Policy and Mechanism in Adaptive Protocols

Ilija Hadžić, William S. Marcus and Jonathan M. Smith
University of Pennsylvania and Bellcore *
ihadzic@ee.upenn.edu, wsm@bellcore.com, jms@cis.upenn.edu

Abstract

Adaptive protocols are protocols which automatically adjust their behavior to runtime phenomena such as traffic or link characteristics. For such protocols, the behavioral adjustment is accomplished with some *mechanism*; the decision as to how much (if any) adjustment is needed is made under control of a *policy*. Design and implementation of policies has often proven more challenging than that of mechanisms.

We make three contributions in this paper. First, we develop a rule of thumb for policy/mechanism separation and lay out a general set of challenges in policy module design. Second, as an illustration, we have analyzed a Forward-Error Correcting Code (FEC) for ATM supporting TCP/IP, and analytically identified a small robust set of "tunable parameters" to delineate regions where the code should be applied. Third, we exploited the analytic results and policy/mechanism separation to implement an adaptive protocol for network errors using programmable hardware to obtain high performance.

Measurements of TCP/IP traffic over a 155 Mbps ATM link augmented with two cooperating Programmable Protocol Processing Pipelines were made with `ttcp`. The new policy module improves throughput by up to 50% over an unaugmented TCP/IP in the face of increasing bit error rates, and continue operation into a range of bit error rates where an unmodified TCP practically ceases to function.

1 Introduction

The complexity of networks and their traffic has been an incentive for constant reexamination of protocols and protocol architectures. Among the recent phenomena motivating the study of *adaptive* protocols are the increasing presence of wireless links in the Internet, as well as the increasing range of traffic types such as HTTP and streaming media. Traditional protocol architectures have provided for adaptation in important ways, such as TCP/IP's congestion-control algorithms, which adapt to the available bottleneck throughput. TCP/IP's algorithms provide illustrative examples of *policy* and *mechanism*. The measurable parameters it uses to detect IP packet loss are checksums, sequence numbers and timers. The "congestion window" size reflects a policy decision in response to these phenomena. However, the adaptation involves no restructuring of the "protocol graph" which describes the protocol's behavior.

The performance of an adaptive protocol organized as a policy controlling use of a mechanism depends on three factors. First, the overhead introduced by a mechanism must be low enough that it improves performance for a significant operating range of the protocol. Second, and demonstrably more important, policy modules must correctly identify opportunities for deployment of mechanisms. Third, the time required to react to the opportunity (*i.e.*, deploy the mechanism) must be short enough to ensure that the opportunity does not evaporate. This third factor, for example, might suggest that, for short fluctuations in network conditions, no action be taken to avoid "thrashing" of the protocol graph. As in the TCP example, the design of policy tends to be more challenging than the choice of mechanism.

One proposal for adaptive protocols are the "Protocol Boosters" proposed by Feldmeier, *et al.*[2, 20]. In this model, functions are dynamically inserted and removed from the protocol. This provides an attractive model for analysis of policy, since the outcome of the policy is simple: insert or remove the function. The goal of the boosters work was to improve end-to-end performance by appropriate runtime selection of functions. When reframed in terms of policy and mechanism, a variety of mechanisms have been demonstrated. However, the inability to provide a convincing example of a policy module has hindered claims that protocol boosters are a viable means of dynamically adapting protocols to traffic and link characteristics.

This paper provides a novel analysis of adaptive protocols using the Protocol Boosters model as a basis. Since design of a policy module is dependent on both the targeted application and the associated mechanism module, we choose particular examples with which to test our analysis. The targeted application is TCP/IP throughput improvement in a noisy environment, and

*This research was supported by DARPA under Contracts #NCR95-20963 and #DABT63-95-C-0073, the AT&T Foundation, the Hewlett-Packard Corporation, the Intel Corporation and the Altera University Grants Program.

the mechanism is forward error correction (FEC). FEC has the interesting property that it trades an increase in bandwidth use (for the redundant information in the code) to achieve an overall increase in throughput (fewer retransmissions, no mistaken congestion backoff, *etc.*).

In the next section, we quickly review previous work in adaptive protocols and identify policy design as the common challenge. Section 3 develops a general framework to analyze policy modules. In section 4, we review the mechanism (an FEC "booster") and its implementation platform the Programmable Protocol Processing Pipeline (P4) and illustrate the policy module design on a case study. Section 5 shows the advantages offered to a policy module by tunable parameters by demonstrating them on the FEC policy module. Section 6 describes and justifies our performance indicator, describes our test setup, and gives our measured results. We outline future work and conclude the paper in Section 7.

2 Previous Work

The benefit of adaptation in protocols is clear, for example in the TCP/IP protocol's adaptation to increases and decreases in bottleneck bandwidth[11]. More aggressive forms of protocol adaptation have been attempted in the wireless domain[14, 4, 12] because the complexity and variation of link behavior is high. In [14], adaptation is performed on the basis of application characteristics. The work reported in [6] showed that link-layer adaptation can significantly improve performance.

Adaptation in the structure of protocols was achieved manually in the x-Kernel dynamic protocol architecture[17], which compiled protocol modules organized into a protocol graph to produce an operational protocol. The Protocol Boosters[2] architecture allows the protocol graph to be dynamically reconstructed at runtime. Perhaps more importantly from an analytic point of view the boosters architecture provides a simple on/off model for use in analyzing policies. Active Networks[3] can subsume protocol adaptation by offering end-user programmability of a network infrastructure, but are faced with the same policy issues we address in this paper for the design of any specific solution. We note that protocol adaptation is not limited to software-based architectures[8].

We reiterate that the major challenge for *any* adaptation scheme is the policy decision, that is, *when* should the protocol graph be reorganized. In a light-hearted manner, "To boost or not to boost!". In this work we provide the common model for making policy decisions and illustrate a proof of concept on a concrete example.

3 Policy Modules

Because policy strongly depends on the type of mechanism it controls and the application it is targeted to, it is not possible to define a general policy independently of the associated mechanism module. It is however possible to identify design challenges that are common for all policy modules. In this section we describe a common framework for implementing policy modules, and move to the case study of an FEC booster and its policy module in the following section.

At the highest level, the role of a policy module is to monitor network conditions and decide to activate or deactivate the mechanism. In general, tasks performed by policy modules are *measurements*, *decision* and *synchronization*. Submodules (illustrated in Figure 1) which perform these functions are called *monitors*, *decision submodules* and *signalling protocols*.

