Transformer Tunnels: A Framework for Providing Route-Specific Adaptations

Pradeep Sudame and B.R. Badrinath
Rutgers University

Transformer Tunnels: A Framework for Providing Route-Specific Adaptations

Pradeep Sudame B. R. Badrinath

Department of Computer Science
Rutgers University
Piscataway, NJ 08855
{sudame,badri}@cs.rutgers.edu

Abstract

In a network using links with diverse properties, a packet flow that is fine tuned for some links (by selecting a proper packet size, transmission rate, encryption method, etc.) may be inappropriate for other links. Ability to change the flow properties over segments of the network allows flows with different characteristics to coexist; making it possible to adapt to diverse link properties. Application-specific adaptation mechanisms (such as proxies) do not force adaptations on every packet flowing over the link and are therefore insufficient for this purpose. We propose the concept of *transformer tunnels* that force adaptations on all the packets flowing through them. Transformer tunnels can coexist with proxies because the adaptations provided by both are independent of each other. Transformer tunnels provide adaptations by means of *transformation functions.* By attaching various transformation functions to such tunnels, we can efficiently fine tune the flow properties. We also provide an API for developing transformation functions. We have implemented transformer tunnels and have used them in our wireless network. In this paper, we present the effects on mobile hosts that use this mechanism to transform flows over the last-hop link for reducing losses during handoffs, and for improving the link utilization.

1 Introduction

The availability of various networking technologies leads to a flow of packets over links with diverse properties. Mobile hosts increase this diversity by using wireless technologies such as WaveLAN [35], CDPD [2], Metricom Ricochet [25], Satellites, cellular modems, and so on. Sometimes such wireless networks are used as the backbone technology as well [17]. In such networks, some links are cheap, fast, reliable, and secure; whereas some links are expensive, slow, lossy, and insecure. Thus, packet flow that is fine tuned for some links (by selecting a proper packet size, transmission rate, encryption method, etc.) may be inappropriate for other links. For example, sending large packets over reliable links improves throughput, whereas sending large packets over lossy links increases the probability of retransmissions. Further, a user can have access to multiple devices such as PDAs, laptops, cellular phones and can connect using different networking technologies at different times and different places (as in overlay networks [7]). Such dynamic changes makes the fine tuning of the flow difficult.

A mobile user invariably encounters a heterogeneous network consisting of segments with diverse properties (bandwidth, asymmetry, reliability, etc.). We need the ability to modify the packet flow over various segments of a route for adapting to such properties. For such adaptations, we propose the concept of *transformer tunnels*. Transformer tunnels transform the packet flow to provide application-transparent, route-specific adaptations. Route-specific adaptations, unlike adaptations that are forced on all hosts using a link, allow different hosts using the link to simultaneously request different adaptations. The adaptations depend on the transformation functions attached to the tunnel. We have implemented the transformer tunnels and have used them over our wireless network to provide adaptations for wireless links. In this paper, we show how mobile hosts can use transformer tunnels to change packet flow over the last-hop link. We also provide an API for adding new transformation functions to the system. Using this API, we have implemented some transformation functions such as encrypting data to provide security, sending packets in bursts to allow energy efficient operations, combining small packets and compressing data over slow links to improve link throughput, and so on.

The paper is organized as follows. Section 2 explains the concept of transformer tunnels, and how mobile hosts use such tunnels to achieve adaptation over the last-hop link. Section 3 details the tunneling mechanism. Section 4 describes the transformation functions that we built and tested. It also describes some other functions that can be built within the same framework. Section 5 describes the

[1]This research work was supported in part by DARPA under contract numbers DAAH04-95-1-0596 and DAAG55-97-1-0322, NSF grant numbers CCR 95-09620, IRIS 95-09816 and Sponsors of WINLAB.

experiments we performed to evaluate the transformer tunnels. Section 6 describes the related work. The paper concludes with our plans to extend this work on transformer tunnels.

2 Transformer Tunnels

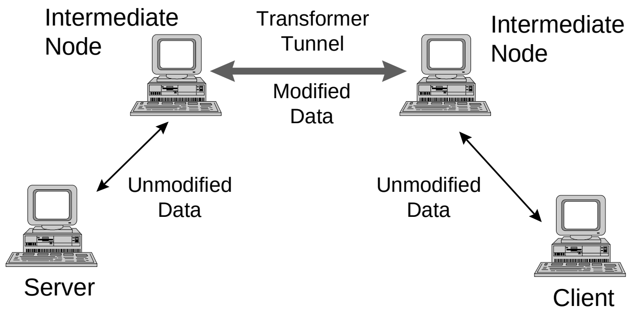

Figure 1: Transformer tunnel

Packet flow over a network segment is modified by placing a transformer tunnel between two end nodes of the segment. Figure 1 shows how packets undergo transformations at the end points of the tunnel. Packets entering the tunnel are modified — either by changing the content, or by changing the way they are transmitted — to fine tune the flow according to the segment properties. Original packets are restored at the other end of the tunnel. Adaptation is thus achieved without affecting rest of the network.

