On Providing Support for Protocol Adaptation in Mobile Wireless Networks

Pradeep Sudame B. R. Badrinath

Department of Computer Science

Rutgers University

{*sudame, badri*}*@cs.rutgers.edu*

Abstract

The availability of a variety of communication devices offers a choice among networks with vastly different characteristics. A mobile host is likely to encounter these different networks and no single protocol or application can be expected to perform well under all these networks. The problem of adapting to a changing network environment is further complicated by the fact that changes in network conditions are usually transparent to the applications. In order to allow automatic adaptation of applications and protocols, awareness of link conditions and network environment is required. In this paper we present a uniform mechanism based on ICMP messages for providing environmental information to all the network protocol layers. We also show how applications and protocols can adapt to changes in the environment and in particular, demonstrate dynamic fine tuning of some of the well known protocols such as UDP and TCP. Performance measurements demonstrate that our mechanism imposes very little overhead and results in better performance of protocols that can adapt to changing network conditions.

1 Introduction

With advances in technology, a variety of communication devices like Ethernet, Wavelan, Rangelan, modems, CDPD, cellular modems etc. have become available at affordable prices. It is not uncommon for a laptop to have access to more than one kind of network [3]. All these technologies offer different network characteristics. Ethernet and wireless LAN offer inexpensive high bandwidth communication but limited coverage. Cellular modems and CDPD on the other hand offer very low bandwidth and wide area coverage but at high cost. Satellite channels offer high latency asymmetric links. All these characteristics have implications for protocol performance. In wired networks, applications and protocols are fine tuned to perform well for the kind of network being used. However, mobility introduces a possibility of dynamic changes in the network. For example, a person may use a modem to communicate from home. While driving to the office, she may have to use a more expensive mode of communication, viz. CDPD or a cellular modem. When she reaches her office, she can have access to a wireless LAN in office corridors and in meeting rooms. Once inside her cubicle/room, she can connect her laptop to an Ethernet.

Given that a mobile user may encounter different networks, it then becomes necessary for network protocols to adapt dynamically to the changing link conditions. The problem of adapting to a changing environment is further complicated by the fact that the changes in network conditions are usually transparent to the applications. In order to allow automatic adaptation of applications and protocols, awareness of the link conditions is needed.

To this end, this paper makes several contributions. First, we propose a uniform mechanism based on ICMP messages to propagate information about the network environment. Second, we

[1]This research work was supported in part by DARPA under contract number DAAH04-95-1-0596, NSF grant number CCR 95-09620, IRIS 95-09816.

provide an API (Application Programmer's Interface) for supporting protocol adaptation and next, we introduce generalized watermarks for achieving protocol stability. Finally, we demonstrate the effectiveness of dynamic protocol adaptation in a mobile wireless environment by incorporating adaptation in well known protocols like UDP and TCP and evaluating their performance.

The rest of the paper is organized as follows: Section 2 describes what constitutes an environment for mobile hosts and how to represent the state of the environment. Section 3 details the mechanism for propagating changes in the environment to applications and protocols. Section 4 describes various adaptations that are possible given the network related information and presents experimental results that demonstrate the effectiveness of protocol adaptation. The paper concludes with a discussion about related work and some relevant issues for future work.

2 What is an Environment?

Our focus in this paper is on providing support for adaptation in network protocols and communication oriented applications. Such an adaptation requires that applications and network protocols be made aware of the network environment parameters. Some parameters like the cost of communication, latency and bandwidth are well known and exist in the fixed networks. However, mobility introduces large variations in these well known parameters and introduces some new parameters that are of interest to mobile wireless computing. Examples include, signal strength, location and energy.

Available bandwidth may suddenly change due to a change in the default communication device. A mobile host may move out of range from a wireless LAN base station but have access to a CDPD network for data transfer. Moving from a wireless LAN network to a CDPD network can change the available bandwidth by two orders of magnitude. Mobility can also make a device temporarily unavailable. For example, a Wavelan (a popular wireless LAN technology) device can be considered unavailable when the mobile host is out of range of any base station. Cost may vary depending on the time of the day and location from where a connection is made; off-peak hour charges are typically lower and long-distance charges are more expensive than local access charges.

Similarly, a mobile host may connect to a satellite link which has long latencies. Also, for satellite links, the ratio of uplink bandwidth to downlink bandwidth can be fairly large. This has a significant impact on TCP performance [8]. Different locations may need communication with different proxies and location dependent applications may have to contact a different server when the location changes. Limited energy supply may force network protocols to be energy efficient [13, 19, 20, 11]. This is a new consideration for network protocols. Energy might have to be saved at the cost of increased latency or lower throughput.

Thus, the network environment constitutes all the parameters of interest such as latency, bandwidth, energy, cost, signal strength, location etc.

3 Reporting Mechanism

Information about environment related parameters is independent of the applications that need them. Thus, an application independent mechanism for reporting parameters is needed. One naive approach would be to make the device layer visible to the entire protocol stack. Any layer which needs to adapt can then continuously poll the parameter values and take appropriate actions. This approach introduces excessive overhead on protocols. Also, not all changes in parameter values are important for adaptation. An alternative approach is to inform interested protocol layers only when the parameter values change *noticeably*. We call such a change an *event*.

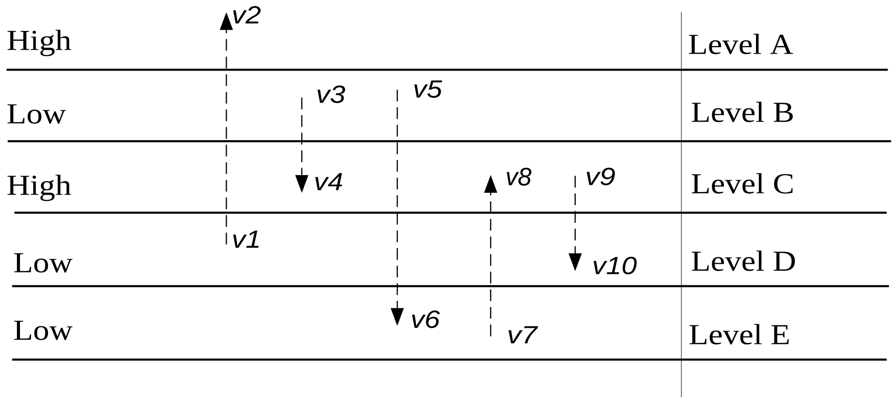


Figure 1: Watermark violations

A change in the behavior of an application or a protocol is required only when an event occurs. Watermarks can be used to decide what constitutes a noticeable change. Use of watermarks helps prevent generation of excessive number of events which can result in protocol thrashing. We introduce a novel concept of generalized watermarks where each parameter is associated with a set of levels (values) as shown in Figure 1. Each level is either a low watermark or a high watermark. A level with a high watermark is violated when the parameter value crosses it from below. For example, in Figure 1, when the parameter value changes from v_1 to v_2, high watermarks C and A are violated. A level with a low watermark is violated when the parameter value crosses it from above. For example, when the value changes from v_3 to v_4, low watermark B is violated. Any violation of any of the watermarks is treated as an event. Sometimes, a change in a parameter value will not result in a watermark violation. When the value changes from v_9 to v_{10}, no watermarks are violated and hence, not events are generated.

