# Pupa: A Low-Latency Communication System for Fast Ethernet

Manish Verma* Tzi-cker Chiueh⋆

∗ Silicon Graphics Inc.
**mverma@engr.sgi.com**

⋆ Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
**chiueh@cs.sunysb.edu**

**Abstract**

*Pupa* is a low-latency communication system that provides the same quality of message delivery as TCP but is designed specifically for a parallel computing cluster connected by a 100 Mbits/sec Fast Ethernet. The implementation has been operational for over a year, and several systems have been built on top of Pupa, including a compiler-directed distributed shared virtual memory system, *Locust* , and a parallel multimedia index server, *PAMIS* . To minimize buffer management overhead, *Pupa* uses per-sender/per-receiver fixed-sized FIFO buffers to optimize for the common case, rather than a shared variable-length linked-list buffer pool. In addition, *Pupa* features a *sender-controlled* acknowledgement and an optimistic flow control scheme to reduce the overhead of providing reliable in-order message delivery. Our performance results show that *Pupa* is twice as fast as the fast path of TCP in terms of latency, and is about 1.5 times better in terms of throughput. This paper presents the design decisions during the development of *Pupa*, the results of a detailed performance study of the *Pupa* prototype, as well as the implementation experiences from *Pupa*-based applications development.

## 1 Introduction

*Pupa* is a communication system specifically designed to turn a NOW (Network of Workstations) into a parallel computing platform by reducing the message latency to the minimum. It is currently implemented in the FreeBSD Unix kernel, and the hardware platform of the prototype is a set of Pentium PCs connected by a 100 Mbits/sec Fast Ethernet. *Pupa* is positioned in parallel with the TCP/IP stack, such that existing workstation applications can still work without modification, while parallel programs developed on the PC cluster can enjoy low-latency message passing.

Many of previous communication systems for NOWs [17] [13] [14] [16] [1] made the assumption that the network hardware is directly accessible from the user mode and/or there are no other communication systems in operation on the hosts so that they can completely bypass existing device drivers. This eliminates most of the OS and device driver overheads. *Pupa* , on the other hand, is designed to work with off-the-self network cards and to co-exist with legacy communication software such as TCP/IP. Many earlier systems also only implemented a rudimentary buffer management scheme and expect either higher level systems or user applications to implement full fledged buffer management services. This artificially reduces the reported message latency measurements. However, the overheads associated with more advanced services such as reliability reappear once they are supported by the higher layer systems. Some of the systems rely on the underlying hardware to provide reliability guarantee to reduce protocol processing overheads. Yet other systems rely on smart network adapters to reduce the number of data copy operations at the receiver. *Pupa* does not have this luxury since it is designed to work with off-the-shelf Ethernet cards. Another that is similar in goal to *Pupa* is *Beowulf* [2], which focuses mainly on the use of multiple Ethernet links for higher throughput, but not on optimizing the message passing software. It uses Linux's communication subsystem and existing MPI or PVM implementations for message delivery. Some other implementations [19] [15] that in many ways are similar in architecture to *Pupa* and have been implemented over off-the-shelf network hardware have achieved even better results in terms of latency and throughput. We believe there is further scope for optimizing the *Pupa* implementation.

Fundamentally, a communication system for parallel computing is required to provide a reliable in-order delivery of messages from one process on one machine to another process on another machine. The studies performed by Karamcheti, Chien and Plevyak [4] [9] [10] [14] demonstrated that the support for reliable and ordered delivery is best provided at the lowest level of the communication layer. Parallel programs require low latency for small message and high bandwidth for large messages Small messages are often generated by time critical operations such as barrier synchronization, reduction operations, request for locks, etc. Small message latency is reduced by optimizing the fast path of the message transfer routine. Large messages are primarily due to bulk data transfers, whose transfer time is mainly determined by the effective bandwidth. Previous studies on the performance problems associated with TCP/IP [7] [11] found that it is not protocol processing, but the overheads associated with buffer management, data copying, checksumming, and operating systems supports such as interrupt scheduling, context switch, timer management etc. that causes the performance problem.

