Distributed Network Computing over Local ATM Networks

Mengjou Lin, Jenwei Hsieh, and **David H.C. Du**[2]
Computer Science Department
University of Minnesota

Joseph P. Thomas[1]
Minnesota Supercomputer Center Inc.
Minneapolis, Minnesota

James A. MacDonald[2]
Army High Performance Computing Research Center
University of Minnesota

Abstract

Communication between processors has long been the bottleneck of distributed network computing. However, recent progress in switch-based high-speed Local Area Networks (LANs) may be changing this situation. Asynchronous Transfer Mode (ATM) is one of the most widely-accepted and emerging high-speed network standards which can potentially satisfy the communication needs of distributed network computing. In this paper, we investigate distributed network computing over local ATM networks. We first study the performance characteristics involving end-to-end communication in an environment that includes several types of workstations interconnected via a Fore Systems' ASX-100 ATM Switch. We then compare the communication performance of four different Application Programming Interfaces (APIs). The four APIs were Fore Systems ATM API, BSD socket programming interface, Sun's Remote Procedure Call (RPC), and the Parallel Virtual Machine (PVM) message passing library. Each API represents distributed programming at a different communication protocol layer. We evaluated two popular distributed applications, parallel Matrix Multiplication and parallel Partial Differential Equations, over the local ATM network. The experimental results show that network computing is very promising over local ATM networks.

Keywords: Distributed Network Computing, Asynchronous Transfer Mode (ATM), Application Programming Interface, Performance Measurement.

To appear in IEEE Journal on Selected Areas in Communications Special Issue of ATM LANs: Implementations and Experiences with an Emerging Technology (Early '95).

[1]This work is supported in part by the Advanced Research Projects Agency (ARPA) through the Air Force Systems Command, contract number F19628-92-C-0072. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by either ARPA or the U.S. Air Force.

[2]This work was supported in part by contract number DAAL02-89-C-0038 between the Army Research Office and the University of Minnesota for the Army High Performance Computing Research Center.

1 Introduction

Distributed network computing offers great potential for increasing the amount of computing power and communication resources available to large-scale applications. The distributed environment in which we are interested is a cluster of workstations interconnected by a local area communication network. The combined computational power of a cluster of workstations, connected to a high speed LAN, can be applied to solve a variety of scientific and engineering problems. It is very likely that the combined power of an integrated heterogeneous network of workstations may exceed that of a stand-alone high-performance supercomputer.

Since the early 1970s, computers have been interconnected by networks such as Ethernet, Token Ring, etc. The communication bandwidth of such networks is limited to the tens of megabits per second (Mbits/sec) and the bandwidth is shared by all of the computers. The communication resources required by a collection of cooperating distributed processors has often been the bottleneck for network computing, limiting its potential. Even the higher speed Fiber Distributed Data Interface (FDDI), which provides a bandwidth of 100 Mbits/sec, can be saturated by data traffic between computers. Therefore, one of the design goals for distributed network computing has been to reduce the amount of communication between computers. However, based on the nature of an application, a certain degree of communication between computers may be necessary. Even when the communication channel is not saturated, the relatively long communication delay may degrade overall computing performance.

The recent introduction of Asynchronous Transfer Mode (ATM) may change this situation. ATM is an emerging high-speed network technology which may satisfy the communication needs required in many distributed network computing applications. ATM, proposed by international standards organizations, uses small 53 bytes cells to transmit data in multiples of OC-1 rates (51.84 Mbits/sec). Popular data transfer rates for ATM are OC-3 (155.52 Mbits/sec) and OC-12 (622.08 Mbits/sec) [7, 10]. ATM was initially developed as a standard for wide-area broadband networks. The fact that Local ATM networks are appearing in advance of long-haul ATM networks, makes ATM an attractive alternative to traditional LANs.

ATM networks are characterized by their switch-based network architecture. All computers are connected to a switch and the communication between each pair of computers is established through the switch. This is in contrast to the situation where all computers share one communication medium, as in the case of traditional LANs such as Ethernet. A switched network is capable of supporting multiple connections simultaneously. The aggregate bandwidth of an ATM switch may be several Gbits/sec or more. Within a predesigned limit, the available aggregate bandwidth of the ATM switch increases as the number of ports increases.

In this paper we discuss the performance of distributed network computing over Local ATM networks. Not only do we consider the end-to-end communication latency and achievable bandwidth, but also the computational performance of distributed network applications. The end-to-end communication latency is primarily affected by hardware and software overhead. Hardware overhead includes the host interface overhead, the switch, signal propagation delay, and the bus architecture of the host computer. As hardware technology improves, the impact of this overhead will decrease. Software overhead includes the delay caused by the interactions with the host operating system, device driver, and higher layer protocols. The device driver overhead is mainly caused by the design of the host interface and the bus architecture of the host computer. The overhead of high-level protocols and the delay caused by the interactions with the host operating system can be varied by using different Application Programming Interfaces (APIs) which are available on different communication layers. Several recent papers have found a significant portion of communication overhead occurring due to these interactions [15, 22, 17].

Our primary goal was to study the performance tradeoffs of choosing different APIs in a local ATM environment. In our test environment, several workstations were interconnected via a Fore's ASX-100 ATM Switch. The details of this local ATM environment will be discussed in Section 2. There are at least four possible APIs available:

- Fore's API [13],
- BSD socket programming interface [20, 21],
- Sun's Remote Procedure Call (RPC) [21], and
- the Parallel Virtual Machine (PVM) message passing library [14].

Fore's API provides several capabilities which are not normally available in other APIs, such as guaranteed bandwidth reservation, selection of different ATM Adaptation Layers (AAL), multicasting, and other ATM specific features. The BSD socket interface provides facilities for Interprocess Communication (IPC). It was first introduced in the 4.2BSD Unix operating system. RPC is a popular client/server paradigm for IPC between processes in different computers across a network. It is widely used as a communication mechanism in distributed systems, such as the V kernel [11] and the Amoeba distributed operating system [4]. PVM was developed at Oak Ridge National Laboratory. It is a software package that allows a heterogeneous network of parallel, serial, and vector computers to appear as a single computational resource. PVM was adopted as the communication primitives for the Cray T3D massive parallel supercomputer [8].

For interprocess communication, any of the four APIs can be chosen. However, the performance of the application may be affected by the decision made. Each API may also represent communicating in a different protocol layer. Some further combinations of APIs are also possible. Figure 1 shows the protocol hierarchy of the different APIs. For instance, Sun's RPC uses External Data Representation (XDR) to encapsulate application messages for ensuring architecture independent data format, and sockets for communicating with the underlying transport layers. In the socket interface, applications can choose different transport protocol combinations such as Transmission Control Protocol/Internet Protocol (TCP/IP), User Datagram Protocol/Internet Protocol (UDP/IP), or even raw sockets for interprocess communication. Finally, ATM can use either ATM Adaptation Layer 3/4 or 5 for IP.

When choosing the most fitting combination for a specific distributed application, in addition to communication efficiency, several other factors must also be considered, such as special interface restrictions and ease of use. In this paper, we focus on the communication efficiency.

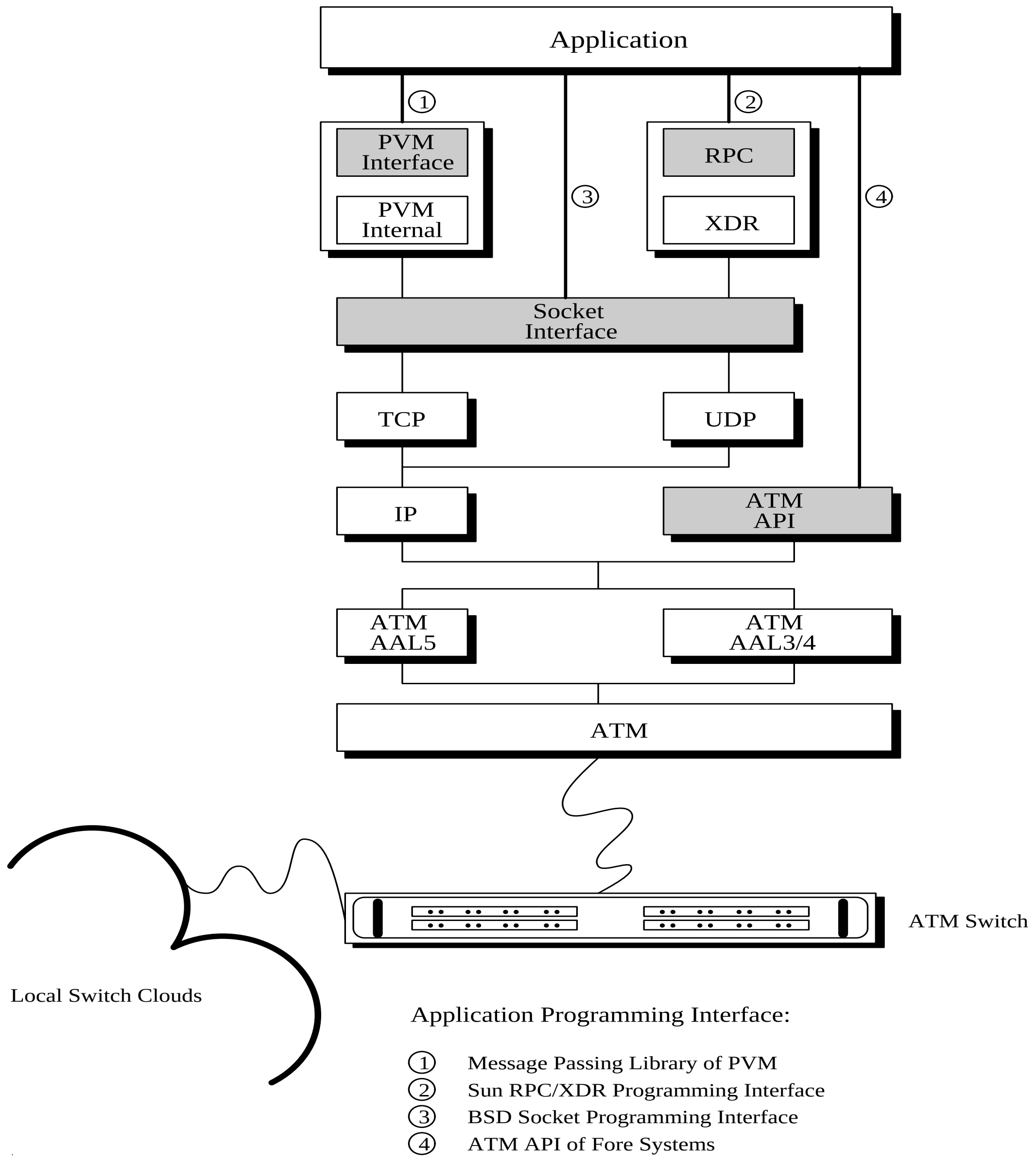

Application
1
2
PVM Interface
PVM Internal
3
RPC
XDR
4
Socket Interface
TCP
UDP
IP
ATM API
ATM AAL5
ATM AAL3/4
ATM
ATM Switch
Local Switch Clouds
Application Programming Interface:
1 Message Passing Library of PVM
2 Sun RPC/XDR Programming Interface
3 BSD Socket Programming Interface
4 ATM API of Fore Systems

Figure 1: Protocol hierarchy

An echo program is used to measure end-to-end communication characteristics (i.e., communication latency for short messages and communication throughput for large messages) and to explore the underlying communication capabilities of different APIs over local ATM, Ethernet, and FDDI networks.