There are various reasons for using generalized watermarks as opposed to just two watermarks: low watermark and high watermark. There may be more than two actions (corresponding to low watermark and high watermark) necessary to fine tune a given protocol. Also, not all applications are interested in all the levels supported by the system. It is possible for applications to define coarser watermarks by choosing a subset of the levels supported by the system. For example, an application may be interested only in level A and level D for signal strength. In such a case, only violations of these levels will be reported to the application.

In addition to detecting violations of watermarks, we also need the concept of clearing watermarks. This concept is needed to reset certain previously detected violations. A high watermark is *cleared* if any low watermark below it is violated. For example, when the value of the parameter changes from v_5 to v_6, watermarks B and D are violated and high watermark C is cleared. Similarly a low watermark is *cleared* if any high watermark above it is violated. For example, when the value changes from v_7 to v_8, watermark C is violated and low watermark D is cleared.

3.1 How to report?

For protocol adaptation, environment related information needs to be propagated to all protocol layers. For example, to make transport layer aware of the link quality, events that pertain to changes in the link quality need to be propagated up the protocol stack. A natural way to communicate this information to a protocol layer is to generate an ICMP message. ICMP messages have already been used for propagating network availability information [16]. Here, we use ICMP messages for reporting changes in all device related parameters.

Using ICMP messages to propagate environment aware information offers distinct advantages. ICMP messages are generic, efficient and already exist. As opposed to exposing everything about the device layer to the higher layers, using ICMP messages provides a controlled mechanism, i.e., a *punch hole*, for exposing device related information. Further, it does not matter where the ICMP message originates. Any daemon running on the system, or running on some other machine can send such a message. Once an ICMP message is received, later actions are independent of the mechanism that generated the event.

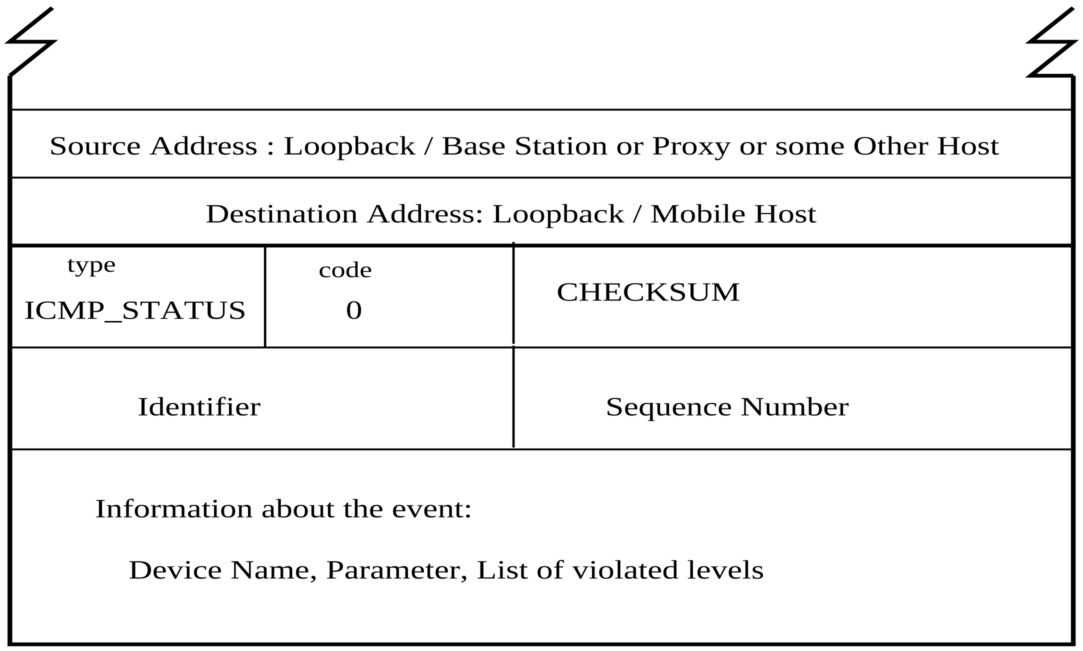

Source Address : Loopback / Base Station or Proxy or some Other Host
Destination Address: Loopback / Mobile Host
type
ICMP_STATUS
code
0
CHECKSUM
Identifier
Sequence Number
Information about the event:
Device Name, Parameter, List of violated levels

Figure 2: ICMP message format

In our system, whenever an event occurs, an ICMP message is generated and sent to the local host. Since the message is interpreted locally, we can use a new ICMP message, *ICMP_STATUS*. The ICMP message contains sufficient information about the event. Structure of such a message is shown in Figure 2. Destination address for the ICMP message is the loopback address if the packet is generated by a local event. In some cases, the base station or some other proxy can report events to the mobile host. ICMP messages used for this will have the address of the mobile host as the destination address and the address of the base station (or the proxy) as the source address. The message contains name of the device for which the event has occured. This device can be a dummy device in case where the event is not associated with any particular device. For example, current workload on a workstation is not a network related parameter and cannot be associated with any network device. The message also contains the name of the parameter that caused the event and a list of watermarks that have been violated.

We also provide a handler for this new ICMP message. The handler invokes a function at the socket layer. This function takes actions for various transport layer protocols and also notifies all interested the applications. Figure 3 shows actions taken in response to an event.

3.2 Reporting events to applications

An application that needs information about events has to create a datagram socket for communicating with the lower layers. It also has to specify what events are of interest. A simple library function is provided for this purpose.

```
request_event (sd, devname, paramname, signal);
int sd; /* socket descriptor */
char *devname, *paramname;
boolean signal; /* Send a signal ? */
```

This function call adds an entry to a table maintained at the socket layer. This entry contains the socket descriptor, process id of the application and the events in which it is interested. Whenever the requested events occur, a message is added to the receive buffer of the socket. Applications can issue a simple read system call on the socket descriptor to read the information. Optionally, a signal can be sent so that events can be reported asynchronously.

