Composing Protocol Frameworks for Active Wireless Networks

Amit Kulkarni and Gary Minden, University of Kansas

ABSTRACT

This article describes a new model for a composible protocol framework that is suitable for the rapid development of any application-specific service and its deployment in the network. The class-hierarchy model described here enables users to compose their own custom, flexible frameworks from either predefined or custom protocol components tailored to an application's needs. We validate experimentally that application-specific frameworks implementing custom protocols can be developed to improve performance of applications over wireless networks.

INTRODUCTION

Recently we have seen a proliferation of wireless devices that connect to the Internet and offer anytime/anywhere Internet access. However, there is a dichotomy between the characteristics of wireless mobile networks and the wired infrastructure. The primary characteristics of wireless mobile networks are relatively lower bandwidth, intermittent connectivity, and higher error rates. Wired networks offer high bandwidth, steady connectivity, and very low error rates. Bandwidth-intensive or near-real-time applications require special handling to surmount the limitations of wireless mobile networks and the interconnection of wireless and wired networks to provide satisfactory performance to the user. In most cases, new and custom protocols are developed and deployed in the wireless network to meet user needs. Unfortunately, the current network infrastructure is rigid, and the protocol stack in the network is usually fixed. Developing and introducing new protocols in the network requires a time-consuming standardization process.

Active networking [1] provides a new paradigm in which the nodes of the network are programmable (i.e., they provide an execution platform on which user code can be executed). *Active nodes* are programmable elements in an active network that enable the deployment of custom services and protocols. *SmartPackets* are executable entities that are the basic means of communication in an active network. SmartPackets are characterized by a unique type and, optionally, a destination address, data, and program code in the form of methods that can be executed locally at any node in the active network. Applications customize network resources for dynamic adaptation by injecting SmartPackets into the active network to modify behavior of the active nodes.

SmartPackets can potentially carry code in the form of application-specific protocol frameworks composed from custom protocols. This raises several interesting questions:

- What kind of framework is required to rapidly create application-specific protocol frameworks?
- What services can be identified and developed that can benefit traditional applications running over wireless networks?

In this article we describe a new model for a protocol framework that is suitable for rapid development of any application-specific service and its deployment in the network. The class-hierarchy model described here enables users to compose their own custom, flexible frameworks from either predefined or custom protocol components tailored to an application's needs.

The article is organized as follows. In the next section we describe the limitations of the traditional layered network model and explain why it is unsuitable for active networking. We investigate properties that customizable protocol frameworks must possess to be useful for active networking. We describe the class-hierarchy model that satisfies these properties and enables us to create customizable protocol frameworks. We also describe an active networking prototype called Magician that uses the class-hierarchy model to enable users to compose application-specific frameworks. We describe two active networking services that were created in Magician for applications transmitting near-real-time data over wireless links. A discussion about the performance of an application using the custom protocol stack in an active network *vis-a-vis* using traditional network infrastructure is also included. Finally, we describe related research in this area before concluding with a summary.

BACKGROUND

Traditional networking protocols were built for largely non-real-time data with very few burst requirements. The protocol stack at a network node is fixed, and the network nodes only manipulate protocols up to the network layer. New protocols such as Real-Time Transfer Protocol

0163-6804/00/$10.00 © 2000 IEEE

(RTP) and Hypertext Transfer Protocol (HTTP) enable the network to transport other types of application data such as real-time and multimedia data. Such protocols cater to specific demands of the application data. But transporting those new data types over a legacy network requires us to transform the new type of data into the type of data carried by the network. Transforming the data to fit legacy protocol requirements prevents one from understanding the transformed protocol. For example, embedding an MPEG frame in MIME format prevents one from easily recognizing an I, P, or B frame. This prevents the network from taking suitable action on the MPEG frame during times of congestion.

Second, introducing new protocols in the current infrastructure is a difficult and time-consuming process. A committee has to agree on the definition of a new protocol. This involves agreeing on a structure, states, algorithms, and functions for the protocol. The time from conceptualization of a protocol to its actual deployment in the network is usually extraordinarily long.

Traditional protocol frameworks use layering as a composition mechanism. Protocols in one layer of the stack cannot guarantee the properties of the layers underneath it. Each protocol layer is treated like a black box, and there is no mechanism to identify if functional redundancies occur in the stack. Sometimes protocols in different layers of the same stack need to share information. For example, TCP calculates a checksum over the TCP message and the IP header [2]; but in doing so, it violates modularity of the layering model because it needs information from the IP header that it gets by directly accessing the IP header. Furthermore, layering hides functionality, which can introduce redundancy in the protocol stack.

