Modeling and Analysis of the Unix Communication Subsystems

Yi-Chun Chu and Toby J. Teorey

Electrical Engineering and Computer Science Department
The University of Michigan
Ann Arbor, MI 48109-2122, USA

Abstract

The performance of host communication subsystems is an important research topic in computer networks.[1] Performance metrics such as throughput, delay, and packet loss are important indices to observe the system behavior. Most research in this area is conducted by experimental measurement; far less attention is paid to the analytic modeling approach. The well-known complexity and dynamic nature of the Transmission Control Protocol/Internet Protocol (TCP/IP) make the performance modeling of communication subsystems extremely difficult. The purpose of this study is to analyze and model the overhead in Unix communication subsystems. The overhead is caused by protocol processing as well as kernel functions for fair allocation of system resources. Our approach is to build analytic models of communication overhead for sending and receiving a message. The analytic models can be applied to analyze the communication overhead for Internet information systems, such as the Internet web servers, or software servers built above midwares, such as the Distributed Computing Environment/Remote Procedure Call (DCE/RPC) servers, that require intensive network I/O.

1 Introduction

With recent advances in networking technology, many services that used to reside in a host system are now provided in a distributed fashion. Distributed file service is a good example of this. Providing services across networks increases communication costs: requests and results must be transported between client and server machines. Intensive overhead caused by network I/O limits the capacity of many distributed servers. This makes the study of host communication subsystems an important research topic.

Earlier research closely examined the overhead generated in host communication subsystems [3,5,6]. The overhead is caused by both protocol-specific processing and operating system (OS) activities such as data movement, context switching, and interrupt handling. Careful analysis of the overhead breakdown can improve the design of communication subsystems. However, it cannot reveal how server machines behave under heavy network traffic [10]. This question has received more attention recently because many distributed servers, such as the World-Wide Web (WWW) servers, have generally experienced performance problems in response time and service availability [11,12].

To answer the above question, we need an analytic solution to study the server system behavior under a varied network load. In this paper, we try to develop analytic models of communication overhead for both sending and receiving a message. These models can be applied to estimate the communication overhead in the distributed server systems.

The rest of this paper is organized as follows. In Section 2, we introduce the software architecture of communication subsystems derived from the Berkeley Software Distribution (BSD) Unix. The overhead breakdown is organized according to the software and protocol layers. The queueing delay

[1]This work is funded by IBM Canada Ltd. Laboratory Centre for Advanced Studies, Toronto.