3.3 Reporting events to protocol layers

Protocols also need to adapt to changes in network conditions. Existing transport layer protocols do adapt to some parameters. In fact, this is one of the main functions of TCP. It does change behavior when congestion and delays are detected. However, mobility introduces many more parameters that need to be considered. At the same time, these parameters can vary suddenly and drastically. Existing support is not sufficient for adapting to these new parameters. Many variations of TCP have been proposed to deal with specific networks. TCP for space communication [16], TCP over SLIP [14] are some examples. However, having a new TCP for every possible network does not help when network conditions change dynamically. We need the ability to automatically change

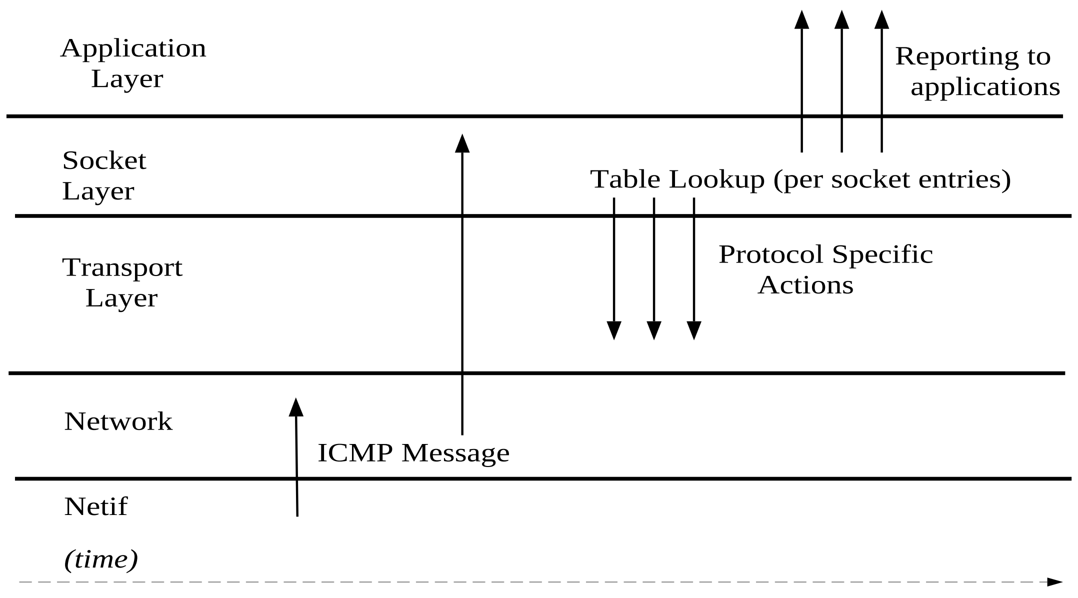

Application Layer
Reporting to applications
Socket Layer
Table Lookup (per socket entries)
Transport Layer
Protocol Specific Actions
Network
ICMP Message
Netif
(time)

Figure 3: Punch holes in the protocol stack

protocol behavior without significant overhead. Hence, we introduce a concept of an action table to allow for automatic protocol adaptation in response to changes in environment related parameters.

3.3.1 Action table

Behavior of a transport layer protocol can be changed by adjusting some protocol specific parameters. For this purpose, we introduce the concept of *actions*. Actions are functions provided by the transport layer to adjust these protocol specific parameters. Allowing applications to perform arbitrary actions often leads to incorrect, insecure or inefficient protocols. Hence, we restrict adaptation to mean choosing from a set of predefined actions provided by the transport layer. This allows efficient and controlled adaptation resulting in a capability of fine tuning protocol behavior. If an application needs extensive adaptation, then it needs to provide that completely in the user space.

Since these actions are invoked whenever an event occurs, they should not be expensive operations. In fact, actions should just set a few protocol specific parameters in the stack. By changing parameters in an appropriate fashion, we can obtain different behaviors from the protocol. For example, when moving from a lossy environment to a lossless environment, TCP timeout values can be reduced. This change can bring TCP out of backoff very quickly. In case there is no parameter to obtain appropriate behavior, new parameters can be introduced. For example, by introducing a *sleep* parameter in UDP, it is possible to put a process to sleep until link conditions improve.

To provide support for protocol adaptation, we use action tables. The socket layer maintains one action table per socket that requires adaptation. Each entry in the action table has the following template.

```
parameter(p), watermark(w) => ACTION(a), arg
```

When parameter p violates watermark w, perform action a where a is an index corresponding to some action provided by the transport layer and *arg* is passed to the action as an argument. Actions that do not require any argument ignore the argument passed to them. Note that no device has been specified here because all the actions in the table depend on the device parameters (like loss and bandwidth) and not the device. When the communication device changes, all the entries of the action table are reexamined to decide the behavior for the new device.

Entries can be added to the action table by a setsockopt on the socket. Again, a simple library function has been provided for this reason.

```
set_simple_action (sd, sol, parameter, watermark, action, arg);
int sd, watermark, action, arg;
int sol; /* Layer at which adaptation is requested: SOL_TCP, SOL_UDP etc. */
char *parameter;
```

Actions are removed from the action table when the socket is closed or the process using the socket terminates. Applications can also explicitly remove actions from the action table.

Each action being added to the table is a simple action, *i.e.,* decision to perform the action depends only on the occurrence of one event. Simple actions like the ones defined above are not sufficient to provide generic behavior. In many cases, actions may depend on a boolean combination of events. For example, putting the process to sleep when both the loss rate and the cost of sending packets are high may be the desired behavior. To achieve such a functionality, complex actions are provided. A boolean combination is achieved by a two layer threshold circuit. Threshold circuits allow efficient evaluation by eliminating the need for reevaluating the entire boolean expression every time. Any time an event occurs, output of the circuit is computed incrementally. If the output of the circuit is above a certain threshold, the action is performed. Figure 4 shows how a complex condition can be expressed as a threshold circuit.

Parameter	Watermark	Action	Arg for action	AND Gate
energy	low_1	Discard	0	*null*
energy	$high_2$	Normal	0	*null*
Cost	$high_1$	Sleep	0	1
Loss	$high_2$	Sleep	0	1
Bandwidth	low_2	Sleep	0	2
Latency	$high_2$	Sleep	0	2

Table 1: Action table

Applications can request such actions by a library function call.