An advantage of active networking is that it enables application-specific processing through the dynamic deployment of user-defined protocol stacks at the network nodes. These protocol stacks are created by composing functionality tailored to the data type being transported. The challenge is to identify an approach that enables creation of flexible, custom, complete, and correct protocol stacks, which can be then injected into the network.

Properties of Customizable Protocol Frameworks

Earlier research on flexible protocol stacks like Horus [3] and the x-kernel [4] showed that it is a good idea to replace monolithic protocols with a composition of small protocol modules, each implementing a specific function i.e., the framework should *support composible protocols*. A single monolithic TCP protocol is replaced by a suite of components, each of which implements various functions of TCP such as checksumming, sequencing, and flow control. Modularity enables the creation of complex but flexible application-specific protocol frameworks from predefined, tested components. This streamlines the protocol stack, and applications do not have to deal with superfluous functionality. Performance measurements on the Horus protocol stack have demonstrated that modularity does not necessarily degrade performance [3].

The traditional protocol implementation is rigidly implemented in the kernel of the operating system of the end system or the control software of a switch or router. Very limited customization is possible in such implementations because the protocol framework at the network nodes provides only limited access to the implementation state. It is also difficult to change the behavior of a protocol installed at a network node to suit the user. The user is forced to find workarounds to implement the desired functionality. This often leads to implementation hacks that are often suboptimal and nonreusable. To promote flexibility, reusability, and the creation of high-confidence protocol components, it is necessary to give the user more control over the creation, customization, and deployment of protocols. One approach to achieve this is to open up the protocol implementation and provide the user with enough tools to control its behavior.

To enhance reusability of protocol components, they should possess a formal, well-defined interface that abstracts the inner workings, but presents the user with a rich representation of its behavior. This is defined as the property of *introspection*. Introspection enables a user to access the implementation state and work with abstractions to the selected implementation state. In the context of active networking, this increases reusability because protocol components satisfying this property can be reused in different protocol frameworks.

The property of *intercession* enables a user to modify the behavior of an abstraction. The user refines existing behavior or adjusts the implementation strategy to improve performance. This is usually achieved through the mechanism of inheritance in object modeling, enabling users to customize previously developed active network protocols to tailor them to the needs of the application without having to write new protocols from scratch. In summary, active network protocols must possess the following properties to enable users to build modular, extensible and verifiable protocol frameworks:

- Modularity — the property of decomposing complex protocols into smaller components, each implementing a piece of communication functionality
- Introspection — the property of being able to access an existing protocol component and work with abstractions to it
- Intercession — the property of being able to modify behavior of an existing protocol component

Protocol Composition in Magician

Magician [5] is a toolkit that provides a framework for creating SmartPackets as well as an environment for executing the SmartPackets. Magician is implemented in Java. Java was chosen because it supports mobile code, serialization, and object-oriented properties such as inheritance and encapsulation. In Magician, the executing entity is a Java object whose state has to be preserved as it traverses the active network. Serialization preserves the state of an

An advantage of active networking is that it enables application-specific processing through the dynamic deployment of user-defined protocol stacks at the network nodes.

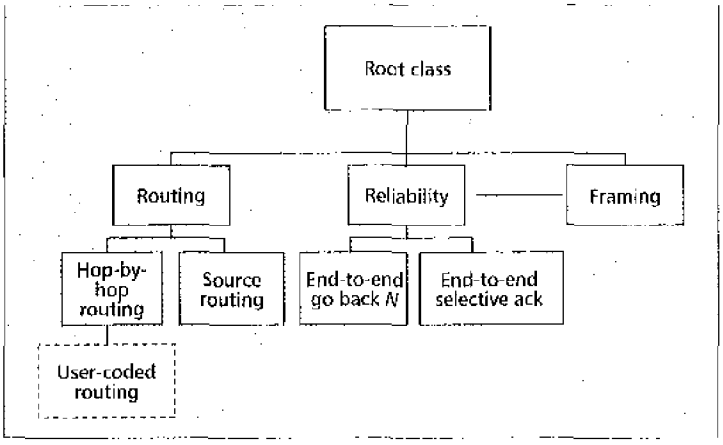


Figure 1. *The class-hierarchy model of an active node.*

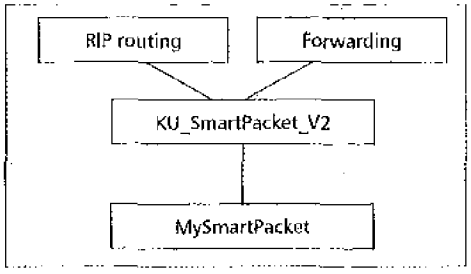


Figure 2. *The composition of a basic unreliable service.*

object so that it can be transported or saved, and recreated at a later time.

