Since the substantial parts of the routing protocols are implemented using packet programming, it is simple and quick to implement the supporting extensions, which constitutes the network infrastructure. Also, packet programming facilitates dynamic protocol implementation and actualization.

Criteria	MANE
Line of Code	1204 in Popcorn + 69 in PLAN

Table 4.7: MANE Implementation of Hybrid Routing Protocol

In MANE, it took one week to implement both basic DSR and optimized DSR. It took one week to implement AODV in MANE. Since there is no comparable hybrid scheme to ours, it is impossible to compare mobility-adaptive routing protocols. It took one week to implement the hybrid protocol and its lines of code is shown in Table 4.7.

Conclusion AN allows shorter and simpler implementations for protocol development, evolution, and hybridizing. Simpler implementations are easier to create and change and thus it is easier to evolve the network from an implementation point of view.

4.7.3 Deployment

Traditional Arch. The traditional architecture requires network-wide software installation to deploy a routing protocol in an ad hoc network, which is feasible only in a small network and before system operation. Once the network is made up, it is difficult to deploy or adopt a new protocol without downtime. Evolving a routing protocol needs a software upgrade on every node accompanied by downtime. Also, for an adaptive protocol, a new phase

of implementation and deployment should start from scratch.

AN Arch. In order to deploy a routing protocol, it is only required for nodes to be equipped with supporting service functions in extensions that can be either dynamically linked or updated. Once the service functions are resident on nodes, realization of protocol is achieved by active packets. Based on the supporting service functions, it is easy to evolve a routing protocol by packet programming. Active packets conveying enhanced features of the protocol actualize the evolution of the routing protocol. It is a lightweight way of evolution to use active packets, while it is a heavyweight way of network evolution to use dynamic updates.

Adaptation by hybridizing can be accomplished by simply combining packet programs rather than complex design and operations. Since active packets carry a hybrid scheme, adaptation is achieved on packet-by-packet basis.

Conclusion The traditional architecture requires software installation and downtime to deploy a new routing protocol or to evolve existing protocols. While, the AN architecture based on programmable infrastructure enables us to deploy or evolve protocols without changes and downtime on nodes. From the deployment point of view, AN provides better evolvability.

4.8 Other Possible Adaptivities

In the previous sections, we discussed the deployment and evolution of the DSR protocol, and the implementation of a hybrid protocol of DSR and AODV. We took them as examples because they are simple and have the basic characteristics of ad hoc routing. We have no reason not to expand these approaches to other routing protocols including proactive routing protocols. We can think of the following examples worthy of further research.

4.8.1 Mobility-based Zone Routing

When we hybridize the routing protocol, we made an assumption that there is a network-wide mobility metric known to all nodes in the network. As a matter of fact, it is hard or impossible to get a mobility metric for a network as the network size grows. Without a special algorithm or information exchange, a node can obtain network mobility only from locally available information. Furthermore, there could be some parts of the network in which nodes' movements are faster or slower than other parts.

If it is allowed for any intermediate nodes to make a decision on routing protocol, there is no need for a global metric of mobility. Because Active Packets allow protocols to change on a packet-by-packet basis, AN provides technology to support this kind of adaptation to different environments.

4.8.2 Connecting to the Internet

While ad hoc networks are efficient for prompt network make-up and break-up, it would be more useful to be connected with the infrastructure-based network, or to be specific, the Internet. Since the Internet and ad hoc networks have different architectures, there needs a special router at the junction working as a gateway connecting the two different networks. In terms of the routing protocol, the Internet has a hierarchical scheme, while ad hoc networks may have either a hierarchical or a flat scheme. To route packets between them, the gateways need to understand both of the routing protocols in use. Since ad hoc networks are ad-hoc, it is quite possible that the gateway is not equipped with the routing protocol in use on the ad hoc networks. In this case, we can take advantage of AN in evolving the gateway either by active extension evolution or by active packet as discussed in Section 2.2.1. Through active extension evolution, the ad hoc network can update and equip the gateway with the routing module to support the routing protocol in use. After that, the following data packets do not need to be aware of the routing protocols. If active packets are aware of the routing protocols and contains code for the routing transformation, ad hoc networks can forward packets to and from the Internet without modifying the gateway.

4.8.3 Dynamic Routing Metric

As ad hoc networks develop, various requirements besides simple routing may arise, such as power-saving operations or quality of service (QoS). The use of an appropriate routing metric, not just hop count, helps the network to route packets optimally for the requirements. AN can facilitate ad hoc networks to meet those requirements through remote execution of active packets. By injecting active packets containing an algorithm for calculating the metric, AN actualizes various QoS routing protocols on the fly. As the active packets propagate through the network, they collect optimal paths for the given metric. Each host is only required to support the interface to supply information for the metric. Given the interface, an ad hoc network is able to find an optimal routing path for varying requirements dynamically.

4.9 Discussion

In order for a protocol to be put into operation in MANE, we need to implement both AEs and APs. APs actualize the protocol by calling services provided by AEs. Adaptability and reusability of the protocol depend on the design of AEs and APs. In the previous section, we demonstrate how to implement the hybrid protocol through *AP-dominant* programming with minimal functionality in AEs. If we include more functions as "built-in" services in AEs, the size of packet programs can be reduced. For instance, we do not need `Get_RREQ_ID()` because built-in services can be designed to generate

a new `RREQ_ID` internally for route discovery. Further, if we implement all the basic functions as AEs, the protocol is embedded in MANE and APs are routed transparently to the upper layer protocols. Even though we do not cover *AE-dominant* programming in this work, currently there are three routing protocols, DSR, AODV and hybrid, embedded in MANE other than the default routing function, `defaultRoute`.

In this experiment, we use a routing-aware AP to show the ease of implementing a protocol in packet programming. By simply combining two packet programs and common services, we can adaptively change the routing protocols and enhance the routing performance. If we expand this approach and combine several routing protocols in a hybrid form, we should be able to achieve an improved protocol that can adapt itself to the conditions changing over various ranges and that performs best under the given conditions. Since APs facilitate protocol realization on the fly, AN can allow routing protocols to change agilely as APs pass through the network.

For the adaptive routing activation, we assume that there is a policy module which monitors network mobility and decides which protocol to use. Since the main purpose of this paper is to show the applicability of AN's adaptability, we can call this assumption is trivial. The module, however, plays an important role in applying the routing protocol. For adaptivity, there is a need to fine-tune the policy module. Further, if the network mobility is varying near the crossover point, the network may switch routing protocols too frequently; thus, it is desirable to allow some hysteresis.

Chapter 5

Case Study III: TCP Over Wireless

In this final case study, we consider how adaptation can be used to address the performance problems of the Transmission Control Protocol (TCP) over wireless links. The importance of TCP is obvious and TCP is one of the central protocols of the IP-based Internetwork. TCP is the transport layer protocol for the most widely-used application layer protocols, such as the Hyper Text Transfer Protocol (HTTP), Telnet, the File Transfer Protocol (FTP), and the Simple Mail Transfer Protocol (SMTP). Further, with the spread of IEEE 802.11 wireless Ethernet technology [99], wireless last hops have become common. Unfortunately, TCP assumes that packet loss is due to congestion. This assumption is not always true for wireless links, which can lead to poor performance [17]. There has been significant research into this problem, but

the solutions either require widespread changes to the network or are architecturally limited.Because so much of the Internet relies on TCP, it is daunting to change TCP. Changing TCP means changing every end host in the Internet. Further, any proposed changes to TCP must be verified to be interoperable with and harmless to the rest of the Internet. For this reason, a substantial emphasis in this chapter is on the architectural approach. A condensed version of this chapter appears in the proceedings of International Working Conference on Active Networking (IWAN) 2004 [101].

This sort of network evolution is exactly the target of Active Networking (AN) and we believe that AN can provide a number of strategies that will help address this issue. Our claim is that if we had a network that incorporated AN in at least some of its nodes, the range of solutions to the problem of TCP over wireless would be greatly increased. Of course, if AN in its most general form penetrated everywhere, it would clearly solve this problem, because it would be easy to simply deploy the best, most general solutions and as new solutions were developed to deploy them. Here we are interested in exploring the implications of having more limited AN penetration on possible solutions. We explore this idea by first presenting a model of how AN might be deployed in the specific case of TCP over wireless. We then present some system requirements, a deployment architecture, and highlight some key AN capabilities. We then use this model and architecture to motivate a series of concrete implementations that address various aspects of the problem. These include an implementation of adaptive link control and of the TCP Snoop

protocol [102]. The implementations also serve as a vehicle for exploring other design possibilities, thus broadening support for our claims.

In addition to presenting background material on TCP over wireless and our AN approach and platform, we make our case in two ways. First, to a large extent the issues at hand have to do with implementation architecture. Thus a key part of our argument is a presentation of a model of TCP over wireless systems that lays out the possible design space, followed by a consideration of the high-level architectural issues. Second, to make things concrete, we present implementations of some of the possible solutions. To further flesh out our understanding of the design space, we also use the implementations to motivate a discussion of alternative possible implementations.

5.1 Background

This section presents an overview of TCP, the problems of TCP over wireless links, and related work.

5.1.1 TCP Overview

The Transmission Control Protocol (TCP) is a connection-oriented transport layer protocol responsible for end-to-end reliable data transmission. TCP is equipped with several mechanisms for reliable transmission: a sliding window-based *Go-back-N ARQ* scheme for in-order delivery, retransmissions for error recovery, and flow control and congestion control for adaptive utilization of

network resources [14].

For in-order delivery and error recovery, TCP uses sequence numbers and retransmissions. Using the sequence numbers of segments and the corresponding acknowledgement (ACK), a TCP sender keeps track of *round trip time* (RTT) variations of the connection [103]. These RTT measurements are used in estimating the retransmission time out (RTO) for segments. If there is no ACK for the duration of a RTO, the sender retransmits the dropped segment. In addition, since TCP uses cumulative ACKs, it will typically retransmit all the segments starting from the lost one even if the subsequent segments have already been successfully received.

TCP's congestion control algorithm manages network resource utilization in a way that adapts to network load. The basic idea of TCP congestion control is for a source to determine the available capacity in the network, controlling the amount of data the source can transmit using the congestion control window size. The source regards the arrival of an ACK as a signal that the available capacity of the network has increased; and TCP gradually increases the pace of transmission by increasing the window. It does so until a packet loss indicates some router is congested. When this happens, TCP reduces the congestion window by a factor of two, which halves the bandwidth used by the sender. This is the basic idea, but in practice the mechanisms are more complex and the basic idea of congestion control has been evolutionarily engineered for better performance. Some of the steps of this evolution are listed in Table 5.1.

TCP Version	Features
Tahoe	Slow Start, Congestion Avoidance, Fast Retransmit
Reno	Fast Recovery, Header Prediction, Delayed ACKs
New Reno	Partial ACKs
Vegas	Rate-based RTT Estimator

Table 5.1: Comparison of various TCP implementations

5.1.2 TCP Over Wireless Links

There are several problems with TCP functionality and performance over wireless links. Over such links, there may be more fluctuations of bandwidth and delay than in typical wired networks, stressing TCP's algorithms ability to adapt. One of the main problems of TCP over wireless links arises from the fact that TCP's error recovery and congestion control are closely coupled. This is due to the assumption that packet drops are only the result of the network congestion. This assumption is a kind of layer violation in that the transport layer is closely correlated with the link layer. This assumption is valid in wired networks because wired links are reliable enough that packet losses can be interpreted as the result of congestion in routers. Thus, TCP's congestion control worked well until wireless links began to be used as an Internet link layer. Wireless links are lossy and cannot be assumed to be reliable in spite of their link-level error recovery schemes [104]. In the case of packet drops caused by link errors, TCP should retransmit the packet without closing the congestion window. These points are reinforced in the literature, where it has been

shown that TCP's performance significantly degrades over wireless links [17].

5.1.3 Related Work

Several solutions have been proposed for the problem of TCP over wireless links [16]. They can be classified into two main categories: *End-to-end* and *Transparent*. An obvious solution is simply to do the necessary research to understand how to mix TCP with wireless links and then deploy those solutions throughout the Internet. Not surprisingly, the first step, researching a solution has had significant success [16, 105, 106, 102, 107, 108]. The most general solutions require updating both the TCP implementations on the end hosts and at least some of the routers handling wireless traffic (end-to-end solutions). Unfortunately, in today's Internet, such an update is very difficult to achieve. As a result, there has also been significant work on solutions that do not require updating the end hosts (or perhaps only the one connected wirelessly), essentially restricting the design space to transparent modifications of the base station connecting the wired network to the wireless one [102, 107]. Unfortunately, this architectural restriction can have adverse performance implications [109].