Two well-known distributed applications, parallel Partial Differential Equations (PDE) and parallel Matrix Multiplication (MM) programs, are used to measure performance for different APIs. Parallel MM is a coarse-grain distributed application. It requires data distribution before and after independent computation of each module. It is frequently used in the area of scientific computation. In contrast to parallel MM, parallel PDE is typically a fine-grain distributed application. Within each iteration of the PDE computation, each node exchanges boundary conditions with its four neighbors. It requires more frequent communication than parallel MM. Parallel PDE is used by a diverse set of applications such as fluid modeling, weather forecasting, and supersonic flow modeling.

After studying the performance characteristics of the different APIs, we compared them according to various aspects, such as functionality, user-friendliness, and semantics. Our goal is to provide general programming guidelines that programmers can use in developing distributed applications for local ATM networks. Several related works have studied point-to-point ATM connections in this regard. Wolman [25] discussed the performance of TCP/IP and showed that a point-to-point ATM connection has approximately a twofold performance increase over Ethernet LANs. Thekkath [23] showed that the overhead of a lightweight RPC is only 170 μsec. However, both measurements did not include the overhead incurred within ATM switches. Thekkath [24] also implemented a distributed shared memory system over ATM and reported that it took 37 μsec to perform a remote write operation through an ATM switch. This result is "raw performance". It is not clear what the end-to-end, application-to-application overhead will be.

This paper is organized as follows. In Section 2, we briefly review the ATM standard, describe the hardware and software test environment, and give an overview of each API. In Section 3, the end-to-end communication characteristics are presented for the four APIs in various configurations. The performance of two well-known distributed applications, parallel PDE and parallel MM, are presented in Section 4. Finally, we close with a conclusion and a discussion of future work.

2 Overview of System Environment

In this section, we give an overview of ATM technology, and then discuss the configuration of our experimental hardware and software. Finally, we present a description of the four different APIs.

2.1 Asynchronous Transfer Mode

ATM is a method for transporting information by using fixed-length cells (53 octets) [7, 10]. It is based on virtual circuit-oriented packet (or cell) switching. A cell includes a 5-byte header and a 48-byte information payload. A connection identifier, which consists of virtual circuit identifier (VCI) and virtual path identifier (VPI), is placed in each cell header. The VPI and VCI are used for multiplexing, demultiplexing, and switching the cells through the network. The communication model of ATM is a layered structure. It includes the physical layer, the ATM layer, and the ATM Adaptation Layer (AAL).

- **Physical layer**: The physical layer is a transport method for ATM cells between two ATM-entities. It provides the particular physical interface format (e.g., the Synchronous Optical Network (SONET) or Block Coded Format). SONET defines a standard set of optical interfaces for network transport. It is a hierarchy of optical signals that are multiples of a basic signal rate of 51.84 Mbits/sec called OC-1 (Optical Carrier Level 1). The OC-3 (155.52 Mbits/sec) and OC-12 (622.08 Mbits/sec) have been designated as the customer access rates in B-ISDN. The Block Coded transmission sub-layer is based on the physical layer technology developed for Fiber Channel Standard [3]. Most of the functions of this sub-layer involve generating and processing the overhead and ATM cell header. In case of OC-3, the baud rate is 194.40 Mbaud or a payload rate of 155.52 Mbits/sec of which 149.76 Mbits/sec is available for user data.

- **ATM layer**: The ATM layer performs multiplexing and demultiplexing of cells belonging to different network connections, translation of the VCI and VPI at ATM switches, transmission of cell information payload to and from the AAL, and functions of flow control and traffic policing.

- **ATM Adaptation layer**: The ATM adaptation layer can be further divided into two sub-layers: the segmentation and reassembly sub-layer (SAR) and the convergence sub-layer (CS). The SAR sub-layer performs the segmentation of the typically large data packets from the higher layers into ATM cells at the transmitting end and the inverse operation at the receiving end. The CS is service-dependent and may perform functions like message identification and time/clock recovery according to the specific services. The purpose of the ATM adaptation layer (AAL) is to provide a link between the services required by higher network layers and the generic ATM cells used by the ATM layer. Five service class are being standardized to provide these services. The CCITT recommendation for ATM specifies five AAL protocols [19] which are listed as follows:

  1. AAL Type 1 - provides constant bit rate services, such as traditional voice transmission.
  2. AAL Type 2 - transports variable bit rate video and audio information, and keeps the timing relation between source and destination.
  3. AAL Type 3 - supports connection-oriented data service and signaling.

4. AAL Type 4 - supports connectionless data services (combined with AAL Type 3 now).
5. AAL Type 5 - provides a simple and efficient ATM adaptation layer that can be used for bridged and routed Protocol Data Units (PDU).

Although ATM network technology was originally developed for public telecommunication networks over metropolitan and wide areas, recent interest has focused on applying this technology to interconnect computing resources within a local area [6]. In this paper, we investigate the feasibility of performing network computing over ATM in local area.

2.2 Network Environment

The experiments described were performed over a variety of host and network architectures. Different host architectures tested include the Sparc 1+, Sparc 2, and 4/690, all from Sun Microsystems. Where possible, each architecture was tested with a variety of network interfaces, and in the case of ATM, with and without the presence of a local area switch.

The ATM environment was provided by the MAGIC (Multidimensional Applications and Gigabit Internetwork Consortium) [18] project. Fore Systems, Inc. host adapters and local area switches were used. The host adapters included a Series-100 and Series-200 interface for the Sun SBus. The physical media for both the Series-100 and Series-200 adapters is the 100 Mbits/sec TAXI interface (FDDI fiber plant and signal encoding scheme). The local area switch was a Fore ASX-100.

The SBA-100 interface (Figure 2) [12] was Fore's first-generation host adapter and interfaced to the host at the cell level. The Series-100 adapter is capable of performing the ATM cell header CRC generation/verification and the AAL 3/4 CRC generation/verification. However, the host is responsible for the multiplexing/demultiplexing of VPI/VCI's, the segmentation and reassembly (SAR) of adaptation layers, and any non-AAL 3/4 CRC generation/verification. These tasks are CPU intensive and thus the network throughput becomes bounded by the CPU performance of the host.

In contrast, the Series-200 host adapter (Figure 3) [16] is Fore's second generation interface and uses an Intel i960 as an onboard processor. The i960 takes over most of the AAL and cell related tasks including the SAR functions for AAL 3/4 and AAL 5, and cell multiplexing. With the Series-200 adapter, the host interfaces at the packet level feeds lists of outgoing packets and incoming buffers to the i960. The i960 uses local memory to manage pointers to packets, and uses DMA (Direct Memory Access) to move cells out of and into host memory. Cells are never stored in adapter memory.

The ASX-100 local ATM switch (Figure 4) [12] is based on a 2.4 Gbits/sec (gigabit per second) switch fabric and a RISC control processor. The switch supports four network modules with each module supporting up to 622 Mbits/sec. Modules installed in the MAGIC switches include two four-port 100 Mbits/sec TAXI modules, one two-port DS3 (45 Mbits/sec) module, and one two-port OC-3c (155 Mbits/sec SONET) module. Connections exist to other ASX-

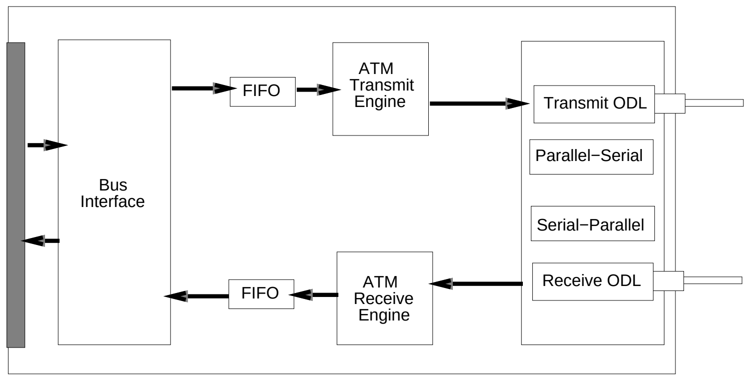


Figure 2: Series-100 host interface

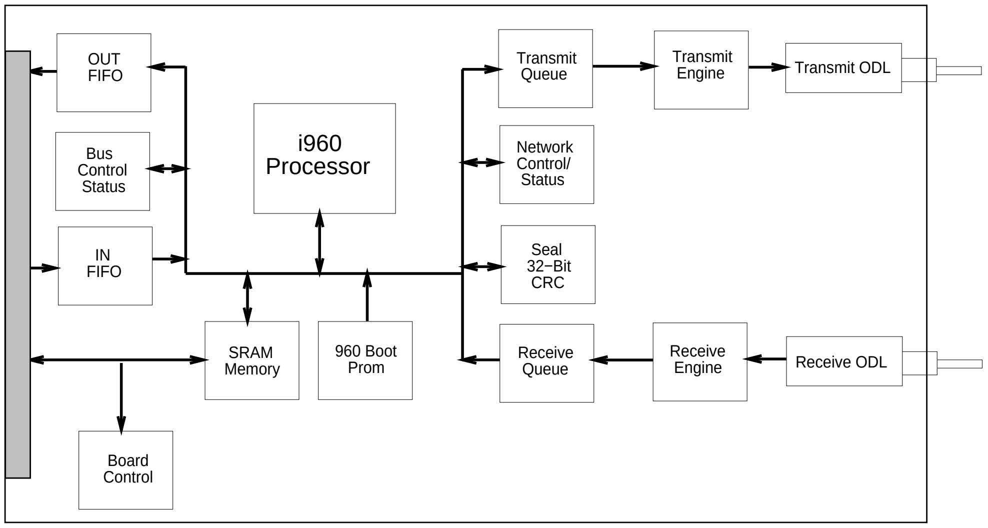


Figure 3: Series-200 host interface

100 switches within both the metropolitan-area and the wide-area environment. The ASX-100 supports Fore's SPANS signaling protocol with both the Series-100 and Series-200 adapters, and can establish either Switched Virtual Circuits (SVCs) or Permanent Virtual Circuits (PVCs). All of the experiments conducted ignored circuit setup time and thus the ATM circuits used can be viewed as PVCs.

The host environment was provided by MAGIC and the Army High Performance Computing Research Center (AHPCRC). Two Sun 4/690's, provided by the AHPCRC, were connected via a local Ethernet subnet, a local FDDI ring, and the MAGIC local ATM network. MAGIC provided two Sun Sparc 1+'s which were connected via a local Ethernet subnet and the MAGIC local ATM network. The Sun 4/690's and Sparc 1+'s were used to characterize various aspects of end-to-end network communication (Section 3). The machines were also connected in a point-to-point manner for the ATM portion in order to characterize the effect of the Fore's ASX-100