Magician provides a model in which an active node is represented as a class hierarchy, as shown in Fig. 1. Every protocol is derived from an abstract base protocol. Every active node provides some basic functionality in the form of certain default protocols (e.g., routing). Users may prefer to utilize these default protocols if they do not wish to implement their own schemes. To foster privacy and safety, a unique state is created for each invocation of the protocol. State that is common to all invocations of a protocol is inviolable and accessible only to users that have appropriate authorization. Providing each user with a protected copy of the state enables the user to customize his/her own state if necessary.

Second, users are permitted to install their own protocols. If necessary, a user creates a new protocol from scratch by extending the base abstract protocol. When the protocol is carried over to the active node by a SmartPacket, a new branch is created for that protocol. Alternatively, the user extends an existing protocol class at the active node and customizes it to implement application-specific behavior, as shown by the example of a user-defined routing algorithm in Fig. 1. Thus, he is able to inherit certain basic properties of the parent protocol and provide minor or major behavioral modifications to tailor it to his application's requirements. In this model, the active node acts as an object whose behavior can be modified dynamically.

The class-hierarchy model for active nodes is advantageous in many respects. It provides the user with the ability to customize default functionality at the active nodes using the *extension-by-inheritance* philosophy. The inheritance model also provides default functionality for user SmartPackets that choose not to implement their own. This enables the active node to support transport of passive packets. The unique state space provided to each invocation of a protocol corresponds to the instantiation of an object from its class definition. This affords privacy and safety to the SmartPacket executing in that state. It also provides an isolated, secure environment for the execution of user code so that the active node can monitor and regulate its activities. The class model approach differs from the data flow composition model [6], wherein protocol functionality is activated only by data flow. In the class model, instead of data being transferred between protocol components, execution control is transferred between the protocol components. Protocol components have access to the user data and can choose to act on the data or perform other activities.

The class model also exhibits the three properties listed earlier: modularity, introspection, and intercession. Modularity is achieved by breaking down communication functionality into distinct protocol components. The functionality of each component is defined in a separate Java class. The property of introspection is satisfied when protocol components are designed as suitable abstractions and specified with well-defined interfaces that promote reuse. Using the principles of inheritance and aggregation, these components are composed together to implement the functionality desired of a service to be injected in the network. This satisfies the intercession property.

Magician provides two basic predefined frameworks to users:

- A framework providing unreliable service
- A framework providing reliable service

The functionality of these frameworks is similar to the UDP datagram and TCP socket abstractions in current networks. In Magician, a user-defined SmartPacket utilizing an unreliable delivery service is created by extending the KU_SmartPacket_V2 class:

```
import magician.Node.*;
public class MySmartPacket
extends KU_SmartPacket_V2 {
:
:
}
```

The basic unreliable service, whose composition is shown in Fig. 2, provides forwarding and routing functions. The RIP routing protocol is the default routing protocol implemented by all active nodes. The API for accessing node resources (getting node name, finding out the next hop, etc.) is provided by the KU_SmartPacket_V2 interface. On the other hand, user SmartPackets requiring assured delivery service extend the ReliableCommFW class:

```
import magician.Node.*;
public class MySmartPacket extends
ReliableCommFW {
:
:
}
```