We regard *backward compatibility* as of importance in TCP issues, thus we discuss two of the transparent methods in more detail. Our approaches, however, are not confined to transparent solutions.

End-to-end Solutions

Basically, end-to-end solutions at the end points require modifying TCP while maintaining end-to-end semantics [16]. They attempt to improve performance by remedying the defects of TCP. Selective ACK (SACK) [105], Explicit Loss Notification (ELN) [16] and Explicit Congestion Notification (ECN) [106] exemplify this approach. SACK aims to reduce unnecessary retransmissions caused by the cumulative ACKs of the original TCP specification. ELN and ECN aim to separate the error recovery and congestion control mechanisms of TCP. In [16], the authors showed that an end-to-end protocol that has both ELN and SACKs is effective in dealing with high packet loss rates.

The drawback of these approaches is that they require fundamental changes to TCP. The need to replace already deployed versions of TCP means that deployment of these approaches will be difficult and slow. Besides, this approach needs more care because it is unclear if the modified TCP will perform well both on wired and wireless links. From experience, we see that it may take time to find problems of newly deployed protocols that were thought to be well-designed.

Transparent Solutions

In this approach, link-level losses are handled by link layer protocols and hidden from the transport layer. Therefore, existing TCP stacks and hosts operate normally without knowing whether the connection is over wireless or

wired links. The TCP snoop protocol [102], AIRMAIL protocol (a combination of FEC and ARQ) [107], and I-TCP (Indirect-TCP, connection split) protocol [108] are examples.

The main advantage of these approaches is that they are more practical than the "end-to-end" approach in terms of incremental deployment. It is easier to modify only the link layer protocols handling lossy links than the TCP protocol deployed on every end-host. However, as we will see from the TCP snoop protocol, link layers should be aware of the transport layers' semantics and session state information, which is a case of layering violation. In addition, this approach has the possibility of redundancy, inefficiency, or even ineffectiveness. In [109], for instance, the authors concluded that competing retransmissions by the link and transport layers often lead to significant performance degradation, unless the packet loss rate is high (more than about 10%). TCP's fast retransmission and associated congestion control degrade TCP performance over wireless or lossy links. We discuss two representative examples of the transparent method: the snoop protocol and I-TCP.

TCP Snoop protocol The TCP snoop protocol is one of the well-known examples of a TCP Performance Enhancing Proxy (PEP). TCP PEPs are adopted to improve the TCP performance in certain circumstances where desired performance is limited by link characteristics [110].

The TCP snoop protocol is a link layer protocol that comprehends TCP semantics on a base station (BS), which is a junction between the wired and wireless links. In order to enhance TCP's performance over wired-

cum-wireless links, the snoop protocol maintains TCP state information by caching TCP data segments, retransmitting lost packets locally (over a one-hop wireless link), and suppressing duplicate ACKs to prevent unnecessary congestion control from launching at the sending fixed host. This scheme is feasible because it only needs modifications at the BS. The snoop protocol, however, results in a solution for TCP data flows of one-direction only, from a fixed host (FH) to a mobile host(MH). Further, there is overlapping retransmission functionality between the link layer and the transport layer, which may cause inefficiency [109].

I-TCP To handle mobile hosts, the indirect model was proposed by B. R. Badrinarth, et al. [111] The authors argued that changes are needed at every level of the OSI model to support mobility and proposed that fixed hosts (FH) and mobile hosts (MH) be handled differently. Based on this indirect model, I-TCP was suggested to improve TCP performance over wireless links [108].

I-TCP splits a TCP connection into two separate links at the border of wired and wireless links, one over wireline links, and the other over wireless links. A transport layer connection between an FH and an MH is established as two separate connections and the BS is the junction point of an I-TCP connection. With the help of I-TCP on the BS, the TCP connection between the FH and the MH can be maintained over the lossy links and during handoff. Packet drops due to losses on

the wireless medium are hidden from the FH and the FH is completely unaware of the wireless links and indirection. Even though I-TCP is a transport layer protocol, it does not sustain end-to-end TCP semantics. For reliable connections, I-TCP requires that some mechanisms for error recovery should be provided from the application layer. This seems a significant limitation of the approach.

5.1.4 The State of The Art: Standardization and Deployment

There have been several proposals for TCP modifications to enhance TCP performance over wireless links [105, 112, 106, 110, 113]. Among them, Explicit Congestion Notification (ECN) is recommended as a possible adoptable scheme [114, 106]. A basic mechanism for ECN was included in the original design of the Internet Control Message Protocol (ICMP) as the *Source Quench Message* [115]. However, there has been significant controversy over implementation of the Source Quench Message [116]. Since TCP's congestion control was shown to be a main reason for performance degradation over wireless links, a standard document for ECN, RFC 2481, was issued as an *experimental* document in January 1999 [117]. In September 2001, RFC 3168 was published as a *standards track* RFC obsoleting RFC 2481 [106]. In the meantime, Cisco supports ECN starting with Internetwork Operating System (IOS^{TM}) Release 12.2(8)T in 2003 [118].

However, ECN requires support from both the routers and the end hosts, i.e., a significant evolution in the network architecture. Every TCP installation on end hosts must be ECN-capable, which means that end hosts need new TCP/IP stacks; every switch or router must be upgraded to support procedures for setting the ECN field in the IP header. Due to the difficulty of adopting ECN, ECN has not been widely deployed in the current network [119].

While end-to-end approaches might be a possible long-term solution, they are not likely to be available throughout the Internet in the short-term. On the other hand, transparent approaches are more feasible and incrementally deployable. They do not require modifications on every end host. One transparent approach is to reduce link errors by using adaptive link error control protocols. Many adaptive schemes have been proposed in the literature [120, 121, 122, 123, 107, 124] and industry also plans to adopt adaptive link control schemes for data service [99, 125, 126]. In general, their link layer protocols achieve adaptivity in various ways: adaptive coding rate, adaptive ARQ schemes, or adaptive frame length. Adaptive modulation and coding was adopted by IEEE 802.11 [99] in 1997, IEEE 802.11b [127] and IEEE 802.11a [128] in 1999. They support multirate operation by adapting modulation schemes, but not error correction schemes. Hybrid ARQ/FEC schemes have been suggested for the channels whose bandwidth-delay product is high, such as the satellite channel [123]. Their adaptivity is limited in that their adaptation is pre-designed and fixed. For example, in the Radio Link Protocol (RLP) specified in the 3rd Generation Partnership Project 2 (3GPP2) speci-

fication, when a frame has errors and cannot be recovered by FEC, the frame is retransmitted a certain number of times, after which RLP gives up [126].

Another transparent approach is to use cross-layering on lower layers by sharing layer-specific information. In this approach, lower layer protocols cooperate to enhance TCP performance without changing TCP. The Berkeley snoop protocol, which is a TCP-aware link protocol, is a well-known example of this approach. The Berkeley snoop protocol was first published in 1995 [102]. Since then, there have been several implementations [129, 130]. However, neither standardization nor deployment of the snoop TCP protocol has started in the 9 years since 1995. In order to support the snoop TCP protocol, the modifications that are required are as follows:

- Architecture
  - New functional entities: snoop proxy on Base Stations
  - Layering violation: link layer protocol (snoop) needs to be aware of transport layer protocol (TCP) semantics.
  - Only one direction of TCP flow (from FH to MH) is supported.
- Functionality
  - The snoop proxy on Base Stations should be aware of TCP semantics.
  - The snoop proxy should be equipped with the following functionality:

* Tracking TCP data and ACK segments by maintaining state for TCP connections
* Caching outstanding TCP data packets
* Maintaining local timer
* Retransmitting by either local timeouts or duplicate TCP ACKs
* Suppressing duplicate ACKs from the mobile host

To address TCP problem over wireless links requires either TCP changes on every end host or transparent modifications of lower layer protocols. Because both of the approaches demand significant network evolution, the standardization for the TCP problem has been slow; no schemes have been widely deployed throughout the Internet.

5.2 How can Active Networking Help?

The goal of our work is to show how AN could help address the problems of TCP over wireless links. At a high-level, this is an architectural question; where and in what form can AN be useful? To answer this question, we begin by creating a model of the underlying system. The section concludes by considering a variety of architectures that map AN capabilities on to this model. The rest of our work is principally an exploration of some specific instances of these mappings.

One obvious point: if sufficiently powerful AN technology were deployed everywhere then the problem would be much easier. In fact, we would be able

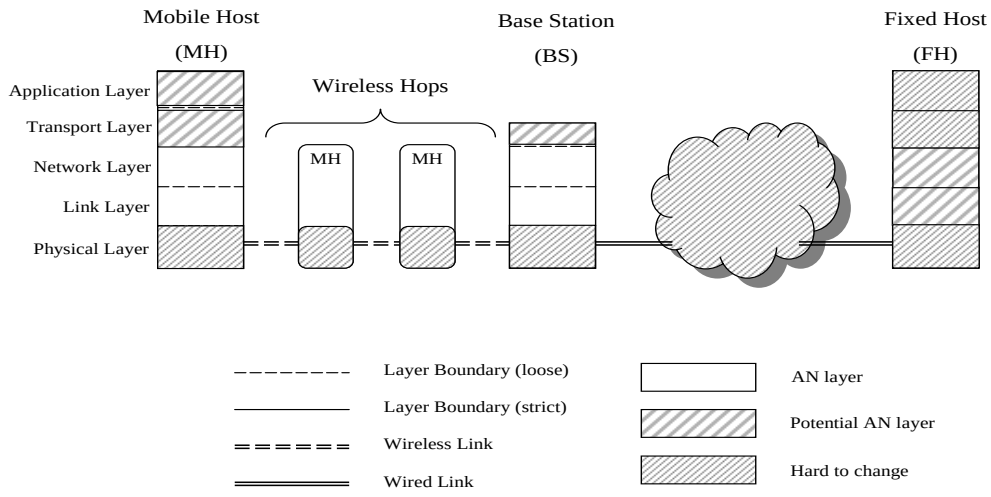

Figure 5.1: AN model for TCP over wireless

to deploy end-to-end, nontransparent solutions easily and to evolve them as better solutions were developed. Our goal here is to consider the possibilities when AN is available in some places and some forms but not in such a widespread manner to allow this trivial solution.

5.2.1 Model

The model of a TCP session shown in Figure 5.1 captures many of the key architectural issues. Communication is between a Mobile Host (MH), and a Fixed Host (FH). A Base Station (BS) connects the wired network where the FH resides to the wireless one where the MH resides. Unlike the bulk of the

related work discussed above, we include the case where the MH may need multiple wireless hops to reach the BS. Also, the related work focuses on the case where the bulk of the data is being transmitted from the FH to the MH. In general, we are also concerned with the case where the MH is the primary source of data.

Figure 5.1 also illustrates some of our thinking about where and what kind of AN technology might be deployed. As described in Chapter 2, our AN architecture has both active packets (APs) containing executable code and active extensions (AEs) which are downloaded dynamically to modify or extend nodes. We assume that we have full control of the MH and thus can expect that both APs and AEs can be used there where needed. Similarly, we entertain the possibility that the wireless network is "all active" and thus that we could potentially deploy both AEs and APs there. Three possibilities exist for the BS. First, if the BS can employ no activeness, then we are restricted to end-to-end solutions (and must have an active enabled FH). Second, perhaps for security reasons, the BS may allow AP processing, but not allow AEs to be downloaded. Finally, the BS may support both APs and AEs. The intermediate links between the BS and FH are not a source of the problems we are trying to address and so without loss of generality, we can assume they are not "active." However, notice that in the case that the BS is not "active," some BS-centric approaches will work if deployed at an intermediate node. Finally, the FH has the same basic options as the BS. Of course, it is likely that an MH will have more control over the BS than the many possible FHs,

so it is likely that the FH will allow fewer "active" options than the BS.