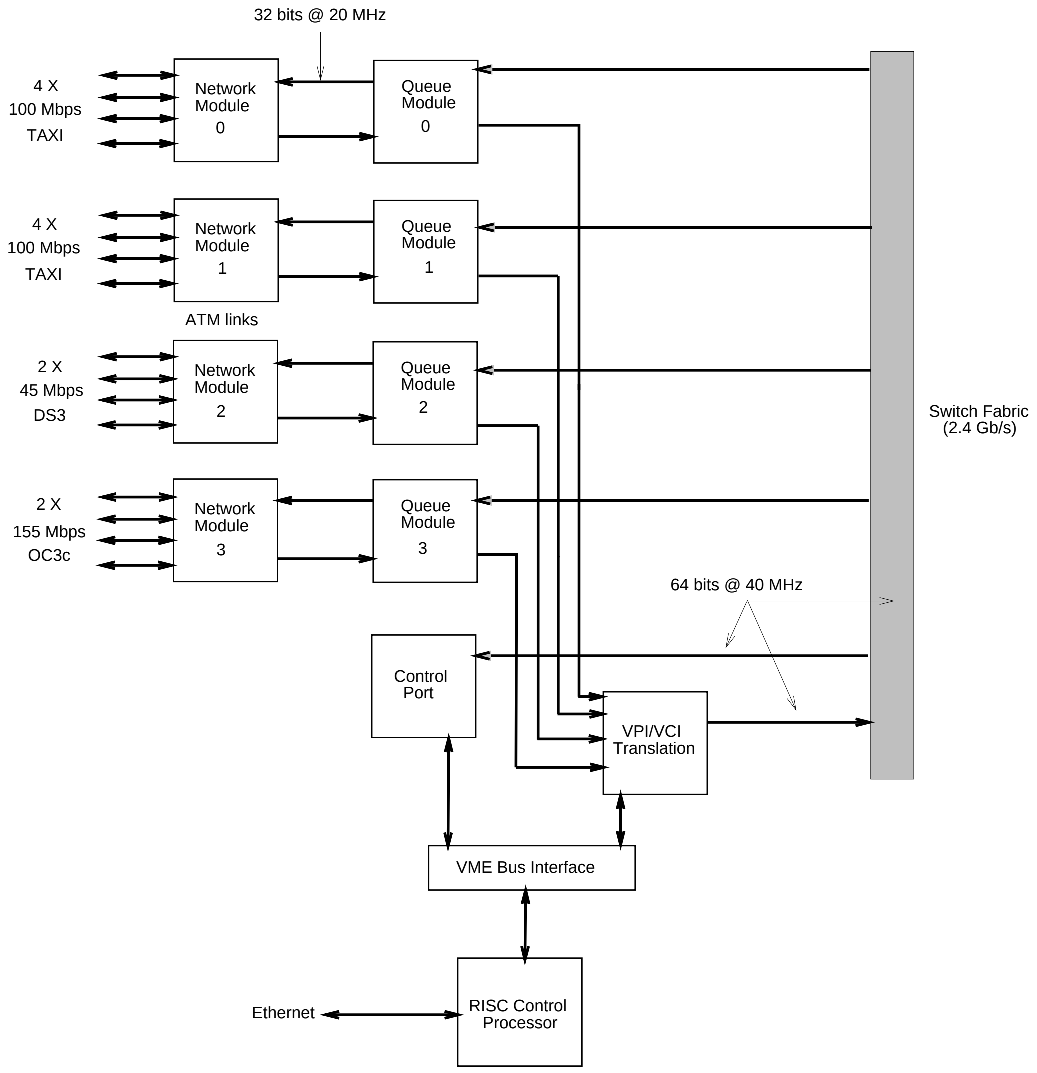


Figure 4: MAGIC ASX-100 ATM Switch functional overview

Switch in the local ATM network.

To characterize the network when used for distributed applications (Section 4), the AHPCRC provided four Sun Sparc 2 machines. These machines were connected via the MAGIC ASX-100 ATM switch, and via a local Ethernet subnet. For experiments run over Ethernet, a Network General Sniffer was used to measure an artificial load created by a client/server socket program on the same Ethernet subnet. Loads between 0 and 30 percent were used to simulate what a typical subnet might look like. We did not have access to equipment necessary to generate loading for the ATM network.

2.3 Application Programming Interfaces

Programmers can choose from a wide variety of APIs. In this section, we briefly review four of them: Fore's API, BSD's socket interface, Sun's RPC/XDR, and PVM's message passing library.

2.3.1 Fore Systems ATM API

With support from the underlying device driver, the user-level library routines provide a portable interface to the ATM data link layer. Depending on the platform, the subroutine library uses either a System V STREAMS interface or a socket-based interface to the device driver. The details of the implementation are hidden from the programmer, so the interface described here is portable across platforms.

The ATM library routines provide a connection-oriented client and server model. Before data can be transmitted, a connection (SVC or PVC) has to be established between a client and server. After a connection is setup between the client and server, the network makes a "best effort" to deliver ATM cells to the destination. During the transmission, cells may be dropped depending on the available resources remaining. End-to-end flow control between hosts and cell retransmissions are left to the applications.

The library routines provide a socket-like interface. Applications first use *atm_open()* to open a file descriptor and then bind a local Application Service Access Point (ASAP) to the file descriptor with *atm_bind()*. Each ASAP is unique for a given end-system and is comprised of an ATM switch identifier and a port number on the switch. Connections are established using *atm_connect()* within the client process in combination with *atm_listen()* and *atm_accept()* within the server process. These operations allow the data transfer to be specified as simplex, duplex, or multicast.

An ATM VPI and VCI are allocated by the network during connection establishment. The device driver associates the VPI/VCI with an ASAP which is in turn associated with a file descriptor. Bandwidth resources are reserved for each connection. If the specified bandwidth is greater than the capability of the fabric, the connection request will be refused due to lack of communication resources.

Applications can select the type of ATM AAL to be used for data exchange. The selected AAL is treated as an argument of *atm_connect()* on the client side. In Fore Systems' implementation, AAL type 0, 1, and 2 are not currently supported by Series-200 interfaces, and type 3 and 4 are treated identically.

atm_send() and *atm_recv()* are used to transfer user messages. One Protocol Data Unit (PDU) is transferred on each call. The maximum size of the PDU depends on the AAL selected for the connection and the constraints of the underlying socket-based or stream-based device driver implementation.

The local ATM network also supports TCP/IP. Either AAL 3/4 or AAL 5 can be used to encapsulate IP packets. On the receiving side, the packet is implicitly demultiplexed. The host uses the identity of the VC to determine whether it is an IP packet. The bandwidth specified for IP connections is zero, which is interpreted by the switch control software as lower priority than any connection with non-zero reserved bandwidth.

2.3.2 BSD Socket-Based Programming Interface

The 4.2BSD kernel introduced an Interprocess Communication (IPC) mechanism (*sockets*) which is more flexible than Unix pipes. A socket is an end-point of communication referred to by a descriptor (just like a file or pipe.) Two processes each create a socket, and then connect those two end-points to establish a reliable byte stream or unreliable datagram. Once connected, the descriptors for the sockets can be read from or written to by user processes similar to regular file operations. The transparency of sockets allows the kernel to redirect the output of one process to the input of another process residing on another machine [20].

All sockets are typed according to their communications semantics. Socket types are defined by the subset of properties a socket supports. These properties are in-order delivery of data, unduplicated delivery of data, reliable delivery of data, preservation of message boundaries, support for out-of-band messages, and connection-oriented communication.

A connection is a mechanism used to avoid having to transmit the identity of the sending socket with each packet of data. Instead, the identity of each end-point of communication is exchanged prior to transmission of any data, and is maintained at each end so that it can be referred to at any time when sending or receiving messages. A *datagram socket* models potentially unreliable, connectionless packet communication; a *stream socket* models a reliable connection-based byte streams that may support out-of-band data transmission; and a sequenced packet socket models sequenced, reliable, unduplicated connection-based communication that preserves message boundaries.

Sockets exist within communication domains. A communication domain is an abstraction introduced to bind common properties of communications. BSD socket supports the Unix domain, the Internet domain, and the NS domain. In our environment, we are limited to use the Internet domain for communication over local ATM networks. In the Internet domain, *stream sockets* and *datagram sockets* use TCP/IP and UDP/IP [21] as the underlying protocols, respectively. We especially focus on the communication performance of the stream socket since it provides a reliable data transmission.

2.3.3 Sun Remote Procedure Call

Remote procedure call (RPC) is a fundamental approach to interprocess communication based on the simple concept known as the procedure call. The RPC model is as follow: a client sends a request, and then blocks until a remote server sends a response back. It is very similar to the well-known and well-understood mechanism known as a procedure call. There are various RPC extensions available such as broadcasting, nonblocking, and batching.

Sun RPC uses both UDP/IP and TCP/IP as its underlying protocols. It supports three RPC features (blocking, batching, and broadcasting) for each transport protocol. So altogether there are five types of RPC calls (broadcast RPC can only use connectionless transport protocols like UDP/IP). Batching RPC is a non-blocking call that does not expect a response. In order to flush previous calls, the last call must be a normal blocking RPC call. In broadcast RPC calls, servers that support broadcast respond only when the calls are successfully completed, otherwise they are silent.

Sun RPC provides two types of interfaces for application programmers. One is available via library routines which consist of three layers. The second interface uses the RPC specification language (RPCL) and the stub generator (RPCGEN). RPCL is an extension of the XDR specification.

2.3.4 PVM Message Passing Library

PVM is a *de facto* standard for distributed computing that uses a basic message passing library. The PVM software system allows a heterogeneous network of computers to be used as a single parallel computer. Thus, large computational problems can be solved by using the aggregate power of many computers.

Under PVM, a collection of serial, parallel, and vector computers appears as one large distributed-memory computer. A per-user distributed environment must be setup before running PVM applications. A PVM daemon process runs on each of the participating machines and is used to exchange network configuration information. Applications, which can be written in Fortran or C, can be implemented by using the PVM message passing library which is similar to libraries found on most distributed-memory parallel computers.

Sending a message with PVM is composed of three steps. First a send buffer must be initialized by a call to *pvm_initsend()* or *pvm_mkbuf()*. Second, the message must be packed into a buffer using any number of *pvm_pk*()* routines. Each of the *pvm_pk*()* routines packs an array of a given data type into an active send buffer. Calls to *pvm_unpk*()* routines unpack the active receive buffer into an array of a given data type. Third, the message is sent to another process by calling the *pvm_send()* routine or the *pvm_mcast()* (multicasting) routine. The message is received by calling either a blocking receive using *pvm_recv()* or non-blocking receive using *pvm_probe()* and *pvm_recv()*.

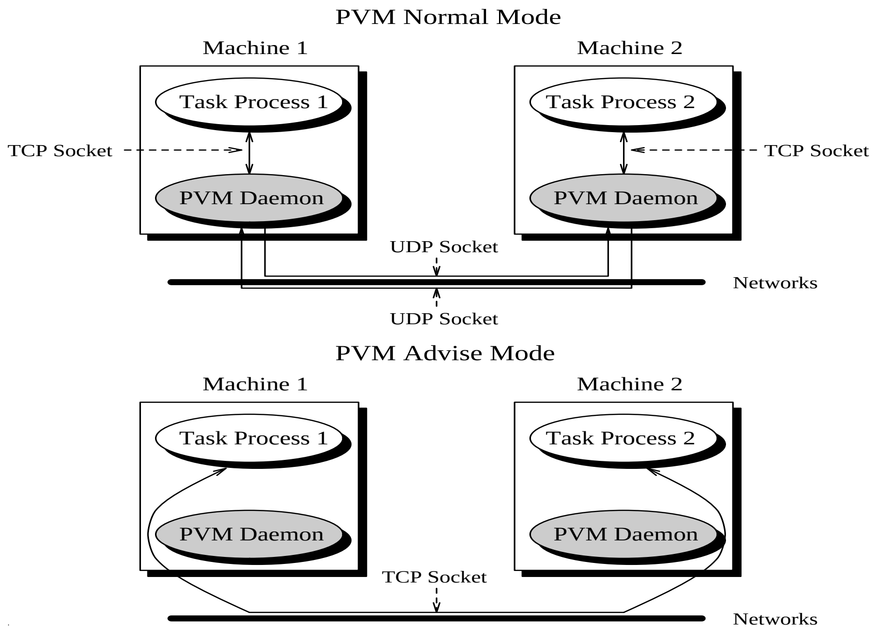


Figure 5: Comparison of PVM Normal and PVM Advise modes

A Dynamic Process Group is implemented on top of PVM version 3. With this implementation, a process can belong to multiple named groups, and groups can be changed dynamically at any time during a computation. Functions that logically deal with groups of tasks such as broadcast and barrier synchronization use the user's explicitly defined group names as arguments. Routines are provided for processes to join and leave a named group.

PVM has two communication modes, PVM *Advise* mode and PVM *Normal* mode. The Advise mode sets up a direct TCP connection between two communicating processes (see Figure 5). The Normal mode uses the existing UDP connections among PVM daemon processes. Each application process creates a TCP connection with its local daemon process. Therefore, two TCP connections and two UDP connections are required for a bi-directional communication between two application processes (see Figure 5). There is no direct communication link between application processes for PVM Normal mode.