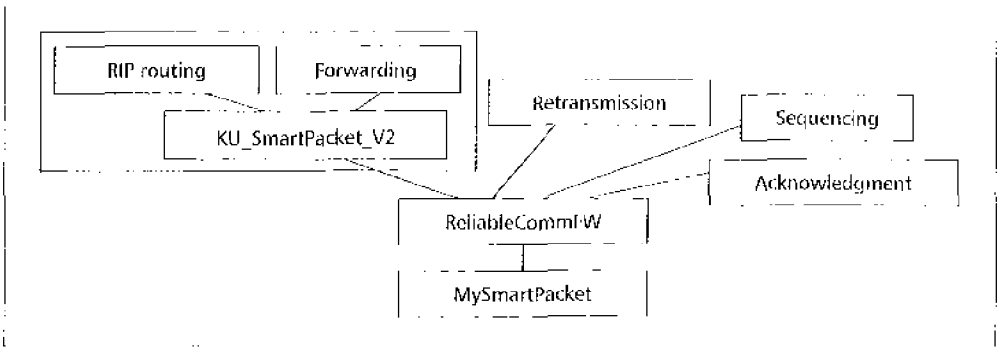


Figure 3. *The composition of a basic reliable service.*

The basic reliable service framework, shown in Fig. 3, adds a sequencing module, a hop-by-hop stop-and-wait-acknowledgment protocol component, and a retransmission component to provide assured delivery. A hop-by-hop protocol is chosen because, in an active networking environment, the destination of a SmartPacket can change during its execution at an active node. This is the simplest reliable delivery mechanism that can handle dynamic destination changes. For example, if a client request packet traversing the network toward a server node encounters information about the location of a proxy that merges duplicate requests to the server, the request packet resets its destination to the location of the proxy. If the packet contains a request that has been made previously, it informs the proxy of the client's request and destroys itself. If an end-to-end acknowledgment/retransmission protocol is used in this case, it is bound to fail because the server node never receives the packet and therefore waits for the sender to send the packet because it is next in sequence.

In Magician, users are not constrained to using only the basic frameworks. Customization of the framework is possible in many ways. Users can replace a particular functionality by replacing the protocol module implementing it. Consider the example of an active application requiring a custom routing algorithm. Active nodes in Magician implement the RIP routing protocol as the default, but the routing manager at the active node provides a registration feature to install custom routing protocols. A custom routing protocol is installed for an application type by registering it with the routing manager at the active node. Once registered, all SmartPackets belonging to the application (and sharing the same type) are routed using the custom routing algorithm. This exemplifies customization by replacement.

Users can also extend functionality of a protocol module. For example, all active nodes are addressed using the naming scheme implemented by the administrator. However, a user can choose to implement his own addressing scheme. A custom addressing protocol can extend the default addressing interface and implement a table that maps configured active node addresses into user-supplied addresses. This enables transparent translation of addresses while adhering to the basic node API.

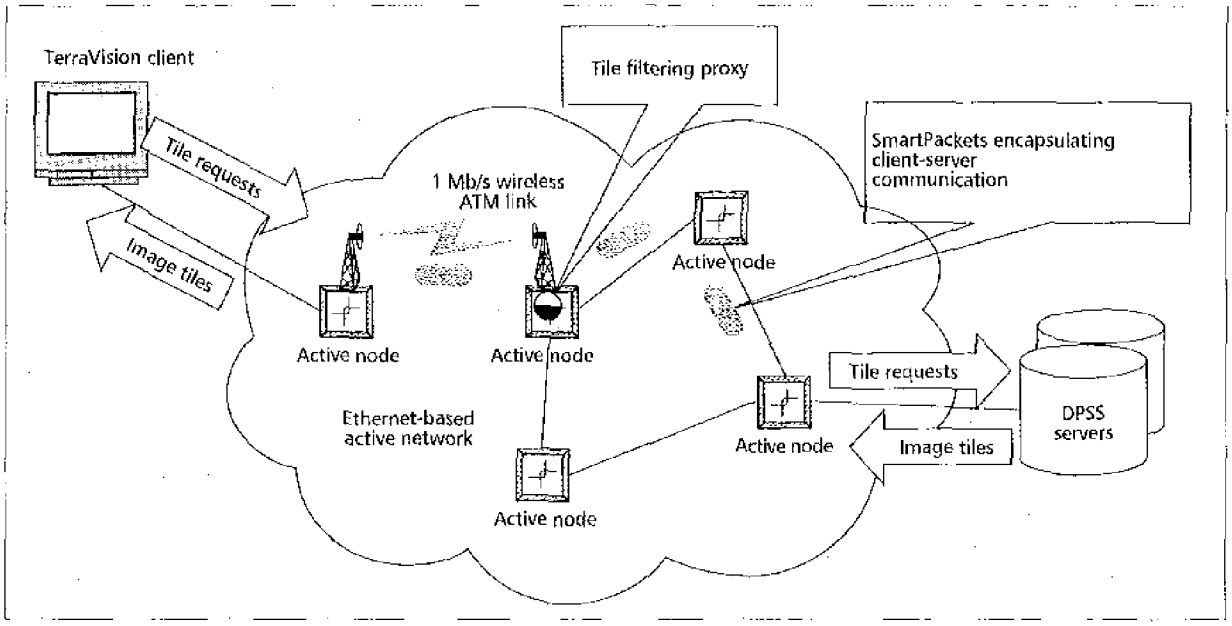
Applications for Wireless Networks

Wireless elements in a network present the problems of intermittent connectivity, low bandwidth, high delay, and high bit error rates. High bit error rates lead to frequent retransmissions of data to ensure reliable delivery to the application. This in turn contributes to lower response times in the case of real-time or near-real-time applications like videoconferencing. Such applications are delay-sensitive as well as bandwidth-intensive, two properties incompatible with the low-bandwidth, high-delay characteristics of wireless links.