Finally, Figure 5.1 also touches on the issue of layer crossing. The problems we are addressing come fundamentally because TCP violates the basic layering principles of the network by making an incorrect assumption about the nature of the wireless physical layer. Thus it is not surprising that many of the approaches to solving these problems also violate layering. In fact, one of our premises is that since AN can support flexible, controlled layer crossing, it is well suited to these solutions. Thus the figure shows which layers we expect to be the most "permeable" as well as at which layers we most expect to deploy either APs or AEs. One important case that is not illustrated is the use of "shim" layers. These are just layers that are inserted between existing layers.

5.2.2 Requirements, Architecture, and Capabilities

Now that the basic system model is stated, we consider the possible architecture of solutions and discuss several AN capabilities that potentially play an important role in the solution space. However, before doing so, we consider two system requirements.

The first requirement is preservation of TCP's end-to-end semantics: reliable, in-order, duplicate free delivery. We view this as a strict requirement for any solution; taking the view that these semantics define what TCP is and that any system that does not provide these features is not TCP. The second requirement is backward compatibility. Since in some scenarios the possibility

exists of using AN to modify the end host's implementation of TCP, we do not view this as a strict requirement. However, many other scenarios exist that deny this possibility and so it is important to consider. Since we view the MH as fundamentally more changeable than the FH, backward compatibility issues focus at the FH. Backward compatibility then takes two forms. First are systems where the FH is "active," but the TCP implementation is not. Such systems admit end-to-end approaches, but must mask any "activeness" from TCP and higher layers. Second are systems in which the FH is unchangeable and transmits standard TCP segments. In this case, any "activeness" must be masked before data segments or ACKs reach the FH. Given our assumption that the Internet is not active, this means "activeness" being masked at or before the BS. In general, we would like to be able to support "islands" of AN functionality isolated by conventional networks. We will illustrate how this may be done in Subsection 5.4.3.

In our view, there are two basic architectural approaches: horizontal and vertical. The horizontal approach works between peer layers and does not cross layer boundaries. For example, link layer protocols over wireless hops can adaptively cope with fluctuating channel conditions and reduce link-level errors. An important special case is when the peer layers are dynamically inserted (and removed) shim layers. This is essentially the idea of Protocol Boosters [22]. A useful analogy is that boosters are like snow chains; you put them on a protocol when they are needed, but remove them when they are not (and may if fact be degrading performance). In section 5.4, we will discuss

how our implementation system makes this idea especially useful.

In contrast, the vertical approach allows layering violations and information sharing between layers. For example, in the PEP architecture discussed previously, the BS is allowed to cross layers in dealing with TCP-aware processing. To avoid congestion control on end hosts, the BS attempts to foil fast retransmit by adaptively manipulating duplicate ACK packets.

One of the key AN capabilities that can be leveraged to assists with TCP over wireless is the ability to adapt quickly, perhaps even on a packet-by-packet basis. This ability derives from the fact that the code (or data used by that code) contained in APs can change in each packet. We will show an example of this based on link error control in Subsection 5.4.3.

A final AN capability of importance centers on AEs. As described in Chapter 2, AEs can be dynamically downloaded and can add to or modify the behavior of node resident code. The implication of this is that even protocols that need new or modified node resident functionality can be incrementally deployed on the fly. As an example, consider a MH that wishes to use an enhanced protocol that requires node-resident functionality at the BS. Assuming the BS supports AEs, then the MH can simply extend the BS. Essentially BS has been adapted to support the new protocol.

5.3 MANE Modifications

To support our current experiments, we need to modify the version of MANE discussed in the previous Chapters. Those changes are discussed here. In addition, many of our examples use the "chunk" feature of PLAN [42] and here we review this feature so that those examples are more understandable.

5.3.1 TCP Itself

Perhaps the most significant addition we made to MANE was an implementation of a TCP-like protocol. To achieve this, we added a data structure called the Transmission Control Block (TCB) [15]. Each TCP host should maintain information about a TCP connection and TCB is used to store this information. Among the variables maintained in the TCB, we added the basic ones needed for the congestion control, which included sequence numbers, round trip time (RTT) measures and variance, timeout values for retransmission, and the congestion window.

TCP senders are responsible for reliable transmission, therefore, they must have a buffer to store outstanding TCP segments and to retransmit them in case of timeout or duplicate ACKs. We equipped MANE nodes with a *send buffer*, which keeps outstanding TCP segments with timeout values. The corresponding ACKs free the segments from the buffer and update RTT measurements and congestion window size.

There are three new service functions for TCP transmission: `tcpSend()`,

`tcpAck()`, and `tcpRcv()`. On the sender side, `tcpSend()` initiates TCP transmission with random segment sequence number, and `tcpAck()` performs reliable delivery of TCP segments following the congestion control and fast retransmit algorithms. A TCP receiver checks if the delivery is in-order by calling `tcpRcv()`. The return value of `tcpRcv()` is the sequence number of the last segment received in order, which is used for cumulative ACKs.

5.3.2 Link Error Issues

For our current experiments, we added a simple error model in which a packets bits are flipped with some probability. Although this is certainly an overly simplistic model, it makes it easy to observe the performance of various approaches in either a low or high error regime.

For link-level error detection, we added a 32-bit Cyclic Redundancy Check (CRC-32) to our frames as a Frame Check Sequence (FCS). The FCS is calculated using the following standard generator polynomial of degree 32:

$$G(x) = x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1.$$

A node-resident service, `crc32()`, calculates a 32-bit CRC from a PLAN chunk. The CRC value is transmitted together with the chunk. As described in Chapter 2, chunks are used to describe the remote execution of PLAN programs on other nodes. When the chunk is evaluated, the named function is invoked with the arguments; remote evaluation is achieved so that the receiver can calculate the CRC of the arriving frame and compare it to the transmitted

value. Based on this error detection function, we can implement automatic repeat request (ARQ) schemes by programming in PLAN.

Some of our approaches also use forward error correction (FEC). Unlike ARQ, FEC requires a coding scheme for error correction as well as error detection. Because of the need to correct bursty errors that are common in wireless links, we adopt Reed-Solomon (RS) Codes as an FEC coding scheme [131, 132]. We use RS codes constituted by 8-bit code symbols, or over the Galois field $GF(2^8)$. An RS code is specified as $RS(n, k)$, which means that an n-symbol codeword is encoded from k data symbols and $n-k$ parity symbols. A $RS(n, k)$ code can correct up to $t = (n - k)/2$ symbols in a codeword. A node-resident function, `fecEncode()`, generates a new chunk of RS codewords from a given chunk with parameters n and k. On the remote host, the new chunk executes to decode the codeword and evaluates the original chunk.

5.3.3 Node-resident Variables for Channel Monitoring

To implement adaptive link error control, we need to monitor the state of our wireless link. Based on the measurements of the link, a host can dynamically change link error control schemes. As an indicator of channel state, we use an error counter (or bit error monitor), which counts corrupted packets in a time window [133]. Each wireless interface of a node maintains a history of packet errors in the time window. According to the value of the counter, policy modules can make decisions on which control scheme to use.

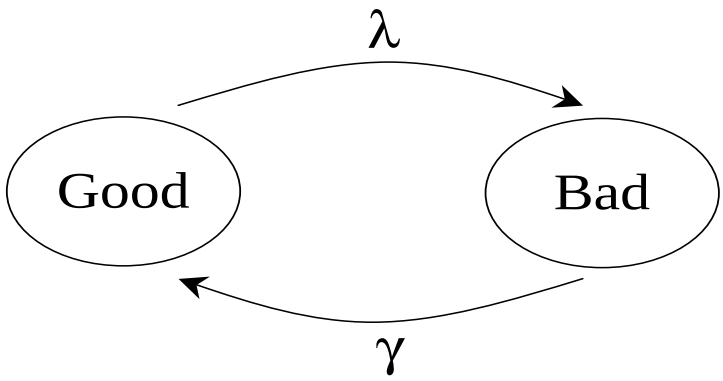

Figure 5.2: Wireless Channel Model in MANE

5.3.4 Interface Queue in the Link Layer

In our experiments, we show how to control link layer protocols using PLAN packets. Therefore, we added interface queues that allow PLAN packets to queue packets for retransmission. We added two functions, `enQueue()` and `deQueue()`. The function `enQueue()` stores a packet in the interface queue and sets a timer for retransmission. While, `deQueue()` clears a packet from the queue.

5.3.5 Channel Model

For the experiment of TCP over wireless links, we need to model wireless channel characteristics in MANE. It is shown that a first-order Markovian model is a good approximation of wireless channel [134, 135]. The wireless

channel in MANE is modelled after Gilbert-Elliott channel model [136, 137], which is a binary symmetric channel with transition probabilities, λ and γ. As in Figure 5.2, it follows a 2-state Markov process, in which the two states are denoted by G (good state) and B (bad state).

Losses occur with a low probability $p_{_G}$ in the good state (G), while they happen with a high probability $p_{_B}$ in the bad state (B). The steady state probabilities of being in states G and B are $\pi_{_G} = \gamma/(\lambda+\gamma)$ and $\pi_{_B} = \lambda/(\lambda+\gamma)$, respectively. Therefore, the average loss rate is $p = p_{_G}\pi_{_G} + p_{_B}\pi_{_B}$.

5.4 Horizontal Adaptive Link Error Control

One obvious way to improve TCP over wireless is simply to improve the error characteristics of the wireless link. As our first example, we consider how we can use the horizontal approach to implement this basic idea. The tricky issue is that how best to do this is a function of the link error rate, which is changing dynamically. If the error rates are very low, it might make sense to have no link-level functionality, much like many links in the wired network. At higher, but still moderate error rates, a basic ARQ scheme is employed because of its simplicity and low overhead. However, as error rates increase, frequent retransmissions degrade performance. Thus at high rates, to control errors more efficiently, FEC is added into ARQ. By combining two coding procedures, hybrid ARQ/FEC can get the benefits of both [138, 139]. In such a ARQ/FEC, the strength of the error-correcting code (ECC) can gradually

adapt [120]. In this section, we show how AN can be used to implement this adaptivity.

We can implement this scheme in PLAN with a supporting shim layer. For adaptive link error control, we place a shim layer between the link layer and network layer. When forwarding data packets, the shim layer protocol saves the packet in a buffer, generates a CRC or codeword from the payload, and sets a timer for retransmission. Then, it sends a PLAN packet encapsulating an error control algorithm, the payload, and the codeword in a chunk. When this chunk is executed at the destination, it completes the algorithm by checking the packet and sending an ACK (which takes the form of another chunk).

5.4.1 Basic ARQ

We begin by considering a simple ARQ scheme. For simplicity, we assume we have only one wireless hop. Thus we expect the round-trip times seen by the link-level ARQ to be small. Therefore, we adopt an idle RQ or *stop-and-wait ARQ* scheme rather than *selective-repeat ARQ* or *go-back-N ARQ* schemes [140]; however, it would be straightforward to include other ARQ schemes when the channel has a long round-trip delay with high transmission rates. In that case, we would not need to change the node-resident services, but would use a different PLAN program containing the required ARQ algorithm.

Figure 5.3 shows the PLAN code for our ARQ scheme. For error detection, the sender calculates a CRC-32 (Line 11) and sends a new chunk (`checkCRC`) containing the original chunk and the corresponding CRC (Line

```
1: fun checkCRC(chk, crc) =
2:  let val crcCalcul = crc32(chk)
3:      val nexthop = defaultRoute(getSrc()) in
4:     ( if(crcCalcul = crc) then (
5:         eval(chk);
6:         OnNeighbor(|deQueue|(), #1 nexthop, getRB(), #2 nexthop)
7:        ) else ()
8:     )  end
9:
10: fun arq(dst, chk) =
11:  let val crc = crc32(chk)
12:      val nexthop = defaultRoute(dst) in
13:  ( enQueue(|checkCRC|(chk, crc), #1 nexthop);
14:    OnNeighbor(|checkCRC|(chk, crc), #1 nexthop, getRB(),
15:                #2 nexthop)
16:  ) end
```

Figure 5.3: PLAN for basic ARQ

14). It also stores the packet in the interface queue for retransmission (Line 13). The destination evaluates the chunk, thus evoking `checkCRC`, which executes to compute the CRC of the received chunk and comparing it with the original CRC (Lines 2 and 4). If the results are the same, the original chunk is evaluated on the destination (Line 5). The destination is also required to generate a chunk to invoke the `deQueue()` function on the sender. This chunk works like an acknowledgment and frees the packet in the interface queue (Line 6). Note in practice, this ACK chunk might also implement other functionality as well, such as updating an RTT estimate.