The advantage of Advise mode is that it provides a more efficient communication path than Normal mode. We have observed more than a twofold increase in communication performance (see Section 3.3). The drawback of Advise Mode is the small number of direct links allowed by some Unix systems, which makes their use unscalable. The terms *Advise* mode and *Normal* mode used here are not explicitly mentioned in the PVM manual.

3 End-to-end Communication Characteristics

In this section, we present the end-to-end communication characteristics of four APIs. A simple client/server echo algorithm was implemented using each API to measure its end-to-end performance. Two performance measurements are used to characterize the communication capabilities. One is the communication latency (latency is especially important when transmitting short messages), and the other is the maximum achievable communication throughput (the maximum achievable throughput is important for applications which may require the transmission of a large volume of data).

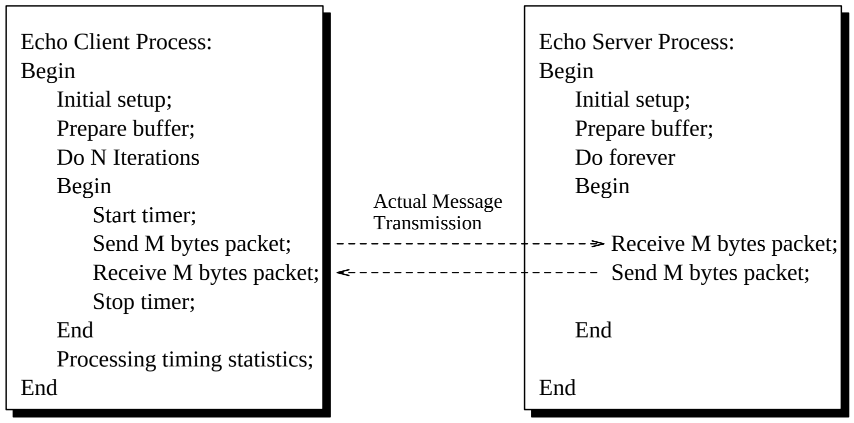
Each API represents programming at a different communication protocol layer (see Figure 1). Fore's API is implemented on top of the ATM adaptation layer. It provides a way to choose either AAL 3/4 or AAL 5 as its underlying communication protocol. In the Internet domain, the BSD socket interface is built on top of either TCP/IP, UDP/IP, or the raw socket. For the stream socket, TCP/IP is employed by default. For IP over ATM either AAL 3/4 or AAL 5 can be used to encapsulate IP packets. However, in our environment, AAL 5 is the default adaptation layer for transmitting IP packets over ATM networks. The PVM message passing library uses BSD sockets as its underlying communication facility. As we pointed out in Section 2.3.4, PVM has two communication modes, Advise and Normal. The protocol combinations needed for an application using PVM totally depends the assumed PVM mode. Sun RPC/XDR is also built on top of the BSD Socket interface. Either TCP/IP or UDP/IP can be chosen as its underlying communication protocol, however we only evaluated Sun's RPC/XDR protocol using TCP/IP.

The BSD Socket interface is used by both PVM and Sun's RPC/XDR as its underlying interprocess communication mechanism. The communication performance of the socket interface has a significant impact on the performance of PVM and Sun RPC/XDR. Therefore, we carefully examined the end-to-end characteristics of stream sockets (see Section 3.2). We also studied the performance of the two PVM communication modes in Section 3.3. The performance of Fore's API using AAL 3/4 and AAL 5 protocols is presented in Section 3.4.

Although there are many possible protocol combinations as indicated in Figure 1, we were specifically interested in the five protocol combinations listed below.

1. Fore Systems ATM API over ATM AAL 3/4
2. Fore Systems ATM API over ATM AAL 5
3. BSD Stream socket over TCP/IP over ATM AAL 5
4. PVM Advise mode using Stream sockets over ATM AAL 5
5. Sun RPC/XDR using Stream sockets over ATM AAL 5

The performance comparison of the five different protocol combinations is presented in Section 3.4. For convenience of presentation, we have defined three performance metrics to capture the end-to-end performance characteristics. Since stream sockets are available on various networks, it is a good candidate to be used to compare the performance of these networks. In Section 3.5 we compare the performance of stream sockets over FDDI, Ethernet and ATM networks. The performance of ATM may be different for different host machines and interface cards. Therefore, in Section 3.6 we compare the performance of the ATM AAL 5 protocol over several different host machines and interface cards.



```
Echo Client Process:
Begin
    Initial setup;
    Prepare buffer;
    Do N Iterations
    Begin
        Start timer;
        Send M bytes packet;
        Receive M bytes packet;
        Stop timer;
    End
    Processing timing statistics;
End
```

```
Echo Server Process:
Begin
    Initial setup;
    Prepare buffer;
    Do forever
    Begin

        Receive M bytes packet;
        Send M bytes packet;

    End

End
```

Figure 6: Pseudo codes for echo server and client processes

3.1 Echo Program

Figure 6 shows the pseudo code of the client and server echo processes. The client sends a M-byte message to the server and waits to receive the M byte message back. The client/server interaction iterates N times. We gather the round trip timing for each iteration in the client process. The timing starts when the client sends the M byte message to the server, and ends when the client receives M bytes of the response message.

The total round-trip time is affected by the the protocol stack, the device driver, the host interface, signal propagation, and switch routing. The echo program is used for measuring the end-to-end communication latency to avoid the problem of synchronizing clocks in two different machines. The communication latency for sending a M-byte message can be estimated as half of the total round-trip time. The communication throughput is calculated by dividing $2 \times M$ by the round-trip time (since $2 \times M$ bytes of message have been physically transmitted).

In our environment, two Sun 4/690s are physically connected to a local ATM network, a FDDI ring, and an Ethernet. Several experiments were conducted on the Sun 4/690s. In Section 3.2 and 3.6, Sun's Sparc 1+, and Sparc 2 computers were also used to measure the communication performance of the BSD sockets and Fore's API. With the exception of several experiments in Section 3.6, each workstation had a Fore Systems' Series-200 interface and was connected to module 0 (with 4 ports of 100 Mbits/sec TAXI interface) of the MAGIC ASX-100 ATM switch. Unless explicitly mentioned, the operating system used by Sun workstations is Sun OS 4.1.2. For the ATM connections, the length of the multi-mode 62.5 micron fiber optic cable was less than 20 meters.

The timing information was collected from the echo program. However, we found that the time required to send and receive the first message takes much longer than the subsequent transmissions. In the case of AAL 3/4 and AAL 5 the timing difference between the first and others is around 10 to 15 milliseconds. For stream socket, it is around 30 to 40 milliseconds.

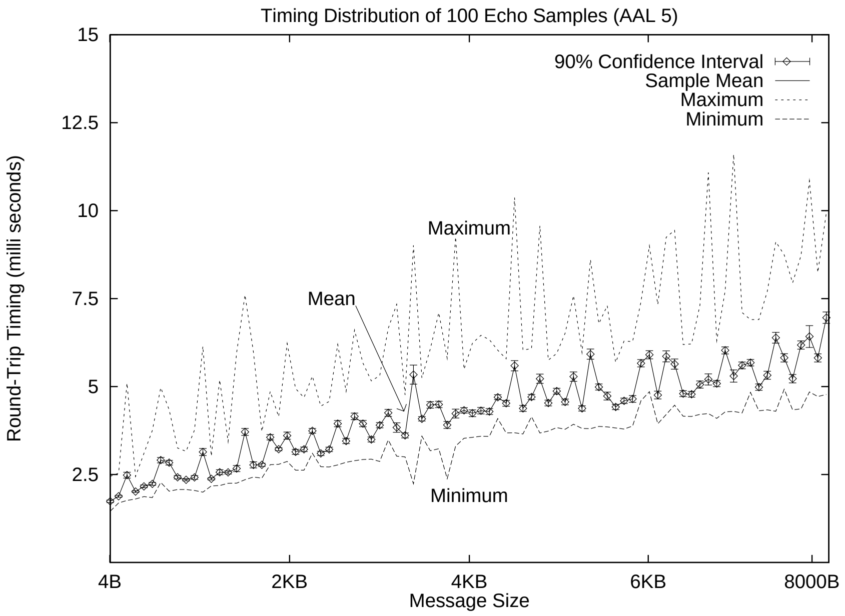


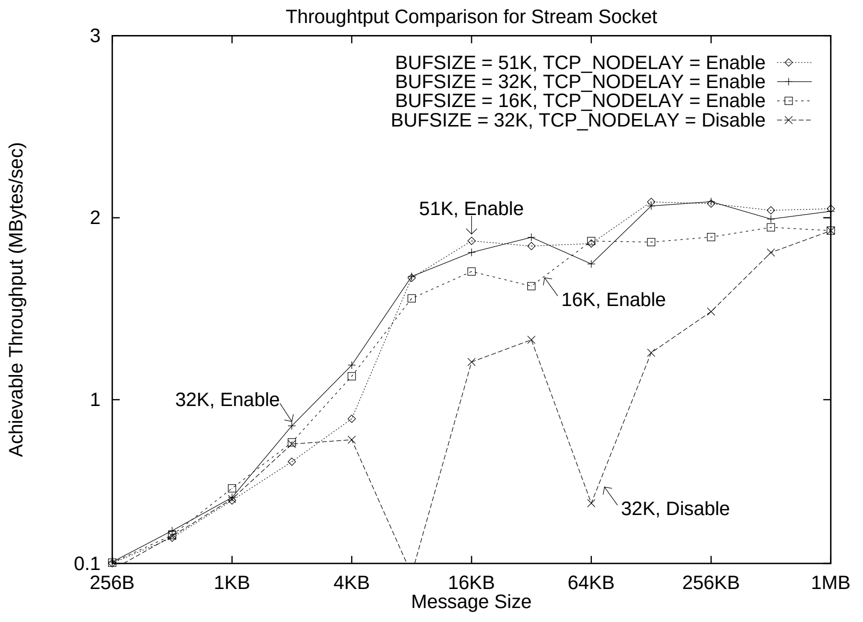
Figure 7: The variation of round-trip timing vs packet size for 100 samples using ATM AAL 5

Further study is required to fully understand the causes of this effect. We have not included the first echo timing in any of the calculations of end-to-end communication latency described in this section.

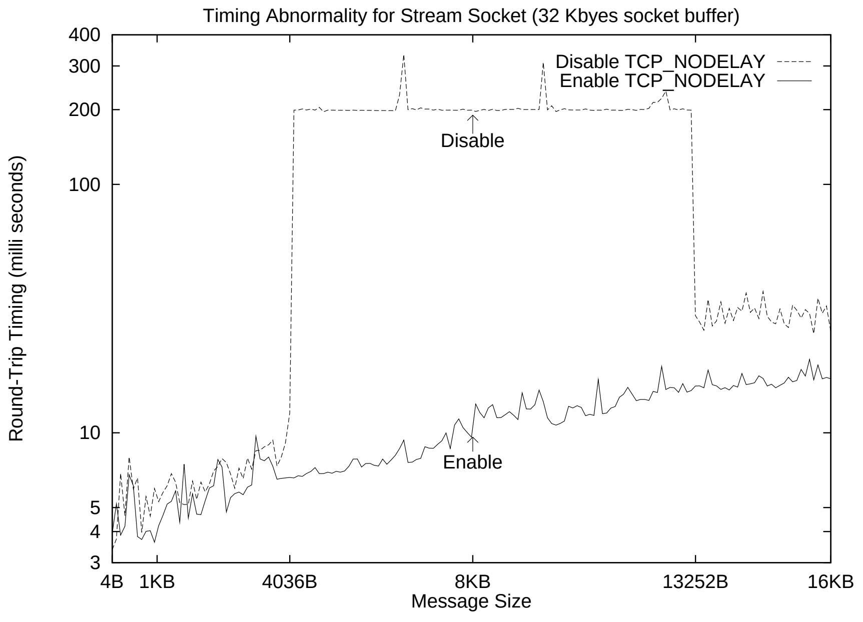
For each 100 timing samples collected, we use three statistics to represent the communication characteristics: sample maximum, sample mean, and sample minimum. Sample maximums and minimums represent the worst and the best of collected timings, respectively. Both are sometimes used to characterize communication performance. However, in our experiments, most of the timing samples collected are very close to the calculated sample mean. For example, Figure 7 shows the distributions of maximum, mean, and minimum for the AAL 5 echo program timing samples. We varied the message sizes as follows: 4 bytes, $4 + 96$ bytes, $4 + 96 \times 2$, ... up to 8 Kbytes. A 96 byte message is equal in length to the data payload of two ATM AAL 5 cells. For each given message size, the echo program iterates 100 times. We gathered the timing of each iteration and collected the maximum, minimum, mean, and computed a 90% confidence interval for each 100 echo timing samples. In Figure 7 the 90% confidence interval is very close to the mean of the samples. Therefore, we chose to only present the mean of the timing samples in the remaining experiments.