Adaptation over the last-hop link for mobile hosts is achieved by placing similar tunnels over the last-hop link. Even on the same last-hop link, different hosts can request different adaptations. As an example, Figure 2 shows three mobile clients, all requiring different adaptations. Client 1 requires data encryption for its sensitive data because wireless links can be easily snooped upon. Client 2 needs bursty transmission so that it can conserve energy by putting its network interface in sleep mode. At the same time, client 3 requires both the adaptations. This is achieved by establishing transformer tunnels between the base station and each of the clients. The base station then acts as a *transformation agent* for the clients. The transformation agent can be placed anywhere on the network as long as it can intercept the client's packets. In our network, the base station is the logical choice for transformation agents.

Use of proxies has been suggested in the literature for similar adaptations [15, 26, 37]. Transformer tunnels are not a replacement for proxies; rather they supplement proxies. Transformer tunnels differ from proxies in three ways.

Layer of operation: Proxies usually operate at the application layer. Transformer tunnels, on the other hand, provide a low-level application-transparent adaptation. Thus proxies and transformer tunnels can coexist. Proxies can perform application-specific filtering of data (for example, dropping frames in a video stream), whereas transformer tunnels can provide link-specific optimization of the flow (for example, deferring packets for a mobile host till it is in range of an infostation [16]).

Mandatory adaptation: High-level proxies operate on streams associated with a particular application or an application-layer protocol like HTTP; whereas transformer tunnels force adaptations on all the packets passing through them. This feature is required for link dependent adaptations that need to control the way packets are transmitted (for example, making a flow bursty).

End-to-end semantics: Transformations performed by transformer tunnels are hidden from higher layers of the protocol stack. So, if the protocols being used have a notion of a connection (TCP, for example), then unlike proxies, transformer tunnels do not affect the end-to-end semantics of such connections.

2.1 Tunneling Mechanism

Applications configure transformer tunnels by attaching various transformation functions to them. The tunnels apply these functions to all the packets passing through them and send the modified packets over the network. The tunnels support composition of transformation functions. For example, small compressed packets can be combined in larger packets by another transformation function to improve the link utilization. Depending on the link conditions, a mobile host can decide what features are required and can ask the base station to provide an appropriate transformer tunnel. Thus, whenever the link conditions change, the tunnel can be reconfigured. This can be achieved by combining the transformer tunnels with our work on exposing link conditions to higher layers of the protocol stack [32].

Depending on the transformation functions attached, a transformer tunnel modifies incoming packets. The modified packets contain sufficient information so that they can be restored at the other end of the tunnel. As with any other tunnel, a transformer tunnel uses the point-to-point address of the tunnel as the packet's destination address. The origi-

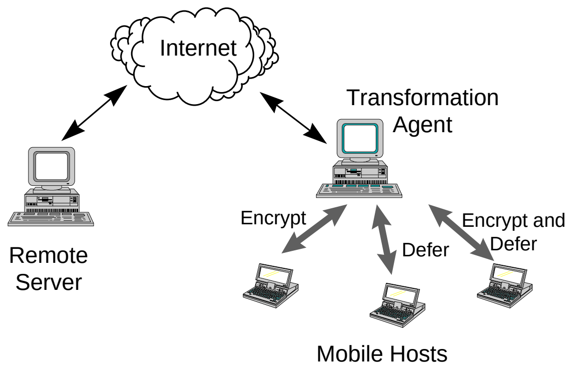

Figure 2: Transformer tunnels on last-hop links

nal destination address, if different, is stored in the data portion of the packet. Some metadata (list of functions to apply to restore the original packet) is also added to the packet.

Transformer tunnels provide a way to build a restricted form of active networks [34]. In case of transformer tunnels, the packets cannot be transformed at intermediate nodes. All the transformations are performed at the end points of the tunnels. This restriction allows an efficient implementation and is still powerful enough to deal with the problems introduced by mobility.

Transformations performed by the transformer tunnels have to be undone at the other end. For this purpose, once the transformations are performed, the transformer tunnel changes the protocol field in the IP header to IP_XFORM. At the other end of the tunnel, the protocol handler for IP_XFORM removes all the metadata from the packet, performs the inverse transformations, restores the original protocol, and puts the packet back in the IP input queue. Some transformations do not change the packet contents (incompressible packets, delayed packets, etc.) and hence do not add any metadata to the IP header. If none of the transformations add any metadata, the packet is delivered with the original protocol, thereby avoiding any overheads at the other end of the tunnel.

A simpler approach would have been to use IP encapsulation. The source address in the new IP header (the outer header used for encapsulation) is not used by the receiver to decapsulate the packet. Moreover, if the the point-to-point address of the tunnel is same as the destination address, the new IP header will have the same destination address. Therefore, to reduce the number of bits used, we modify the original IP header. We add the original protocol number (2 bytes), a flag indicating whether the original destination address is same as the point-to-point address of the tunnel (1 byte), and if required, the original destination address (4 bytes) to the metadata. IP encapsulation would require 20 bytes instead of 7 bytes used by our scheme. (We need just 3 bytes if the destination address is same as the point-to-point address of the tunnel.) Therefore, we save at least 49 μsec on a 2-Mbps WaveLAN.