This particular code is specialized for a single wireless hop because it always does the crc check on its neighbor. However, it could be used from

either the BS or the MH. Further, it would be easy to generalize this approach to support multiple wireless hops. In this case, if transmitted from the BS, it would simply defer execution of `checkCRC` until it reached its final destination and it would also need to carry with it the address of the BS to provide the "ack" with a destination. If transmitted from the MH, it would need to either know the address of the BS in advance, or, as we will show later, the BS would need to have a service function that identified it as a BS.

5.4.2 ARQ/FEC

By utilizing ARQ/FEC at high error rates, we can maintain constant throughput at the expense of encoding/decoding overhead and complexity. Figure 5.4 presents the PLAN code for a FEC scheme using Reed-Solomon codes (or type I hybrid ARQ [140]).

This code is similar to that for the basic ARQ. The key difference is that before the original chunk is transmitted it is encoded using Reed-Solomon coding (Line 11) and then when it is received, it is decoded (Line 2).

Although this code does not take advantage of this feature, by including the FEC strength in the chunks, we could control the level of error correction on a packet-by-packet basis. An example of where this might be of value would be in a system like 802.11 which precedes each data transmission with a request-to-send (RTS)/clear-to-send exchange(CTS). Then the RTS could act as a channel probe, while the CTS could return the channel state to the sender, which would then use it to determine how strong to make the FEC.

```
1: fun decode(codeword, scheme, n, k) =
2:  let val chk = fecDecoding(codeword, scheme, n, k)
3:      val nexthop = defaultRoute(getSrc()) in
4:     ( if(#1 chk = 0) then (
5:         eval(#2 chk);
6:         OnNeighbor(|deQueue|(), #1 nexthop, getRB(), #2 nexthop)
7:       ) else ()
8:     )  end
9:
10: fun fec(dst, chk) =
11:  let val codeword = fecEncoding(chk, "RS", 255, 223)
12:      val nexthop = defaultRoute(dst) in
13:     ( enQueue(|decode|(codeword, "RS", 255, 223), #1 nexthop);
14:       OnNeighbor(|decode|(codeword, "RS", 255, 223),
15:                       #1 nexthop, getRB(), #2 nexthop)
16:     ) end
```

Figure 5.4: PLAN for FEC

5.4.3 Adaptive Link Control

Now that we have both basic ARQ and ARQ/FEC the question is how to combine them so that the appropriate one is used based on the quality of the channel. Figure 5.5 presents code that does this by hybridizing three schemes: No Error Correction, ARQ and ARQ/FEC. This particular implementation is designed to be sent by a FH. It depends on the BS to identify itself by returning true when `isThisHostBS` is called as well as to maintain a measure of channel quality, queried by `isChanGood` or `isChanSoSo`. The basic idea is that the packet single hops through the network (Lines 2 and 16) looking for the BS (Line 3). At the BS, it queries the channel state (Line 4) and if it is

```
1: fun adapLink(dst, chk) =
2:  let val nexthop = defaultRoute(dst) in
3:  (if(isThisHostBS()) then (
4:     enQueue(|adapLink|(dst, chk), #1 nexthop);
5:     if(isChanGood(#1 nexthop)) then (
6:       let val crc = crc32(chk) in
7:         OnNeighbor(|checkCRC|(chk, crc), #1 nexthop, getRB(),
8:                       #2 nexthop) end )
9:     else (
10:      let val codeword = fecEncoding(chk, "RS", 255, 223) in
11:        OnNeighbor(|decode|(codeword, "RS", 255, 223),
12:                              #1 nexthop, getRB(), #2 nexthop)
13:      end ) )
14:  else
15:    OnNeighbor(|adapLink|(dst, chk), #1 nexthop, getRB(),
16:                  #2 nexthop)
17:  ) end
```

Figure 5.5: PLAN for Adaptive Link Control

good, it doesn't use error control (Line 5). If the channel error rate is medium (Line 8), uses basic ARQ (Lines 9 – 11) otherwise it uses ARQ/FEC (Lines 13 – 15). Note that `checkCRC` and `decode` are the same as in the preceding examples. This example shows that we can essentially implement protocol boosters that switch their protocol on a packet-by-packet basis.

This particular example assumes that the FH can send APs, but it could be easily adapted to the case where only the BS and MH were active. In that case, the BS would receive a non-active packet, which it would then encapsulate in either no error correction chunk, or a ARQ chunk or a ARQ/FEC chunk as appropriate. The MH would execute the chunk, passing the non-active packet

up the stack if appropriate. This idea on encapsulating "normal" packets inside of PLAN chunks is essentially the key to allowing "active" islands to exist in a sea of "normal" networks.

We could also use a similar approach to the above when the MH is the sender. In that case, if there were multiple wireless hops we would still need to search for the BS and would evaluate the CRC or FEC there and send the "ack" back to the MH. In any of these approaches, the fact that the algorithm is encoded in the packet means that we can apply this adaptation on a packet basis.

5.4.4 AN for Channel Monitoring

For adaptive link control, we need to track the state of the channel. One approach is for the sender to use ACKs (or rather their lack) to tell when the channel is bad. With AN it is easy to do better. The key observation is that the receiver is in the best position to monitor the channel, while the sender is the one that needs this information. Assuming the receiver records channel information, we can use APs to query this state.

Figure 5.6 shows the code for an out-of-band channel monitor. The function `getChanInfo()` (Line 2) defines a standard interface to get information on channel characteristics. This function returns various channel information depending on the parameter, `indicator`, such as the Received Signal Strength (RSS) or Signal-to-Noise Ratio (SNR). The code composes a query and sends it (Line 9). The query executes on the receiver and returns the result to the

```
1: fun report(indicator) =
2:   let val measure = getChanInfo(indicator)
3:       val src = defaultRoute(getSrc()) in
4:         OnNeighbor(|print|(measure), #1 src, getRB(), #2 src)
5:   end
6:
7: fun probe(dst) =
8:   let val nexthop = defaultRoute(dst) in
9:     OnNeighbor(|report|("RSS"), #1 nexthop, getRB(), #2 nexthop)
10:  end
```

Figure 5.6: PLAN for Monitoring RSS

sender (Line 4). Note that this is much more flexible than the conventional approach, which would require specifying a special packet format and protocol for such queries.

An important variation on this idea would be to piggyback the query chunk on a data packet. This is easy to do because chunks are data and it is easy to compose various chunk oriented calculations. The result is that such queries can be done without sending additional packets and yet remain transparent to the data flow. This ability to piggyback control on data transparently, solves a key problem with Protocol Boosters, controlling when to add or remove a booster.

Finally, consider our example above where RTS/CTS packets were used to monitor the channel just before sending a data packet. In a conventional network, this would require changing the format and function of the RTS and CTS. However, if the wireless link sent APs for its RTS and CTS, then adding

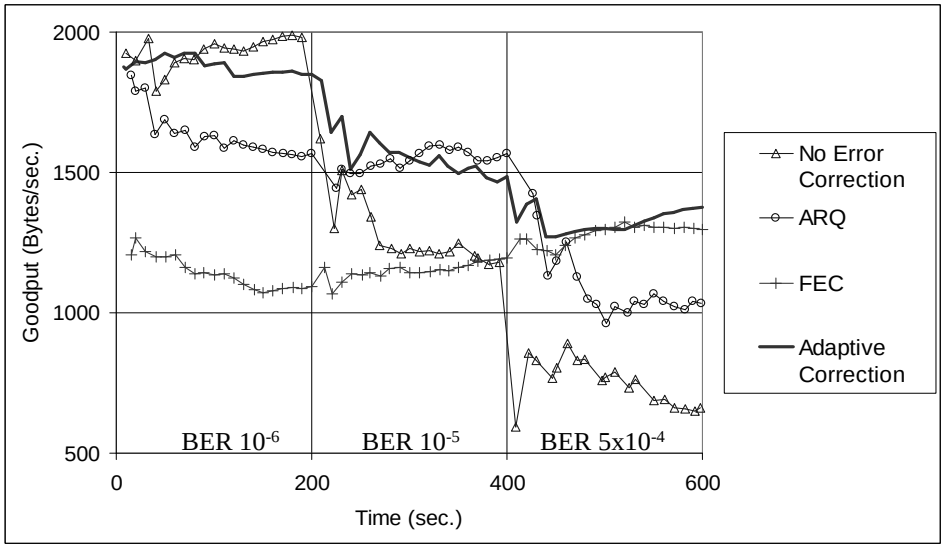

Figure 5.7: Comparison of Link Error Control Techniques

to or modifying the function of these parts of the protocol would become just a matter of packet programming.

5.4.5 Performance Evaluation

In Figure 5.7, we present a performance comparison of various versions of the link error control mechanisms discussed above: No Error Correction, ARQ, FEC, and Adaptive Correction. In this evaluation, the MH is the TCP sender and the FH is the TCP receiver, while adaptive link control is over the wireless link between the MH and the BS. Adaptive Correction is based on ideal channel estimation, which assumes that it is possible to track channel states quickly

and accurately. When the error rate is very low, the adaptive protocol does not use error correction. At low error rate, the adaptive scheme adopts the ARQ scheme. When the error rate is increased, it uses the FEC scheme.

Figure 5.7 has time along the X-axis and goodput in bytes per second along the Y-axis. Initially, the channel is quite reliable with a low error rate (Bit Error Rate (BER) $\approx 10^{-6}$); but its error rate increases at time 200 (BER $\approx 10^{-5}$) and then at time 400, the channel state worsens (BER $\approx 5 \times 10^{-4}$). At low error rate, the channel is reliable and the performance of No Error Correction is better than those of ARQ and FEC due to less overhead; Adaptive Correction adopts No Error Correction and shows better performance than that of ARQ and FEC. At medium error rate, the ARQ scheme outperforms No Error Correction and FEC due to light error correction. In this period, Adaptive Correction adapts and uses ARQ. At high error rate, FEC's performance does not change significantly, while No Error Correction and ARQ is severely affected and their performance deteriorates; Adaptive Correction adapts to use FEC and avoids severe performance degradation. While the performance of the adaptive protocol is best overall, sometimes it does not show exactly the same performance of the best protocol in a period. For example, between 100 and 200 seconds, the adaptive protocol is outperformed by No Error Correction. We think that this might result from experimental errors.

In Figure 5.7, our main goal is to show the adaptivity achievable using AN based on exact channel monitoring, not to devise a better channel estimation scheme. Figure 5.8 shows goodput comparison of the adaptive protocol

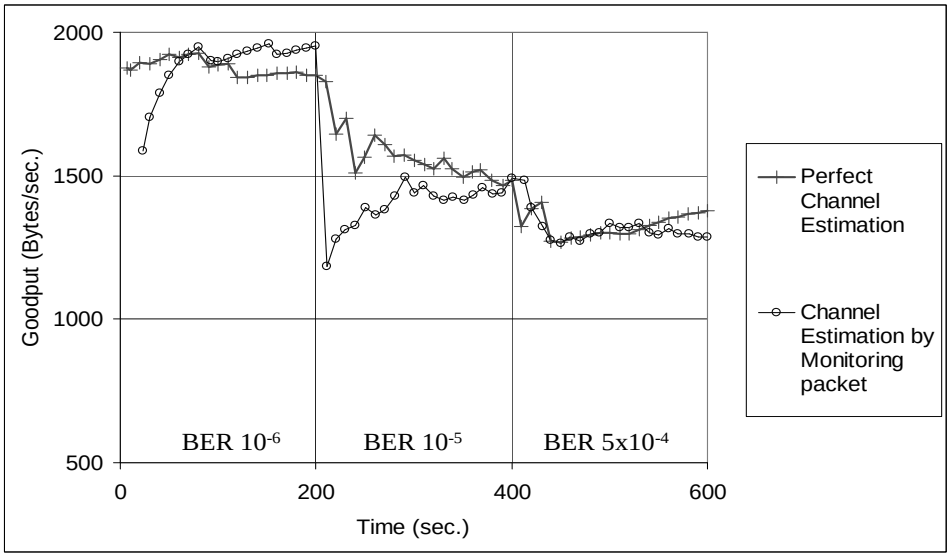

Figure 5.8: Comparison of Ideal Estimation and Active Hybrid

when the monitor traces the channel changes perfectly and when the monitor uses out-of-band PLAN packets to monitor the channel state and thus drive adaption. In this case, it does not do as well as the adaptive protocol based on perfect channel estimation at medium error rate, because sometimes the monitor believes that the channel is better than it really is and sends packets without link control rather than doing ARQ. Thus, packet errors are recovered by TCP retransmission rather than local retransmission of ARQ. Since TCP retransmission takes longer than ARQ local retransmission, the adaptive protocol with out-of-band PLAN packets does not do as well as the one based on ideal channel monitoring. We believe that more experience would allow us to track the channel more closely.