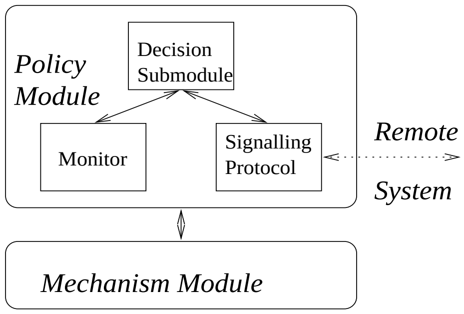

Figure 1: Substructure of a policy module

3.1 Measurements

When designing a monitor submodule, it is necessary to identify a set of relevant measurable parameters that will be used to detect a need for protocol adaptation. These parameters can range from something as simple as requests (*i.e.*, messages) from the user or other policy modules to measurable quantities such as checksum errors, buffer utilization, signal-to-noise ratio (SNR) etc. Measurements can be taken at different layers of a protocol stack and at multiple points in the network (*i.e.*, measurements may be distributed).

3.2 Decisions

The second task involves making decisions based on measurements and/or requests. Collected information which can be of any nature is processed and transformed into a result that represents a decision. The decision can be to insert/remove a new functional element or to remain in the current state. In other words, a decision submodule maps the output of a monitor to a three element set: { *insert*, *remove*, *continue* }. A decision submodule separates mechanism-specific parts from the rest of a policy module. A decision submodule must be aware of the time needed to instantiate the new functional module. In the Protocol Boosters model, this time may include module instantiation time for boosters implemented as kernel loadable modules[16], device downloading time for FPGA implemented boosters[8], propagation time over a high latency network, etc. If the event that would trigger the modification of a protocol graph is shorter than the mechanism instantiation time, switching the module in would result in a mismatch between current network conditions and protocol instantiation. This is obviously undesirable, and the policy module should avoid reacting to such events. In other words, some amount of *inertia* must be built into a decision module.

3.3 Synchronization

In general, functions inserted into a protocol stack are distributed over the network, and thus some form of signalling is necessary. The role of a signalling protocol is twofold. Firstly, it must ensure that distributed functions are activated synchronously to avoid mismatched protocol stacks at transmitter and receiver.

The second function of a signalling protocol is to allow distributed decisions. In the FEC example presented in the next section, only the receiver has the information about bit errors and therefore a monitor submodule must be located at the receiver side. However, the transmitter must mark the point at which the coding starts so that the receiver can start decoding at the correct point in time. Placing the decision submodule at the transmitter side is impractical because it would be necessary to constantly transfer measurements across the network. A better solution is to place the decision submodule at the receiver and use the signalling protocol to transfer decisions across the network. Decision information may be sent only when the protocol state needs to be changed, which will result in much less overhead traffic.

4 Case Study: FEC Booster

In this section we illustrate a policy module design using the example of an FEC booster described in [9].

4.1 Platform

Since a large group of protocol processing functions are bit-oriented (*e.g.*, checksum calculation, FEC, encryption, *etc.*), potential exists for speedups of orders of magnitude if these functions are implemented in hardware. Many other protocol processing functions are packet-oriented (*e.g.*, ARQ, flow and congestion control etc.) and thus amenable to software implementation. A dynamically reconfigurable hardware platform, the Programmable Protocol Processing Pipeline (P4)[8] has been designed to implement protocol functions in hardware while maintaining software-like flexibility.

The architecture of the P4 is shown in Figure 2. The P4 organizes a set of RAM based field-programmable gate arrays (FPGAs) in a pipeline, with a switching array selecting which devices are engaged in processing a data stream. FPGAs allow protocol processing with run-time reconfigurable hardware. Packets are received by the input interface (IIF) processed by the system and sent back into the network by the output interface (OIF).

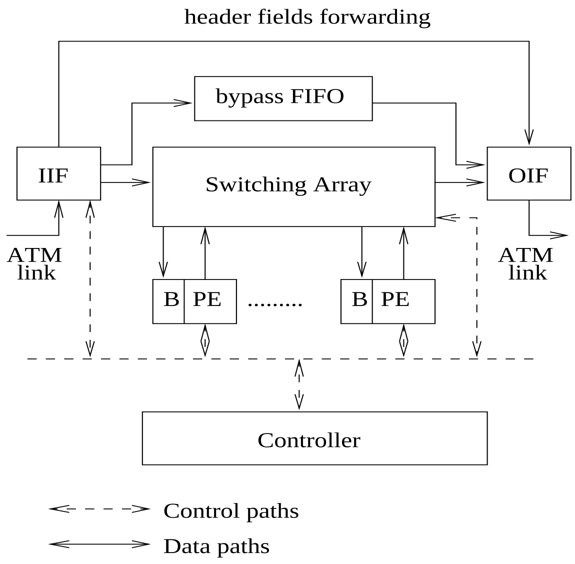

Figure 2: P4 architecture

Each device (processing element, PE) has a FIFO buffer (B) associated with it. A processing element reads the data from its FIFO buffer, performs its processing, and writes into the FIFO buffer associated with the next device in the chain. Connection to the next device is achieved via the switching array. The switching array can dynamically include or exclude processing elements, or reorder them on an as-needed basis.

When needed, a protocol processing function (in the form of an FPGA configuration) is added by downloading a free device, and inserting this device into the pipeline chain. Unnecessary functions are switched out of the processing chain and the device becomes free.

The P4 prototype uses ATM cells as a convenient unit of processing. While the architecture is *not* ATM-specific, ATM simplifies the hardware implementation of processing algorithms and allows validation of performance in 100+ Mbps operating regimes.

In the software running on the bus-attached controller[1] each mechanism has the associated thread that executes the protocol described in Section 4.5 and inserts or removes the associated mechanism module.

4.2 A Forward Error Correction Mechanism

Forward error correction (FEC) is an example of a protocol processing function that might be dynamically inserted or removed from a protocol stack. An FEC coding algorithm can thus be viewed as a mechanism module for an FEC protocol booster. Experiments showed[9] that adding the FEC "as-needed", improves TCP throughput. An updated version of the experimental results[2] is given in Figure 3.

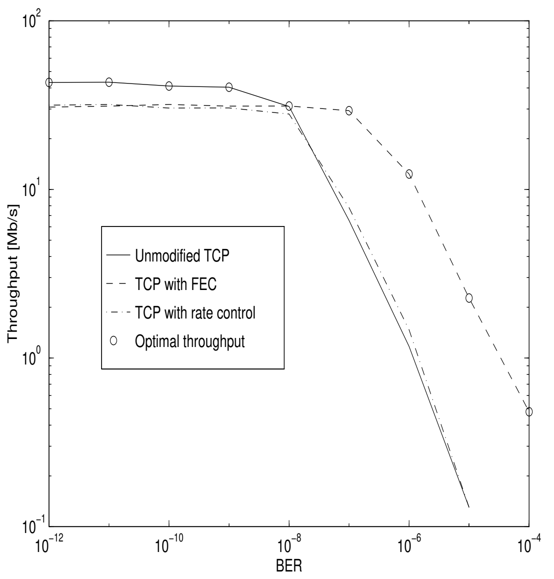

Figure 3: Mean value of measured TCP throughput

The graph shows the result for the R=1/2, constraint length 3, convolutional code. Similar results have been observed for the H(7,4) block code applied to every group of four bits in the packet[3].