2.1.1 Adding New Transformation Functions to the System

Transformer tunnels do not assume a fixed set of transformation functions. The transformation functions can be arbitrary as long as the mobile host and the base station agree on them. We provide an API to add new functionalities (using the support for loadable modules provided by Linux) to the system. Any module defining a new functionality has to register the transformation function, and an optional private *ioctl*, with the system by calling the following function.

```
int add_tunnel_func (unsigned char id,
    int (*func) (struct device *dev,
        struct sk_buff **skb,
        struct stunnel_info *info),
    int (*ioctl) (struct device *dev,
        int cmd, void *data, int len));
```

id: unique id for the functionality
func: the transformation function
ioctl: private ioctl for the functionality

The transformation function *(func)* is called with three parameters. The first parameter *(dev)* is the tunnel device. The second parameter *(skb)* is the entire packet with all the headers. (Linux uses *sk_buff* structure to pass packets around.) The third parameter *(info)* must be filled in by *func* with the id of the inverse transformation that should be performed at the other end to restore the packet.

The transformation function has to return either of the two values: PACKET_SENT or PACKET_NOT_SENT. In general, the function will just change the packet contents and let the tunnel handle the actual transmission by returning PACKET_NOT_SENT. Functions that need to control the way packets are transmitted should return PACKET_SENT.

Some modules may want to provide additional *ioctls*. For example, a module that provides buffering requires an *ioctl* to flush all the buffered packets. Other modules may not need any *ioctls*. The corresponding parameter can be *null* in such cases.

While unloading the modules, the functionality has to be removed from the system by invoking the following function.

```
int remove_tunnel_func (unsigned char id);
```

id: id specified when loading the module

2.1.2 Attaching Transformation Functions to Tunnels

Applications attach transformation functions to a transformer tunnel using the following *ioctl* call for the tunnel device.

```
sd = socket (AF_INET, SOCK_DGRAM, 0);
struct ifreq rq;
rq.ifr_name = <tunnel name>;
rq.ifr_data = <id for the function
         to be added, priv>
ioctl (sd, SIOCATTACHFUNC, &rq);
```

The parameter *priv* is used for passing information to the transformation function. For example, an encryption function may use this as the encryption key. A new SIOCDETACHFUNC *ioctl* is used to detach transformation functions from the tunnel.

3 Establishing and Configuring Transformer Tunnels

A transformer tunnel is established like any other tunnel: by specifying the local host's IP address as one end of the tunnel and the remote host's IP address (point-to-point address) as the other end. The tunnel transforms all the packets that are given to the tunnel device, stores the original destination address and the original protocol in the body of the packet, and replaces the destination address with the address of the other end of the tunnel. It then performs a routing-table lookup and prepares the packet for delivery. Normal IP routing then reroutes the packet to the new destination address. In rest of the paper, discussion about transformer tunnels is restricted to tunnels over the last-hop link for mobile hosts. Nevertheless, the discussion is valid for any other transformer tunnel.

A transformer tunnel for a mobile host is established between the mobile host and its base station. All packets destined for the mobile host are forced to go over the tunnel by adding an entry to the routing table at the base station. Once established, the tunnel has to be configured (using the API described in section 2.1.2) to specify the required transformations. The transformation functions are applied in the order in which they are attached to the tunnel.

To allow the mobile host to control the tunnel configuration, we have provided a *tunnel manager* at the base station. Whenever the mobile host requests some adaptation on the last-hop link, the tunnel manager establishes and configures a transformer tunnel accordingly. Moreover, whenever the mobile host moves to a new location, it informs the tunnel manager at the previous location. The tunnel manager then changes the end point of the corresponding tunnel to the new location, forwards all the pending packets, and closes the tunnel. This is a client-server architecture where the client (the mobile host) can remotely configure tunnels at the server (the base station). Thus, the mobile host can add or delete transformation functions at any time. Typically, such a decision will be taken by the mobile host when conditions change. For example, when operating on battery power, it may request that the packets be delivered in bursts so that the network interface can be put in sleep mode in between the bursts. It may then request removal of this feature when operating off an automobile battery. For certain adaptations, the mobile host may have to perform additional functions. For example, if the mobile host requests bursty flow, then it has to monitor the flow, and if the inactivity exceeds a certain threshold, it has to put its network interface in sleep mode.

The tunnel manager has the following features.

Reliability of the adaptation requests: Whenever the tunnel manager receives a request, it sends back a reply indicating whether the action was successful. It is the mobile host's responsibility to ensure that the request is reliably sent to the base station. If it fails to receive the reply, it should retransmit the request. If the reply from the tunnel manager is lost, then retransmitting the request may reconfigure the tunnel with the same parameters. However, reconfiguration leaves the behavior of the tunnel unchanged and hence is harmless.

```
Initialization code
    delayed_packet = *null*;

reassemble (p) {
    if (p is large) {
        send delayed_packet; send p;
        delayed_packet = *null*;
        return PACKET_SENT;
    }
    if (delayed_packet exists) {
        if (p and delayed_packet can be combined) {
            combine and send;
            delayed_packet = *null*;
            return PACKET_SENT;
        } else {
            send delayed_packet;
            delayed_packet = p;
            set timer; return PACKET_SENT;
        }
    } else {
        delayed_packet = p;
    }
}

Timer expires:
    send delayed_packet; delayed_packet = *null*;
```

Figure 3: Reassembly algorithm

Ability to bypass the transformations: Transformer tunnels do not transform the reply packets sent by the tunnel manager. (Otherwise the reply may be delayed if the tunnel transformation function requires so.) This is achieved by specifying that the packets associated with the socket that the tunnel manager uses should not be modified by the tunnel. Other applications that need to bypass some of the transformations can make similar requests. Such requests are made using an *ioctl* call for the tunnel device.