5.5 Vertical Snoop Protocol

Even with adaptive link error control, packet drops may be still possible and the resulting congestion control action can cause performance degradation. In vertical adaptivity, collaboration and information sharing across layers on BS are allowed to adaptively control the TCP flow. We claim that AN is advantageous because AN facilitates cross-layering implementation by allowing layer-specific information to be included in active packets.

In this approach, the lower layer protocols on the BS are aware of TCP semantics and adjust TCP flow information to prevent congestion control from taking place due to packet drops over wireless links. Further, by following up the parts of the end hosts' TCP Control Block (TCB) [15], the BS can take actions on incorrect congestion control, such as adjusting RTT measures and screening three duplicate ACKs. In this section, we show how to deploy the snoop protocol onto the BS, which can improve performance of TCP connections from fixed hosts (FH) to mobile hosts (MH).

5.5.1 Snoop Protocol

Figure 5.9 shows the PLAN code for an AN version of the snoop protocol. Like our adaptive link example, this particular implementation is designed to be sent by a FH, but, as we described for the previous protocol, a similar version could be implemented on the BS in a system where the FH was not active.

The FH initiates data transmission by invoking `tcpSend()` (Line 35)

```
1: fun snoopack(src, seq) =
2:  let val nexthop = defaultRoute(src) in (
3:    if(thisHostIs(src)) then
4:      tcpAck(seq)
5:    else (
6:      if(isThisHostBS()) then (
7:        if(isDupAck(seq)) then ()
8:        else (
9:          OnNeighbor(|snoopack|(src, seq), #1 nexthop,
10:                     getRB(), #2 nexthop) );
11:       deQueue(seq) )
12:     else
13:       OnNeighbor(|snoopack|(src, seq), #1 nexthop,
14:                 getRB(), #2 nexthop)) ) end
15:
16: fun snoop(src, dst, seq, payload) =
17: if(thisHostIs(dst)) then (
18:    let val last_seq = tcpRcv(seq, payload)
19:        val nexthop2src = defaultRoute(src) in
20:          OnNeighbor(|snoopack|(src, last_seq),
21:                 #1 nexthop2src, getRB(),#2 nexthop2src)
22:    end )
23:  else (
24:    let val nexthop2dst = defaultRoute(dst) in (
25:      if(isThisHostBS()) then
26:        enQueue(|snoop|(src, dst, seq, payload), dst, seq)
27:      else ();
28:      OnNeighbor(|snoop|(src, dst, seq, payload),
29:                  #1 nexthop2dst, getRB(), #2 nexthop2dst))
30:    end )
31:
32: fun tcpsnoop(src, dst, payload) =
33:  let val seq = tcpGetSeq()
34:  in
35:    tcpSend(|snoop|(src, dst, seq, payload), dst, seq, getRB())
36:  end
```

Figure 5.9: PLAN for the Snoop Protocol

with a sequence number returned by `tcpGetSeq()` (Line 33). The first parameter of the function is a chunk containing the algorithm for the snoop protocol. When the packet arrives at the BS (Line 25), it is saved in the queue for retransmission (Line 26). At the destination, an ACK is sent back in a chunk `snoopack` (Lines 20 – 21). This chunk not only delivers the ACK segment to the source, but completes the realization of the snoop protocol on the BS. Duplicate ACKs are discarded (Line 7), and the saved packet is freed from the interface queue (Line 11).

This example shows how to deploy a new protocol easily. There is no need to update protocol stacks on the BS. Service extensions on the BS mainly implement the cross-layering mechanisms. As an adaptation layer, the service extensions transform active packets to TCP segments or vice versa. Evaluation of the PLAN packet on BS actualizes the snoop protocol and enhances TCP performance over wireless links.

5.5.2 Performance Evaluation

Figure 5.10 presents the simulation result to compare performances between regular TCP and the snoop protocol. In this case, the FH is the sender and the MH is the receiver. The X-axis represents simulation time, while the Y-axis shows goodput. The evaluation time is 800 seconds and divided to four periods of 200 seconds. Initially, there are no wireless channel errors (BER = 0) during the first 200 seconds. In the second period from 200 to 400 seconds, the error rate is very low (BER $\approx 10^{-6}$). During the first half of the

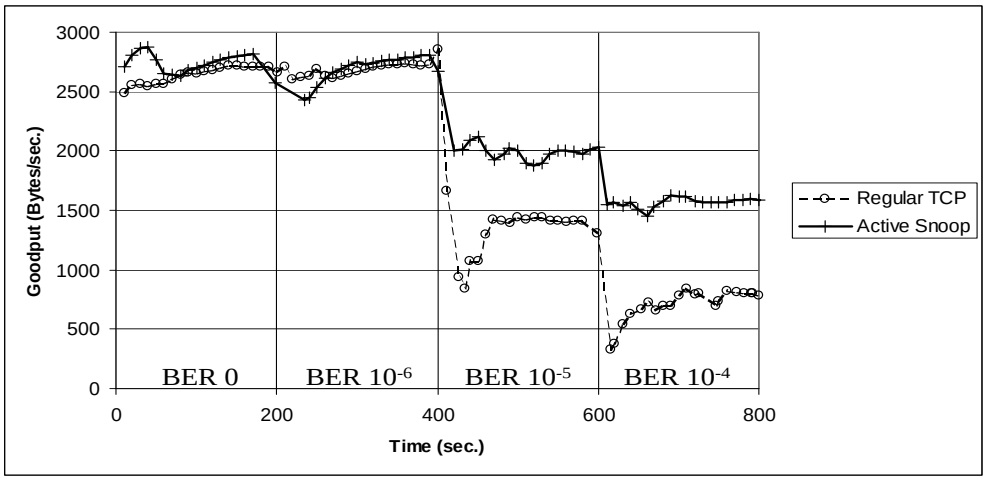

Figure 5.10: Goodput Comparison between Regular TCP and Snoop Protocol

evaluation, in which the error rates are zero or quite low, the performances of the two protocols are not much different. On the other hand, when the error rates are getting higher at 400 second (BER $\approx 10^{-5}$) and 600 second (BER $\approx 10^{-4}$), the snoop protocol outperforms regular TCP. As expected, coarse timers and unnecessary congestion control deteriorate the regular TCP over wireless links with high error rates. When the error rates are high, the snoop protocol enhances performance by preventing unnecessary congestion control and by locally retransmitting the lost frames.

5.6 Advantages of AN for Adaptation and Evolution

In this section, we compare the adaptivity and evolvability of the traditional network to those of the AN-enabled network. We make this comparison in terms of the ability to adapt link error control protocols and to evolve the network to support transparent solutions. In each of these areas AN facilitates adaptation and evolution.

5.6.1 Adaptive Link Control

Standardization

Traditional Arch. To achieve adaptivity, it is necessary to design an adaptive scheme based on anticipated environments. The design should be verified before standardization and deployment. As in the case of RLP of 3GPP specifications [126], there is no room for further adaptation beyond the one designed in. If new link environments are seen, the intended adaptivity could be suboptimal. To meet the new link characteristics, a new standardization process would have to be started from scratch.

AN Arch. Since AN facilitates adaptation by supporting protocol implementation in a lightweight way using packet programming, there are no substantial demands on standardization for adaptive schemes. Also, AN provides flexibility to cope with the environments not anticipated in the design phase

without further need for standardization.

Conclusion AN eases or eliminates the burden imposed by standardization.

Implementation

Traditional Arch. For the hybrid ARQ/FEC scheme, a specific policy about hybridizing needs to be defined and their implementations should be fixed before deployment. Once they are deployed, their operation cannot be changed when the channel conditions are changed unexpectedly. Further, if there is a need to measure channel states, there is a need to design and implement special control protocols. The control protocol should be implemented as part of the link layer protocol. Software implementation of adaptive link control is large and complicated, since the link layer protocol is implemented in static software containing each link error control schemes, control schemes for channel monitoring, and interfaces between them.

AN Arch. AN enabled us to actualize link control protocols by packet programming. Since the link error control scheme is embedded in packets in a lightweight way, we were able to adapt the link control protocol on a packet-by-packet basis. Also, we achieved channel monitoring by piggybacking function calls independently of the packet payload. It took one week to implement the link error control protocols and the numbers of line of code are shown in Table 5.2. For the ARQ and FEC schemes, we needed to implement the supporting service functions in Popcorn to calculate CRC 32 values and to

Control Scheme	Line of Code
ARQ	140 in Popcorn + 22 in PLAN
FEC	673 in Popcorn + 22 in PLAN
Hybrid ARQ/FEC	0 in Popcorn + 41 in PLAN

Table 5.2: MANE Implementation for Adaptive Link Control

encode/decode Reed-Solomon codes, respectively. Hybrid ARQ/FEC scheme was implemented by combining PLAN code without changes of the service functions on nodes.

Conclusion Because AN enables us to implement protocols by packet programming, it allows shorter and simpler implementations. Thus it is easier to evolve the network from an implementation point of view.

Deployment

Traditional Arch. In order to deploy an adaptive protocol, it is required to model target environment and to design the protocol within the target environment. Since the link layer protocol is closely coupled with the physical layer characteristics, adaptive link layer protocols should take the physical environment into consideration. Once the protocol is designed and deployed, adaptivity is limited to the target environment. If new adaptivity is required, new link layer protocols should be designed, standardized, implemented and deployed from scratch.

AN Arch. AN supports packet level adaptation, thus given the standard interface to extensions, it can flexibly deploy an adaptive protocol in a lightweight way. If there is no existing interface with active extensions, AN supports network evolution by dynamically updating active extensions. After the deployment, AN facilitates adaptation to new environments either by packet programming or extension updating.

Conclusion In the traditional architecture, each adaptive protocol has its own target environment and their adaptivity is determined when deployed. Further adaptivity or evolution is difficult without changing the system installation. Since AN supports lightweight system installation and upgrade, AN is better from the viewpoint of deployment.

5.6.2 Transparent Performance Enhancing Proxy (PEP)

Standardization

Traditional Arch. The snoop protocol is one of the well-known transparent PEP approaches. The snoop protocol was shown to outperform the regular TCP over wireless links in 1995 [102]. Also, there have been several implementations of the snoop protocol [129, 130]. In spite of that, neither standardization nor deployment of the snoop protocol have been started yet. An *informational* RFC mentions the snoop protocol as an example of TCP PEP with the discussion of limitations, such as the interoperability problem with IPSec, duplicate efforts of error recovery, and layering violation [110]. To

standardize the snoop protocol, it is necessary to address and resolve those problems. Other transparent approaches, such as I-TCP [108], have been only discussed in literatures without any further standardization.

AN Arch. In the AN architecture, layering is relaxed and layering violations are feasible in both design and standardization. Besides, since protocols are implemented in packets and supporting extensions, it is flexible on a packet basis to avoid the interoperability problem or duplicate functionality. Thus, AN makes much simpler demands on the standardization.

Conclusion With the simpler demands on standardization and less layering restriction, AN facilitates network evolution for transparent PEPs.

Implementation

In order to implement the snoop protocol, we need Base Stations that can understand TCP semantics and maintain state for TCP flows. Specifically, the link layer protocol on the Base Stations should;

- Track TCP data and ACK segments,
- Cache outstanding TCP data packets,
- Maintain local timers,
- Retransmit by either local timeouts or duplicate TCP ACKs, and
- Suppress duplicate ACKs from the mobile host.