The solid line presents the throughput without the P4 or with idle P4 in the datapath. The dash-and-dotted line shows the TCP throughput with an inactive P4 and rate control[4] at the transmitter.

Finally, the dashed line shows the measured throughput with P4 running the FEC booster. It can be observed that the major overhead of the FEC booster is in the rate control. Because the FEC is implemented in hardware, encoding and decoding do not impose substantial overhead (*i.e.* dashed and dashed-and-dotted lines overlap for low bit error rate).

A nearly exponential drop-off in throughput for bit error rates above 10^{-8} occurs due to TCP's congestion control scheme. This is consistent with predictions made by the theoretical model derived in [10]. Adding FEC to the protocol on an as-needed basis improves TCP throughput in a noisy environment and does not modify the protocol stack when the bit error rate is low. On the other hand, FEC requires extra bandwidth which increases the probability of congestion. The congestion probability dominates for low bit error rates while the bit error probability dominates in the high bit error rate region.

The goal of a policy module associated with the FEC booster is to insert the mechanism (*i.e.* the encoder and decoder) only where a reduction in bit errors from the code actually pays off. For the case shown in Figure 3, this is the region where bit error rate is greater than 10^{-8}.

A generalization of this scheme uses multiple FEC modules of different strength and coding schemes. Each FEC scheme would perform best for a range of bit error rates. The role of a policy module would be to match the BER range to the FEC scheme.

4.3 BER Monitor

In our FEC booster implementation, the FPGA implemented bit error monitor (BER monitor) measures the number of corrupted packets inside a time window W. For each packet, a moving average sum is calculated (recursive version is used in implementation):

$$s[k] = \sum_{i=0}^{W-1} r[k-i] = s[k-1] + r[k] - r[k-W] \quad (1)$$

where $s[k]$ is the output for the k-th packet, W is the size of a moving average window, and $r[k]$ indicates if the packet is in error ($r[k] = 1$ for packet in error and 0 otherwise). Clearly, if we could identify bad packets, the output of a BER monitor would be proportional to packet error rate.

Determining if the packet has incurred an error is specific to the underlying network infrastructure. In non-boosted mode, we perform the CRC check of the AAL-5

[1] In our experiments we use a PC running Linux as a controller. If this was a production system, a PC would be replaced with an embedded control system

[2] These results show remarkably close consistency with the model developed in [10], and thus provides experimental confirmation of those results.

[3] For background on error correcting codes, see, *e.g.*, [13]

[4] Rate control is necessary to reduce the bandwidth utilized by a transmitter so that the P4 can insert packets generated by the encoder. Rate control should theoretically introduce an idle period of *at least* one packet length per each transmitted packet. In our experiments, rate control introduced a throughput reduction by roughly a factor of 1.5 (measured throughput was around 45Mb/s without rate control and around 30Mb/s with rate control). Since the workstation we used can not fully utilize the OC-3 pipe, a rate control factor less than 2 was acceptable.

unit in which the packet is encapsulated. If the network is using early packet discard (EPD)[19], AAL-5 CRC is the perfect indicator of bit errors since packets are entirely dropped in the case of network congestion and almost all CRC errors will be due to bit errors rather than cell losses. Without EPD, CRC error can occur in the case of a single cell loss due to network congestion. In this case the BER monitor would make an assumption contrary to the one made by TCP (TCP assumes that all errors are due to congestion, while BER monitor assumes that all errors are due to bit errors).

When the FEC booster is on, using CRC check after decoding, is an inaccurate criteria for determining if the packet has encountered an error. Since many bit errors are now corrected, passing the CRC check still does not necessarily imply that the bit error rate is sufficiently low to remove the FEC booster. A better indicator is the decoder itself. Instead of returning a moving average of CRC failures, the BER monitor in boosted mode returns a moving average of the number of packets which have been corrected by the decoder. In the case of a convolutional code (*i.e.* Viterbi decoder on the receiver), a non-zero accumulated Hamming distance for the selected path is a good indicator that the packet has incurred an error[13]. For a block code, appearance of non-zero syndromes can be used to detect packets with bit errors.

4.4 Decision Submodule

Our policy module uses *event counting* to make its decisions. Mapping measurements to a set of decisions[5] is based on crossing certain *thresholds* in the number of events. This principle is not limited to FEC and it can be used in policy modules that control other types of boosters. In all cases a fundamental design question is how to determine thresholds at which a policy module decides to activate or deactivate the booster. In this section we describe the event counting decision submodule and in Section 5 we develop and experimentally verify a model which can be used to determine decision thresholds for the FEC booster.

The decision submodule used in our implementation is a software implemented error counter which polls the BER monitor and reads $s[k]$. Alternatively, it can be viewed as a state machine (see Figure 4). Initially, the counter is set to zero (state machine is in its leftmost state) and the module returns *remove*. If the value read from the BER monitor is greater than zero, the counter increments by this value (state machine advances to the right). If the counter reaches its maximum (rightmost state), *insert* is returned. Counter may not count above its maximum nor below zero. Reading zero from the BER monitor, will decrement the counter by 1 (move the state to the left). When the counter reaches zero, *remove* is returned. In all intermediate states of a counter, the decision submodule returns *continue* meaning that the current state should not be changed.

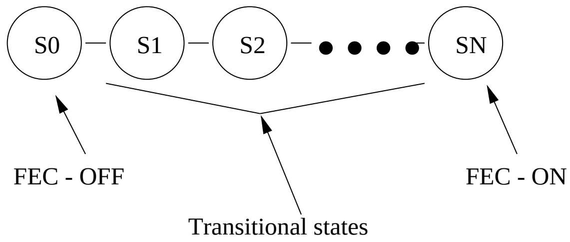

Figure 4: State machine representation of a decision submodule

4.5 Signalling Protocol

We have implemented a signalling protocol which allows a policy module located at one point of the network to inform remote policy modules about its decisions and synchronously perform the transition to boosted mode. We briefly describe the implementation of our signalling protocol, of which details can be found in [7].

