Appendix A

Performance Analysis Tools

In this appendix, we describe briefly the software tools that we used for our performance analysis. These include existing tools as well as new ones that we developed.

***sock*:** The *sock* program [101] allows us to create sources and sinks of data that communicate with each other using the TCP and/or UDP protocols. The original *sock* program only allowed sending data in one continuous stream. We have enhanced this program to emulate Web traffic with its short bursts and relatively long pauses. We also added hooks to selectively enable the various performance optimization techniques that we develop in later chapters, which allow us to analyze the benefits of each technique in isolation.

tcpdump: The *tcpdump* program [69,68] is packet capture tool which passively captures a subset of or all packets on a network link. This is very useful for analyzing the behavior of protocols such as TCP. We use *tcpdump* in our experiments to capture packet traces at the sender, receiver, and/or an intermediate point (such as a Web proxy). We wrote a set of scripts to process the *tcpdump* packet traces, extract data and ack sequence number traces for the individual connections, and identify interesting events such as packet loss, reordering, duplicate acknowledgement and retransmission.

tcpstats: We developed a trace tool called *tcpstats* to monitor the evolution of the TCP protocol state. This state includes the congestion window size, the slow start threshold, the retransmission timer setting, etc. Together with the sequence number traces obtained using *tcpdump,* this information enables us to construct a comprehensive picture of the functioning of the protocol in action.

The variables of interest are traced by instrumenting the TCP/IP protocol stack in BSD/OS 3.0 [16] with trace points installed where the variables are modified. To minimize the impact of our instrumentation on the protocol execution, all tracing activity is confined to the kernel, with the traced values (and the corresponding timestamps) stored in an in-kernel buffer. Only after the experiment has concluded are the contents of the trace buffer collected and saved in a file by a user-level program.

linkemu: The *linkemu* tool we developed emulates a wide range of network link characteristics in software (much as tools like *dummynet* [94]). With this tool, we can carefully control a number of link parameters including bandwidth, delay, buffer size, and a variety of scheduling and buffer management schemes (e.g., strict priority scheduling and priority dropping of packets). Using *linkemu* we can experiment with the real TCP/IP protocol code over a variety of emulated network links, ranging from dialup phone lines to geostationary satellite links, and yet have complete control over the traffic that traverses these links.

We implemented *linkemu* in the device-independent part of the BSD/OS driver for Ethernet-class devices (such as 10/100 Base-T Ethernet, WaveLAN, etc.). Our implementation adds two stages of queuing in addition to the existing interface queue. The first is used for bandwidth emulation, i.e., to hold packets for a length of time corresponding to their transmission time on the emulated link. This is also where the scheduling and buffer management policies are implemented. The second stage is a callout queue used to emulate propagation delay.

Note that the bandwidth of the underlying physical channel used for emulation (e.g., Ethernet) imposes a limit on the bandwidth of the emulated link.