3.2 BSD Socket Interface over Local ATM Networks

In BSD, IPC the basic building block for communication is the socket. A socket is a communication endpoint. In the Internet communication domain, a stream socket is built on top of TCP/IP. The socket interface provides a way to manipulate options associated with a socket or its underlying protocols. Each option represents a special property of the socket or underlying protocol. Among those options the two most interesting to change are, the sending and receiving



(a)



(b)

Figure 8: End-to-end performance measurement of BSD Stream Socket on two Sun 4/690s over local ATM networks

buffer sizes, and enabling and disabling TCP_NODELAY. The socket buffer size is a socket layer option. The default socket buffer size depends on the system's initial configuration. The buffer size could range up to 51 Kbytes in our environment (some operating systems can provide socket buffer size greater than 64 Kbytes).

The choice of the socket buffer size will affect the time required to assemble or dis-assemble a message. TCP_NODELAY is a TCP layer option. When it is enabled, TCP will not queue any small packet (smaller than the low water mark of TCP flow control) to form a larger packet. The TCP protocol uses the low and high water marks to ensure appropriate flow control between two communicating processes. The TCP_NODELAY option is disabled by default.

The echo program, which is implemented using BSD sockets, was used to measure the achievable throughput and round-trip echo time of the stream sockets. The performance of different combinations of the above two options are presented. Figure 8(a) shows the achievable throughput when varying message size from 4 bytes to 1 Mbyte[3] with the message size doubled each time. We first studied the performance of the stream socket with TCP_NODELAY enabled for the following three socket buffer sizes; 16 Kbytes, 32 Kbytes, and 51 Kbytes. For Sun 4/690, there is no significant performance difference among the three socket buffer sizes.

Another experiment examined the effect of disabling and enabling TCP_NODELAY. We fixed the socket buffer size to 32 Kbytes. The choice of 32 Kbytes is intentional, since the socket buffer size of PVM Advise mode is set to 32 Kbytes. The achievable throughput with TCP_NODELAY enabled is better than that of the one with TCP_NODELAY disabled. As shown in Figure 8(a), the achievable throughput drops dramatically in two places around message sizes of 8 Kbytes and 64 Kbytes when TCP_NODELAY is disabled. In order to explore the detailed timing information for the message sizes around 8 Kbytes in greater detail, we also ran an experiment with message sizes that varied from 4 bytes to 16 Kbytes in 96 byte increments. The results are presented in Figure 8(b).

This is a known TCP timing abnormality which has been reported by Crowcroft [9] and others. We have observed the same timing abnormality in our local ATM network. However, after enabling TCP_NODELAY, this timing abnormality disappears.

We performed experiments with the same TCP echo program, over both FDDI and Ethernet for message sizes less than 16 Kbytes. We found that similar abnormalities also exist for FDDI and Ethernet. For FDDI, the timing abnormality occurred for message sizes in the range of 4072 to 12800 bytes. For Ethernet, the timing abnormality ranged from 4064 to 6976 bytes, 8416 to 9872 bytes, and 11328 to 12768 bytes. This timing abnormality is caused by the TCP protocol. However, the range of message sizes in which this abnormality occurred and the frequency of the timing abnormality are affected by the physical network on which TCP is used.

In a concurrent server model, a new socket will be created when the server accepts a connection request from a client. We would like to point out that this newly created socket will inherit only the options of the socket layer. The options of lower layer protocols such as TCP, UDP and IP will be set to their defaults. Therefore, the TCP_NODELAY option should be explicitly enabled by the server to ensure that this abnormality will not occur.

[3] 1 Mbyte equals to 2^{20} Bytes. For throughput figures used here, MBytes/sec equal 10^6 Bytes/sec.

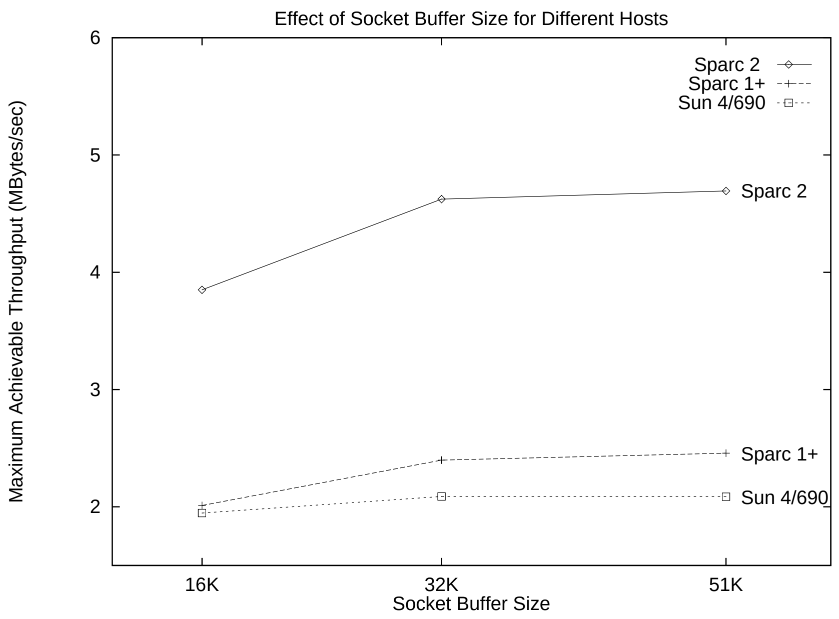


Figure 9: Maximum achievable throughput for different socket buffer sizes and host machines

To understand the effect of socket buffer size for different host machines, we also examined the TCP echo program over Sun's Sparc 1+, and Sparc 2. The TCP_NODELAY option is enabled in this experiment. The maximum achievable throughput for three socket buffer sizes are shown in Figure 9. We found that the larger the socket buffer size, the better the maximum achievable throughput for machines other than Sun 4/690.

3.3 PVM Characteristics over Local ATM Networks

PVM provides both Normal and Advise modes as described in Section 2.3.4. The Advise mode creates a direct TCP connection between two communicating application processes. The Normal mode uses an existing UDP connection between PVM daemon processes. Each application process creates a TCP connection with its local daemon process. Therefore, two TCP connections and two UDP connections are required for a bi-directional communication between two application processes.

Versions 3.2.4, 3.2.5, and 3.2.6 of PVM were used. In the PVM Advise mode (TCP) of version 3.2.4 the TCP_NODELAY option is disabled. Thus, timing abnormalities similar to stream socket were observed. This abnormality has been fixed in PVM version 3.2.5 by enabling TCP_NODELAY at the sending side. It also sets the socket buffer size to 32 Kbytes.

Figure 10 shows the performance effect of increasing the message size for both Normal and Advise modes. The PVM Advise mode provides at least a twofold performance jump over the PVM Normal mode. Each echo timing consists of the time spent on packing the message in the PVM buffer, and the time required for transmitting the message through the network. As shown in the figure, the total time is dominated by the message transmission time.

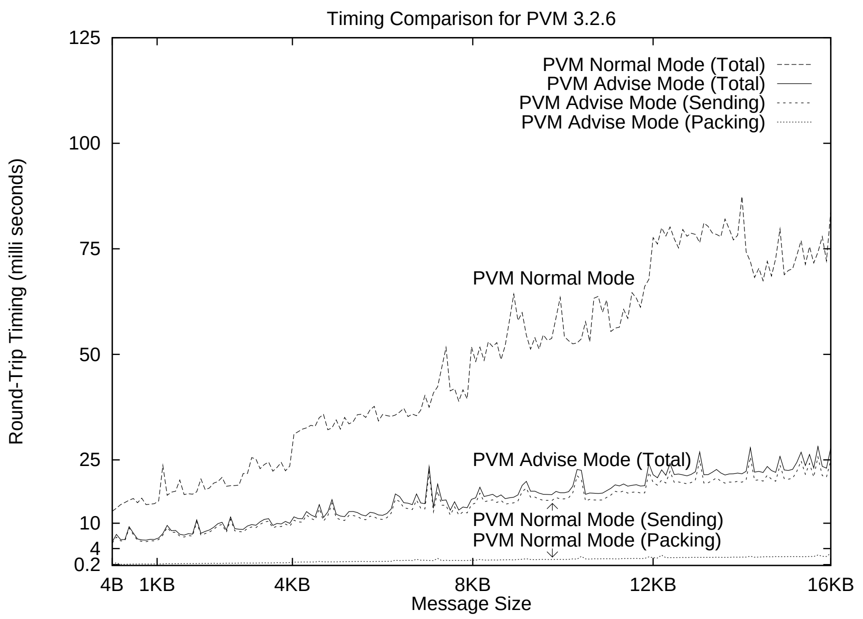


Figure 10: Performance of two PVM communication modes over local ATM networks

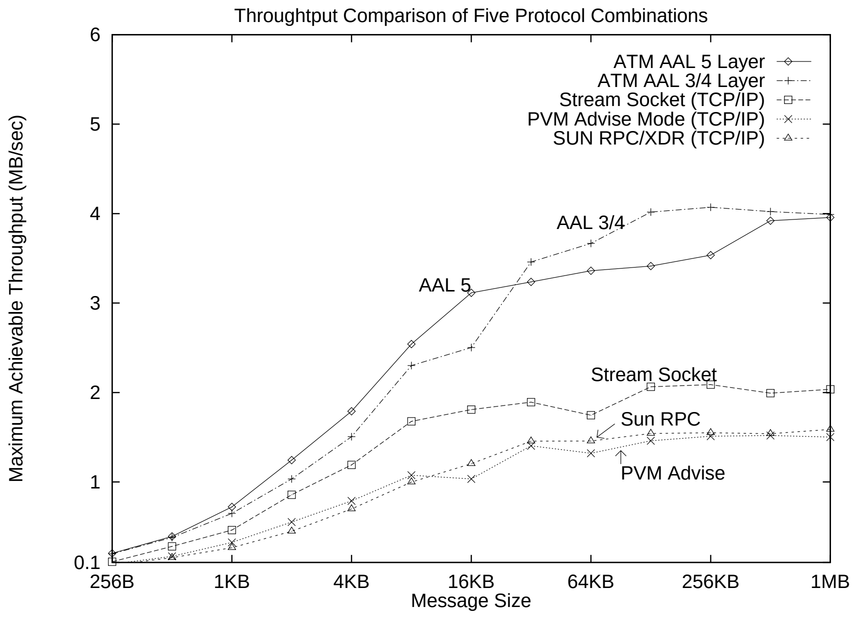
3.4 Four APIs over Local ATM Networks

In this subsection, we compare the performance of the following five protocol combinations.