Criteria	Berkeley	Linux Snoop	MANE
Line of Code	1632 in C	1156 in C	140 in Popcorn + 68 in PLAN

Table 5.3: Comparison of Snoop TCP Implementations

Traditional Arch. In order to deploy the snoop protocol in the traditional network, the protocol stack on the base stations needs to be changed to support the aforementioned TCP-aware operations. As in the case of the snoop protocol, transparent PEPs require a certain functionality to reside on the proxy. For the proxy functionality, it is usually required to implement this in a kernel module. Even if the snoop protocol is implemented, it is a limited solution that only addresses the problem of TCP-specific data flow from the fixed host to mobile host.

AN Arch. Since substantial parts of the snoop protocol can be implemented by packet programming, AN needs a small set of basic services implemented on base stations. Also, AN relaxes layering boundaries and supports more flexible design and extensible approaches.

Table 5.3 shows the lines of code of two snoop implementations and MANE implementation. The Berkeley Snoop protocol software was developed by the Daedalus Research Group at UC Berkeley and released in August 1998 [129]. Linux Snoop is the implementation of the Snoop Protocol by the National

University of Singapore [130]. Linux Snoop was released in February 2004. In MANE, it took a day to implement the snoop protocol. The MANE implementation is about 8 times fewer lines than the Berkeley implementation and 5.5 times fewer lines than Linux Snoop.

Conclusion It appears that AN admits significantly shorter and simpler implementations. Simpler implementations are easier to create and change and thus it is easier to evolve the network from an implementation point of view.

Deployment

Since neither standardization nor real deployment of the snoop protocol has been available yet, it is hard to discuss its deployment. However, the following presents how the functionality discussed in Section 5.1.4 might be deployed in the traditional architecture and the AN architecture.

Traditional Arch. To support the snoop protocol, there is a need for a protocol stack modification on every Base Stations; the program maintains soft state of TCP connections and intervene in transport. It would take significant time to deploy a new program throughout the current Internet or to change the deployed program on Base Stations. Furthermore, only manufactures can change the already-deployed base stations. Users and network administrators are dependent on the manufactures for program deployment and system upgrade.

AN Arch. Based on soft state maintenance, the new functionality is deployed just by sending the appropriate packets. This only takes as long as the time to transmit the packets and execute. AN would make it possible for base station owners to evolve their own base stations.

Conclusion Until now, transparent PEPs, such as the snoop protocol, have not been available in the network. To install the new protocol on base stations, the current network will take downtime of several minutes to deploy on each base station, as compared to seconds for AN. Further, in the AN architecture, it is possible for users rather than manufactures to deploy new functionality to the existing infrastructure. From the deployment point of view AN provides superior evolution ability.

5.7 Other Possibilities

Using the model and architecture we developed, we explored a series of implementations of adaptive link control and of the TCP snoop protocol. These concrete examples are also a vehicle for exploring other design possibilities, thus broadening support for our claims. Here are some other ideas that are thought to be worth exploring and which may prove useful in improving end-to-end connections over wireless links and demonstrating the usefulness of AN.

Horizontally Adaptive Routing Another way to improve link performance is to utilize multiple routes. Since it is possible for a MH to connect with

more than one BS, the MH can communicate with a FH through different routes redundantly. The network layers of the MH and the BSes adaptively change paths between them so that link fluctuation does not affect the end-to-end flow. Furthermore, we can expand this approach into support for handoff.

Adaptive MTU size Because wireless links are prone to bit errors, the size of a frame is important; the larger a frame is, the higher is the probability of frame errors [141]. The frame size is determined by the Maximum Transmission Unit (MTU) of the wireless link. The MTU is a link layer restriction on the maximum size of a datagram in a single transmission. If an IP datagram is larger than an MTU, it is fragmented by a network layer protocol. MTU size is link layer specific, but has significant impacts on overall performance over wireless links [124]. Therefore, the MTU size for a wireless link can adapt to link conditions. Fragmentation and link error correction protocols are required to cooperate because there is a close correlation between them.

End-to-End Adaptivity If the FH is AN-capable, we can develop more efficient adaptive control over TCP flows. We expect to apply this advantage to handling handoff, during which harsh link deterioration and route changes happen at the same time.

Chapter 6

Contributions

Our main contribution is to show that Active Networking's adaptability and evolvability can provide significant benefits to mobile networks. Through three case studies, we demonstrated how AN can evolve networks to support mobility and adapt mobile networks for better performance. Detailed contributions include:

- Evolving networks to support Mobile IP style mobility
  - Active Packet Evolution
    - Demonstrated that when an application is aware of the possibility of mobility, mobility can be supported using no more than Active Packets and simple soft state at a home router.
    - Demonstrated that PLAN chunks can be used to construct tunnels, in particular between a Home Agent and a node acting as

a Foreign Agent. Unlike conventional Mobile-IP, the Foreign Agent node does not need to be preprogrammed to serve this role, all that is needed is support for APs.

    * By using APs to correct the proxy ARP bug, showed that APs can support dynamic error-correction.

  - Update Extension Evolution

    * Demonstrated that by using update extensions, we can support mobility transparently even when the system design has not anticipated adding the mobility service; mobility can be supported for normal packets as well as application-aware packets.

    * Demonstrated that new functionality can be easily added or modified and that dynamic updating provides flexibility to support wider variety of protocols than APs or plug-ins only.

- Ad Hoc Routing Deployment, Evolution, and Adaptation

  - Demonstrated that if activeness prevails throughout an ad hoc network, protocol deployment can be accomplished in a timely manner and that the network can overcome routing heterogeneity, which is quite likely in ad hoc networks.

  - Demonstrated that APs facilitate the evolution of a routing protocol on the fly for better performance.

  - Demonstrated that, based on generic services, it is easy to adapt a

routing protocol to its environment, such as level of mobility, using APs.

  - Showed that AN is beneficial for customizing a routing protocol, such as changing the caching policy or adopting a routing metric in a lightweight way.

- Enhancing TCP Performance over Wireless Links

  - Architectural model

    * Modelled general TCP sessions in wireless networks including the cases of multi-hop wireless and mobile host being a TCP sender.

    * Identified where and what kind of AN technology can be deployed, and showed a strategy for incremental deployment of AN into the current network.

    * Suggested two main architectural approaches, horizontal and vertical, when AN and non-AN entities are mixed in the network.

  - Adaptive Link Error Control

    * Demonstrated that, as an example of the horizontal approach, AN facilitates the implementation of link-error control protocols and the adaptation of the link control on the fly.

    * Channel Monitoring

· Demonstrated that APs piggybacking PLAN chunks for channel monitoring provide a novel technique for multiplexing logically out-of-band control on top of physically in-band data transmissions.

· Demonstrated that PLAN chunks are useful for policy customization.

– Snoop Protocol Deployment

* Demonstrated that, as an example of the vertical approach, Active Extensions and Active Packets together promote flexible cross-layer protocol design and implementation.

Chapter 7

Conclusion

Mobile networking implies new services and protocols; however, it is not easy to add these new services and protocols to the current network because of architecture limits. Mobile networking is an area in which adaptivity and evolvability is likely to be especially important.

In order to support our thesis that Active Networking (AN) can provide useful adaptivity and evolvability to mobile networks, we demonstrated AN's adaptivity by developing a series of designs, simulation studies, and working prototypes. Because AN enables highly flexible packet functionality, on-the-fly protocol deployment, even on a packet-by-packet granularity, and cost-effective network expansion, it is possible for mobile networks to evolve agilely and to adapt to changing network environments.

In the first case study of Mobile IP, we could easily deploy the new protocol so that the network supports the new service of mobility. Using AN's

evolution techniques, the network was able to tunnel and forward packets to the mobile host either with or without the help of a foreign agent. In this case, we presented two kinds of Mobile IP examples that illustrate what expressibility gains are possible as successively more powerful techniques are used.

In the case of ad hoc routing, we showed how easily a routing protocol can be deployed and evolved. Furthermore, we addressed the routing heterogeneity problem using the extensibility of AN. As another example of adaptivity, we presented how to adapt a routing protocol to mobility. In this experiment, we used a routing-aware AP to show the ease of implementing a protocol in packet programming. By simply combining two packet programs and common services, we could adaptively change the routing protocol and enhance routing performance. If we expand this approach and combine several routing protocols in a hybrid form, we are able to achieve an improved protocol that can adapt itself to the conditions changing over various ranges and that performs best under the given conditions.

In the last case study of the TCP problem over wireless links, we addressed the architectural design and suggested a design space followed by a number of possible options to explore. We presented two approaches: horizontal and vertical. In the horizontal approach, AN enabled the link layer protocol to control link errors adaptively and to improve performance transparently to TCP hosts. We also presented AP's usability in monitoring wireless channel states. In the vertical approach, we took the snoop protocol as an example

to show that AN's flexibility facilitates cross-layering to improve TCP performance.

As we pointed out in Chapters 4 and 5, there are several adaptations worthy of future work. For example, we can extend AN's adaptivity to proactive ad hoc routing protocols, QoS routing, and power-aware routing. Also, for the TCP problem, if sufficiently powerful AN technology were deployed everywhere, then we would be able to deploy end-to-end, nontransparent solutions easily and to evolve them as better solutions were developed. With this future work, we expect the larger result will be continued support of our thesis.

Bibliography

[1] K. G. Anagnostakis, "Congestion Control in Packet-Switching Internetworks," Technical Report, University of Pennsylvania, April 2001.

[2] M. S. Blumenthal and D. D. Clark, "Rethinking the Design of the Internet: The End-to-End Arguments vs. the Brave New World," *ACM Trans. on Internet Technology*, vol. 1, pp. 70–109, August 2001.

[3] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden, "A Survey of Active Network Research," *IEEE Communications Magazine*, vol. 35, pp. 80–86, January 1997.

[4] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura, "Active Networking and the End-to-End Argument," in *Intl. Conf. on Network Protocols (ICNP'97)*, (Atlanta, GA), pp. 220–228, IEEE, Oct. 1997.

[5] "The Network Simulator - ns-2." http://www.isi.edu/nsnam/ns/.

[6] "VINT Project." http://www.isi.edu/nsnam/vint/index.html.

[7] S.-K. Song, S. Shannon, M. Hicks, and S. Nettles, "Evolution in Action:

Using Active Networking to Evolve Network Support for Mobility," in *Proc. of IWAN 2002*, pp. 146–161, Dec. 2002.

[8] M. Hicks, J. T. Moore, and S. Nettles, "Dynamic Software Updating," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 13–23, ACM, June 2001.

[9] J. T. Moore, M. Hicks, and S. Nettles, "Practical Programmable Packets," in *Proc. of the Twentieth IEEE Computer and Communication Society INFOCOM Conference*, pp. 41–50, IEEE, April 2001.

[10] E. Nygren, S. Garland, and M. F. Kaashoek, "PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems," in *Proc. of IEEE Conference on Open Architectures and Network Programming (OPENARCH'99)*, pp. 78–89, IEEE, March 1999.

[11] J. Ioannidis, D. Duchamp, and Gerald Q. Maguire, Jr., "IP-based Protocols for Mobile Internetworking," in *Proc. of SIGCOMM'91 Conference on Communications Architecture and Protocols*, pp. 235–245, September 1991.

[12] C. Perkins, "IP Mobility Support for IPv4," RFC 3344, IETF, August 2002. http://www.ietf.org/rfc/rfc3344.txt.

[13] C. E. Perkins, ed., *Ad Hoc Networking*. Addison-Wesley, 2000.

[14] W. R. Stevens, *TCP/IP Illustrated*, vol. 1. Addison-Wesley, 1994.

[15] J. Postel, "Transmission Control Protocol," RFC 793, IETF, Sep. 1981. http://www.ietf.org/rfc/rfc793.txt.

[16] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz, "A Comparison of Mechanisms for Improving TCP Performance over Wireless Links," in *Proc. of ACM SIGCOMM'96*, pp. 256–269, 1996.

[17] K. Pentikousis, "TCP in Wired-cum-Wireless Environments," *IEEE Communications Surveys*, pp. 2–14, Fourth Quarter 2000.

[18] S. Dawkins, G. Montenegro, M. . Kojo, and V. Magret, "End-to-end Performance Implications of Slow Links," RFC 3150, IETF, July 2001. http://www.ietf.org/rfc/rfc3150.txt.