To optimize for both low latency and high bandwidth, we base *Pupa*'s design on the following assumptions: the physical network hardware is relatively reliable and therefore software error checksumming can be eliminated, the main memory capacity of each cluster node is comparatively large and can be traded for high performance, and the Ethernet architecture doesn't allow multipath routing between any two network nodes and therefore packets rarely arrive out of order at the receiver. In particular, *Pupa* chooses a simple and fast receive message buffering scheme that is optimized for the common case: small messages arriving in order. The buffering scheme uses a FIFO queue, each entry of which is of fixed size that can accommodate most small messages. Therefore both allocation and deallocation take only a pointer update. When a message arrives out of order, it is immediately dropped and a negative acknowledgment is sent to the sender to request for re-transmission. In the case that received messages are too large to fit into the FIFO queue entry, they are allocated from a shared secondary buffer, with its pointer stored into the corresponding FIFO queue entry.

Other techniques used in *Pupa* include an optimistic flow control policy to maximize the network throughput, and a sender-controlled acknowledgment scheme to reduce both the number of control messages and the interrupt and context switch overheads. *Pupa* also exploits Ethernet's broadcast mechanism to support multicast and broadcast communication primitives efficiently.

## 2 Software Architecture

Figure 1 shows how *Pupa* interacts with the device driver and user applications. *Pupa* exposes different programming interfaces to message passing programs and shared memory programs. The architecture cleanly separates interface-dependent functionalities from interface-independent functionalities.

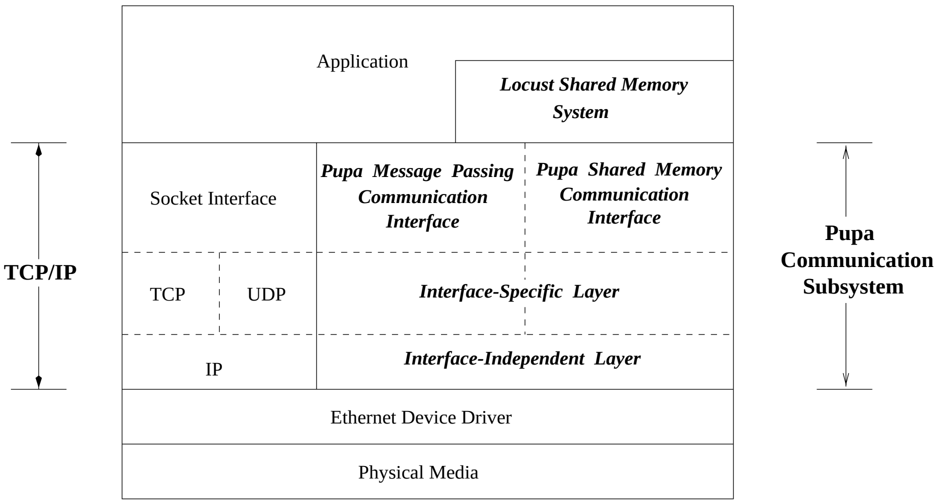

Application
Locust Shared Memory System
Socket Interface
Pupa Message Passing Communication Interface
Pupa Shared Memory Communication Interface
TCP/IP
TCP
UDP
Interface-Specific Layer
IP
Interface-Independent Layer
Pupa Communication Subsystem
Ethernet Device Driver
Physical Media

Figure 1: *The organization of Pupa with respect to the network device driver and user applications.*

## 2.1 Buffer Management

Because previous studies have shown that the efficiency of buffer management dictates the performance of the message passing system, *Pupa*'s buffer management scheme is highly optimized by fast-pathing the common case: small messages that arrive error-free and in order. Each node participating in a parallel computation has a set of *primary buffers*, some of which are *primary send buffers* and others are *primary receive buffers*. These buffers are not system wide buffers but are allocated to each parallel program at the start and deallocated when the program terminates. Primary buffers are FIFO queues with fixed size entries and are used to store the message headers of the messages that are in transit. These buffers also hold the bodies of small messages. In addition, there are two *secondary buffer pools*, again allocated to each program, from which large message bodies are stored, one for outgoing messages and one for incoming messages.

Each node participating in a parallel computation has one primary send buffer for every other node, which stores the headers for messages destined to that particular node. The organization of the primary receive buffers, however, is dependent on the high-level system that builds on top of *Pupa* . For a message passing subsystem, each node has one primary receive buffer for every other node and stores the headers of messages arriving from that particular node. In a shared memory subsystem the receive buffers are organized according to generation numbers [5] [18]. In this paper, we will focus only on the message passing system.