1. **ATM AAL 3/4**: Fore Systems ATM API over ATM AAL 3/4
2. **ATM AAL 5**: Fore Systems ATM API over ATM AAL 5
3. **Stream socket (TCP)**: Stream sockets over ATM AAL 5
4. **PVM Advise mode (TCP)**: PVM Advise mode using Stream sockets over ATM AAL 5
5. **Sun RPC/XDR (TCP)**: Sun RPC/XDR using Stream socket over ATM AAL 5

In these experiments the socket buffer size is set to 32 Kbytes and the TCP_NODELAY option is enabled for all TCP/IP connections. Figure 11 shows the times required to exchange a message of 4 bytes using ATM AAL 5, ATM AAL 3/4, and the socket interface to be 1738 μs, 2068 μs and 3920 μs respectively. The time required by either PVM or RPC is at least three times more than that of ATM AAL 5 and the maximum achievable throughput is about half of that of ATM AAL 5. As expected, Fore's API exhibited a better communication performance than the others. The major causes of the long latency and low throughput of PVM and RPC is the protocol processing overhead of TCP/IP, the communication overhead of the PVM daemon and RPC daemon processes, and heavy interaction with the host operating system.

The latency of ATM AAL 5 is still too large for any communication intensive application. It is believed that modifications to the ATM interface device driver could reduce the overhead to less than one hundred μsec.



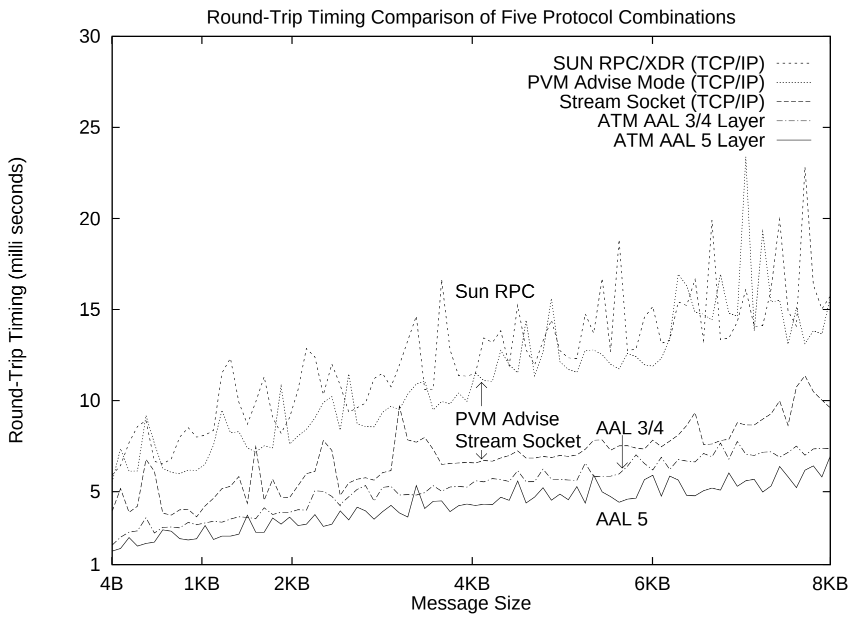


Figure 11: Five protocol combinations

Table 1: Echo measurements of $(r_{max}, n_{1/2})$ for five different protocol combinations

Protocol Hierarchy	r_{max} MBytes/sec	$n_{1/2}$ Bytes	t_0 μsec
ATM AAL 5	3.96	5134	869
ATM AAL 3/4	4.07	6823	1034
BSD Stream Socket	2.09	3204	1960
PVM Advise	1.52	3853	2766
Sun RPC/XDR	1.59	5407	2957

We further characterize the experimental results using some other performance metrics. Three performance metrics are introduced below.

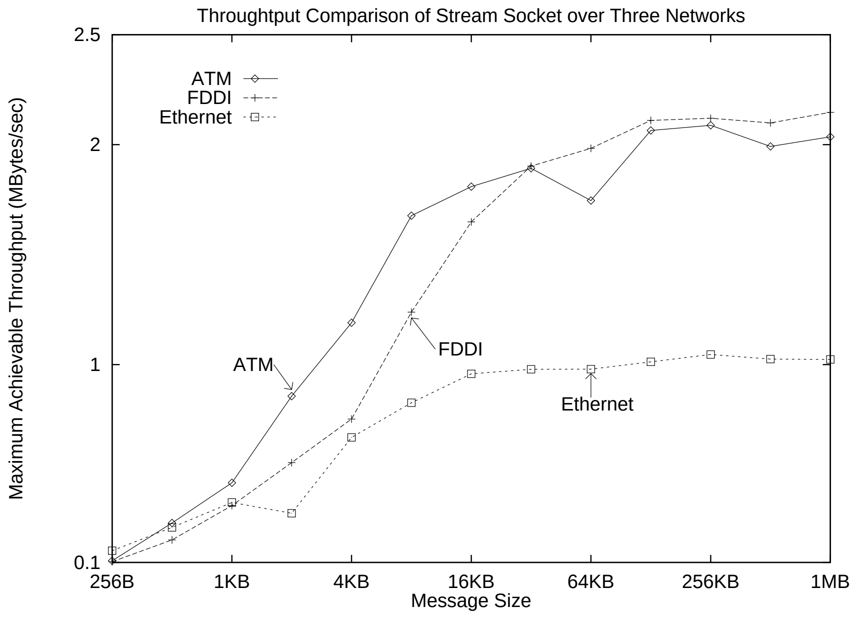
- r_{max} (*maximum achievable throughput*) : the maximum achievable throughput which is obtained from experiments by transmitting very large messages.
- $n_{1/2}$ (*half performance length*) : the message size needed to achieve half that of the maximum achievable throughput. This number may not be compared with the corresponding numbers from different hardware and software configurations, since the maximum achievable throughputs may be different for different configurations.
- t_0 (*startup latency*) : the time required to send a message of minimum size. This is set to half of the time required by the echo program when sending a message of 4 bytes.

These three performance metrics provide a quick reference for the communication characteristics of the different protocol combinations. The maximum achievable throughput is the maximum throughput which could be observed by applications in different software and hardware combinations. It is important for applications which may require a large volume of data transmission. The startup latency is the minimum required time to send messages. It is especially important when transmitting short messages. The half performance length provides a reference point to reach half of the maximum achievable throughput.

In Table 1 we characterize five protocol combinations using these three metrics, r_{max}, $n_{1/2}$, and t_0. The *startup latency* for ATM AAL 5 is 869 μsec; ATM AAL 3/4 yields the largest maximum achievable throughput, 4.07 MBytes/sec. There is no significant difference for communication overhead of PVM Advise mode and Sun RPC/XDR.

3.5 BSD Stream Socket Over Different Networks

In this experiment, we compared the performance of stream sockets (TCP/IP) over local ATM, Ethernet, and FDDI networks. The ATM, FDDI, and Ethernet interface were assigned with different IP addresses. By giving the desired IP address, the IP protocol can choose the corresponding network interface to transmit messages.



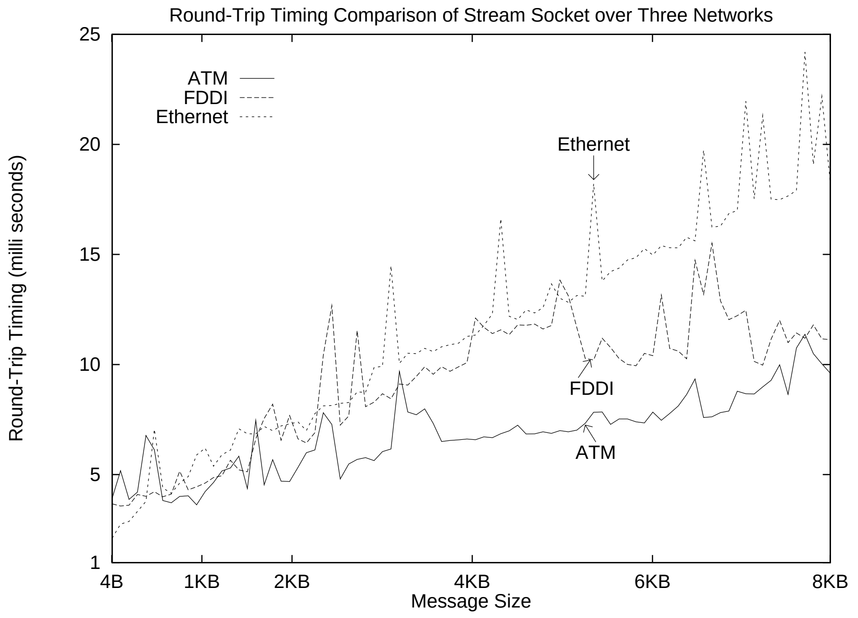


Figure 12: Performance comparison of Stream Socket over different networks

Table 2: Echo measurements of $(r_{max}, n_{1/2}, t_0)$ for BSD Stream Socket over different networks

Protocol Hierarchy	r_{max}	$n_{1/2}$	t_0
	MBytes/sec	Bytes	μsec
ATM Networks	2.09	3204	1960
FDDI Ring	2.15	6818	1833
Ethernet	1.05	6482	1053

Table 3: Echo measurements of $(r_{max}, n_{1/2}, t_0)$ using AAL 5 echo program for various point-to-point connection

End Host	Interface	Operating	I/O Bus	r_{max}	$n_{1/2}$	t_0
	Card	System	Standard	MBytes/sec	Bytes	μsec
Sun Sparc 1+	Series-100	4.1.1	S-Bus	0.96	838	734
Sun Sparc 2	Series-100	4.1.2	S-Bus	1.44	811	547
Sun Sparc 1+	Series-200	4.1.2	S-Bus	2.60	3784	811
Sun 4/690	Series-200	4.1.2	S-Bus	4.40	4237	858
Sun Sparc2	Series-200	4.1.2	S-Bus	5.76	6650	469

Figure 12 shows the time required by the echo program for short messages and the achievable throughput for long messages. As stated previously, all experiments are performed in the absence of other network traffic. Ethernet shows the lowest latency for message sizes less than 500 bytes. It is believed that the firmware code for Ethernet has been fine-tuned for better communication latency. The communication latencies of ATM and FDDI are similar. Both ATM and FDDI sustain around 2 MBytes/sec throughput. Their network utilization is only around 16 % (16 Mbits/sec out of 100 Mbits/sec) over that of TCP/IP. This indicates the possibility of further improvement of TCP/IP over ATM.

The Ethernet can sustain a 1.05 MBytes/sec throughput. This figure represents 84 % network utilization of Ethernet. Table 2 summarizes our results.

3.6 Performance Comparisons of Different Hardware Configurations

Table 4: Echo measurements of $(r_{max}, n_{1/2}, t_0)$ using AAL 5 echo program for various configuration via an ASX-100 switch in a local area

End Host	Interface	Operating	r_{max}	$n_{1/2}$	t_0
	Card	System	MBytes/sec	Bytes	μsec
Sun Sparc 1+	Series-100	4.1.1	0.94	748	736
Sun Sparc 2	Series-100	4.1.2	1.36	707	537
Sun Sparc 1+	Series-200	4.1.2	2.82	6675	742
Sun 4/690	Series-200	4.1.2	3.96	5134	869
Sun Sparc 2	Series-200	4.1.2	5.61	4475	478

All the previous end-to-end communication performance measurements were done on two Sun 4/690 machines. In this subsection, experiments on a variety of host machines and different host interfaces were tested. The host machines tested include Sun Sparc 1+, Sparc 2, and 4/690 The two ATM interface cards include Fore's Series-100 and Series-200. The performance metrics obtained were tabulated in Table 3 for a point-to-point connection without going through an ATM switch and in Table 4 for communication via a local ATM switch.