[19] S. Dawkins, G. Montenegro, M. . Kojo, V. Magret, and N. Vaidya, "End-to-end Performance Implications of Links with Errors," RFC 3155, IETF, August 2001. http://www.ietf.org/rfc/rfc3155.txt.

[20] J. Stamos and D. Gifford, "Remote Evaluation," *ACM Trans. on Programming Languages and Systems*, vol. 12, pp. 537–565, October 1990.

[21] A. Fuggetta, G. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Trans. on Software Engineering*, vol. 24, pp. 342–361, May 1998.

[22] D. C. Feldmeier, A. J. McAuley, J. M. Smith, D. S. Bakin, W. S. Marcus, and T. M. Raleigh, "Protocol Boosters," *IEEE Journal on Selected Areas in Communications*, vol. 16, pp. 437–444, April 1998.

[23] D. Alexander, W. Arbaugh, M. Hicks, P. Kakkar, A. Keromytis, J. Moore, C. Gunter, S. Nettles, and J. Smith, "The SwitchWare Active Network Architecture," *IEEE Network Magazine*, vol. 12, no. 3, pp. 29–36, 1998.

[24] D. Wetherall, J. Guttag, and D. Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," in *Proc. of IEEE Conference on Open Architectures and Network Programming (OPENARCH'98)*, April 1998.

[25] B. Schwartz, A. Jackson, W. Strayer, W. Zhou, R. Rockwell, and C. Partridge, "Smart Packets: Applying Active Networks to Network Management," in *ACM Trans. on Computer Systems*, vol. 18, pp. 67–88, February 2000.

[26] M. Hicks, P. Kakkar, J. Moore, C. Gunter, and S. Nettles, "PLAN: A Packet Language for Active Networks," in *Proc. of ACM SIGPLAN International Conference on Functional Programming Languages*, pp. 86–93, ACM, September 1998.

[27] I. Wakeman, A. Jeffrey, and T. Owen, "A Language-Based Approach to Programmable Networks," in *Proc. of IEEE Conference on Open Architectures and Network Programming (OPENARCH'00)*, pp. 128–137, IEEE, March 2000.

[28] K. Hino, T. Egawa, and Y. Kiriha, "Open Programmable Layer-3 Net-

working," in *Proc. of the Sixth IFIP Conference on Intelligence in Networks (SmartNet 2000)*, pp. 133–150, September 2000.

[29] S. Bhattacharjee, *Active Networking: Architecture, Compositions, and Applications*. PhD thesis, Georgia Institute of Technology, August 1999.

[30] Y. Yemini and S. da Silva, "Towards Programmable Networks," in *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, October 1996.

[31] S. Karlin and L. Peterson, "VERA: An Extensible Router Architecture," *Computer Networks*, vol. 38, no. 3, pp. 277–293, 2002.

[32] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner, "Router Plugins: A Software Architecture for Next Generation Routers," *ACM SIGCOMM Computer Communication Review*, vol. 28, pp. 229–240, October 1998.

[33] D. S. Decasper, B. Plattner, G. M. Parulkar, S. Choi, J. D. DeHart, and T. Wolf, "A Scalable High-Performance Active Network Node," *IEEE Network*, vol. 13, pp. 8–19, Jan.-Feb. 1999.

[34] M. Hicks, J. Moore, D. Alexander, C. Gunter, and S. Nettles, "PLANet: An Active Internetwork," in *Proc. of IEEE INFOCOM'99*, pp. 1124–1133, IEEE, March 1999.

[35] D. S. Alexander, W. A. Arbaugh, A. D. Keromytis, and J. M. Smith,

"A Secure Active Network Environment Architecture: Realization in SwitchWare," *IEEE Network*, vol. 12, pp. 37–45, May-June 1998.

[36] A. W. Jackson, J. P. Sterbenz, M. N. Condell, and R. R. Hain, "Active Network Monitoring and Control: The SENCOMM Architecture and Implementation," in *Proc. of the DARPA Active Networks Conference and Exposition (DANCE'02)*, pp. 379–393, May 2002.

[37] O. Frieder and M. E. Segal, "On Dynamically Updating a Computer Program: From Concept to Prototype," *Journal of Systems and Software*, vol. 14, pp. 111–128, February 1991.

[38] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes, "Runtime Support for Type-Safe Dynamic Java Classes," in *Proc. of the 14th European Conference on Object-Oriented Programming*, pp. 337–361, June 2000.

[39] G. Hjálmtýsson and R. Gray, "Dynamic C++ Classes: A lightweight mechanism to update code in a running program," in *Proc. of the USENIX Annual Technical Conference (NO 98)*, pp. 65–76, June 1998.

[40] M. Hicks and S. Nettles, "Active Networking means Evolution (or Enhanced Extensibility Required)," in *Proc. of the Second International Working Conference on Active Networks*, pp. 16–32, October 2000.

[41] "Objective Caml." http://caml.inria.fr/ocaml/.

[42] J. T. Moore, M. Hicks, and S. M. Nettles, "Chunks in PLAN: Language Support for Programs as Packets," in *Proceedings of the 37th Annual Allerton Conference on Communication, Control, and Computing*, September 1999.

[43] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From System F to Typed Assembly Language," in *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (San Diego, CA, USA), pp. 85–97, January 1998.

[44] G. C. Necula, "Proof-Carrying Code," in *The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (Paris, France), pp. 106–119, January 1997.

[45] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic, "TALx86: A Realistic Typed Assembly Language," in *ACM SIGPLAN Workshop on Compiler Support for System Software*, pp. 25–35, May 1999.

[46] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, vol. 63, pp. 1278–1308, September 1975.

[47] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Addison-Wesley, 2nd ed., 1999.

[48] M. Hicks, S. Weirich, and K. Crary, "Safe and Flexible Dynamic Linking of Native Code," in *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation* (R. Harper, ed.), vol. 2071 of *Lecture Notes in Computer Science*, Springer-Verlag, September 2000.

[49] M. Hicks, A. D. Keromytis, and J. M. Smith, "A secure PLAN (extended version)," in *Proceedings of the DARPA Active Networks Conference and Exposition (DANCE)*, pp. 224–237, IEEE, May 2002.

[50] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles, "Network Programming Using PLAN," in *Proceedings of the IEEE Workshop on Internet Programming Languages* (L. Cardelli, ed.), vol. 1686 of *Lecture Notes in Computer Science*, pp. 127–143, Springer-Verlag, May 1998.

[51] J. T. Moore, *Practical Active Packets*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 2002.

[52] M. Hicks, J. T. Moore, and S. Nettles, "Compiling PLAN to SNAP," in *Proceedings of the Third International Working Conference on Active Networks* (I. W. Marshall, S. Nettles, and N. Wakamiya, eds.), vol. 2207 of *Lecture Notes in Computer Science*, pp. 134–151, Springer-Verlag, October 2001.

[53] M. Hicks, *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August 2001.

[54] M. Hicks and S. M. Nettles, "Dynamic Software Updating," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2005. To appear.

[55] IAB, "IAB Concerns and Recommendations Regarding Internet Research and Evolution," RFC 3869, IETF, August 2004. http://www.ietf.org/rfc/rfc3869.txt.

[56] D. D. Clark, K. Sollins, J. Wroclawski, and T. Faber, "Addressing Reality: An Architectural Response to Real-World Demands on the Evolving Internet," in *ACM SIGCOMM 2003 Workshop on Future Directions in Network Architecture (FDNA-03)*, August 2003. http://www.isi.edu/newarch/DOCUMENTS/Principles.FDNA03.pdf.

[57] W. Willinger and J. Doyle, "Robustness and the Internet: Design and Evolution." http://netlab.caltech.edu/pub/papers/part1_vers4.pdf, March 2002.

[58] D. B. Johnson, A. Myles, and C. Perkins, "Route Optimization in Mobile IP," internet draft, IETF, July 1994. http://k-lug.org/ griswold/Drafts-RFCs/draft-route-optimization-mobile-ip-00.txt.

[59] C. Perkins, "IP Mobility Support," RFC 2002, IETF, October 1996. http://www.ietf.org/rfc/rfc2002.txt.

[60] C. Perkins, "IP Mobility Support for IPv4," RFC 3220, IETF, January 2002. http://www.ietf.org/rfc/rfc3220.txt.

[61] C. Perkins, "IP Mobility Support for IPv4, revised," internet draft, IETF, June 2004. http://www.ietf.org/internet-drafts/draft-ietf-mip4-rfc3344bis-00.txt.

[62] C. Perkins and D. B. Johnson, "Route Optimization in Mobile IP," internet draft, IETF, September 2001. http://www.ietf.org/proceedings/02mar/I-D/draft-ietf-mobileip-optim-11.txt.

[63] D. Johnson, C. Perkins, and J. Arkko, "Mobility Support in IPv6," RFC 3775, IETF, June 2004. http://www.ietf.org/rfc/rfc3775.txt.

[64] 3GPP, "Technical Specification Group Services and System Aspects: Combined GSM and Mobile IP Mobility Handling in UMTS IP CN," Technical Report 3G TR 23.923 version 3.0.0, 3GPP, May 2000. http://www.3gpp.org/ftp/Specs/archive/23_series/23.923/23923-300.zip.

[65] 3GPP2, "The Technical Specifications Group - Core Networks (TSG-X): cdma2000 Wireless IP Network Standard: Introduction," TSG-X Specifications 3GPP2 X.S0011-001-C v1.0, 3GPP2, August 2003. http://www.3gpp2.com/Public_html/specs/X.S0011-001-C_v1.0_110703.pdf.

[66] C. Perkins, "IP Encapsulatioin within IP," RFC 2003, IETF, October 1996. http://www.ietf.org/rfc/rfc2003.txt.

[67] "Mip4 Working Group Status Pages - Implementations." http://www.mip4.org/2004/implementations/.

[68] "Rice Monarch Project IETF Mobile IPv4." http://www.monarch.cs.rice.edu/mobile_ipv4.html.

[69] L. Lehman, S. Garland, and D. Tennenhouse, "Active Reliable Multicast," in *IEEE INFOCOM*, March 1998.

[70] U. Legedza, D. Wetherall, and J. Guttag, "Improving the Performance of Distributed Applications Using Active Networks," in *IEEE INFOCOM*, March 1998.

[71] S.-K. Song and S. M. Nettles, "Using Active Networking's Adaptability in Ad Hoc Routing," in *Proc. of IWAN 2004*, October 2004.

[72] S. Corson and J. Macker, "Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations," RFC 2501, IETF, Jan. 1999. http://www.ietf.org/rfc/rfc2501.txt.

[73] E. M. Royer and C.-K. Toh, "A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks," *IEEE Personal Communications*, pp. 46–55, April 1999.

[74] J. Broch, D. Maltz, D. Johnson, Y. Hu, and J. Jetcheva, "A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols,"

in *Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'98)*, pp. 85–97, Dallas, TX, Oct., 1998.

[75] C. Perkins and P. Bhagwat, "Highly Dynamic Destination-Sequenced Distance-Vectore Routing (DSDV) for Mobile Computers," in *Proceedings of the conference on Communications architectures, Protocols, and Applications (SIGCOMM) 1994*, pp. 234–244, 1994.

[76] P. Jacquet, P. Muhlethaler, T. Clausen, A. Laoulti, A. Qayyum, and L. Viennot, "Optimized Link State Routing Protocol for Ad Hoc Networks," in *Proc. of IEEE INMIC 2001*, pp. 62–68, IEEE, 2001.

[77] G. Pei, M. Gerla, and T.-W. Chen, "Fisheye State Routing: A Routing Scheme for Ad Hoc Wireless Networks," in *Proc. of IEEE ICC 2000*, vol. 1, pp. 70–74, IEEE, 2000.

[78] D. Johnson and D. Malz, "Dynamic Source Routing in Ad Hoc Wireless Networks," in *Mobile Computing*, ch. 5, pp. 153–181, Kluwer Academic Publishers, 1996.

[79] C. Perkins and E. Royer, "Ad hoc On-Demand Distance Vector Routing," in *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pp. 90–100, Feb. 1999.

[80] Z. J. Haas, "A New Routing Protocol for the Reconfigurable Wireless

Networks," in *Proceedings of the Sixth International Conference on Universal Personal Communications*, pp. 562–566, Oct. 1997.