Soft state: Mobile hosts can move away without proper deregistration. The base station cannot keep the tunnel open indefinitely because it has finite resources. The mobile host thus has to renew the request periodically. If the tunnel is not renewed periodically, the base station removes the tunnel. In other words, the base station maintains soft-state information about the transformation functions required by the mobile host. The state is regenerated by periodic renewals.

4 Transformation Functions

The API provided for developing transformation functions takes away developer's responsibility of dealing with devices. New transformation functions can therefore be developed easily. Using this API, we have implemented five transformation functions.

Reassembly: A packet flowing from a fixed host to a mobile host typically traverses links with different MTUs (Maximum Transfer Unit). To eliminate fragmentation, TCP sometimes uses path-MTU–discovery mechanism and selects the smallest MTU along the path as the TCP segment size. (In practice, many TCP implementations select a pessimistic segment size of 536 bytes to avoid the overheads of path-MTU discovery.) Some wireless devices, such as WaveLAN, have a large MTU (1500 bytes). A narrow link along the path means that packets much smaller than the link MTU will be sent over the wireless link. The reassembly transformation function combines such packets to improve the link utilization and reduces cost if the users are charged on a per-packet basis (as in CDPD networks) for their network usage. The combined packet requires just one link-layer header, thereby offsetting the extra bits sent as metadata. Moreover, reducing the number of packets results in lowered contention over broadcast links.

The reassembly function uses a mechanism similar to TCP delayed ACKs. Every small packet is delayed by a small amount of time. If another small packet arrives in this interval, both the packets are combined. Otherwise, the packet is sent as is. This reassembly mechanism is different from IP reassembly (performed after IP fragmentation). A reassembly function combines small packets, along with their IP headers, to get a larger packet. The mobile host regenerates all the original packets when it receives such a reassembled packet. Thus, this mechanism can be used even for protocols that honor message boundaries (for example, UDP).

We use the reassembly algorithm shown in Figure 3. A reassembly tunnel maintains a single packet buffer for every mobile host that requests this adaptation. The maximum time for which packets are delayed is a parameter specified by the mobile host. The mobile host has to decide the maximum delay it can tolerate. A larger delay value increases the probability of two small packets being combined to create a larger packet. It is possible to extend the strategy to combine more than two packets if more delay is tolerable. To simplify the implementation we combine at most two packets.

Energy savings: Wireless devices usually have power-saving features that are built in the hardware. Software strategies to take advantage of these features are gaining importance [24]. We demonstrate a simple energy-saving strategy by using the transformer tunnels.

For devices like WaveLAN, energy consumed while waiting for a packet is about seven to eight times higher than the energy consumed when the card is in sleep mode. Thus the amount of time the card can be put in sleep mode determines how much energy can be saved [31]. The mobile host may put its WaveLAN card in sleep mode only if it knows when *not* to expect packets. This knowledge can be provided by making the traffic over the wireless link bursty and by informing the mobile host about the time slots when the bursts are sent.

In our implementation, the base station buffers all packets for the mobile host whenever the energy-savings adaptation is requested. The base station also sends out periodic beacons that contain a list of mobile hosts who have a packet pending. The mobile host periodically wakes up to listen to these beacons. If it has a pending packet, it requests that the packet be sent ("card spin-up"). In response, the base station sends all the buffered packets for this host and removes the energy-savings transformation function from the tunnel. The mobile host then stays awake. Based on some inactivity threshold, it decides to enter the energy-savings mode again ("card spin-down"). This "spin-up–spin-down" mechanism has already been used for CDPD links [23]. The energy-savings transformation function allows any other device to use the same mechanism without modifying the corresponding device driver, but is useful only when energy required to power up and power down the device is low (for example, WaveLAN [31].)

The mobile host can sleep for as long as it wants if the base station can buffer any number of packets. However, the buffer space at the base station is limited. Therefore, the mobile host has to wake up periodically. (Mobile-IP implementation [11] requires that the mobile hosts renew their registration every 5 seconds. So in our implementation, a mobile host cannot sleep for more than 5 seconds.) Moreover, the beaconing interval at the base station should be small enough to avoid buffer overflow during the interval.

Buffering: Whenever a mobile host performs a handoff, some packets are lost. These losses can be reduced by buffering the last few packets and forwarding them to the mobile host's new location [9].

A buffering tunnel acts like a write-through buffer that buffers the last few packets sent to the mobile host. Whenever the mobile host experiences a burst loss, which may be due to a handoff, it asks the tunnel manager to flush the buffer (retransmit the last few packets). We have modified the mobile-IP code at the mobile host so that it informs the tunnel manager at the previous base station whenever a handoff occurs. In response, the tunnel manager changes the end point of the tunnel to point to the new base station and forwards the buffered packets.