When arriving messages are small, free of errors and in order, they are inserted in to the receive FIFO queue associated with the sender. Sending a small message involves nothing but pushing the message to the receiver's send FIFO queue. Consequently, the buffer management overhead associated with sending or receiving messages can be as little as one pointer update. More buffer management work is required in the case of large messages or messages that arrive out of order.

Primary and secondary buffers are preallocated at the beginning of a parallel program. These buffers are mapped into both kernel and user address spaces and are pinned in the physical memory. Buffers are mapped into both address spaces so that they can be accessed from both user and kernel modes of execution directly. Pinning buffers in memory is necessary because the buffers need to be manipulated by network interrupt handlers when a message or an acknowledgment arrives.

### 2.1.1 Primary Buffers

Primary buffers store the headers of the messages in transit, and are organized as fixed size circular FIFO queues. There are four pointers *low, high, next* and *sent* for each primary send buffer to keep track of the status of the messages in the buffers:

*low* points to the beginning of the first message in the send buffer.

*high* points to the end of the last message in the send buffer

*next* points to the beginning of the next message to be transmitted,

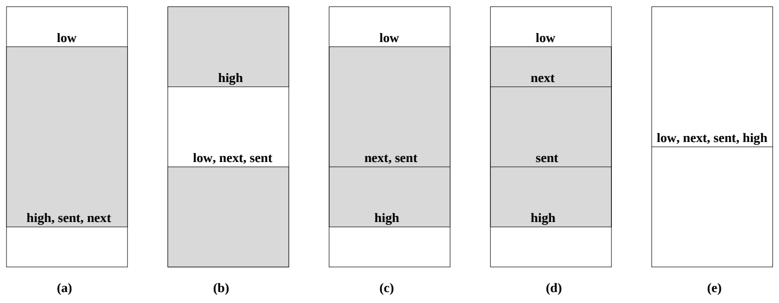
*sent* points to end of the last message that has been transmitted at least once.

The *high* pointer is incremented by the library after it inserts a message to the send buffer. The *low* pointer is advanced by the interrupt handler when a message previously sent is acknowledged, thus indicating to the user library that more space has become free in the send buffer. The library is not allowed to modify this pointer. The *next* pointer points to the next message to be transmitted. The *sent* pointer is used by the acknowledgment processing routine to determine if an arriving acknowledgment actually acknowledges a valid message. The *sent* and *next* pointers are updated when a send call is made or when a message is re-transmitted after a timeout. The *next* pointer can also be updated in an acknowledgment processing routine called by the interrupt handler if an acknowledgment is received for a message that was marked to be retransmitted. The *next* and *sent* pointers are maintained by the kernel and are always updated in critical sections where network interrupts are disabled. Figure 2 shows several possible states of the primary send buffer. Similarly, two pointers *low* and *high* are used to keep track of the state of messages in each primary receive buffer.

*low* points to the beginning of the first message in the buffer.

*high* points to the end of the last message in the buffer.

The area between the *low* and *high* pointers holds the headers of the messages that have been received by *Pupa* but have not yet been retrieved by the user application. The *high* pointer is advanced by the receive network interrupt handler when it adds a message to the receive buffer, thus indicating to the user library the availability of more data. The library increments the *low* pointer when it extracts a message from the receive buffer, thus indicating to the interrupt handler of the availability of more free space in the buffer.



(a) : All messages in the buffer have been sent once, waiting for acknowledgements
(b)  There are several messages in the buffer yet to be sent. No unacknowledged messages.
(c) : Some sent (unacknowledged) and some unsent messages
(d) : Some sent and some unsent messages. Retrasmitting previously sent messages.
(e) : No messages in the send buffer.

Figure 2: *Different states of a primary send buffer. Shaded regions represent the parts of the buffer that hold messages that are either yet to be transmitted or that have been transmitted but have not yet been acknowledged.*

When a primary buffer pointer reaches the end of the buffer, it wraps around to the beginning of the buffer. The system maintains a separate flag for each buffer to distinguish between the completely full and completely free buffers.

In both primary send and receive buffers, each of the *low* and *high* pointers is modified exclusively by either the network interrupt handler or the user library but not by both. As a result, the user library can update send messages to or receive messages from the primary buffers without entering any critical section to disable network interrupts. This make it possible for applications to receive messages without crossing the user-kernel boundary.