Each mechanism module known to the system has an associated control block and a supervisory thread which executes the protocol described later in this section. Driven by timeouts or signalling message arrivals, it periodically wakes up and runs different decision submodule depending wether the mechanism is engaged of not.

The decision submodule returns one of the decision codes: `BOOSTER_ON` (insert the booster), `BOOSTER_READY` (remain in the current state), `BOOSTER_OFF` (remove the booster).

Depending on a policy decision or the semantics of a received signalling message, the booster updates its state and if necessary sends a signalling message.

If a policy module determines that the booster should remain in current state (`BOOSTER_READY`), no changes to the protocol stack will be made. However, the system will check if the devices on P4 are configured and if necessary download the FPGA configurations from the list in the control block. The purpose of this check is to ensure that all devices are configured in a timely manner so that the booster insertion is not stalled by slow device downloading.

A supervisory thread is a finite state machine which is different for booster and debooster. Its role in the system is determined by a field in the control block. The state transition diagram for a booster is shown in Figure 5.

When the booster is not engaged, the supervisory thread will be in `IDLE` state and no messages

[5] See the set defined in Section 3.2. Later, when we speak about the implementation, we will refer to these elements by names used in our source code: `BOOSTER_ON`, `BOOSTER_OFF`, `BOOSTER_READY`.

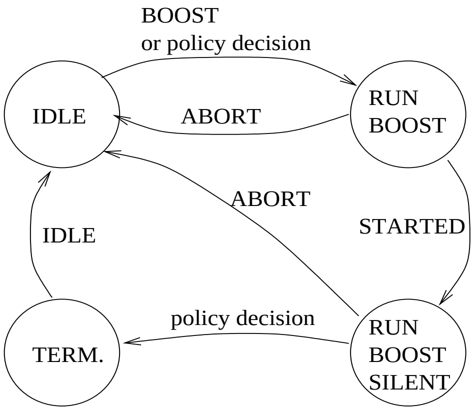

Figure 5: Booster state transition diagram

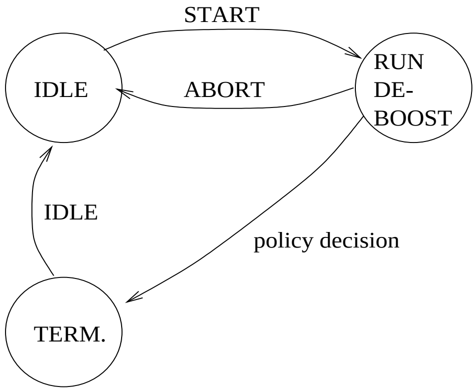

Figure 6: Debooster state transition diagram

will be exchanged. If the policy module anticipates that the booster might be needed soon (*i.e.*, returns `BOOSTER_READY`), devices on P4 board will be downloaded, but the booster will not be switched in and the state will be unchanged. Also a `PREPARE` message will be sent, to make the remote party aware of a local policy decision. Transition to boosted mode (`RUN_BOOST` state) can occur either as a result of the policy decision or on demand via a signalling message (*i.e.* a remote policy decision).

In the `RUN_BOOST` state, booster is active and begins sending `START` messages to force the remote side to go into boosted mode. Once the remote side acknowledges the transition, there is no need to send further signalling messages and the booster goes into `RUN_BOOST_SILENT` mode. A booster is removed if the policy module decides so (local policy decision) or if the `ABORT` message is received (remote policy decision). In the former case, booster must remain in `TERMINATING` state, and keep sending `ABORT` messages to force the remote side to remove the debooster. Once the removal is acknowledged, both sides will be in `IDLE` state and the booster will not be engaged.

The debooster state transition diagram is shown in Figure 6.

In the `IDLE` state, debooster also runs the policy module which may decide that boosting is necessary. However, this will only result in sending the `BOOST` message to the remote side (booster), and the transition will not occur until `START` message is received.

Receiving a `START` message will result in the transition to `RUN_DEBOOST` state and `STARTED` message will be sent in response. When engaged, a debooster can be removed due to a local or remote policy decision (*i.e.* receiving the `ABORT` message). If the policy decision was local, the thread will go into `TERMINATING` state where it will periodically send `ABORT` message to inform the remote party of its decision. A booster will acknowledge the transition with `IDLE` message and booster and debooster will be removed.

5 Tunable Parameters

Our policy module can be tuned to a particular error model with three parameters: moving average window size in packets (W), counter maximum (N) in error events, and the polling period (T) in seconds. In this section, we develop models that can be used to determine these parameters given the network conditions. Our analysis assumes that bit errors are independent random events (*i.e.* Bernoulli process) and in our experiments bit errors are emulated using this model. Though we recognize that different models should be used for different types of links (*e.g.*, terrestrial wireless link is modeled differently from a satellite link), the error model discussion is out of the scope of this paper.

5.1 Moving Average Window Size, W

Moving average window size, determines the bit error probability for which the FEC will be turned on or off (*i.e, critical BER*). To statistically guarantee booster insertion, the expectation $E\{s[k]\}$ must be greater than $1/2$. From the equation 1 and linearity of an $E\{.\}$ operator, the requirement can be written as:

$$E\{s[k]\} = \sum_{i=0}^{W-1} E\{r[k-i]\} > \frac{1}{2} \tag{2}$$

By definition, $r[k]$ is a Bernoulli random variable and its expectation equals the probability of a packet error at time k. If the bit error rate changes slowly, we can assume that the error probabilities for all $r[k]$ inside the moving average window are almost equally distributed. Given this assumption and the packet length of L bits, condition (2) becomes:

$$W > \frac{1}{2 \cdot L \cdot P_{bit}} \tag{3}$$

The inequality (3) allows us to determine the W so that the transition to boosted mode is guaranteed when the bit error rate reaches P_{bit}. Similar inequality, with the opposite sign, can be derived for the window size that guarantees the transition from boosted to non-boosted mode.

We have experimentally verified this result by varying the window size W and observing behavior of the policy module. Results for 1500 bytes packet size are plotted in the Figure 7. It is important to note that when the system is in boosted mode, the effective packet size is 3000 bytes, because the FEC encoder adds one extra cell to each original cell: redundancy needed for bit error corrections.

The figure compares the predicted critical bit error rates with bit error rates for which the actual booster insertion/removal has occurred (*i.e*, measured critical BER). The left graph shows the probability of error above which the FEC booster is inserted into the protocol stack as the function of W. The right graph shows the critical BER for booster removal.

It is important to note that the equation (3) specifies when the booster insertion is *guaranteed*. It is still possible that the actual insertion occurs earlier which is the reason for the differences between measured and predicted critical BER. However, a good match in the order of magnitude, validates the linear model of (3).