Most solutions offered so far to enable applications to function over wireless links are end-to-end or reactive. Applications respond to problems in the network (e.g., congestion) in one of two ways. Applications act at the endpoints, for example, by backing off at the source to reduce load on the network until the congestion is alleviated; or applications provide hints in the headers of protocols that enable the network to drop packets of least value to the application. These solutions are based on the notion of the network trying to provide best-effort delivery to the application. But in the case of real-time or near-real-time traffic, this best-effort delivery can be too late! We argue that such applications require proactive solutions to handle their data. Proactive solutions are those that reduce the load on the network by either discarding data that is no longer useful to the application or reducing bandwidth demand itself. In an earlier paper we outlined a classification of services that can benefit from active networking [7]. There we argued that two classes of services, filtering and merging, are particularly suitable for applications running over wired/wireless networks. In this section we discuss two protocols derived from those classes that exemplify the concept of proactive solutions.

In the context of the MAGIC-II project, we developed a new filtering protocol that utilizes the current state of the application to proactively reduce load on the wireless links in the network. The new protocol enables legacy MAGIC applications like TerraVision to tailor network resources, especially for clients connected over low-bandwidth links. These applications are terrain visualization clients that fetch and display imagery of terrain data stored on remote servers. Image tiles are requested in the form of tile

Proactive solutions are those that reduce the load on the network by either discarding data that is no longer useful to the application or reducing bandwidth demand itself.



■ **Figure 4.** *Active tile filtering.*

request identifiers that are bundled to form a request frame. The requests in each frame correspond to tiles that are part of the current view. The image data is returned as a set of tiles the client application processes to render the terrain. When a user pans the terrain, TerraVision requests a large number of tiles. If the user pans the terrain rapidly and there is significant response lag, it is entirely possible that by the time some of the tiles reach the client, the frame of view has changed, and the tiles are no longer needed and never displayed.

This scenario occurs frequently when TerraVision runs on a host connected to the network over a wireless link. If the user pans very rapidly, the wireless gateway becomes congested, resulting in deteriorated performance. A traditional TCP/IP-based network connection attempts to deliver all requested tiles to the client, including those that are eventually dropped. An application-specific solution to this problem is to prevent tardy tiles (i.e., tiles that are candidates to be dropped by the client) from being transmitted across the wireless link. This reduces the demand on network resources, which in turn enhances the ability of the application to respond to user input because there are more network resources available for tiles in the current frame of view.

In active networks, the solution is implemented in the following manner. The client customizes the network by dropping a filtering proxy at the gateway (Fig. 4) which maintains the identifiers of tiles that are currently requested. Tiles returned from the servers that are not part of the current request list are dropped at the gateway itself. This prevents congestion at the wireless gateway. Since only tiles from the current request list are allowed to pass through, the user is not limited by the capacity of the wireless link. In effect, we have combated the problem by reducing the demand on the network. When the user stops panning, the request list at the gateway does not change as often, enabling tiles in the current view to pass through. The application is thus able to quickly synchronize with the user's demand. The wireless gateway is identified and set up as a special filtering proxy. The existence and location of the proxy is advertised by sending out beacon packets in an n-hop neighborhood of the gateway that installs this information in named caches at the active nodes. Tile request packets check for this information as they traverse the network. If existence of the proxy is detected, the tile request packets reroute themselves to the location of the proxy where the request frame is deposited.

The application is created by defining separate SmartPackets that perform the functions of setting up the connection, requesting the tiles from the server, delivering the requested tiles from the server to the client, locating and setting up the wireless gateway for filtering, and so on. Since we want assured delivery of SmartPackets, all SmartPackets extend the `ReliableCommFW` interface. The default RIP routing protocol is utilized for basic routing capabilities for all packets except beacon packets. Beacon packets simply multicast themselves on all interfaces of a node. A maximum hop limit and checks for previous visitation of a beacon packet are used to prevent excessive flooding.