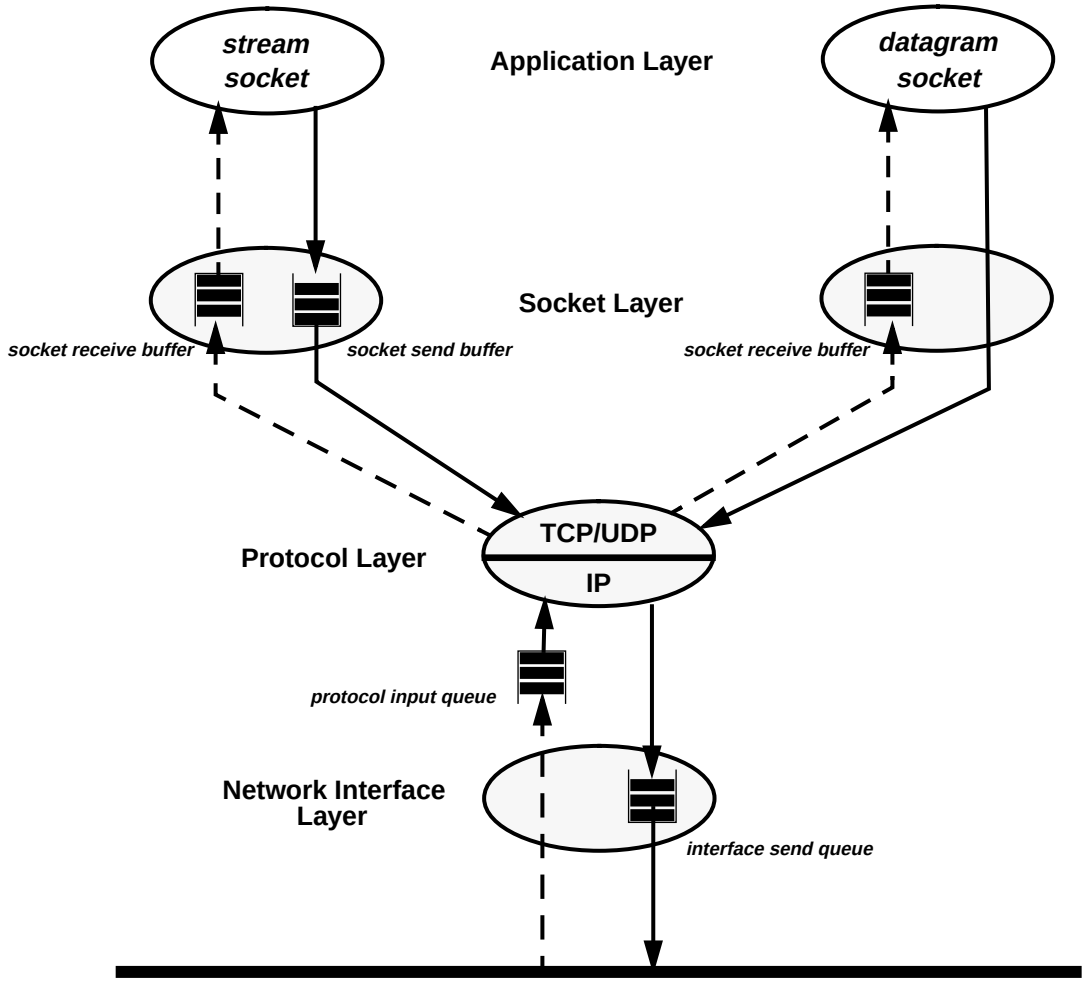

Figure 1. Software Architecture of Unix Communication Subsystems

in communication subsystems is also described here. In Section 3, analytic models for communication overhead are developed, along with a detailed analysis of the critical path for sending and receiving a message. Both transport layer services, TCP and User Datagram Protocol (UDP), are considered in our study. In Section 4 we present two case studies: the Internet web server and the DCE/RPC server with our analytic models. Conclusions and future work are outlined in Section 5.

2 Unix Communication Subsystems

The Unix communication subsystem is divided into three software layers: 1) the socket layer, 2) the protocol layer, and 3) the network-interface layer [8]. The software architecture is shown in Figure 1.

The socket layer hides the complexity of network communication and provides an abstract interface similar to a generic I/O device. *The protocol layer* covers protocol-specific processing in the transport layer (TCP/UDP) and the network layer (IP). *The network-interface layer* is mainly concerned with the link-layer encapsulation/ decapsulation and driving the transmission media. The TCP/IP protocol specification puts no restriction on the layered structure of network software. Most implementations, however, put the code in the kernel with tightly integrated software layers for efficiency considerations [2].

The specific communication subsystem we studied is a DEC Alpha AXP workstation running the OSF/1 1.0 [1]. The workstation is attached to a department LAN with an Ethernet adaptor. The implementation of the OSF/1 network software follows the design of the 4.3 BSD Reno release [8].

2.1 The Processing Overhead

In communication subsystems, processing overhead can be caused by data-touching operations, such as data movement and checksum computation, as well as non-data-touching operations, such as context switching and interrupt handling. Before developing any analytic models to describe the system behavior, a thorough understanding of the overhead is necessary. The major kinds of overheads discovered in communication subsystems are described below.

Data Movement

Data movement is a principal overhead in the communication subsystems. This overhead is significant because memory bandwidth has not kept pace with the speed of microprocessors [3,6]. Generally, two data movements are needed in both sending and receiving paths. First, data has to be copied between user space and kernel space for protection reasons. This work is done by CPU, and we denote them as $M_{uk}(m)$ and $M_{ku}(m)$ separately (m is the size of data to be moved).[1] The other data movement is between kernel space and network adaptor buffer (which is in I/O space). The real work can be done either by CPU (programming I/O) or by Direct Memory Access (DMA). Which does the work depends on the hardware I/O architecture, and it also depends on if the network adaptor has DMA capability. In the system we studied, the data movement from kernel to adaptor is done by PIO, $M_{ka}(m)$; but the data movement in the reverse direction is done by DMA, $M_{ak}(m)$.