Encryption: Wireless links are more susceptible to snooping. Some applications may want to encrypt their data over

Transformation Function	When to use	Side effects
Reassembly	large MTU, cost per packet	increases latency for isolated packets, large packets more likely to be lost on noisy links
Encryption	insecure links	processing overheads
Compression	slow links	overheads for incompressible data
Buffer	mobile clients	requires buffers at the base station
Energy Savings	power constraints	increases latency for first few packets in idle mode

Table 1: Currently implemented transformation functions

such links. A transformer tunnel with a simple encryption function may be used here.

Depending on the security requirements, various encryption methods can be used. To demonstrate the feasibility of such a transformation function, we have used a simple XOR function to encrypt data [33]. A more secure method can be used where the mobile host and the base station share a secret key. This secret key can be established (by using public key encryption) when the mobile host requests encryption over the link. If required, a sophisticated IP-encryption mechanism [1] may be used. We do not deal with the issue of key management as it is orthogonal to the mechanism of transformer tunnels. Once the key is obtained, it can be passed as an argument to the encryption function (by using the API described in section 2.1.2).

Compression: On slow links, the time for transmitting packets is large and compressing packets before transmission improves performance. Header compression techniques [10, 21] have been proposed to improve performance for such links. On very slow links, compressing the data portion leads to even more gains [28, 29]. Compressing data is not useful on fast links, because the compression and decompression overheads offset the savings obtained by sending fewer bits. On slow links, however, a fast compression function (where the time per byte for compression and decompression multiplied by the bandwidth is less than the fraction of bytes saved) leads to improved performance.

We have used a simple compression function provided by the minilzo library [27]. If the packet is incompressible, we send the original packet without any modifications. The LZO compression method is fast enough to be useful even on WaveLAN. For slower devices like CDPD, it will lead to more gains. To increase the gains further, a more expensive compression function can be used on slow devices.

The conditions under which these transformations are useful are enumerated in Table 1. Many other transformation functions can be built in the same framework. We describe some of them here. These functions have not been implemented yet in the transformer tunnels framework.

Snoop TCP: Losses on wireless links are usually due to noise on the link. TCP interprets all losses as a symptom of congestion and slows down the transmission. Snoop TCP [6] deals with this problem by having faster retransmissions of lost packets on the wireless link. Transformation tunnels provide a simple framework for implementing Snoop TCP.

Dealing with asymmetry: Asymmetric links (where the uplink bandwidth is small as compared to the downlink bandwidth) result in poor TCP performance [12, 5]. A slow uplink results in underutilization of the fast downlink because TCP senders decide the transmission rate based on the frequency of incoming ACKs.

A transformation function that suppresses a few TCP ACKs reduces the load on the uplink [5]. Dropped ACKs do not affect the reliability of the protocol because TCP ACKs are cumulative. However, ACK filtering leads to bursty transmission from the sender [5]. To alleviate this problem, the base station can regenerate filtered ACKs and can send them to the sender at a steady rate [4].

Support for proxies: Transformation functions at the link layer are unaware of the semantics of the data. Proxies can do better filtering based on the data-type–specific operations. A module can be written that provides support for developing proxies. If required, this module can propagate packets to some filter application. Other packets can be sent over the link without changes. Application filters can take appropriate actions and send the filtered packets back onto the network. Such a module would be similar to the "low-level proxies (LLP)" [37] for application-independent adaptation.

5 Evaluation

This section describes the experiments we performed to evaluate the transformer tunnels. For all our experiments, we used a wireless LAN configuration. The wireless LAN consists of fixed hosts (or base stations) and mobile hosts. The fixed hosts are Pentium (133 MHz) based desktop machines running the Linux operating system (version 2.1.24).

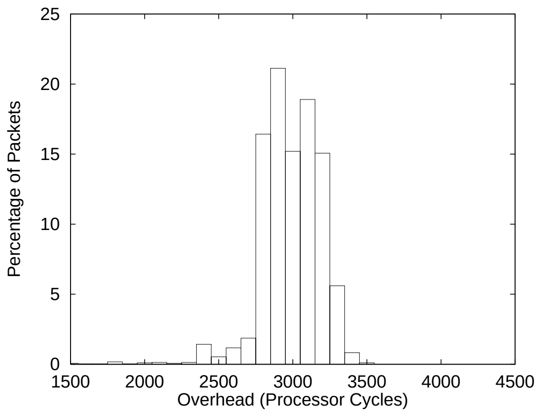

Figure 4: Overhead at the sender

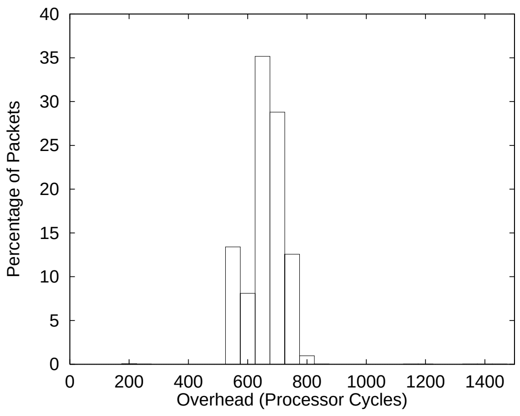

Figure 5: Overhead at receiver

The mobile hosts are Pentium (133 MHz) based laptops and also run Linux (version 2.1.24). The base stations as well as the mobile hosts use 2-Mbps WaveLAN technology for wireless communication. In addition, the base stations have a wired interface to a 10-Mbps Ethernet. To support mobility, the machines run the mobile-IP code developed at the State University of New York at Binghamton [11].