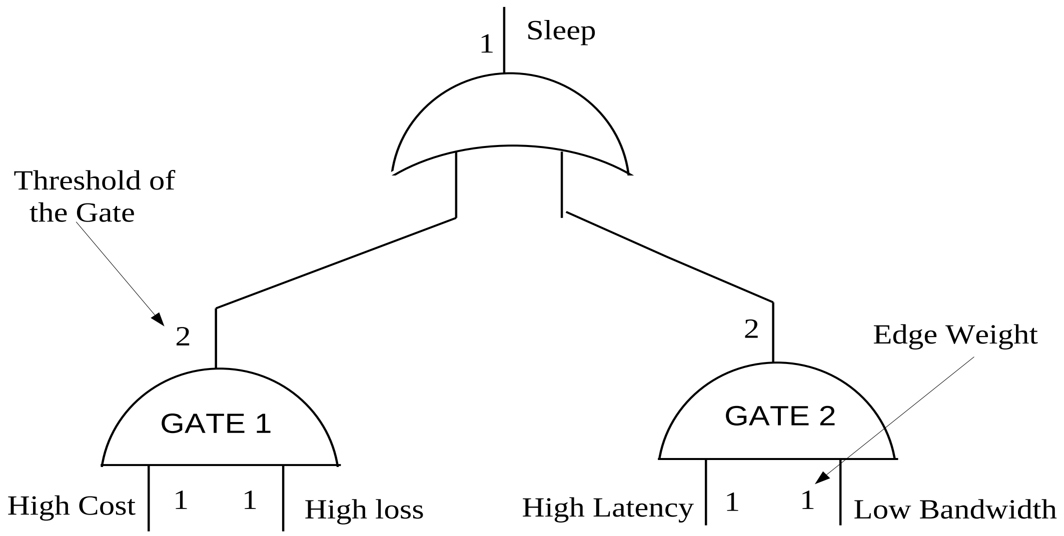
```
set_complex_action (sd, sol, action, arg, fanin, p1, w1, p2, w2, ...);
int sd, sol, nactions, arg;
char *p1, *p2, ...;
int action, w1, w2, ...;
```

Table 1 shows entries of the action table when an application requests the following actions:

```
/* when energy is low, discard all outgoing packets */
set_simple_action (sd, SOL_UDP, "energy", LOW_1, UDP_DISCARD, 0);
set_simple_action (sd, SOL_UDP, "energy", HIGH_2, UDP_NOP, 0);

/* Sleep when cost is high and loss is high
   OR when latency is high and bandwidth is low */
set_complex_action (sd, SOL_UDP, UDP_SLEEP, 0, 2, "cost", HIGH_1, "loss", HIGH_2);
set_complex_action (sd, SOL_UDP, UDP_SLEEP, 0, 2, "latency", HIGH_2, "bw", LOW_2);
```

HIGH_*k* corresponds to *k-th* high watermark. Similarly, LOW_*k* corresponds to *k-th* low watermark. The first two entries in the table corresponds to the simple action. Calls to set_complex_action result in the remaining entries in the table. The resulting complex action is also shown in Figure 4.



Sleep if bandwidth is low and latency is high, or if cost is high and loss is also high

Figure 4: Threshold circuits for representing a complex action

For supporting complex actions by means of a two level threshold circuit, the action table has been augmented with an entry for specifying the identity of the AND gate. For simple actions, AND gate field is set to *null* and is ignored. For complex actions, every time one of the inputs to an AND gate changes, the output of the threshold circuit is computed. For example, in Figure 4, output of GATE 1 will be 1 when cost becomes high, which is less than the gate threshold. When loss also becomes high, GATE 1 output becomes 2 indicating that both the inputs for this gate are true. This sets the input of the second level gate to 1 which is equal to the gate threshold. So, the action *sleep* has to be performed. This action will set the *sleep* parameter for UDP. GATE 2 works in a similar way. When a watermark is cleared, the output of the corresponding AND gates is decremented. Clearing watermarks allows us to take the action again when another violation is reported. Every time an event occurs, a number of actions may have to be performed. Once all the actions are decided, they are performed in order.

3.4 Setting the parameter values

How to detect or measure values for various parameters is independent of the reporting mechanism. Detection can take place at various layers. Some device drivers may be able to determine loss probability (Wavelan can infer loss probability based on signal strength) whereas an application layer code may be needed to detect any change in service provider charges.

Since, detection is a parameter dependent process, various access points have been provided to set and to read these parameters. At an application level, ioctl on the device can be used to set the parameters. Typically, parameters like cost need to be set this way by applications. For this purpose, library functions have been provided so that applications can set or get parameters for various devices.

```
set_dev_param (devname, paramname, value);
char *devname, *paramname;
int value;

get_dev_params (devname, buffer);
char *devname;
struct ifreq_param *buffer;
```

However, the kernel (the protocol stack in particular) can make a simple function call to set these parameters. For example, a Wavelan device driver sets signal quality information by using a simple function. Once the parameters are set, rest of the propagation mechanism and mechanism for adapting to changes is uniform.

The interface provided for applications as well as the kernel code checks watermark violations whenever a parameter value is set.

4 Protocol Stack Adaptation

This section describes a few possible adaptations for the well known transport layer protocols viz. UDP and TCP.

UDP does not offer many parameters that can fine tune its behavior. However, UDP can be made to defer transmission or send redundant packets when the link condition deteriorates. This type of UDP behavior can be obtained by adding additional parameters *sleep, discard, buffer* and *fail* to the protocol. These parameters can be checked just before sending out a new datagram. A process can be made to sleep when events indicate that it not worth sending packets. These events may be increase in cost, increase in loss etc. As soon as the conditions improve, the process can resume execution. For real time applications, lost packets are not worth retransmitting; packets can be dropped if network conditions indicate a very high likelihood of packets getting lost. In such cases, not sending the packet over the network results in energy savings. Alternatively, *write* system call can be made to fail when bad network condition is detected and applications can decide on further actions. Also, another possibility is to buffer the packets in the protocol stack so that applications can continue as if nothing has happened. These packets can be transmitted when the conditions improve. An action is also needed to set the behavior back to normal or default when conditions change.

TCP behavior can be changed similarly but not with equal ease. TCP offers a lot of parameters that can change its behavior (a closer examination of the socket structure gives an idea). We suggest a few possible actions in this section. These actions have not been studied in detail for determining their effect on TCP. There are definitely a lot more possible actions and more work is needed to decide the smallest useful set of actions.

- Reset Retransmission Timers (TCP_RST_TIMERS) : In a lossy network, TCP tends to back-off. Then, even when the network conditions improve, it takes time to get back to the normal flow. This is done to avoid congestion and congestion is a normal reason for losses in fixed networks. However, in wireless networks, losses usually are real losses. So, when the conditions improve, TCP can safely resume normal flow.

- Send duplicate ACKs when a loss is detected (TCP_DUPACKS): Three duplicate ACKs will start a fast retransmit on the sender side [4]. If losses are known to be real losses, this can be used to initiate retransmission on the sender side without waiting for the coarse timeout. Additional duplicate ACKs can be used to open up the congestion window at the sender. However, this should be done only when it is known that the sender has shrunk the congestion window. Thus for isolated losses dues to lossy behavior of the wireless links, three duplicate ACKs can be sent. For known bursts of losses like those during a handoff, we can send more duplicate ACKs to open up the sender side congestion window.