Checksum Computation

Network communication relies on checksums to preserve end-to-end data integrity. In the Internet protocols, a 16-bit checksum field is used for error detection in IP header (20 bytes), UDP datagram (12-byte IP pseudo header, 8-byte UDP header and UDP data), and TCP segment (12-byte IP pseudo header, 20-byte TCP header and TCP data) [4,15]. We denote the overhead of checksum computation as $CS(m)$. Both checksum computation and data movement are data-touching operations; the overhead, hence, grows linearly with the data size to be processed.

Protocol-specific Processing

Protocol-specific processing contributes different overhead in each protocol layer. Measurement results show that the overhead tends to be fixed if the checksum computation is not included [6]. Therefore, we can use constants to represent the fixed part of overhead for protocol-specific processing in each layer, and we denote them separately as TCP_{in}, TCP_{out}, UDP_{in}, UDP_{out}, IP_{in}, and IP_{out}.

Demultiplexing

Demultiplexing is a table-lookup operation in the transport layer. It searches protocol control blocks (PCBs) for the socket connection associated with an incoming packet. Most implementation derived from the BSD Unix uses a link-list structure with a one-entry cache[2] (for the latest lookup result) to improve performance [9]. The search cost depends on the number of socket connections in the system. Here we consider it as part of the fixed overhead in transport-layer input routines, TCP_{in} and UDP_{in}. However, it has been shown that the overhead can grow significantly in a busy Internet information server that has peak connections for more than a thousand [10].

Interrupt Handling

Network communication generates two device hardware interrupts: the receiving interrupt and the transmission-complete interrupt. The overhead for interrupt handling receives less attention than other processing overhead. This is probably

[1] Since both $M_{uk}(m)$ and $M_{ku}(m)$ are memory to memory copy, we use $M_{mm}(m)$ to represent $M_{uk}(m)$ or $M_{ku}(m)$.

[2] The one-entry cache is called “1-behind cache.”

Table 1. Overhead Breakdown in Software Layers and Network Adaptor

	Overhead	Description	Device
Socket Layer	$M_{uk}(m)$	data copy from user space to kernel space	CPU
	$M_{ku}(m)$	data copy from kernel space to user space	CPU
Protocol Layer	$CS(m)$	checksum computation	CPU
	TCP_{in}/TCP_{out}	TCP protocol-specific processing	CPU
	UDP_{in}/UDP_{out}	UDP protocol-specific processing	CPU
	IP_{in}/IP_{out}	IP protocol-specific processing	CPU
Network Interface Layer	$M_{ka}(m)$	data copy from kernel space to I/O space (adaptor)	CPU
	$M_{ak}(m)$	data copy from I/O space (adaptor) to kernel space	DMA
	ETH_{out}	link-layer processing	CPU
	I_s	transmit complete interrupt	CPU
	I_r	receive interrupt and link-layer processing	CPU
Network Adaptor	$Tx(m)$	packet transmission time	Adaptor
	$Rx(m)$	packet reception time	Adaptor

because of its asynchronous nature and the difficulty for measuring it. However, careful analysis of interrupt handling in the network device driver reveals how it critically affects the performance during heavy network traffic [14]. In the system we studied, the overhead of the receiving interrupt, denoted as I_r, covers the entire link-layer processing. It does not include the data movement overhead from network adaptor to kernel, $M_{ka}(m)$ (which is done by DMA). The overhead for the transmission-complete interrupt involves data movement from kernel to network adaptor (which is done by PIO) and the initiation of next packet transmission. Hence, it is further divided into a fixed part, denoted as I_s, and a varied part, denoted as $M_{ka}(m)$.

Context Switch

Socket system calls for sending or receiving a message are synchronous. As a consequence, it blocks the current running process and might cause a context switch. Incoming packet processing, which is driven by asynchronous interrupt, will "wakeup" the blocked receiving process at the final stage. Here we denote the fixed overhead for a context switch as C.