We have also developed a new protocol, called a Request Merging protocol, to reduce bandwidth demand over wireless links. We trap requests from clients and transparently redirect them to nodes on which a proxy exists that keeps track of the requests made to the server (Fig. 5). The proxy acts like a beacon and sends information about itself to neighboring nodes periodically in a beacon SmartPacket. The information includes proxy location, time sent, expiration time, and number of hops to the proxy. This

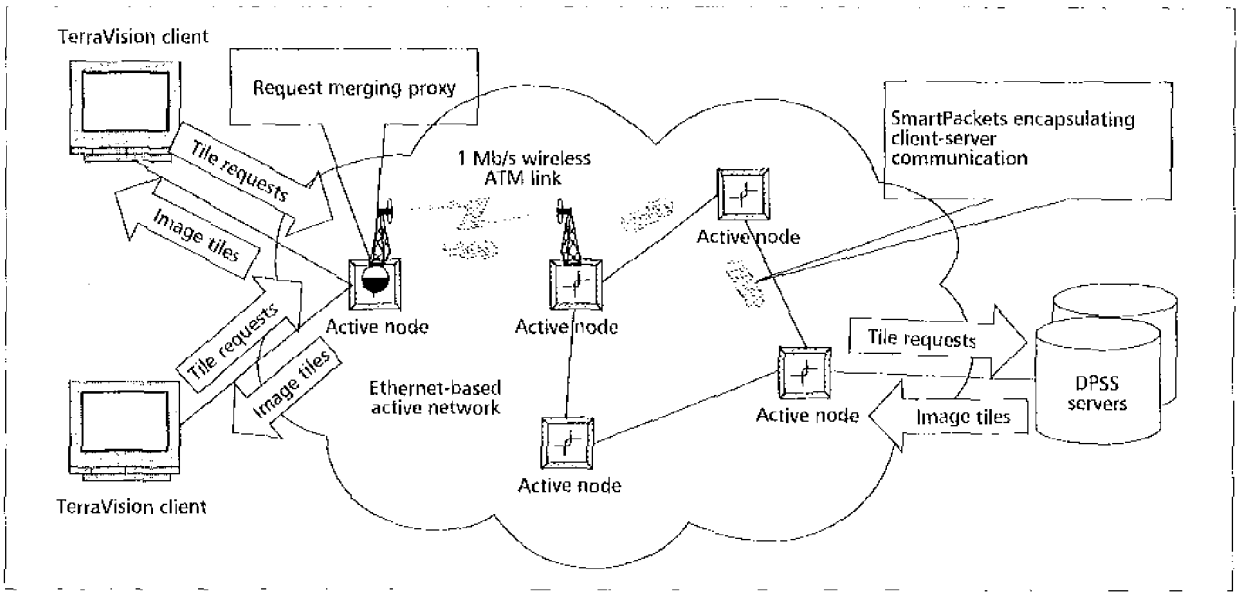


Figure 5. *Active tile merging.*

information is stored at the neighboring nodes and updated periodically. When a request reaches an active node, it is redirected to the nearest proxy based on the information stored at the node. At the proxy, if an incoming request has already been made on behalf of another client, the current request is cached. The server's reply is multicast to all requesting clients.

Most of the packets designed for the filtering application are reused here. The packets for setting up the connection, setting up the proxy, and the beacon packets are the same. The tile request packet and tile data packet frameworks require modification. The tile request packet is composed with an interface that identifies and takes action on a duplicated request. Similarly, the tile data packet, instead of checking for validity of the tile request identifier at the proxy, retrieves the number and location of the clients that made a request for the current tile, and multicasts itself to all the clients.

PERFORMANCE

We conducted experiments to measure the performance benefits obtained by implementing the active tile filtering in the network. The performance of TerraVision running over the traditional TCP/IP network stack is compared with TerraVision running over an active network implementing active tile filtering at the wireless gateway. The network, whose topology is shown in Fig. 4, included a 1 Mb/s wireless link. The network consists of five active nodes, a workstation running the TerraVision application, and another running the image server connected by 10 Mb/s Ethernet. TerraVision fetches 49,332-byte image tiles from the remote server over the wireless link. For the TCP/IP network, a static route ensures that packets are always forwarded over the wireless link. In the active network, routing was controlled so that the route between the client and the server included the wireless link.