In the current implementation each primary buffer along with the above mentioned pointers and other metadata takes 4Kbytes, out of which 4064 bytes are used for storing messages and 32 bytes are used to maintain the state information.

### 2.1.2 Secondary Buffer Pools

Secondary buffers are used to store large messages that can not fit into the FIFO queues directly. Each node participating in a parallel program maintains two secondary buffer pools - a send and a receive pool. Buffers from these pools are shared among all nodes, and are allocated on demand. *Pupa* keeps track of free buffers from each pool with a *free list*. The free lists are accessible by both the kernel and the user library. Send buffers are allocated from the send pool by the user library during a send operation and are returned back to the pool by the network interrupt handler when the message held in that buffer is acknowledged. Receive buffers are allocated from the receive pool by the network interrupt handler when a packet arrives at the receiver and are returned back to the pool by the user library when the user process consumes the message. Allocating a buffer from a pool involves removing the buffer from the corresponding free list and returning the buffer to the pool involves adding that buffer to the corresponding free list. *Pupa* keeps at least one buffer in the free list at all times. Since at any time there can be at most one thread adding a buffer to the a free list and at most one thread removing a buffer from the same free list, it is guaranteed that allocating and returning buffers from and to the secondary buffer pools can be done without entering a critical section.

The number of buffers in each pool and size of each buffer is a tunable system parameter that can be set at the beginning of the program and they do not change during a particular run. By default the system allocates 128 send buffers each 512 bytes in size and 512 receive buffers each 128 bytes in size.

### 2.1.3 Use of *mbufs* at Receiver End

The SMC EtherPower 10/100 card used in the prototype does not have any on board receive buffers. Packets arriving from the network are transferred directly into the main memory using DMA *before* interrupting the CPU. The buffers in which the packet would be transferred must be preallocated and registered with the card. The original driver for this card uses preallocated 2Kbyte *mbufs* for this purpose. The *mbufs* are then passed on to the upper network layers such as IP. *Pupa* uses the same organization for simplicity. We have modified the device driver so that a *Pupa* packet is passed to the RC layer instead of IP. If the messages in the packet pass the acceptability tests – such as they are not duplicates and have not arrived out-of-order – message headers are copied into the primary receive buffers. Small message bodies are copied into the primary buffers also. We have chosen 32 bytes as the cutoff length – the choice is rather arbitrary and is driven by the goal of reserving approximately 50% of the primary buffer space for message headers which are 28 bytes in length. Larger message bodies are either retained in the *mbufs* or are copied into buffers allocated from the secondary buffer pools. Since the message headers are copied into the right primary buffers in all cases, the user program finds the message of interest at the head of the expected primary buffer even when the message body actually resides in an *mbuf* or in the secondary buffer pool.

Retaining the message body in the *mbuf* has the advantage that it avoids an additional data copy and the secondary buffer allocation overhead. But messages stored in the *mbuf* can only be accessed through a system call, whereas messages stored in the secondary buffers are directly retrievable from the user level. For relatively small message bodies, the copy overhead is not too large and is compensated for to some extent by the elimination of the system call overhead otherwise required to access an *mbuf*. The *mbufs* used to receive a packet are 2Kbytes and occupy physical memory. Using a large *mbuf* to store modest sized messages is a waste of physical memory and hence we have chosen to copy message bodies smaller than a threshold, the default being 128 bytes, to secondary buffers and then free the *mbufs*. The threshold is again a tunable parameter.

In summary, when the user program sends a message, it makes a call to the *Pupa* library, which creates a message header and stores it in the primary buffer associated with the destination node. If the message body is small (<32 bytes) the message body is also copied in the primary buffer. Otherwise, a list of secondary send buffers are allocated and the message body is copied into them. The library then makes a system call to send the message. The device driver reads the message from the buffers and packs them into an Ethernet packet and hands the packet to the card.

When the packet arrives at the receiver, the card DMAs it into a 2Kbyte *mbuf* pre-registered with the card by the device driver. The RC layer examines the message header and copies it into the appropriate primary receive buffer. If the message body is small (<32 bytes), it is also copied into the primary buffer after the header. Otherwise, if the message body is larger than 32 bytes and smaller than or equal to 128 bytes, a secondary buffer is allocated from the secondary buffer pool and the message body is stored there with a pointer to it in the primary buffer. If the message body is larger than 128 bytes, it is retained in the *mbuf* and a pointer to it is stored in the primary buffer. When the user program makes a receive library call the message is copied from the *Pupa* buffers to the user address space and the buffers are freed.