We list the performance comparison as follows:

- **Effect of Host Interfaces:** The Fore's Series-200 interface has much better communication throughput than the Series-100 interface. For example, the Sun Sparc 2 can achieve a 5.76 MBytes/sec maximum throughput using a SBA-200 interface, but only 1.44 MBytes/sec using a SBA-100 interface.
- **Effect of Host Machines:** Faster machine CPUs yield higher throughput and the lower latency. One special case which was unexpected was the Sun 4/690. It had a larger latency than the Sparc 1+. Further study is required.
- **Effect of Switch Component:** The signal propagation delay through the switch was measured from the timing differences between point-to-point direct connection and connection via an ASX-100 switch. Using the Series-200 interface, the delay through the switch was 9 μsec and 11 μsec for the Sparc 2 and Sun 4/690 respectively.
- The Sun Sparc 2 had the lowest communication latency and the largest user throughput.

In the next section, we investigate the performance of PVM, BSD Sockets, and Fore's API when carrying out distributed applications over Ethernet and local ATM networks.

4 Performance Evaluation of Distributed Applications

Distributed network computing is one of the possible application areas that may benefit from the use of high-speed local area networks such as ATM. Computers which are distributed in a local area can be used together to cooperatively solve large problems. Previously a supercomputer would have been required to solve such problems. The echo program used in the previous sections provided the latency and achievable throughput of different protocol combinations over several networks. In this section, we consider the impact of using different protocol combinations over local ATM networks for distributed applications. We especially want to understand the performance of two popular distributed programs, parallel partial differential equations (PDE) and parallel matrix multiplication, over ATM LANs and Ethernet.

The partial differential equations and matrix multiplication examples were chosen because they represent two typical types of communication and computation patterns. The parallel matrix multiplication consists of several phases including a distribution phase, a computation phase, and a result collecting phase. During the distribution phase and result collecting phase, a high volume of data will be transferred between processing nodes. The matrix multiplication can be used to compare the throughput of different protocol combinations and networks. The parallel partial differential equations can be characterized as a communication intensive application. During the execution, each processing node repeatedly exchanges its boundary values with its immediate neighbors. Since only boundary values need to be exchanged, most of the messages are short. Parallel PDE can be used to compare the latency impact of different protocol combinations and networks. We present a brief description of the PDE and matrix multiplication in later subsections.

The hardware environment we used included four Sun Sparc 2 workstations. These four workstations were exclusively used for all of our experiments. Each Sparc 2 is equipped with an Ethernet adapter connected to a 10 Mbits/sec Ethernet, and a Fore Systems SBA-200 ATM adapter connect to a Fore Systems ASX-100 Switch. The parallel PDE and matrix multiplication applications were implemented using the master/slave programming model. In a master/slave model the master program spawns and directs some number of slave programs which perform computations. The master program is also responsible for recording timing information and collecting computation results. One of the four Sparc 2 workstations was used to execute the master and a slave at the same time.

When running distributed programs on the ATM LAN, the ATM switch was dedicated to our experiments. For the Ethernet experiment, we executed the distributed programs over the network with two different background traffic loads: silent and 30% loaded. Since the bandwidth in an Ethernet network is a shared, having additional load is more realistic. In our Ethernet experiments, a Network General Sniffer (Ethernet sniffer) was used to monitor the traffic of the Ethernet to secure a fully controlled testing environment.

We would like to point out an important difference between a local ATM network and the Ethernet. A local ATM network is scalable; an Ethernet is not scalable. An ATM switch of n ports is capable of supporting n parallel channels. Therefore, in a mesh-connected distributed application each processor needs to communicate with four immediate neighbors. That means the channel connected to the processor will be shared by four other processors. As long as the switch is capable of supporting n ports, as n increases there are always only four processors sharing the same channel. However, this is not the case for Ethernet. As the number of processors increases, the total traffic amount increases as well. This is the reason that we did not consider extra traffic loads for local ATM network in our experiments. Due to the availability of equipment, we were unable to investigate the issue of scalability further.

In the following two experiments, the communication APIs and networks that we compared are:

- BSD stream socket interface over Ethernet and ATM networks: The TCP_NODELAY option was enabled, and both socket's send and receive buffers were set to 32 Kbytes during the execution of the distribution applications.
- PVM over Ethernet and ATM networks: The PVM Advise mode was used so that processing nodes could communicate with each other over direct task-to-task links.

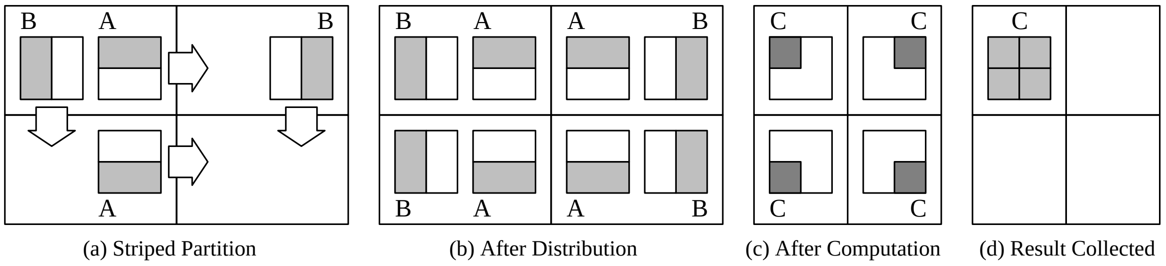


Figure 13: A simple parallel implementation of matrix multiplication

- Fore's API: The ATM AAL 5 was used due to its lower communication latency.

4.1 Parallel Matrix Multiplication

There are many possible parallel algorithms for matrix multiplication. We used a straightforward approach to address the problem of parallel multiplication of two $n \times n$ square matrices A and B to yield the product matrix C = A $\times$ B. The cluster of Sun Sparc 2 workstations is viewed as a simple 2-D mesh (2×2). Before the distribution phase, matrix A was partitioned by row-stripping such that each processing node (Sparc 2 workstation) in the leftmost column of the 2-D mesh will have half of the number of rows of matrix A. Matrix B was partitioned by column-stripping in a similar way such that each processing node in the topmost row of the mesh will have half of the number of columns of matrix B as shown in Figure 13(a).

The distribution phase consists of two steps. In the first step, processing nodes in the leftmost column transmit the partitions of matrix A to those processing nodes in the same row. In the second step, processing nodes in the topmost row will transmit the partitions of matrix B to those nodes located in the same column. After the distribution phase (Figure 13(b)), each processing node computes a submatrix of C using the appropriate partitions of matrices A and B (Figure 13(c).) The result of each submatrix of C will be sent back to the *master*, a designate processing node, in the result collecting phase as shown in Figure 13(d).

Table 5 shows the total execution time for three matrix sizes, i.e., 32×32, 128×128, and 256×256. The timing information in table 5 is the mean value of 50 executions of the same distributed program. In the ATM LAN environment, the performance of PVM and BSD socket are similar to that of Fore's API since the required computation becomes the dominant part of the execution. From Table 5, we can see that Fore Systems API has the best throughput. This is because it uses the AAL 5 directly. A 3.93 speedup is achieved when running the 256×256 matrix multiplication over ATM.

We also conducted the same experiments over Ethernet with two background traffic loads (silent and 30% load.) The traffic on the Ethernet was monitored by the sniffer during the execution. The sniffer is capable of capturing all traffic over the network. The performance of both PVM and BSD socket over silent Ethernet is comparable to that over ATM.

Table 5: Execution time of matrix multiplication (unit: second)

Protocol hierarchy/	**Matrix Size**		
Network	**32×32**	**128×128**	**256×256**
Sequential	0.0988	6.6205	64.0001
PVM			
ATM	0.0524	1.9493	16.4005
Ethernet (Silent)	0.0134	1.9693	16.9130
Ethernet (30% loaded)	0.0341	2.0355	17.2416
BSD Socket			
ATM	0.0736	1.9177	16.4030
Ethernet (Silent)	0.0627	1.9136	16.7187
Ethernet (30% loaded)	0.0714	1.9932	16.9256
Fore's API			
ATM	0.0629	1.7758	16.2709

To setup a 30% loaded Ethernet, we used two more Sparc 2 workstations on the same Ethernet to generate background traffic and used the sniffer to verify the traffic load. One of the Sparc 2 workstations periodically sent an 1460 byte UDP packets to the other workstation. The 1460 byte UDP packet can be packed into a single Ethernet packet, then transmitted. We used the sniffer to adjust the interval between UDP packets to achieve the desired background traffic load. A 128 byte UDP packet with shorter interval has also been used to created the same amount of background traffic. However, we observed a similar effect as the previous approach.

4.2 Parallel Partial Differential Equations

PDE is widely used in many applications of large-scale scientific computing such as weather forecasting, modeling supersonic flow, and elasticity studies. One of the parallel algorithms which uses a 2-d mesh topology is briefly described below. For a detailed description, refer to [2].

One class of the PDE problems can be represented by a uniform mesh of $n+1$ horizontal and $n+1$ vertical lines over the unit square as shown in Figure 14(a), where n is a positive number. The intersections of these lines are called mesh points. For the desired function *u(x,y)* at each mesh point, an iterative process can be used to obtain an approximate value for *u(x,y)*. When computing the approximate value for *u(x,y)*, we need the values from its four neighboring mesh points (except those boundary mesh points, which have less than four neighbors). Let e_k denote the absolute value of the difference between the approximate value u_k(x,y) and the exact value of u at (x, y). The iterative process continues until $e_k \leq e_0/10^v$. It can be shown that the iterative process converges after $g * n$ iterations, where $g = v/3$ and v is the required accuracy. For example, for 10^{-6} accuracy, v is 6 and g is 2.

In our implementation of the parallel PDE, the cluster of Sun Sparc 2 workstations was used as a 2D-mesh. The mesh points in the unit square are partitioned equally in checkerboard

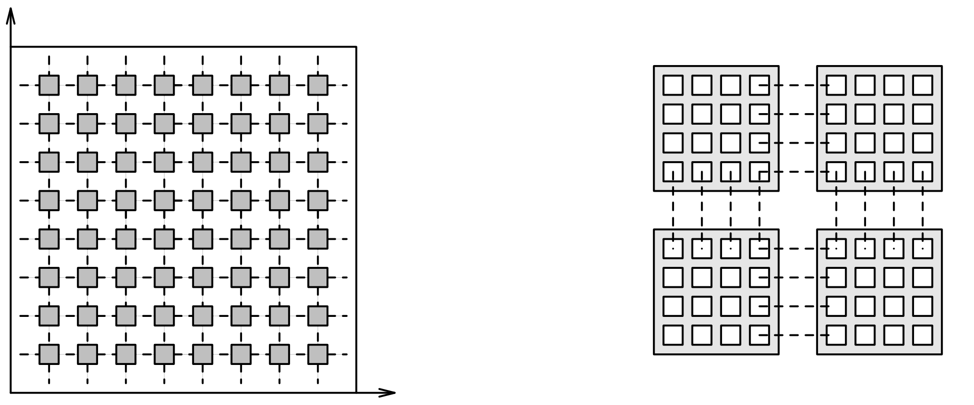

(a) Mesh points inside the unit square (b) Partitioned by 2-D mesh

Figure 14: Mesh points and mesh partition for parallel PDE

Table 6: Execution time of Partial Differential Equation (unit: second)

Protocol hierarchy/ Network	Mesh size					
	16×16		64×64		256×256	
Accuracy	10^{-6}	10^{-12}	10^{-6}	10^{-12}	10^{-6}	10^{-12}
Sequential	0.0868	0.1713	5.1483	10.2823	330.7060	661.4495
PVM						
ATM	0.2994	0.5821	3.0942	6.1306	137.2770	273.8295
Ethernet (Silent)	0.3326	0.6472	3.2719	6.4996	138.3927	276.7819
Ethernet (30% loaded)	0.3519	0.6750	3.4066	6.6960	140.2437	279.1801
BSD Socket						
ATM	0.1142	0.2608	2.4681	4.9136	133.6884	266.8405
Ethernet (Silent)	0.1432	0.2824	2.6505	5.1868	134.7947	268.7918
Ethernet (30% loaded)	0.1943	0.3651	2.6854	5.4361	135.9601	271.7504
Fore's API						
ATM	0.1222	0.2208	2.4506	4.8273	133.2512	266.0706

style and mapped to the processing nodes as shown in Figure 14(b). The dash lines between the mesh points represent the cross-machine communications. In each iteration, each processing node sends values of its boundary mesh points to its neighbors and waits for data from its four neighboring nodes. It then recomputes the approximate values of those mesh points which reside inside its partition.