- Increase the number of delayed ACKs: TCP specification allows delaying one ACK. If one is willing to violate the specifications, it is possible to delay more than one ACK. This may be useful when the uplink is extremely slow compared to the downlink.

- Delayed ACK timer: Again, delayed ACK timer, according to TCP specifications, has to be less than 500 msec. This may not be useful when the round trip time is much larger. Delayed ACK timer can be increased to reduce uplink traffic.

- Change estimates: Round trip time and MTU estimates can be revised when a device change occurs.

5 Measurements

To show the effectiveness of adaptation mechanism, we performed a number of experiments. For all our experiments, we used a wireless LAN configuration. The wireless LAN consists of fixed hosts (or base stations) and mobile hosts (MH). Fixed hosts are pentium based desktop machines running the linux operating system (version 2.1.24). Mobile hosts are pentium based laptops and also run linux. Base stations as well as mobile hosts use 2Mbps Wavelan technology for wireless communication. In addition, base stations have a wired interface to 10 Mbps Ethernet. To support mobility, mobile IP code developed at State University of New York at Binghamton [7] is used.

This section shows how some simple actions are effective in improving performance of the well known protocols like UDP and TCP. These are by no means the only actions that are possible. The purpose of these measurements is to show that our framework allows easy adaptation to changing network conditions.

5.1 Overhead measurements

	TCP Throughput Mb/s (Wavelan)	TCP throughput Mb/s (Ethernet)
Unmodified kernel	1.16	7.43
Modified kernel no events	1.16	7.47
Modified kernel 100 events per sec	1.16	7.39
Modified kernel 100 events per sec (10 interested apps)	1.16	7.46

Table 2: Overhead measurements

To measure overheads caused by the event reporting mechanism and protocol adaption mechanism, we measured TCP throughput by using the *ttcp* program for three different cases. The throughput reported in table 2 is average throughput of 10 runs. During each run the mobile host sent 10240 packets of 1400 bytes each. The granularity of the clock used for measurements is 10 msec.

In the first case, mobile host was running an unmodified kernel. This was done to determine the throughput without any overhead of propagating event information *i.e.* to obtain a base line for comparison. Next the mobile host was running our modified kernel which has support for event notification. However, no events were generated. In the third case, the mobile host was again running the modified kernel but events were generated at various frequencies from one per 500 msec to one per 10 msec. The action table had ten entries and the experiment was conducted with and without applications requesting this event information. This was used for measuring overheads of delivering the events and reacting to them.

All the measurements were done for Wavelan as well as Ethernet to see if a faster interface shows any significant overheads.

As shown in Table 2 the measured throughput was same in all the cases. For Ethernet, the values reported do not show any fixed pattern. When the tests were repeated at different times, even for the unmodified kernel, the average Ethernet throughput varied from 7.35 Mb/s to 7.60 Mb/s. No particular case consistently has a higher or a lower throughput than any other case. That implies that the added processing required because of our modifications is negligible as compared to the protocol processing time.

5.2 UDP adaptation

UDP is an unreliable protocol. Applications that use UDP are aware of unreliable nature of UDP and can tolerate packet loss to a certain extent. However, in a wireless environment, these losses become unacceptably high when mobile hosts cross cell boundaries. If transmission can be deferred temporarily while crossing cell boundaries, losses can be reduced to a great extent. Using our framework, we provide a simple way to achieve such a behavior.

To report handoffs, we use the device parameter *availability*. The watermarks for this parameter are shown in Table 3. These watermarks are set by calling the following library function. The parameter *wms* contains the values and the types (*high or low*) for the all the watermarks from 0 to 4.

```
set_dev_param_wms (devname, paramname, wms);
char *devname, *paramname;
struct watermarks wms;
```

Watermark	Description	Value	Type
0	Device is unavailable	10	Low
1	Device is available	20	High
2	Dummy	30	Low
3	About to handoff	40	High
4	Handoff over	50	High

Table 3: Watermarks for handoff notification

We modified the mobile IP daemon to set these watermarks appropriately. The parameter value is initialized to 0 indicating that the device is not available. When the mobile host detects presence of a base station (by listening to the beacons), the value of this parameter is set to 25. That violates watermark 1 indicating that the device is now available. Whenever the mobile IP daemon decides that a handoff is necessary, it sets the value of the *availability* parameter to 45, violating the watermark 3. On completion of the handoff procedure, it sets the value to 55. That indicates that the handoff procedure is over - either the mobile host has moved to a new location or the handoff failed and the mobile host is still with the old location. In either case, the communication is possible. Following this, the value is set back to 25 immediately. This violates the low watermark 2 (dummy) and clears the watermarks 3 and 4. The cycle can repeat again when another handoff takes place. In normal operations, the steady state value of the parameter is 25.