## 2.2 Reliability Guarantee

Reliability guarantee is provided by the RC layer through sequence numbers, positive and negative acknowledgments and retransmissions. The RC layer at the sender attaches a sequence number to each message transmitted, which is examined by the receiver to detect lost messages. Each node maintains an *expected sequence number* for every other node to determine which sequence number to expect from that node next. Senders are notified of acceptance of messages through acknowledgments. To conform with the FIFO buffer management scheme, receivers accept only in-order messages. When messages are dropped at the receiver or when they are lost in the network, the RC layer at the sender times out and retransmits them.

### 2.2.1 Acknowledgment Management

Acknowledgments are required to provide a reliable data stream. Every message sent by a node is retained in the send buffers at that node until the receiver acknowledges the receipt of that message. If a message is not acknowledged within a certain interval, the sender retransmits it assuming that the previous transmission was lost in the network or dropped by the receiver. To maintain a constant stream of data between a pair of nodes, it is necessary that acknowledgments arrive fast enough so that the message pipeline is not stalled due to send buffer overflow. However, each acknowledgment sent has its own cost - the receiver needs to assemble the acknowledgment packet and transmit it, and the sender has to process the acknowledgment packet on its receipt. If the receiver sends the acknowledgment before delivering the packet to the application program, the time to assemble the acknowledgment packet and to send it is added to the critical path of message transfer. In addition, acknowledgment packets consume network bandwidth without actually transferring useful data.

A widely used solution to reduce acknowledgment overhead is to *piggy-back* acknowledgments with outgoing messages. This solution works well when the communication traffic volume is symmetric in both directions between each pair of nodes. In the case of largely one-way data transfers, *piggy-back* acknowledgments alone cannot provide good performance. Usually the *piggy-back* acknowledgment scheme is augmented with timer controlled acknowledgments at the receiver end [12]. A timer expires at regular intervals and acknowledgments are sent on all live connections. Although, this protocol alleviates some of the deficiencies of the vanilla piggy-backed scheme, it is not responsive enough in the sense that it is up to the receiver to decide the right time to send acknowledgments. Since the receiver does not have an idea of when the sender actually needs an acknowledgment, it might end up sending too many acknowledgments or it might not be able to send acknowledgments immediately when the send buffer overflows.

*Pupa* augments the piggy-back scheme with *sender-controlled acknowledgments*. When the amount of free space in a primary sends buffer falls below a low water mark, the sender enters an *acknowledgment required* state with respect to that receiver. Every subsequent message is sent with an *acknowledgment requested* flag on until the arrival of an acknowledgment that lifts the free space in the send buffer above a high water mark. On receipt of this acknowledgment, the sender leaves the *acknowledgment required* state with respect to that receiver. When a message with the *acknowledgment requested* flag arrives at a receiver, the receiver sends an acknowledgment immediately. This acknowledgment informs the sender of the successful delivery of all messages accepted so far. When the number of free buffers in the send secondary buffer pool falls below a certain low water mark, acknowledgment requests are sent to all receivers from which acknowledgments are pending for messages stored in the secondary buffer pool.

When an arriving message is rejected at the receiver because of buffer overflow or due to out-of-order arrival, the receiver sends a *negative acknowledgment* to notify the sender. The negative acknowledgment carries the sequence number of the next message that the receiver expects from that sender.

To accommodate the case of lost acknowledgments and lost acknowledgment requests, an acknowledgment timer is also implemented. The timer expires every 10ms. Every time this timer expires all pending acknowledgment request and acknowledgments are sent to the respective nodes. After every 50 timer expiration intervals, each node sends broadcast acknowledgments to all nodes participating in the computation to refresh acknowledgment pending and acknowledgment required states.

In summary, acknowledgments are sent to the sender in one of the following five ways.

- Every data packet carries a piggy-backed acknowledgment.
- Whenever a node receives a packet with the *acknowledgment requested* flag on, an acknowledgment is sent immediately.
- When a message is dropped due to a buffer overflow or when an out-of-order message arrives at a node, a negative acknowledgment is sent immediately.
- After every 10ms an acknowledgment timer expires and all pending acknowledgment requests and acknowledgments are sent.
- Broadcast acknowledgments to all nodes are also sent after every 50 timer expirations.