From the Figure 3, it can be observed that, in our example, the critical point is at BER=10^{-8}. It is also desirable to provide a hysteresis to avoid unnecessary switching between boosted and non-boosted mode when the system is operating near the critical point. Small fluctuations in bit error rate around the critical point will be damped by the hysteresis. Given the above result and assuming a packet size of 1500 bytes (3000 in boosted mode), we have chosen the window sizes of 512 for non-boosted mode and 768 for boosted mode which results in the hysteresis loop placed between[6] $3 \cdot 10^{-8}$ and $4 \cdot 10^{-8}$.

5.2 Counter Maximum, N

Counter maximum (N) provides the inertia of the system and can be determined through a trade-off between the system speed and robustness. Larger N prevents switching the booster in or out when the average bit error rate is near critical with occasional short intervals between errored packets. On the other hand, system reaction to bit error rate changes will be slower for larger N. We show this trade off in the analysis to follow.

The policy module can be modeled as a Markov chain shown in the Figure 8. The top subchain models transitions from non-boosted to boosted mode, while the opposite transition is modeled with the bottom subchain. States $0 = 0'$ and $N = N'$ represent bounds of the counter. Two subchains are necessary because of different transition probabilities in boosted and non-boosted mode.

Transition probabilities $P(s)$ and $P'(s)$ represent the probabilities that the BER monitor will read s for non-boosted and boosted mode respectively. They are defined as:

$$P(s) = p(s)\Big|_{W=512, L=1500\cdot 8}$$

$$P'(s) = p(s)\Big|_{W=768, L=3000\cdot 8} \tag{4}$$

$$p(s) = \binom{W}{s} (P_{bit} \cdot L)^s \cdot (1 - P_{bit} \cdot L)^{W-s}$$

Since all the parameters necessary to fully describe a Markov chain have been specified, it is a straight forward mathematical task to find the steady state solution vector $[q_i]$, where q_i is the probability of being in state i. Of special interest for this discussion is the behavior of probabilities of being in state 0 and probability of being in state N. We call these probabilities q_{remove} and q_{insert} respectively, as they represent probabilities of removing and inserting a booster into the protocol stack. Figure 9 shows the behavior of q_{remove} (dashed line) and q_{insert} (solid line) for different values of N and error probability around critical point. As expected for error probability well below critical point q_{remove} dominates while q_{insert} is negligible and vice-versa. There is also the uncertainty region in the vicinity of critical point where q_{remove} and q_{insert} are comparable. In this area the system is most likely to oscillate between boosted and non-boosted mode. It can be observed from Figure 9 that for larger N uncertainty region becomes narrower with the minor shift of the critical point to the left. A desirable property is to have the uncertainty region as narrow as possible, and the dominant effect of the counter maximum value N is to control the width of this region.

On the other hand, large counter maximum value will make the system inert as more time will be needed to make the transition. As discussed earlier in the paper, some amount of inertia is necessary to filter out short events that are likely to cause the system oscillation. However, too much inertia will make the system too slow and filter out some events that are otherwise addressable

[6] The left boundary of the loop should actually be below 10^{-8}, but this would require at least three times larger moving average window. Since the size of a shift register needed to implement a moving average equals the size of a window, three times more memory elements would be needed. Due to a limited size of the FPGA devices we used and the amount of FPGA space needed by a Viterbi decoder, 768 was the largest implementable window size