5.1 Overheads

The overheads introduced by the transformer tunnels depend on the adaptations requested. A complex encryption function will lead to reduced throughput. Tunnel overheads should thus be measured when the transformation function introduces a little extra overheads of its own. For this purpose, we have implemented the identity transformation function that leaves all packets unmodified. The same function is used to recover the packets at the mobile host. All packets are sent with the new protocol IP_XFORM, and the corresponding protocol handler is invoked at the mobile host. (In practice, such unmodified packets would be sent without the new protocol, and no processing would be required at the other end.)

Thus the overheads involved in the process are as follows.

- Sender: The tunnel changes the IP header, invokes the identity function, adds the metadata to the IP packet, and recomputes the IP checksum. It then sends the packet over the wireless interface.

- Receiver: The protocol handler for IP_XFORM removes the metadata, calls the identity function, restores the old protocol in the IP header, and recomputes the IP checksum. It then sends the packet back to the protocol stack.

To measure the overheads introduced by the transformer tunnels, we measured (using the internal Pentium counters) the processor cycles consumed by an identity transformation function. A transformer tunnel was established between a fixed host and a laptop. During the experiment, no other user processes were running either on the fixed host or on the laptop. The processor cycles were measured at both ends of the tunnel. The measurements were taken for ICMP (using *ping*) packets as well as UDP and TCP packets (using *ttcp*) from the base station to the mobile host. The overheads were also measured for various packet sizes. The results were same in all the cases. This is expected, because the operations performed by the tunnel and the identity function do not depend on the packet size or on the protocol being used. Figures 4 and 5 show the distribution of the overheads observed. At the base station, overheads are around 3100 processor cycles ($\approx 23\ \mu$s). Overhead for restoring the packet at the mobile host are around 700 processor cycles ($\approx 5\ \mu$s). In both the graphs shown above, a few (less than 5 in 3000) stray values have been discarded.

5.2 Buffering

	Avg loss % [St. Err]	Avg retransmissions % [St. Err]
No buffering	2.32 [0.36]	14.19 [0.54]
With buffering	0.63 [0.16]	6.91 [0.72]
% reduction	73	54

Table 2: Effect of buffering tunnel on handoffs

To evaluate the buffering tunnel, we used *raplayer* and *raserver* (from RealNetworks, Inc. [20]). For this experiment, we used two base stations, an audio server (another fixed machine), and a mobile host. Handoffs were forced every 10 seconds. The mobile host fetched audio files (total 1.2 MB). The buffer size used was 5 packets. Table 2 shows average percentage of packets lost (could not be played) due to handoffs and the percentage of packets that had to be resent by the server. The statistics were gathered from the server's access log files. The server was run in a mode where it logs all the information about lost packets.

Data type	Bytes saved (per KB) [St. Err]	Compression time (ms/KB) [St. Err]	Decompression time (ms/KB) [St. Err]
text	375 [0.789]	0.458 [0.001]	0.087 [0.000]
text (compressed)	0 [0]	0.303 [0.001]	0 [0]
image (GIF)	0 [0]	0.289 [0.001]	0 [0]
PostScript	469 [0.241]	0.386 [0.000]	0.090 [0.001]
PostScript (compressed)	0 [0]	0.290 [0.000]	0 [0]

Table 3: Compression

5.3 Reassembly

To evaluate the reassembly tunnel, we performed the following experiment. A realaudio server was placed on a fixed machine in our network. The mobile host accessed an audio file on the server. As before, we used *raplayer* to fetch this file. *raplayer's* statistics window was used to measure the number of packets that were delayed (and hence dropped). We also measured the total number of packets on the link. *raplayer* saw exactly the same number of packets because the original packets were regenerated when the reassembled packets reach the mobile host. The actual number of packets on the link was measured by snooping on the link (using *tcpdump*).

For the data stream in our experiment, we observed that the inter-arrival time for packets was close to 20 ms. Thus, we configured the tunnel with the delay parameter of 25 ms *i.e.* small packets (of size 256 bytes) that could be combined were delayed by at most 25 ms. Combined packets (each of size 512 bytes) were delivered as soon as possible. We observed that even with the reassembly tunnel, no packets were lost due to delay. In other words, the quality of the audio player was not affected by the reassembly transformation. At the same time, the number of packets sent over the link (and hence the bytes used by link-layer headers) were reduced by a *factor of two*.