[81] C. E. Perkins, E. M. Royer, S. R. Das, and M. K. Marina, "Performance Comparison of Two On-Demand Routing Protocols for Ad Hoc Networks," *IEEE Personal Communications*, vol. 8, pp. 16–28, Feb. 2001.

[82] M. Takai, J. Martin, and R. Bagrodia, "Effects of Wireless Physical Layer Modeling in Mobile Ad Hoc Networks," in *Proc. of the 2001 ACM International Symposium on Mobile ad hoc networking and computing*, pp. 87–94, ACM, 2001.

[83] C. Perkins, E. Belding-Royer, and S. Das, "Ad hoc On-Demand Distance Vector (AODV) Routing," RFC 3561, IETF, July 2003. http://www.ietf.org/rfc/rfc3561.txt.

[84] D. A. Maltz, J. Broch, J. Jetcheva, and D. B. Johnson, "The Effects of On-Demand Behavior in Routing Protocols for Multihop Wireless Ad Hoc Networks," *IEEE Journal on Selected Areas in Communications*, vol. 17, pp. 1439–1453, August 1999.

[85] "Mobile Ad-hoc Networks (manet) Charter." http://www.ietf.org/html.charters/manet-charter.html.

[86] C. Huitema, J. Postel, and S. Crocker, "Not All RFCs are Standards," RFC 1796, IETF, April 1995. http://www.ietf.org/rfc/rfc1796.txt.

[87] T. Clausen and P. Jacquet, "Optimized Link State Routing Protocol (OLSR)," RFC 3626, IETF, October 2003. http://www.ietf.org/rfc/rfc3626.txt.

[88] R. Ogier, F. Templin, and M. Lewis, "Topology Dissemination Based on Reverse-Path Forwarding (TBRPF)," RFC 3684, IETF, February 2004. http://www.ietf.org/rfc/rfc3684.txt.

[89] "Monarch Project IETF Activities." http://www.monarch.cs.rice.edu/ietf.html.

[90] D. B. Johnson, D. A. Maltz, and Y.-C. Hu, "The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR)," internet draft, IETF, July 2004. http://www.ietf.org/internet-drafts/draft-ietf-manet-dsr-10.txt.

[91] C. E. Perkins, E. M. Belding-Royer, and S. R. Das, "Ad hoc On-Demand Distance Vector (AODV) Routing," internet draft, IETF, February 2003. http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-13.txt.

[92] C. E. Perkins, E. M. Belding-Royer, and I. D. Chakeres, "Ad hoc On-Demand Distance Vector (AODV) Routing," internet draft, IETF, October 2003. http://moment.cs.ucsb.edu/pub/draft-perkins-manet-aodvbis-00.txt.

[93] "Implementation of DSR." http://www.monarch.cs.rice.edu/dsr-impl.html.

[94] "picoNet." http://piconet.sourceforge.net/.

[95] "The Click DSR Router Project." http://pecolab.colorado.edu/DSR.html.

[96] "AODV Public Implementations." http://moment.cs.ucsb.edu/AODV/aodv.html#Implementations.

[97] "The Monarch Project." http://www.monarch.cs.rice.edu/.

[98] "MOMENT Lab @ UCSB." http://moment.cs.ucsb.edu/.

[99] IEEE Computer Society LAN MAN Standards Committee, "Information Technology- Telecommunications and Information Exchange Between Systems - Local And Metropolitan Area Networks - Specific Requirements - part 11: Wireless LAN Medium Access Control (MAC) And Physical Layer (PHY) Specifications." IEEE Std 802.11-1997, 1997.

[100] J. Jubin and J. D. Tornow, "The DARPA Packet Radio Network Protocols," *Proceedings of the IEEE*, vol. 75, pp. 21–32, January 1987.

[101] S.-K. Song and S. M. Nettles, "Active Networking for TCP over Wireless," in *Proc. of IWAN 2004*, October 2004.

[102] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz, "Improving TCI/IP Performance over Wireless Networks," in *Proc. of ACM MobiCom'95*, pp. 2–11, Nov. 1995.

[103] P. Karn and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols," *ACM Trans. Comput. Syst.*, vol. 9, pp. 364–373, Nov. 1991.

[104] A. Chockalingam, M. Zorzi, and V. Tralli, "Wireless TCP Performance with Link Layer FEC/ARQ," in *ICC'99 Proceedings*, vol. 2, pp. 1212–1216, IEEE, 199.

[105] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options," RFC 2018, IETF, October 1996. http://www.ietf.org/rfc/rfc2018.txt.

[106] K. Ramakrishnan, S. Floyd, and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP," RFC 3168, IETF, September 2001. http://www.ietf.org/rfc/rfc3168.txt.

[107] E. Ayanoglu, S. Paul, T. F. LaPorta, K. K. Sabnani, and R. D. Gitlin, "AIRMAIL: A Link-Layer Protocol for Wireless Networks," *ACM Wireless Networks*, vol. 1, no. 1, pp. 47–60, 1995.

[108] A. Bakre and B. Badrinath, "I-TCP: Indirect TCP for Mobile Hosts," in *Proc. of the 15th Int. Conf. on Distributed Systems*, pp. 136–143, May 1995.

[109] A. DeSimone, M. C. Chuah, and O.-C. Yue, "Throughput Performance of Transport-Layer Protocols over Wireless LANs," in *Proc. of IEEE GLOBECOM'93*, vol. 1, pp. 542–549, Dec. 1993.

[110] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby, "Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations," RFC 3135, IETF, June 2001. http://www.ietf.org/rfc/rfc3135.txt.

[111] B. R. Badrinarth, A. Bakre, T. Imielinski, and R. Marantz, "Handling Mobile Clients: A Case for Indirect Interaction," in *4th Workshop on Workstation Operating Systems*, Oct. 1993.

[112] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP," RFC 2883, IETF, July 2000. http://www.ietf.org/rfc/rfc2883.txt.

[113] S. Bhandarkar and A. L. N. Reddy, ""improving the robustness of tcp to non-congestion events"," internet draft, IETF, August 2004. http://www.ietf.org/internet-drafts/draft-ietf-tcpm-tcp-dcr-01.txt.

[114] G. Montenegro, S. Dawkins, M. Kojo, V. Magret, and N. Vaidya, "Long Thin Networks," RFC 2757, IETF, January 2000. http://www.ietf.org/rfc/rfc2757.txt.

[115] J. Postel, "INTERNET CONTROL MESSAGE PROTOCOL," RFC 792, IETF, September 1981. http://www.ietf.org/rfc/rfc792.txt.

[116] S. Floyd, "TCP and Explicit Congestion Notification," *ACM Computer Communication Review*, vol. 24, pp. 10–23, October 1994.

[117] K. Ramakrishnan and S. Floyd, "A Proposal to add Explicit Congestion Notification (ECN) to IP," RFC 2481, IETF, January 1999. http://www.ietf.org/rfc/rfc2481.txt.

[118] "WRED - Explicit Congestion Notification (ECN)," January 2003. http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122newft/122t/122t8/ftwrdecn.htm.

[119] "ECN Problems." http://www.icir.org/floyd/ecnProblems.html.

[120] S. Choi and K. G. Shin, "A Class of Adaptive Hybrid ARQ Schemes for Wireless Links," *IEEE Trans. on Vehicular Technology*, vol. 50, pp. 777–790, May 2001.

[121] H. Minn, M. Zeng, and V. K. Bhargava, "On ARQ Scheme With Adaptive Error Control," *IEEE Trans. on Vehicular Technology*, vol. 50, pp. 1426–1436, November 2001.

[122] H. Lin, S. K. Das, and H. Y. Youn, "An Adaptive Radio Link Protocol to Improve TCP Performance over Correlated Fading Wireless Channels," in *Personal Wireless Communications*, pp. 222–236, September 2003. http://crewman.uta.edu/ hlin/papers/pwc2003.pdf.

[123] S. Lin and P. S. Yu, "A Hybrid ARQ Scheme with Parity Retransmission for Error Control of Satellite Channels ," *IEEE Trans. on Communications*, vol. 30, pp. 1701–1719, July 1982.

[124] P. Lettieri and M. B. Srivastava, "Adaptive Frame Length Control for Improving Wireless Link Throughput, Range, and Energy Efficiency," in *INFOCOM'98 Proceedings*, vol. 2, pp. 564–571, IEEE, 1998.

[125] 3GPP, "Digital cellular telecommunications system (Phase 2+); Radio Link Protocol (RLP) for data and telematic services on the Mobile Station - Base Station System (MS - BSS) interface and the Base Station System - Mobile-services Switching Centre (BSS - MSC) interface," Technical Specification 3G TS 04.22 version 8.0.0, 3GPP, July 1999. http://www.3gpp.org/ftp/Specs/html-info/0422.htm.

[126] 3GPP2, "Data Service Options for Spread Spectrum Systems: Radio Link Protocol Type 3," TSG-C Specifications 3GPP2 C.S0017-010-A v1.0, 3GPP2, June 2004. http://www.3gpp2.org/Public_html/specs/C.S0017-010-A_v1.0_040617.pdf.

[127] IEEE Computer Society LAN MAN Standards Committee, "Supplement to IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Higher-speed Physical Layer Extension in the 2.4 GHz Band." IEEE Std 802.11b-1999, 1999.

[128] IEEE Computer Society LAN MAN Standards Committee, "Supplement to IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: High-speed Physical Layer in the 5 GHz Band." IEEE Std 802.11a-1999, 1999.

[129] "The Berkeley Snoop Protocol." http://daedalus.cs.berkeley.edu/software/pub/snoop/.

[130] "Linux Snoop - An Implementation of Berkeley Snoop on Linux." http://www.cir.nus.edu.sg/research/software/snoop/snoop.html.

[131] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, pp. 300–304, June 1960.

[132] T. S. Rappaport, *Wireless Communications: Principles & Practice*, ch. 6. Prentice-Hall, Inc., 1996.

[133] I. Hadžić, *Applying Reconfigurable Computing to Reconfigurable Networks*. PhD thesis, The University of Pennsylvania, 1999.

[134] H. Wang and P. Chang, "On Verifying the First-Order Markovian Assumption for a Rayleigh Fading Channel Model," *IEEE Trans. on Vehicular Technology*, vol. 45, pp. 353–357, May 1996.

[135] M. Zorzi, R. Rao, and L. Milstein, "On the accuracy of a first-order Markov model for data transmission on fading channels," in *Proc. IEEE ICUPC'95*, pp. 211–215, Nov. 1995.

[136] E. N. Gilbert, "Capacity of a burst-noise channel," *Bell Syst. Tech. J.*, vol. 39, pp. 1253–1265, Sept. 1960.

[137] E. O. Elliott, "Estimates of error rates for codes on burst-noise channels," *Bell Syst. Tech. J.*, vol. 42, pp. 1977–1997, Sept. 1963.

[138] K. Brayer, "Error Control Techniques Using Binary Symbol Burst Codes," *IEEE Trans. on Communication Technology*, vol. 16, pp. 199–214, April 1968.

[139] H. O. Burton and D. D. Sullivan, "Errors and Error Control," *Proc. of IEEE*, vol. 60, pp. 1293–1310, Nov. 1972.

[140] S. Lin and Daniel J. Costello, Jr., *Error Control Coding: Fundamentals and Applications*, ch. 15. Prentice-Hall, Inc., 1983.

[141] B. S. Bakshi, P. Krishna, N. H. Vaidya, and D. K. Pradhan, "Improving Performance of TCP over Wireless Networks," in *Proc. of Int. Conf. on Distributed Computing Systems*, 1997.

Vita

Seong-Kyu Song was born in In-Cheon, South Korea on February 27, 1969, the son of Jong-Heon Song and Hyun-Suk Jung. He received the degrees of Master of Science in Engineering and Bachelor of Science in Engineering from Seoul National University in February 1994 and February 1992, respectively. He worked for Korea Telecom in Korea from 1994 to 1999 as a member of technical staff. In August 1999 he entered the University of Texas at Austin to pursue his doctoral degree in Electrical and Computer Engineering. During his studies at the University of Texas at Austin, he has worked as a member of the Wireless Networking and Communications Group.

Permanent Address: 1620 W. 6th St. Apt. #W

Austin, TX 78703

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.