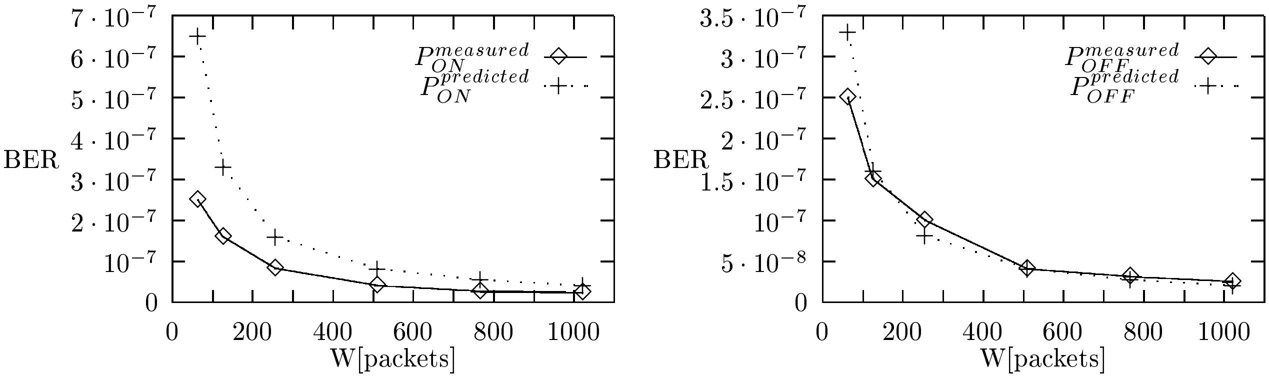

Figure 7: Predicted and measured critical bit error rates

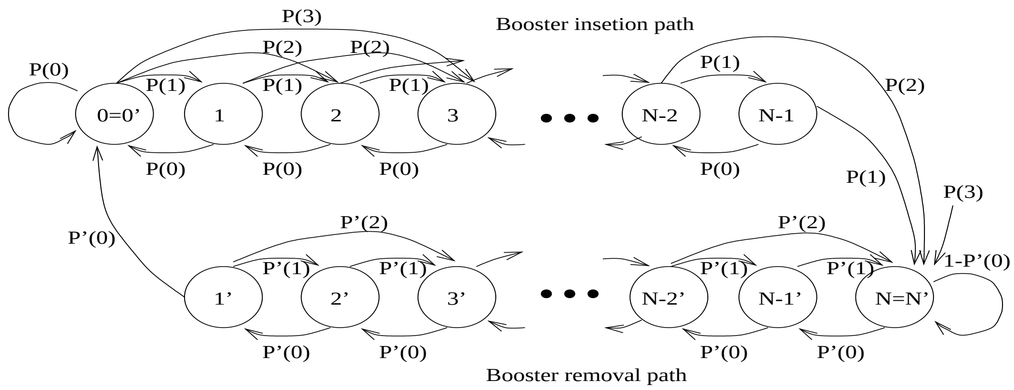

Figure 8: Markov model of a decision submodule

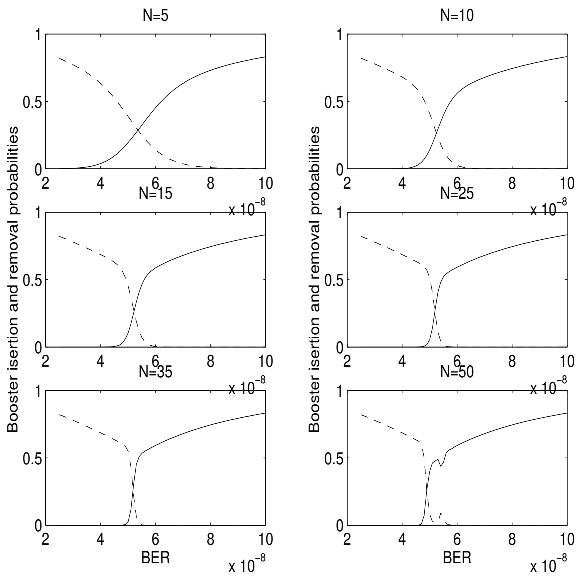

Figure 9: Steady state solutions: Probabilities of switching the booster in (solid line) and out (dashed line) as function of BER

by the booster. In the Figure 10 we have plotted q_{remove} and q_{insert} as the function of time and error probability fixed to 10^{-7} (BER for which it is clear that the booster should be switched in). The plots are shown for several values of N and the influence of N to inertia is obvious.

In our example we have chosen $N = 25$ as a trade-off between inertia and width of the uncertainty region.

5.3 Polling Period, T

Polling period (T) determines how often a processor reads data from the BER monitor and runs the decision submodule. The system will operate faster with shorter polling period. Also the assumption about slowly changing bit error probability favors small T. On the other hand, too short polling interval may result in overlapping moving average windows, which can have further impact on the accuracy of models (*i.e.*, Markov chain model) we used to determine other tunable parameters. A model that accurately describes the role of the polling interval remains yet unclear and will be developed in our future work. However, in our experiments, we observed higher probability of booster oscillation with shorter T suggesting that there is a lower limit on the polling interval. For OC-3 bit rate, we have empirically found that for $T = 150ms$ probability of oscillation is suf-

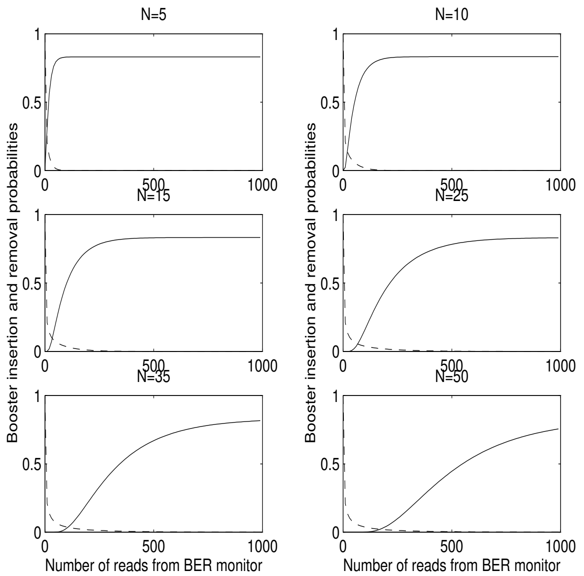

Figure 10: Transitional behavior: Probabilities of switching the booster in (solid line) and out (dashed line) as a function of time

ficiently small and that our models still give accurate predictions.

6 Results and Discussion

In this section we begin by defining some performance measures which allow us to segue from the analysis to measurements of the implementation. We then describe the testbed, and report the performance results.

6.1 Performance Measures

In this section we define measurable quantities which can be used to characterize the performance of an FEC booster mechanism module. Due to the instantiation latency discussed in Section 3.2, a policy module takes a finite time to adapt the protocol graph to new network conditions. During this transitional period, the protocol will not be matched to current network conditions, and suboptimal performance will be observed.

We are interested in how far the measured throughput deviates from the ideal. We define relative throughput error:

$$\bar{e} = \begin{cases} \frac{\bar{\xi}-\bar{\eta}}{\bar{\xi}} & \bar{\eta} > \bar{\xi} \\ 0 & otherwise \end{cases} \tag{5}$$

where $\bar{\eta}$ is the measured TCP throughput with our policy module controlling booster insertion and removal. Relative throughput error compares the measured throughput with the optimal:

$$\bar{\xi} = \frac{1}{T_c}\int_0^{T_c} max[\xi(t)]dt \tag{6}$$

where T_c is the duration of the connection.

Ideal throughput, $\bar{\xi}$, would be achieved assuming an ideal policy module that switches the booster in and out infinitely fast and an ideal TCP protocol which opens up the congestion window infinitely fast when the bit error rate improves. This is the maximum possible throughput, achievable only under the ideal and non-realistic circumstances. However, since the basis for the calculation of $\bar{\xi}$ is the experimental curve of Figure 3, measurement errors can potentially result in small negative value of $\bar{e}$. The lower branch of equation (5) will round negative errors up to zero.

Ideally, $\bar{e}$ would be zero; smaller values indicate better performance of the policy module. Sources of error are the finite time to switch between boosted and non-boosted mode, the inability to track very fast changes in the BER, errors in BER estimation, and the short outages possible when the signalling cells are lost or corrupted.

A second performance indicator, *boost factor*, is defined as:

$$\bar{b} = \begin{cases} \frac{\bar{\eta}-\eta_{static}}{\eta_{static}} & \bar{\eta} > \eta_{static} \\ 0 & otherwise \end{cases} \tag{7}$$

where $\bar{\eta}$ is the same measured throughput from equation 5. The boost factor compares the measured throughput of the system with the policy module to the measured throughput of a static implemented system.

The optimal throughput (ξ) represents the upper limit for our policy module, while the η_{static} represents the baseline if the policy module was not operating. Two baselines exist: (1) the throughput of unmodified TCP; and (2) the throughput of FEC augmented TCP without a policy module to dynamically insert and remove the FEC. The measured boost factor compared with both bottom lines and the relative throughput error is a good indicator of how well the policy module performs its task.

6.2 Experimental Measurement Setup

The experimental setup is shown in Figure 11. The host is a 300 Mhz Pentium II processor with 64 MB of 60ns DRAM, 16KB of L1 cache, 256 KB of L2 cache and a 33Mhz PCI bus. It operates Linux 2.0.29 with support for ATM[1] and uses a Fore Systems PCA200E ATM adaptor[5] for the networking experiments. The device driver was slightly modified to support the rate control scheme mentioned in Section 4.2.