5.4 Compression

We tested the compression function over WaveLAN. We measured the effect of compression on three data types: text, image and PostScript. In all the cases, we measured the number of bytes saved by compression, the compression overheads (measured in processor cycles, shown as time) and the decompression overheads. We also measured these parameters after compressing the original files with *gzip*. (The GIF files were not compressed as they are already in a compressed format.) Table 3 shows the results of the experiment. The numbers are averages over 10 sessions. During each session, 20 files were retrieved using *ncftp*. The results show that for compressible files (text and PostScript), the bytes saved due to compression more than compensate for the compression overheads. (On a 2-Mbps WaveLAN, assuming no transmission overheads, transferring one byte requires 3.8 μsec.)

6 Related Work

The problem of adapting to a changing environment has been studied extensively in the literature. Support from the base station has been used for adaptation at various layers of the protocol stack. I-TCP [3] and Snoop TCP [6] use base stations to improve TCP performance for mobile hosts. At the network layer, base stations have been used to improve the performance during handoffs [9]. Even at the link layer, use of base stations has been suggested for scheduling packet transmissions to reduce losses [8]. These mechanisms provide a fixed transformation function to solve specific problems introduced by mobility. Transformer tunnels suggested in this paper is not an alternative for above solutions. It merely provides a simple way of using such features whenever they are appropriate.

Some other mechanisms extend system functionality by interposing agents [22] between applications and the kernel. The University of Arizona's *x*-Kernel [19] provides support for composing protocols from simple elements. Protocol boosters [13] provide a way to insert such elements into the kernel "on-the-fly". Such protocol boosters are similar to the transformation functions described in this paper. Support for protocol boosters requires that every packet be inspected by the booster code to decide if boosting/deboosting is required (and a change in the kernel is also required for providing the booster support). The transformation functions described in this paper are used for dealing with specific link conditions rather than providing a generic support for modifying protocols. We use a simple and restricted mechanism to deal with transformations at the link layer. In our framework, only the packets to be transformed are sent to the tunnel, and only the packets that require inverse transformations are intercepted by the other end of the tunnel. All this is achieved without any changes to the kernel code.

There are application-layer adaptation techniques that allow greater flexibility. These techniques provide adaptations that are specific to some applications or application-layer protocols. In the Odyssey architecture [26], the granularity of data being sent over the network is decided based on the available resources. The Daedalus system [15] uses proxies to performs on-demand distillation of data. That includes converting color images to black-and-white images, converting PostScript to Rich Text Format (RTF), dropping video frames, and so on. Another way of adapting to link conditions is by using "high-level proxies (HLP)" [37]. An HLP allows developing proxies similar to the Daedalus system.

There are several systems that provide mechanisms for safely adding code to the system for configuring protocols. For example, the SPIN operating system [14] allows dynamic configuration of protocols. The ANTS toolkit [36] allows even more flexibility by shipping the processing code along with the packet. Another approach is at language level where the PLAN (Programmable Language for Active Networks) [18] language is used for programs that are carried in the packets of a programmable network. The SwitchWare [30] project suggests use of "switchlets" for dynamically linking new functionalities with network elements.

7 Conclusions and Future Work

Today's networks are composed of links with diverse properties. In such networks, packet flow that is fine tuned (by selecting a proper packet size, transmission rate, encryption method, etc.) for some links may be inappropriate for other links. In this paper, we have shown that *transformer tunnels* provide an efficient mechanism for transforming the flow on various segments of a network, without affecting rest of the network. We have demonstrated the effectiveness of this technique by implementing it over our wireless network to improve the link utilization by compressing data and reassembling small packets, and to reduce losses during handoffs by buffering packets. We have also provided an API for easy development of transformation functions. This API has been used to implement the transformation functions described in this paper. We have implemented the tunnel manager as well that allows clients to control the transformations performed by transformation agents.

In the current implementation, the transformations to be performed depend on a packet's destination. We are investigating an extension to this approach where more restrictive filters can be specified. This can be useful for rerouting certain packets (such as rerouting all e-mail traffic to the home server). Moreover, packets can be selectively sent to a proxy server allowing easy development of client-proxy-server applications.

We also plan to combine transformer tunnels with a mechanism for exposing link conditions to higher layers of the protocol stack [32]. This will allow automatic reconfiguration of the tunnels in response to changes in the link conditions.

8 Acknowledgments

We would like to thank the anonymous reviewers for their valuable insight, detailed comments, and also for providing pointers to related work on protocol boosters. The reviews helped in improving the quality of the paper. We would also like to thank William Marcus for providing details on the work about protocol boosters.

References

[1] R. Atkinson. RFC1825: Security architecture for the Internet Protocol, August 1995. http://globecom.net/(nobg)/ietf/rfc/rfc1825.shtml.

[2] N. G. Badr. Cellular Digital Packet Data CDPD. In *Proceedings of the IEEE 14th Annual International Phoenix Conference*, pages 659–665, March 1995.

[3] A. Bakre and B.R. Badrinath. I-TCP: Indirect TCP for mobile hosts. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 136–143, Vancouver, Canada, May 1995.

[4] Hari Balakrishnan. Discussion on the end2end mailing list, 20 February 1996. ftp://ftp.isi.edu/end2end/end2end-interest-1996.mail.

[5] Hari Balakrishnan, Venkata Padmanabhan, and Randy Katz. The effect of asymmetry on TCP performance. In *Proceedings of the 3rd MOBICOM Conference*, pages 77–89, Budapest, Hungary, September 1997.