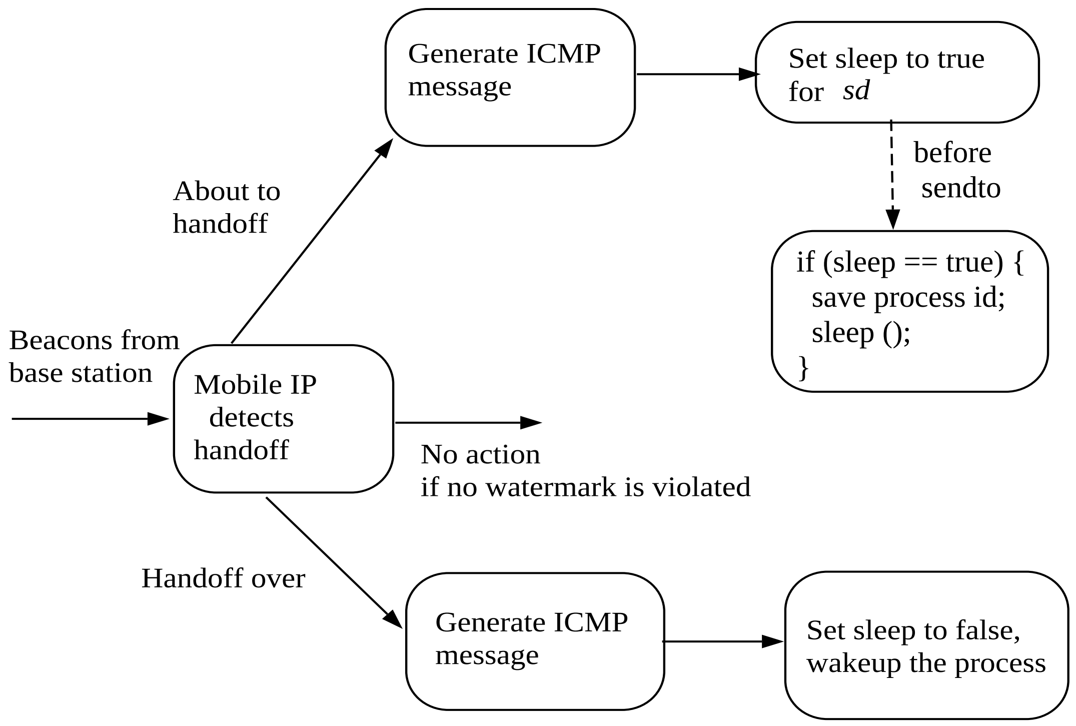


Figure 5: UDP adaptation: message flow

Our mechanism allows an easy way to defer transmission during handoffs. Code required for this modification is simple.

```
sd = socket (AF_INET, SOCK_DGRAM, 0);
set_simple_action (sd, SOL_UDP, "avail", 3, UDP_SLEEP, 0);
set_simple_action (sd, SOL_UDP, "avail", 4, UDP_WAKEUP, 0);
```

Behavior of UDP with this modification is shown in Figure 5. Whenever the mobile host is about to perform a handoff procedure, an ICMP message is generated. A table lookup indicates that the socket *sd* has requested deferred transmission. Value of *sleep* parameter for this socket is then set to true. When a send is issued on this socket at some later time, the process using the socket is put to sleep. When the handoff procedure is over, another ICMP message is generated. The *sleep* parameter is then reset to false and the process is woken up so that transmission can be resumed.

This is an acceptable behavior for many simple applications. However, some applications may be interested in performing some special processing while the handoff is being performed. In such

Beaconing interval	# Packets lost: Unmodified # Handoffs				# Packets lost: Modified # Handoffs				Savings per handoff # Packets
	1	2	3	6	1	2	3	6	
500 msec	154	242	390	654	99	150	237	432	43
1 sec	277	476	672	1170	168	244	447	762	81
2 sec	386	812	1188	2298	236	412	702	1236	174

Table 4: UDP adaptation: Packet loss during handoffs

cases, the action UDP_SLEEP can be replaced by another action which will buffer the packets in a transparent manner, or which can force the write system call to return with an error. In any case, the loss pattern that we see will be similar as long as there is no transmission during handoff.

To determine effectiveness of this adaptation, we performed the following experiment. We compared packet loss for standard UDP and the modified UDP. In the modified protocol, transmission was suspended while the handoff was being performed. The experiment was repeated for three beaconing intervals (interval between two beacons sent by the base stations): 500 msec, 1 sec and 2 sec. For each beaconing interval, loss was measured for four different cases; one, two, three and six handoffs per run. During each run, the mobile host sent out 10240 packets of 1400 bytes each. Ten such runs were conducted for each of the four cases mentioned above.

Results of the experiment are summarized in Table 4. The figures indicated are average loss per handoff at various beaconing intervals. The modified protocol can save data equivalent to the beaconing interval *i.e.* data that can be sent in one beaconing interval.

This modification raises a few interesting questions. How long should an application sleep? What happens if the signal quality does not improve for a long time? Applications may need to specify maximum time for which transmission can be deferred. We provide a mechanism where applications can specify a parameter for actions. However, exact nature of the action to be taken depends on the application and needs further work.

5.3 TCP adaptation: Sender

In wired networks, losses are infrequent and are usually a result of congestion. Thus, whenever a packet loss is detected, TCP assumes the reason to be congestion and backs off exponentially to slow down transmission. However, this does not work very well in wireless networks where losses are real losses over the wireless link. Unfortunately, TCP reacts to these losses by backing off; resulting in underutilization of the link. TCP can be modified to resume normal flow (without waiting for non-existing congestion to subside) when conditions are no longer lossy. Such an adaptation can easily be achieved with our adaptation mechanism. A socket can request use of modified TCP by calling a simple function.

```
sd = socket (AF_INET, SOCK_STREAM, 0);
set_simple_action (sd, SOL_TCP, "sq", HIGH_MARK, TCP_RST_TIMERS, 0);
```

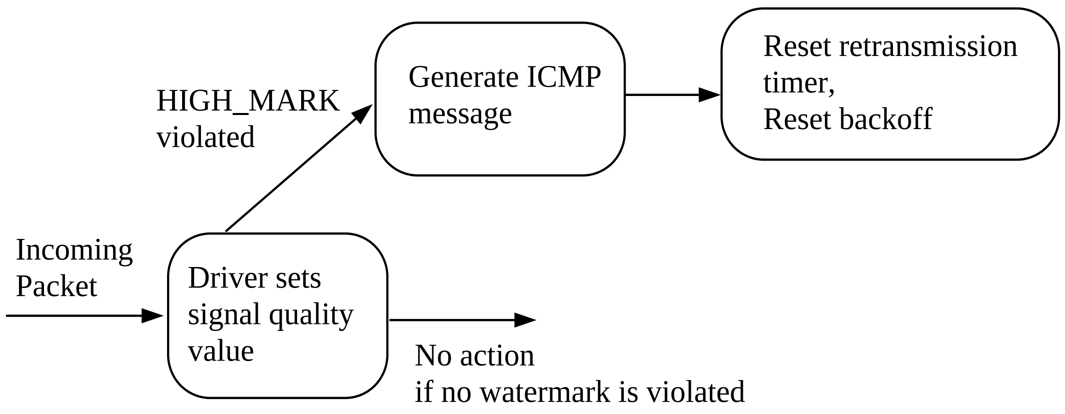

HIGH_MARK violated
Generate ICMP message
Reset retransmission timer, Reset backoff
Incoming Packet
Driver sets signal quality value
No action if no watermark is violated

Figure 6: TCP adaptation: message flow

Figure 6 shows actions taken when signal quality violates the high watermark. High watermark violation leads to an ICMP message. Since the socket *sd* has requested adaptation, the retransmission timer and backoff associated with this socket are reset to a small value. The retransmission timer then expires earlier and retransmission of the lost packet occurs sooner. Without the modification, TCP would wait for the retransmission timer to expire (which has been set to a high value by the exponential backoff). Thus, whenever link conditions improve, modified TCP gets out of backoff much faster than unmodified TCP.

We performed a simple experiment to demonstrate effectiveness of modified TCP. We measured TCP throughput across wireless link by using the *ttcp* program. Observations were made at various points from the base station. The figures for the *Near* region in Table 5 are obtained when the mobile host is near the base station and the signal strength is either 14 or 15. The figures for the *Far* region are obtained by moving the mobile host in the region where the signal quality oscillates around 13.