In active networking, we trade processing time at the network nodes for benefits gained by sending information rather than data through the network. The benefits can be in the form of reduced bandwidth demand, better utilization of node resources such as buffers and queues, and/or application-specific customization. Performance of the network is traditionally measured in terms of throughput and delay; but simply calculating performance in terms of tiles delivered per second is inappropriate. Because TerraVision drops tiles that are outside the user's frame, a better measure is to find out the number of tiles the application actually displays. Therefore, we define *effective utilization* as the fraction of the data delivered to the *application* that is *actually utilized*. Thus, in our experiment, effective utilization is the ratio of the number of tiles *displayed* by TerraVision to the number of tiles *received* by it.

We ran TerraVision using both TCP/IP and active networking to determine qualitatively and quantitatively if active tile filtering improved its performance. Subjectively, it is observed that when running TerraVision over the TCP/IP network, navigation was rather jerky as the flight neared the end of its path. This effect is not observed when TerraVision runs over the active network implementing tile filtering. The most probable reason for this is that at high panning speeds, TerraVision continuously made requests for a large number of tiles. TCP/IP, in attempting to deliver all the requested tiles to the application, caused the wireless link to get congested. This caused the rendering thread of TerraVision to wait intermittently for the congestion at the wireless gateway to be alleviated, which in turn caused a large number of tiles to be discarded when they did arrive because they were not part of the current view. On the other hand, the tile filtering code in the active network proactively drops tiles that are not in the current view. This prevents congestion at the wireless gateway and enables the network to deliver the required tiles

The higher the utilization, the lower the bandwidth wasted by the network attempting to deliver tiles eventually dropped by the application. In our experiment, this corresponds to a savings of over 18 percent in network bandwidth over the wireless link without significantly affecting performance.

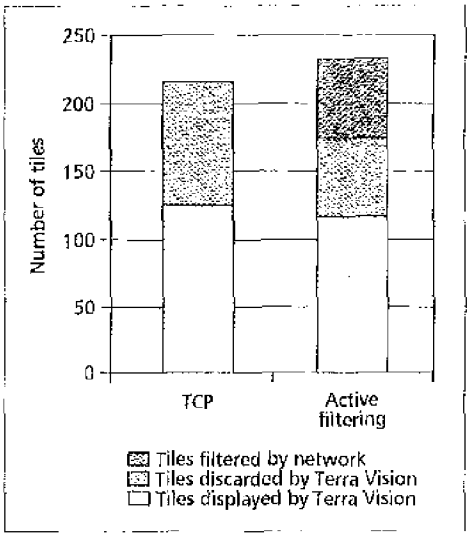


Figure 6. *Comparative performance of TerraVision with and without active tile filtering.*

to the application in a timely manner with higher probability of being displayed.

This is borne out from quantitative analysis of the performance, as shown in Fig. 6. The chart compares the number of tiles displayed by TerraVision, the number of tiles received but not displayed, and the number of tiles filtered by the network for the two cases. The number of tiles displayed by the application is roughly the same in both cases, which validates observed performance that the perceived quality of the image is approximately the same. We see that active filtering provides an effective utilization of 0.67 as compared to 0.58 for the TCP/IP network because approximately half of the tardy tiles are filtered at the wireless gateway itself, as shown by the shaded area of the chart. The higher the utilization, the lower the bandwidth wasted by the network attempting to deliver tiles eventually dropped by the application. In our experiment, this corresponds to a savings of over 18 percent in network bandwidth over the wireless link without significantly affecting performance.

Related Research

Other active networking prototypes utilize some form of protocol composition or other. Boosters are in-kernel protocol modules that adapt to the environment [8]. They are transparently inserted into and deleted from protocol graphs on an as-needed basis. However, booster protocols cannot be dynamically created and injected into the network. They are preinstalled in the kernels of the nodes of the network. It is not clear if booster protocols are developed in a modular fashion or if their functionality can be extended or overridden by applications.

ANTS [9] provides a mechanism for protocol composition using the *CodeGroup* attribute of the ANTS capsule. Code belonging to the same protocol suite (or stack) shares the same *CodeGroup* attribute. Protocol modules are demand-loaded into the active node prior to capsule execution. A wrapper module (which is the main class for the protocol) puts together the protocol modules in the manner desired by the programmer. Protocol frameworks in ANTS are written in Java and can therefore potentially support the properties of modularity, introspection, and intercession.

NetScript [6] is a dataflow programming language in which computation consists of a collection of interconnected programs, called *boxes*, that process data streams flowing through them. The language supports the construction of packet processing programs, interconnecting these programs and allocating resources to them. Protocol composition is modular, achieved by using NetScript constructs to define and establish interconnection between various boxes. A new protocol framework is created by defining a composite box template that describes the protocol components and their interconnections. The protocol components themselves are either box templates or primitive programs. The framework is deployed in the network by dispatching it to individual NetScript engines at the network nodes. It is obvious that frameworks in Netscript are highly modular. It is also shown in [9] that Netscript supports the properties of introspection and intercession.