[6] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy Katz. Improving TCP/IP performance over wireless networks. In *Proceedings of the 1st MOBICOM Conference*, Berkeley, CA, November 1995.

[7] Bay Area Research Wireless Access Networks (BARWAN). http://http.cs.berkeley.edu/˜randy/Daedalus/BARWAN /BARWAN_over.html.

[8] P. Bhagwat, P. Bhattacharya, A. Krishna, and S. K. Tripathi. Enhancing throughput over wireless LANs using channel state dependent packet scheduling. In *Proceedings of the INFOCOM*, March 1996.

[9] Ramón Cáceres and N. Padmanabhan. Fast and scalable handoffs for wireless internetworks. In *Proceedings of the 2nd MOBICOM Conference*, pages 56–66, November 1996.

[10] Mikael Degermark and Stephen Pink. Soft state header compression for wireless networks. In *Proceedings of the 2nd MOBICOM Conference*, pages 1–14, November 1996.

[11] Abhijit Dixit, Vipul Gupta, and Ben Lancki. Linux mobile IP: Implementation overview. http://anchor.cs.binghamton.edu/˜mobileip/.

[12] Robert Durst, Gregory J. Miller, and Eric J. Travis. TCP extensions for space communication. In *Proceedings of the 2nd MOBICOM Conference*, pages 15–26, November 1996.

[13] D. C. Feldmeier, A. J. McAuley, and J. M. Smith. Protocol boosters. To appear in IEEE JSAC Special Issue on Protocol Architecture for the 21st Century, 1997.

[14] Marc E. Fiuczynski and Brian N. Bershad. An extensible protocol architecture for application-specific networking. In *1996 Winter USENIX Technical Conference*.

[15] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand distillation. In *Proceedings of the ASPLOS*, pages 160–170, October 1996.

[16] R. H. Frenkiel and Tomasz Imielinski. Infostations: The joy of “many-time many-where” communications. Technical Report 119, WINLAB, Rutgers University, April 1996.

[17] J. J. Garcia-Luna-Aceves et al. Wireless internet gateways (WINGS). In *Proceedings of the IEEE MILCOM*, Monteresy, California, November 1997.

[18] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A programming language for active networks. Submitted to PLDI, 1998.

[19] Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[20] RealNetworks, Inc. Live and on-demand audio, video and animation for the Internet. http://www.real.com.

[21] V. Jacobson. RFC 1144: Compressing TCP/IP headers for low speed serial links, 1990.

[22] M. B. Jones. Interposing agents: Transparently interposing code at the system interface. In *Proceedings of the 14th SOSP*, pages 80–93, Asheville, NC, 1993.

[23] R. Frank Quick Jr. and Kumar Balachandran. An overview of the cellular digital packet data (CDPD) system. In *Proceedings of the Fourth International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, Yokohama, Japan, September 1993.

[24] Jacob R. Lorch and Alan Jay Smith. Software strategies for portable computer energy management. To appear in IEEE Personal Communications.

[25] Metricom ricochet modem. http://www.metricom.com/ricochet/.

[26] Brian Noble, M. Satyanarayanan, Dushyanth Narayanan, J. Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th SOSP*, 1997.

[27] Markus Franz Xaver Johannes Oberhumer. miniLZO – mini version of the LZO real-time data compression library.
http://www.infosys.tuwien.ac.at/Staff/lux/marco /lzo.html.

[28] A. Sacham, R. Monsour, R. Pereiera, and M. Thomas. IP payload compression protocol (IPComp), October 1997. Internet draft, Work in progress.

[29] V. Schryver. RFC1977: PPP BSD compression protocol, August 1996.
http://globecom.net/(nobg)/ietf/rfc/rfc1977.shtml.

[30] J. M. Smith, D. J. Farber, C. A. Gunter, Scott Nettles, D. C. Feldmeier, and W. D. Sincoskie. Switchware: Accelerating network evolution (white paper), 1996.
http://www.cis.upenn.edu/˜jms/white-paper.ps.

[31] Mark Stemm, Paul Gauthier, Daishi Harada, and Randy Katz. Reducing power consumption of network interfaces in hand-held devices. In *Proceedings of the 3rd International Workshop on Mobile Multimedia Communications (MoMuc-3)*, pages 130–142, 1996.

[32] Pradeep Sudame and B. R. Badrinath. On providing support for protocol adaptation in mobile wireless networks. Technical Report 333, Department of Computer Science, Rutgers University, July 1997.
http://www.cs.rutgers.edu/pub/technical-reports/dcs-tr-333.ps.Z.

[33] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, second edition, 1993. Section on cryptography.

[34] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. Minden. A survey of active network research. In *IEEE Communications*, January 1997.

[35] Wavelan. http://www.wavelan.com/.

[36] David J. Wetherall, John V. Guttag, and David L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. Submitted to IEEE OPENARCH, 1998.

[37] Bruce Zenel and Dan Duchamp. A general purpose proxy filtering mechanism applied to the mobile environment. In *Proceedings of the 3rd MOBICOM Conference*, pages 248–259, Budapest, Hungary, September 1997.