The logical ATM link operates at the SONET OC3c rate of 155 Mps. In reality, the logical link consists of several subsegments, starting from the transmit port of the adaptor:

1. link to the Fore ASX-200 ATM switch
2. link to "encoder" P4 and link back to the switch
3. link to the Network Impairment Emulator[18] (it is a VXI card in an HP 75000 Broadband Network Analyzer and is used to emulate a noisy line by inserting bit errors) and link back to the switch
4. link to "decoder" P4 and link back to the switch
5. link from the switch to the workstation adapter

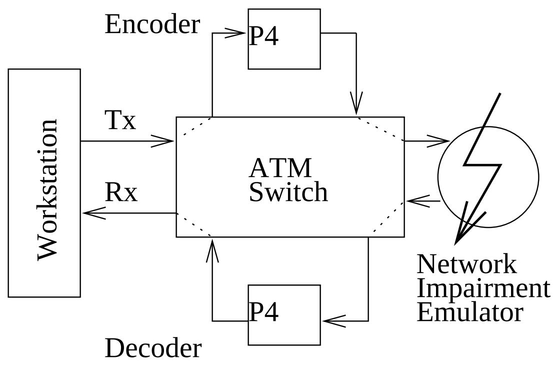

Figure 11: Experimental setup used in testing the FEC booster

Throughput testing is done with `ttcp`. For convenience, we used a single test machine with source and sink running as two separate processes. The performance impact on TCP/IP throughput of the policy module in the P4 is independent of the PC's performance except for scaling.

6.3 Measured Results

We tested the performance of our policy module using the `ttcp` throughput testing utility with very long data streams (`-n 100000`). During the connection the network impairment emulator kept the link error free for time T_0 and then switched to error mode and kept it for time T_0 after which the process repeated. The total duration of a connection (T_c) was always large compared to T_0 resulting in many transitions from error free to error mode.

The BER in the errored intervals was fixed throughout the `ttcp` measurement session; multiple measurements were made for each BER. Conveniently, for this test scenario, the integral of (6) depends on neither T_0 nor T_c and can be written as:

$$\bar{\xi} = \frac{R_{good} + R_{bad}}{2} \quad (8)$$

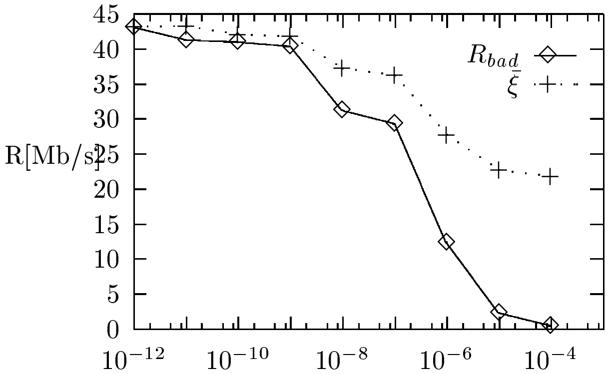

Figure 12: Ideal TCP throughput: Calculated from measured data under the assumption that the policy and TCP react infinitely fast

where R_{good} is the throughput for the error free connection and no FEC in place and R_{bad} is the throughput in the error mode assuming that the FEC will be inserted only for bit error rates above critical (see Figure 3). We have measured R_{good} and R_{bad} for a wide range of bit error rates. R_{good} is approximately 43.9Mb/s. Results for R_{bad} and the calculated $\bar{\xi}$ are plotted in Figure 12.

We ran the `ttcp` tests using our policy module to control the FEC booster, and compared measured throughput against the ideal. Figure 13 shows the relative error (equation (5)) as the function of bit error rate for different values of T_0. We see that if the bit error rate changes sufficiently slowly, the relative throughput error remains below 20% keeping the performance of the policy module reasonably close to optimal.

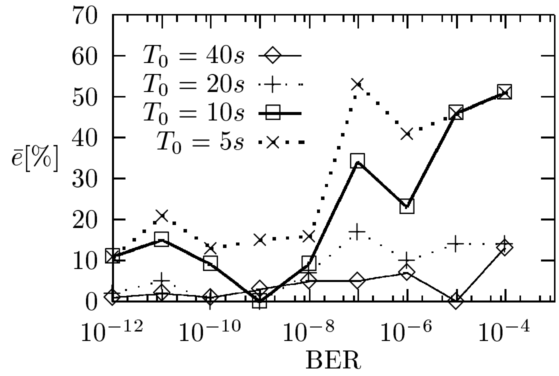

Figure 13: Relative throughput error

For smaller values of T_0, the interval where the protocol instantiation is mismatched to network conditions is longer and the throughput error is larger. Factors that drive the measured throughput away from optimal are the finite time to instantiate the booster (*i.e.*, inertia of the decision module), finite time needed for the TCP to open up the congestion window when the BER improves, and bit errors in signalling messages that could cause a temporary loss of synchronization between the coder and decoder.

To estimate how much improvement in TCP throughput is due to the protocol adaptation we measured the TCP throughput of the system with the policy module and calculated the boost factor defined by (7). Figure 14 shows the boost factor with respect to the throughput of unmodified TCP protocol stack (*i.e.*, the reference throughput η_{static} was measured without FEC).

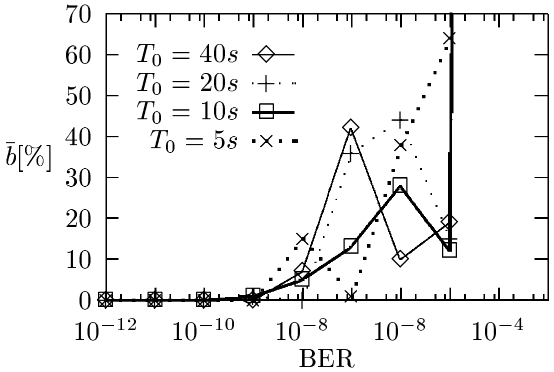

Figure 14: Boost factor with respect to the throughput of unmodified TCP protocol stack

For low bit error rates, improvement is near or equal zero, because the booster is never engaged and the system acts as unmodified TCP protocol. The improvement rapidly grows with the bit error rate and becomes practically infinite for $BER > 10^{-4}$. In the BER region above 10^{-4}, the unmodified TCP closes its congestion window and almost completely stalls. The values of T_0 for which we performed our experiments (up to 40s) were insufficient to allow recovery when the link returned to an error free state.

Figure 15 shows the boost factor with respect to the throughput of a TCP protocol stack with FEC running on the P4. To determine the reference throughput for this case, we manually forced FEC booster insertion before the test session and ran the throughput test without removing the booster.