Switchware [10] provides protocol composition using a two-level architecture consisting of *switchlets* and user programs in active packets. Each switchlet provides services in the form of methods manipulating protocols implemented for that switchlet. The switchlets services are implemented in a high-level language and dynamically loaded at the network nodes. Active packets are user packets that invoke the services provided by the switchlets. The code in the active packets is written in a strongly typed language called PLAN that is specifically designed for active networks. PLAN programs are allowed to compose new protocols from the switchlet services in a modular fashion. But it is not clear if users can extend or replace the functionality of the switchlet services to customize processing arbitrarily.

Summary

In this article we describe the class-hierarchy composition model that enables users to compose and deploy custom, flexible protocol frameworks that cater to an application's immediate requirements. We outline properties required of customizable frameworks and show that the class-hierarchy model satisfies those properties. We implemented this model in the Magician toolkit and demonstrated its use to design application-specific protocol frameworks that tackle problems in wireless networks. Finally, we report on studies on the performance of the protocols and show that active network protocols provide benefits to application data over wireless networks.

References

[1] D. Tennenhouse and D. Wetherall, "Towards an Active Network Architecture," *Comp. Commun. Rev.*, vol. 26, no. 2, Apr. 1996.
[2] D. Comer, *Internetworking with TCP/IP Vol. 2: Design, Implementation and Internals*, Prentice Hall, 1998.
[3] R. van Renesse, K. Birman, and S. Maffeis, "Horus, A Flexible Group Communication System," *Commun. ACM*, vol. 39, no. 4, Apr. 1996, pp. 76–83.

[4] N. Hutchinson and L. Peterson, "The x-Kernel: An Architecture for Implementing Network Protocols," *IEEE Trans. Software Eng.*, vol. 17, no. 1, Jan. 1991, pp. 64–76.
[5] A. Kulkarni *et al.*, "Implementation of a Prototype Active Network," *IEEE Conf. Open Architectures and Network Programming*, San Francisco, CA, Apr. 1998.
[6] A. Kulkarni and G. Minden, "Active Networking Services for Wired/Wireless Networks," *Proc. INFOCOM '99*, vol. 3, New York, 1999, pp. 1116–23.
[7] W. Marcus *et al.*, "Protocol Boosters: Applying Programmability to Network Infrastructures," *IEEE Commun. Mag.*, Oct. 1998.
[8] D. Wetherall, J. Guttag, and D. Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," *IEEE Conf. Open Architectures and Network Programming*, San Francisco, CA, Apr. 1998.
[9] S. da Silva, D. Florissi, and Y. Yemini, "Composing Active Services in NetScript," position paper, DARPA Active Networks Workshop, Tucson AZ, Mar. 1998.
[10] S. Alexander *et al.*, "The SwitchWare Active Network Architecture," *IEEE Network*, Special Issue on Active and Controllable Networks, vol. 12, no. 3, 1999, pp. 29–36.

BIOGRAPHIES

AMIT B. KULKARNI (amit.kulkarni@crd.ge.com) is a computer scientist at General Electric Research and Development (GE CR&D), Niskayuna, New York. Before joining GE CR&D, he was a researcher at the Information and Telecommunications Technology Center (ITTC) at the University of Kansas, where he worked on the DARPA Multi\dimensional Applications and Gigabit Inter-network Consortium (MAGIC-II) project. He received his B.E. in electronics and telecommunications engineering from the University of Pune, India, in 1989, and his M.S. from the University of Kansas in 1996. He has worked for many years in the industry as a systems engineer. He is currently pursuing his Ph.D. at the University of Kansas.

GARY J. MINDEN (B.S.E.E. 1973, Ph.D. 1982, University of Kansas) is professor of electrical engineering and computer science at the University of Kansas. He is a principal investigator on the MAGIC testbed, the Rapidly Deployable Radio Network, and the Innovative Active Networking Services project. From June 1994 through December 1996 he was on leave at the Defense Advanced Research Projects Agency (DARPA) Information Technology Office and served as a program manager in the area of high-performance networking systems. While at DARPA he formulated and initiated a new research program in active networking. His research interests are in the areas of large scale distributed systems which encompass high performance networks, computing systems, and distributed software systems.

It is obvious that frameworks in Netscript are highly modular. It has also been shown that Netscript supports the properties of introspection and intercession.