When there is no packet loss, modified TCP behaves similar to standard TCP. However, while moving from a bad signal zone to a good signal zone, modified TCP recovers from backoff much faster.

Distance from base station	Throughput Mb/s (unmodified)	Throughput Mb/s (modified)	Approx. Backoff (sec)
Near	1.16	1.16	0
Far	0.99	1.05	$\approx$ 5
Far	0.51	0.62	$\approx$ 10
Far	¡ 0.1	0.30	$\approx$ 50

Table 5: TCP adaptation: Sender

Table 5 shows results of this experiment. In the region near the base station, modification does not have any advantage neither does it have any overheads. The throughput figures are same as normal TCP. However, as one starts moving away from the base station, TCP starts performing more and more poorly. This is because every packet loss leads to a backoff. Every time TCP backs off, next try for sending the packet occurs less frequently. The modified TCP improves this situation. Whenever, it receives strong beacons from the base station, it resets its retransmission timeout to a small value. Thus, retransmission attempts are more in number. As the backoff increases, better gains are observed. This will lead to problems if losses are really because of congestion. However, in this experiment, TCP end points are the end points of the wireless link. Thus, losses were real losses. In other cases, same modification can be used if losses can be detected [2]. (Note that loss is another device parameter).

5.4 TCP adaptation: Receiver

A similar simple adaptation is possible when the mobile host is the receiver. Whenever a handoff takes place, the sender backs off due to losses during the handoff. Then, even when the conditions are back to normal, sender does not resume the transmission immediately. The mobile host can send extra duplicate ACKs to trigger a retransmission at the sender. Three duplicate ACKs are sufficient for this purpose. However, we send five ACKs. Every additional duplicate ACK opens up the congestion window at the sender. An application on the mobile host can request such a behavior by making a function call.

```
sd = socket (AF_INET, SOCK_STREAM, 0);
set_simple_action (sd, SOL_TCP, "avail", 4, TCP_DUPACKS, 5);
```

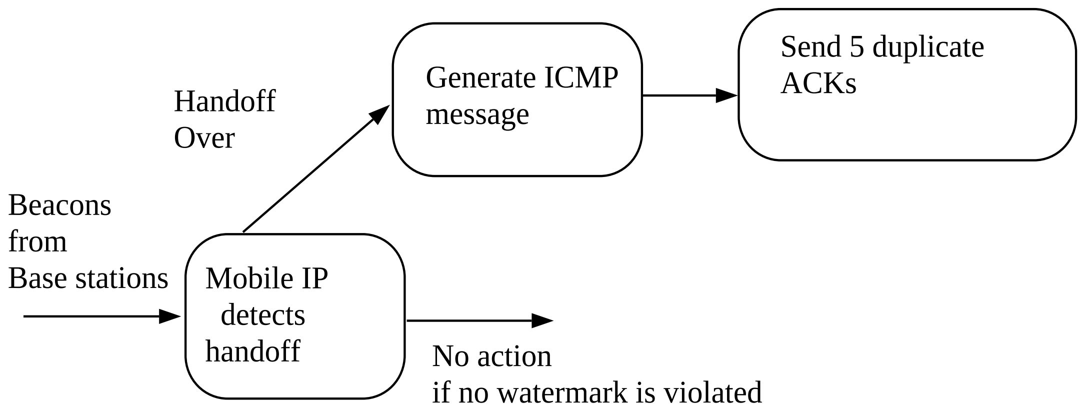

Handoff
Over
Generate ICMP message
Send 5 duplicate ACKs
Beacons from Base stations
Mobile IP detects handoff
No action
if no watermark is violated

Figure 7: TCP adaptation: message flow

Figure 7 shows the flow of messages when such an adaptation is requests by an application. To measure the effect of this adaptation, we again compared normal TCP and modified TCP. The experiments were performed for three different beaconing intervals: 500 msec, 1 sec and 2 sec. For every beaconing interval, time required to complete the transfer of 10240 packets of 1400 bytes each was measured. Different runs were performed with one, two, three and six handoffs per run. Table 6 shows the average time required for the transfer. The TCP throughput for the mobile host with the home agent is larger than the TCP throughput for the mobile host with the foreign agent. Hence, the total time required depends on the ratio amount of time the mobile host spends

Beaconing interval	Time (sec): Unmodified # Handoffs				Time (sec): Modified # Handoffs				Savings per handoff (sec)
	1	2	3	6	1	2	3	6	
500 msec	85.37	86.63	87.08	88.95	84.24	85.05	85.15	85.67	0.66
1 sec	84.58	88.01	90.96	100.71	83.05	85.90	87.87	92.97	1.20
2 sec	84.44	89.01	86.77	108.13	82.11	84.50	88.73	96.32	2.22

Table 6: TCP adaptation: Receiver (time required for transferring 10240*1440 bytes)

with the home agent to the time it spends with the foreign agent. To remove disparity because of this reason, the handoffs were forced at fixed times. From Table 6, we can see that during every handoff the modified TCP can save time approximately equal to the beaconing frequency. When the experiments were repeated at a different time, modified TCP consistently showed a better performance.

5.5 Other adaptations

This section showed one possible binding between some events and actions. There are other possibilities that can be exploited by applications. For example, *sleep* action may be associated with other events. A mobile host may temporarily go in a *blind* region (the car carrying the mobile host may pass through a tunnel) and lose communication with all the base stations. For some applications which usually run in background (like transfer of mail messages), *sleep* can be used when the cost of sending packets increase. Similarly, TCP can use the TCP_RST_TIMERS action after a handoff. The mechanism presented in this paper allows all such bindings.

In some cases, the handoffs may be across different types of networks. Moving from an Ethernet to satellite communication can increase the asymmetry of the network. The BARWAN project [3] at the University of California, Berkeley suggests how applications can take various actions when such changes occur. Our mechanism can be used as a framework for such adaptations.

6 Issues

This new mechanism for adaptation gives rise to a number of important research issues.

- Setting watermarks: Watermarks that are widely separated may fail to catch changes in the parameter values. On the other hand, watermarks that are too close may lead to thrashing. Also, fine grained watermarks imply that there will be more watermark violations and consequently, there will be more overheads because of the reporting mechanism. The system has to be initialized to set the watermarks that are appropriate for every parameter. Further study is required regarding changes in various parameter values and how applications adapt. This will give some insight into what constitutes a good set of watermarks for each parameter.