Transmission Time

Transmission time depends on the speed of transmission media the workstation is attached to. Transmission speed can vary several orders of magnitude, e.g. from 10 Mb/s (Ethernet) to 622 Mb/s (ATM). Packet transmission and reception are also data-touching operations; we denote the overhead as $Tx(m)$ and $Rx(m)$. Generally, it takes equal time to transmit or receive a packet.

Others

Other kinds of overheads not listed above, such as mbufs allocation, are not significant to our analysis. As a result, we treat them as part of the fixed overhead in protocol-specific processing.

2.2 Overhead Breakdown in Software Layers

The breakdown of processing overhead in communication subsystems is categorized in Table 1. Overhead for data-touching operations and for non-data-touching operations is organized according to the software layers to which it applies.

The overhead breakdown help us see where the overhead is generated. In Section 3, we use this table to develop analytic models of communication overhead by detailed analysis of sending and receiving paths.

2.3 Queueing Delay in Communication Subsystems

The processing overhead described above does not account for the entire delay accumulated in communication subsystems. There is a queueing delay

introduced by buffers or queues within or between the software layers. These queues and buffers are also shown in Figure 1 and described below.

Socket Send Buffer

The socket send buffer holds data not sent yet or sent but not acknowledged by the receiving end. Since UDP does not provide flow control or reliable message delivery, a UDP message is never placed into the socket send buffer.[1] For TCP, the queueing delay is determined by its flow-control algorithms such as the slow start and congestion avoidance.

Socket Receive Buffer

The socket receive buffer is used to hold data received, but not yet delivered to the application. The queueing delay, hence, depends on how quickly the receiving process can accept the data. For TCP, its flow-control algorithm will prevent the sender from sending more data when the buffer is full. For UDP, which has no flow control, any message that arrives when the buffer is full is simply dropped.

Protocol Input Queue (IP Queue)

The protocol input queue holds IP datagrams delivered by the network interface that are waiting for protocol layer input processing. This queue normally will not build up unless there are burst packets arriving at the network adaptor. IP input routine is scheduled as an asynchronous software interrupt in the kernel. This software interrupt is posted by the receiving interrupt handler. It has a lower priority than the device hardware interrupt. IP input routine is usually scheduled immediately to process incoming IP datagrams after the receiving hardware interrupt returns.

Interface Send Queue

Outgoing packets that wait to be transmitted by the network adaptor are placed in the interface send queue. The queueing delay depends on the Medium Access Control (MAC) protocol and the bandwidth of the transmission media.

Developing analytic models to estimate delay accumulated in the communication subsystems is a challenging task. Several factors make it extremely complicated. First, incoming packet processing is divided into two stages in the kernel and scheduled as asynchronous activities with two different priorities. This applies to both TCP and UDP. Second, the dynamics of transport layer processing, such as TCP, is sensitively influenced by its flow-control algorithms. This turns out to be an end-to-end issue that we have to also consider how quickly the remote peer can accept the packet and the end-to-end network latency.

We cannot currently develop analytic models about the delay accumulated in communication subsystems because further study is required to capture the end-to-end dynamics in TCP. For now, we develop a mean value model of the overhead delay to be used as the service demands for queueing models of delay in the future.

3 Analytic Model for Overall Communication Overhead

In Section 2, we introduced the different categories of overhead generated in communication subsystems. In this section, we use them to develop analytic models of the overall overhead for sending and receiving a message. Since TCP has a much richer transport functionality than UDP does, it is impractical to use a single model to describe both of them. Four overhead models, $TCP_{send}(m)$, $TCP_{recv}(m)$, $UDP_{send}(m)$, and $UDP_{recv}(m)$, are built, with m denoting the size of the message to be sent.

3.1 Processing Overhead for Sending and Receiving a Packet in the Bottom Layer

We analyze the bottom layer first because both TCP and UDP employ the same processing steps in this layer. The bottom layer corresponds to the link layer or the MAC sublayer in the Open System Interconnection (OSI) reference model.

[1] Although UDP message is never copied into the socket send buffer, the buffer size restricts the maximum size of UDP messages can be sent.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.