Improvement occurs for both low and high bit error rates. In the low BER regime the overhead of statically instantiated FEC booster prevents the TCP from fully opening the congestion window and the improvement of about 30% is due to rate control at the transmitter. For high bit error rates, the FEC improves the TCP throughput when the link is in the error mode, but does not take the advantage of error-free periods. For fast changing bit error rates (*e.g.*, $T_0 = 5s$), there is almost no throughput improvement in the high BER region. Here, the inertia of the decision submodule prevents the booster from being removed during short fluctuations in network conditions.

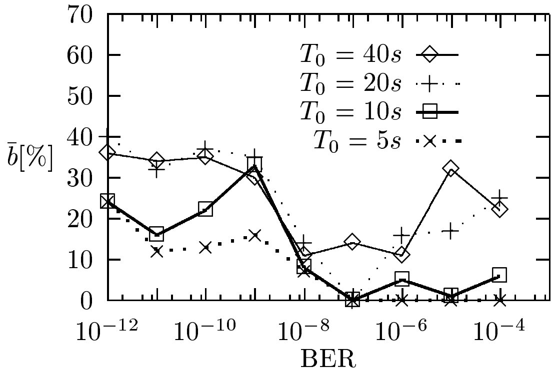

Figure 15: Boost factor with respect to the throughput of TCP protocol stack with FEC on P4

7 Conclusions and Future Work

Adaptive protocols, and the approach of run-time protocol construction, such as that of Protocol Boosters, can offer significant performance improvements. However, there are subtleties which must be addressed in a realization, particularly in the design of the policy module. In this paper, we used a particular example, a policy for insertion and removal of an FEC from a high-speed ATM path, to identify issues and provide a case study moving from analysis to policy realization. Our implementation of an FEC Protocol Booster in a set of gate arrays included both mechanism and policy modules, and showed that both the mechanism and the application specific elements of a policy module can be isolated in two small submodules (the *monitor* and *decision* submodules).

We exploited the on/off behavior of mechanisms in the protocol boosters scheme to achieve a novel analysis of policy performance. The analysis predicted that not all of the changes in network conditions can be tracked, due to the time to instantiate a booster and the network propagation delay. This poses a limit to system agility in the face of rapid network dynamics. We used the analytic results to guide a set of experiments, which confirmed the model's predictions about policy, and showed an extensive range of operating conditions under which a policy module can lead to performance increases.

The work conclusively demonstrates that an adaptive protocol architecture can be effectively designed using a policy/mechanism separation. Once the mechanism (FEC) is chosen, analysis of the mechanism in on/off states results in a robust policy module. For appropriate mechanisms and policies, hardware implementation in gate arrays can lead to a system with an attractive tradeoff of performance and flexibility. In particular, TCP/IP throughput improved by up to 50% with increasing BERs (up to 10^{-6}). For larger BERs where TCP/IP becomes dysfunctional (*e.g.*, 10^{-4}) comparison is unfair (but 4500+% does sound impressive!).

There are a number of interesting directions to pursue in the future. First is an attempt to use our hardware for an even more aggressive approach to adaptation in network architectures such as active networks. In this domain, we would try to selectively accelerate active packets (such as ANTS Java capsules[21] or PLAN packets[15]) with the hardware fastpath to achieve an overall performance increase. Second is an attempt to apply the programmable hardware to a domain more likely to be encountered with high-speed networks than high bit-error rates, which is security. IPSec offers a number of opportunities where hardware acceleration could make dramatic improvements in end-to-end performance.

References

[1] ATM on Linux home page and source code. URL `http://lrcwww.epfl.ch/linux-atm/`.

[2] D. C. Feldmeier, *et. al.* Protocol boosters. *IEEE JSAC Special Issue on Protocol Architectures for the 21st Century*, 16(3):437–444, April 1998.

[3] D. L. Tennenhouse, *et al.* A Survey of Active Network Research. *IEEE Communications*, 35(1):80–86, January 1997.

[4] D. Duchamp. Issues in Wireless Mobile Computing. In *Proceedings of the 3rd Workshop on Workstation Operating Systems*, pages 2–10, April 1992.

[5] Linux device driver for Fore PCA200 adaptor, home page and source code. URL `http://os.inf.tu-dresden.de/project/atm/`.

[6] H. Balakrishnan, *et. al.* A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. In *Proceedings of ACM Sigcomm 96 Conference*, pages 256–269, August 1996.

[7] I. Hadžić. Signalling Protocol for P4 (SPP4), version 1.0. Technical Report MS-CIS-98-20, University of Pennsylvania, CIS Department, 1998.

[8] I. Hadžić and J. M. Smith. P4: A Platform for FPGA Implementation of Protocol Boosters. In *Field-Programmable Logic and Applications: 7th International Workshop, FPL'97, Proceedings*, LNCS, 1304, pages 438–447. Springer, September 1997.

[9] I. Hadžić, J. M. Smith, and W. S. Marcus. On-the-fly Programmable Hardware for Networks. In *Proceedings of IEEE Globecom 98*, November 1998.

[10] J. Padhye *et. al.* Modeling tcp throughput: A simple model and its empirical validation. In *Proceedings of ACM Sigcomm 98*, August 1998.

[11] V. Jacobson. Congestion avoidance and control. In *Proceedings of ACM Sigcomm 88 Conference*, August 1988.

[12] R. Katz. Adaptation and Mobility in Wireless Information Systems. *IEEE Personal Communications*, 1(1):6–17, 1994.

[13] S. Lin and D. J. Costello. *Error Control Coding*, chapter 3, 10 and 11. Prentice-Hall, 1st edition, 1983.

[14] M. Satyanarayanan *et. al.* Application-Aware Adaptation for Mobile Computing. *Operating Systems Review*, 29(1), January 1995.

[15] M. W. Hicks, *et. al.* PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998.

[16] W. S. Marcus, A. J. McAuley, and T. Raleigh. Protocol Boosters: A Kernel-Level Implementation. In *Proceedings of IEEE Globecom*, November 1998.

[17] S. O'Malley and L. L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems*, 10(2), May 1992.

[18] R. W. Dmitroca, *et al.* Emulating ATM Network Impairments in the Laboratory. *Hewlett-Packard Journal*, 48(2):45–50, April 1997.

[19] A. Romanow and S. Floyd. Dynamics of TCP Traffic over ATM Networks. In *Computer Communications Review, Proceedings of ACM Sigcomm 94 Conference*, volume 24, pages 79–88, October 1994.

[20] W. S. Marcus *et. al.* Protocol Boosters: Applying Programmability to Network Infrastructures. *IEEE Communications Magazine*, 36(10):79–83, October 1998.

[21] D. J. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Proceedings of IEEE Openarch'98*, April 1998.