- Hysteresis: Hysteresis means lagging behind of an effect. This is used to decide the current state of any of the device parameters. In other words, current state is a function of last few values of the parameter rather than the value at a particular instance. So, a sudden transient change in the value does not affect the current state drastically. Change in the current sate usually implies some action on protocol's part. Protocol behavior usually changes to force the state back to steady state. For example, TCP decides retransmission timeout (RTO) based on round trip delay. RTO depends not only on the round trip time for the last packet but also the previous value of RTO. Changing retransmission timers is a corrective action on TCP's part to reach a steady state. Since these corrective actions take some time to show their effect, they are meaningful only when the changed state lasts for some significant amount of time. For transient departures from steady states, these actions are not required.

  Such transients can be avoided by using hysteresis. Instead of using the last measured value, a weighted average of last few values can be used. It gives rise some new problems. What should the weights be for calculating average? Should these weights be unique for a parameter or should they vary from application to application?

- What is a good set of actions: The set of actions provided by the transport layer should be such that the protocol behavior can be changed in a useful manner. A good set of actions is needed to show that most of the desirable results can be obtained just by setting the parameters. For a complex protocol like TCP, care has to be taken that the actions provided are consistent, *i.e.* an event does not lead to two conflicting actions. A lot of work has gone into modifying TCP for various network conditions. A good action set should enable us to select any of these easily.

7 Related Work

Mechanisms for making applications aware of the network conditions has been suggested in the literature. ICMP messages have been used for allowing isolated functioning of disconnected systems [16]. A more generic system for application aware adaptation [17] describes a required supporting architecture. In case of protocol adaptation, TCP has been very widely studied. Numerous flavors of TCP have been suggested for dealing with various network conditions. TCP for space communication [8], header compression in wireless environment [6], semi-connected TCP/IP [10] are just a few examples. Adaptation for mobile IP [5] has also been proposed. Allowing applications to control protocol behavior [9, 12] has also been studied.

8 Future Work

To allow a very generic adaptation, work is needed on providing support for hysteresis and deciding a set of actions for various transport layer protocols. We are currently working on achieving hysteresis by introducing a *hysteresis thread.* This thread can generate an ICMP message to report changes *consistent with hysteresis policy.*

All the adaptation mechanisms described in this paper are performed at the mobile host without any support from the network. We are currently studying mechanisms to support an environment aware network. Such a network can provide adaptation at two end points of a connection; one at the network end and the other at the mobile host end. For example, a mobile host can request that data be encrypted. Moreover, such a behavior might be required only in foreign networks. Another example is that if a data flow is made bursty, the mobile host can periodically go in doze mode to match the burstiness of the data. This will result in lower energy consumption at the mobile host.

We are investigating the use of special purpose tunnels like *encryption tunnel, energy tunnel etc.* on the last (or the first) hop or between the mobile host and a proxy. Such tunnels can provide required behavior on the wireless link. Again, the same mechanism of ICMP messages can be used to dynamically configure the tunnels.

It is also conceivable that hosts that communicate with mobile hosts are aware of mobility. In such cases, some other type of adaptations [15, 18, 1] are possible.

9 Conclusion

The results in this paper shows that it is possible to make network information available to applications and protocol stack with little processing overheads. We have described a simple and effective mechanism for adaptation using action tables by which protocols can change their behavior dynamically.

Experimental evidence shows the effectiveness of the adaptation mechanism and the simplicity of the API proposed in this paper. Even with a single simple action, it is possible to get better performance from UDP and TCP for a specific environment. With availability of a good set of actions, it will be possible to change protocol behavior easily and efficiently. A reasonably rich action set will allow protocols to adapt to a wide variety of environments.

References

[1] Bikram S. Baksi, R. Krishna, N.H.Vaidya, and D.K.Pradhan. Improving performance of TCP over wireless networks. In *17th International conference on distributed computing systems*, may 1997.

[2] Hari Balakrishnan, Venkat Padmanabhan, Srinivasan Seshan, and Randy H. Katz. A comparison of mechanisms for improving tcp performance over wireless links. In *In Proceedings of the ACM SIGCOMM'96*, pages 256–269, 1996.

[3] Bay area research wireless access networks (barwan). http://http.cs.berkeley.edu/r̃andy/Daedalus/BARWAN/BARWAN_over.html.

[4] R. Caceres and L. Iftode. Improving the performance of reliable transport protocols in mobile computing environments. *IEEE Selected Areas in Communications*, 13(5):850–857, June 1994.

[5] Stuar Cheshire and Mary Baker. Internet mobility 4x4. In *Proccedings of the ACM SIGCOMM*, pages 318–329, August 1996.

[6] Mikael Degermark and Stephen Pink. Soft state header compression for wireless networks. In *Proceedings of the second annual international conference on mobile computing and networking*, pages 1–14, 1996.

[7] Abhijit Dixit, Vipul Gupta, and Ben Lancki. Linux mobile IP:implementation overview.

[8] Robert Durst, Gregory J. Miller, and Eric J. Travis. TCP extensions for space communication. In *Proceedings of the second annual international conference on mobile computing and networking*, pages 15–26, 1996.

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

[10] Jørgen Sværke Hansen, Torben Reich, and Birger Andersen. Semi-connected TCP/IP in a mobile computing environment. ftp://ftp.diku.dk/pub/diku/users/cyller/taco/imc96paper.ps.gz.

[11] David P. Helmbold, Darrel D. E. Long, and Bruce Sherrod. A dynamic disk spin-down technique for mobile computing. In *Proceedings of the MOBICOM'96*, pages 130–142, 1996.

[12] 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.

[13] T. Imielinski, S. Vishwanathan, and B.R. Badrinath. Energy efficient indexing on air. In *Proceedings of the international conference on Management of Data - ACM SIGMOD*, 1994.

[14] V. Jacobson. Compressing TCP/IP headers for low speed serial links, 1990. Requests for Comments 1144.

[15] Matthew Mathis and Jamshid Mahdavi. Forward acknowledgment: Refining TCP congestion control. In *Proceedings of ACM SIGCOM*, pages 281–291, August 1996.

[16] Gabriel Montenegro and Steve Drach. System isolation and network fast-fail capability in solaris. In *Second USENIX Symposium on mobile and location-independent computing proceedings*, April 1995.

[17] Brian Noble, Morgan Price, and M. Satyanarayanan. A programming interface for application-aware adaptation in mobile computing. In *Second USENIX Symposium on mobile and location-independent computing proceedings*, April 1995.

[18] S.Keshav and S.P.Morgan. Smart retransmission: Performance with overload and random losses. In *Proceedings of INFOCOM*, 1997.

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

[20] Mark Weiser, Brent Welch, Alan Demeres, and Scott Shenker. Scheduling for reduced CPU energy. In *Proceedings of the first USENIX symposium on OSDI*, pages 13–23, 1994.