Table 6 shows the time spent executing the parallel version of partial differential equations for different mesh sizes, protocol hierarchy, and networks. Since the PDE example is one of the most communication intensive distributed applications, the overhead of different APIs become more important. In the ATM LAN, Fore's API has the lowest protocol overhead when compared with the other APIs. A speedup of 2.49 was achieved. The BSD socket API provided good performance and a reliable communication interface. The PVM message passing library had the worst performance in this scenario because PVM provides additional support for distributed programming which results in additional overhead. The performance gets even worse when running in PVM Normal mode instead of Advise mode.

In the Ethernet environment, we first executed the PDE over a silent Ethernet. According to the sniffer, we observed a consistent traffic load on the Ethernet. This is because each processing node needs to exchange data with its neighbors for every iteration. When running the 64 $\times$ 64 mesh size PDE, we measured a 6% traffic load, and for 256 $\times$ 256 mesh-size PDE a 3% traffic load. In the case of 64 $\times$ 64, each processing node has a 32 $\times$ 32 partition of mesh points, for 256 $\times$ 256 mesh-size, 128 $\times$ 128 of mesh points. The ratio of communication time to computation time of the former is larger than that of the later. Thus, the 64 $\times$ 64 mesh-size PDE generates more traffic than the 256 $\times$ 256 mesh-size PDE.

Executing the PDE on a silent Ethernet is an extreme and unusual example. Most Ethernets are not silent, there is usually other traffic such as X Windows, Network File System (NFS), remote printing, telnet, and file transfer. The performance measured by running the distributed applications over a Ethernet with some degree of background traffic, is closer to the real world. Therefore, we used the same approach as mentioned before to generate background traffic with 30% load. The performances of both PVM and BSD sockets degraded.

Several issues need to be pointed out:

- With a small number of dedicated workstations and a silent Ethernet, distributed network computing over Ethernet can accomplish performance comparable to that of ATM LANs. But without the support from dedicated communication channels like ATM links, the scalability of Ethernet becomes a problem. When the number of processing nodes increases, the performance of distributed programs over local ATM networks are scalable. For example in the case of parallel PDE, each processing node uses four bidirectional links to communicate with four neighbors. The number of links is still fixed when we employ more processing nodes to solve the problem. On the other hand, an Ethernet will saturate quickly when the number of processing nodes increases.
- Some limitations imposed on current implementation of Fore's API. These include a 4 Kbytes maximum transfer size, no concurrent server model support, and machine dependencies.
- In our two previous experiments, we did not include the time for connection management, application topology setup, and resource reservation. Since these tasks are related to the implementation of the communications API.
- We restricted ourselves from using any unique facilities of individual communications APIs. For example the multicasting and barrier synchronization of PVM, and the single-client-multiple-server multicasting model of Fore's API. For a fair performance comparison, the distributed programs used in our experiments only used the common facilities of each API.

5 Conclusions and Future Work

In this paper, we studied the feasibility of carrying out distributed programs over a cluster of workstations interconnected by a local ATM network. The end-to-end performance of several

Table 7: Functional and Efficient Comparison of Four Available APIs

Property	Fore's API	BSD Socket	PVM Interface	Sun RPC/XDR
Communication Model	Message Passing	Message Passing	Message Passing	RPC
Underlying Mechanism	Device Driver	Transport Protocols	Socket	Socket
Maximum Transfer Unit	4 Kbytes	8 Kbytes (UDP) † (TCP)	†	8 Kbytes (UDP) † (TCP)
Protocol Selection	AAL 3/4/5	TCP/UDP	Advise/Normal	TCP/UDP
Send-semantic	Bufferred	†	†	†
Remote Process Spawn	No‡	No‡	Yes	No‡
Concurrent Server	No	Yes	Yes	Yes
Dynamic Process Group	No	No	Yes	No
Data-type Encapsulation	No	No	Yes	Yes
Authentication	No	No	No	Yes
Reliability	No	Protocol-dependent	Yes	Protocol-dependent
Application Complexity	High	High	Low	Medium
Throughput	Good	Fair	Fair	Fair
Latency	Short	Medium	Long	Long

† The property is implementation-dependent.
‡ It can be supported by Unix system calls.

protocol combinations based on four different APIs has been presented. We have also studied the performance speedup for a parallel PDE and a parallel matrix multiplication programs which are executed on four Sun Sparc 2 workstations over a local ATM network. The experimental results demonstrated that executing communication-intensive distributed programs over local ATM networks appears to be very promising.

We have focused our discussion mainly on the communication performance aspect. When designing and implementing distributed programs, many other factors need to be considered.

Table 7 shows both functional and performance comparison of four APIs. Fore's API, BSD Socket interface, and PVM provide a general message passing capability to users, i.e., processes on different machines communicate with each other via send and receive commands. Depending on the underlying communication protocol, a connection should be set up before actual data transmission. Sun RPC/XDR uses a remote procedure call to invoke remote services. Basically, both message passing and RPC can provide similar communication capability. For a comprehensive comparison of message passing and RPC mechanism, refer to Chapter 5.3.14 of Goscinsk's book [1].

Each API discussed in this paper represents a distributed programming environment over a different communication layer in the protocol hierarchy. A distributed program using an API in a lower layer, like Fore's API, can take advantage of better communication performance. However, it usually lacks of distributed programming support which is available in the higher layers. Without distributed programming support from the API, extra effort will be required for users to develop distributed applications. On the other hand, higher layer APIs provide a convenient distributed programming environment with versatile facilities like remote process spawn, process synchronization, and multicasting. The consequence is that some degree of overhead has to be incurred in order to provide this convenient programming environment.

Fore's API provided the best performance among the four APIs studied in this paper. However, because the maximum transfer unit of Fore's API is 4 Kbytes, a user level message segmentation/reassemble is required. The unreliable data transmission of Fore's API forces application programs to process the message loss and retransmission explicitly. The communication interface of Fore's API is similar to that of a socket interface. The current implementation of Fore's API does not support multiple clients communicating to a specific server. This makes it much more complicated to implement a distributed applications with Fore's API than with other APIs.

BSD Sockets are a well-accepted interprocess communication protocol. However, it does not provide machine transparent access. Sun Microsystems has claimed that their RPC library will use the Transport Layer Interface (TLI) API [21] instead of sockets in future operating system releases.

Sun RPC/XDR is suitable for client/server applications such as remote database access, remote file access, and transaction processing. However, it is not clear whether the RPC programming paradigm is good for implementing high performance computing applications over a cluster of networked workstations.

In general PVM provides a higher level programming support than Fore's API. This support includes data-type encapsulation, process group communication, remote process spawn, and dynamic process control. It also introduces more protocol overhead than Fore's API.

In order to provide a user-friendly distributed programming interface and good performance, one possibility is to implement the PVM message passing library using Fore's API. ATM could become a good candidate for providing the kind of communication capability needed by distributed network computing. The multicasting capability of ATM can be utilized by PVM multicasting subroutines. The ATM signaling protocol Q.93B [5] or Fore's SPANS could be used to maintain application specific topologies such as 2-D mesh, hypercube, or tree. PVM over Fore's API has the potential to become an appropriate API for running distributed programs over a local ATM network. One disadvantage of implementing PVM over Fore's ATM API is that Fore's API is not a standard. Porting PVM to an ATM LAN from another vendor would require a significant amount of work. A project to implement PVM over ATM using Fore's API instead of BSD Sockets is currently under investigation. We are also developing a new device driver for Fore's Series-200 host interface to reduce the internal overhead of a stream-based device driver. Another ongoing project is to study and to improve the performance of TCP/IP over local ATM networks.

ACKNOWLEDGEMENT

The authors wish to express their sincere gratitude to Thomas Pfenning at the University of Cologne (Germany), Eric Lampland with Network Services at the University of Minnesota, Timothy J. Salo and Jeffrey Kays at the Minnesota Supercomputer Center Inc., Ronald J. Vetter at the University of North Carolina, and our colleague James Schnepf and Rose Tsang for their valuable comments and support.

References

[1] A. Goscinski. *Distributed Operating Systems: The Logical Design.* Addison-Wesley, 1991.

[2] S.G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, 1989.

[3] ANSI X3T9.3. *Fiber Channel - Physical and Signaling Interface (FC-PH)*, 4.2 edition, November 1993.

[4] A.S. Tanenbaum, R. van Renesse. Distributed Operating System. *Computing Surveys*, 17(4), 1985.

[5] ATM Forum. *ATM User-Network Interface Specification*, 3.0 edition, September 1993.

[6] Bellcore. *Network Compatible ATM for Local Network Applications*, 1.0 edition, April 1992.

[7] J.D. Cavanaugh and T.J. Salo. Internetworking with ATM WANs. In *Advances in Local and Metropolitan Area Networks.* William Stallings, IEEE Computer Society Press, 1994.

[8] Cray Research. *PVM and HeNCE Programmer's Manual*, SR-2501 version 3.0.

[9] J. Crowcroft, I. Wakeman, Z. Wang, and D. Sirovica. Is Layering Harmful? *IEEE Network*, 6(1):20–24, January 1992.

[10] C.T. Lea. What Should Be the Goal for ATM. *IEEE Network*, September 1992.

[11] D.R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3), 1988.

[12] E. Biagioni, E. Coope, and R. Samsom. Designing a Practical ATM LAN. *IEEE Network*, pages 32–39, March 1993.

[13] Fore Systems, Inc. *ForeRunner SBA-200 ATM SBus Adapter User's Manual*, 1993.

[14] G.A. Geist, A.Beguelin, J.Dongarra, W.Jiang, R.Manchek, V.Sunderam. *PVM 3 User's Guide and Reference Manual.* Oak Ridge National Laboratory, May 1993.

[15] Z. Haas. A Protocol Structure for High-Speed Communication over Broadband ISDN. *IEEE Networks*, 5(1):64–70, January 1991.

[16] Intel Insight i960. *Fore Systems 200-Series ATM Adapters for High-Speed Workstations and PCs*, 3rd Quarter 1993.

[17] M. Zitterbart. High-Speed Transport Components. *IEEE Network*, 5(1):54–63, January 1991.

[18] Minnesota Supercomputer Center Inc. *An Overview of the MAGIC Project*, 1993.

[19] Request for Comment 1483. *Multiprotocol Encapsulation over ATM Adaptation Layer 5*, July 1993.

[20] S.Leffler, M.McKusick, M.Karels, J.Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System.* Addison-Wesley, 1990.

[21] Sun Microsystems. *Network Programming Guide*, March 1990.

[22] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *The 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[23] C.A. Thekkath and H.M. Levy. Limits to Low-Latency Communication on High-Speed Networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[24] C.A. Thekkath, H.M. Levy, and E.D. Lazowska. Efficient Support for Multicomputing on ATM Networks. Technical Report TR 93-04-03, Department of Computer Science and Engineering, University of Washington, April 1993.

[25] Alec Wolman, Geoff Voelker, and Chandramohan A. Thekkath. Latency Analysis of TCP on an ATM Network. Technical Report TR 93-03-03, Department of Computer Science and Engineering, University of Washington, March 1993.