### 2.2.2 Flow Control

The purpose of flow control is to control traffic between pairs of nodes so that senders do not overwhelm the receivers. A receiver might temporarily run out of buffers and may not be able to accept any further messages from the sender. Flow control is also needed to ensure that senders do not flood the network with messages that will be dropped anyway. A flow control scheme that allows senders to transmit only when it has definite information about availability of buffers at the receiver avoids wasting bandwidth. However, it is overly conservative and might result in reduced network throughput if it cannot keep the data transfer pipeline full at all time.

*Pupa* implements an *optimistic* flow control scheme. Each node has a *send credit* associated with it for every other node participating in the parallel computation. The send credit specifies the number of messages that the sender can send to the receiver at any point of time. Every time a message is sent this credit is decremented by one. Initially each node gives an unlimited credit to every other sender node, essentially turning off flow control. When a primary buffer overflow occurs at a receiver, a credit update with available credit of zero is sent to the sender. From this point on the sender will not transmit any data destined to that receiver until it receives a subsequent positive credit update. When the receive buffers become free again, the receiver sends credit updates based on the amount of free space available in the primary buffers to the sender. Once the free space in the primary receive buffers rises above a high water mark, the receiver sends an unlimited credit update to the sender. Credit updates are sent with all positive and negative acknowledgments. Special credit update packets are also sent when credit information for a particular sender changes but there are no acknowledgments due for that sender.

This flow control policy allows senders to send data at the maximum rate possible as long as no buffer overflows occur. Only when a buffer overflow does occur, will the overheads of flow control become visible. Additionally, users have the option of turning flow control off completely. This option is useful in a high bandwidth network with no other applications running so there is not much to be gained by trying to reduce redundant network traffic.

*Pupa* also takes advantage of the large MTU of the Ethernet hardware and packs multiple messages into one packet whenever possible. This is accomplished by not sending small messages as soon as they are added to the send buffers, but waiting till a sufficient number of messages are accumulated in the buffers. The result is fewer packets on the network and hence less contention for the shared media and fewer collisions. The disadvantage is of course longer message latency. An exception to this policy is made when the user specifies a *fastpath* option when adding the message to the send buffers.

## 3 Performance Evaluation

Our experimental setup consists a network of PCs connected by a single segment Fast Ethernet. Each PC has a P90 CPU, 16 Mbytes of main memory and 256 Kbytes of external cache. The PCs are connected to the network by SMC EtherPower 10/100 adapters which are based on DEC 21140 chipsets. The hub used to connect the network is a SMC TigerHub 100. The PCs run FreeBSD 2.1.0 [8] which is an x86 implementation of 4.4 BSD UNIX operating system [3]. The kernel has been modified to include the *Pupa* module. There are 12 PCs on the network which have been used for the experiments presented in this chapter. Another similar PC on the same network serves as the NFS server for all the machines involved in the experiments. During the experiments there were no other computation or communication activities on the nodes or in the network. UNIX daemons.

To put the performance of *Pupa* in perspective, we have also compared the latency and bandwidth characteristics of *Pupa* with those of TCP on the same platform. We do not perform any comparisons with UDP because while both TCP and *Pupa* are reliable message delivery systems, UDP does not provide any reliability guarantee. Reliable message delivery is important for parallel programming and UDP without any reliability layer built on top it does not qualify as an usable programming platform.

### 3.1 Latency Measurements

The latency for messages to be sent from one user process at one node to another user process at another node, and its breakdowns, are presented in Table 1. All timing measurements are in microseconds. The library routine `gettimeofday` and the kernel routine `microtime` were inserted along the message transfer path to perform the timing measurements. `Gettimeofday` and `microtime` take $6\mu s$ and $2\mu s$ respectively. The presented numbers have been adjusted for these values.

Table 1 presents the total one-way latency of messages of different sizes for *Pupa* and TCP. For small

| Message size (bytes) | 4 | 32 | 64 | 128 | 256 | 512 | 1024 | 1436 |
|---|---|---|---|---|---|---|---|---|
| 1. Sender end delay | 77 | 89 | 93 | 95 | 102 | 109 | 126 | 140 |
| 2. Receiver end delay | 101 | 106 | 122 | 126 | 119 | 129 | 147 | 165 |
| 3. Transimission & DMA | 20 | 23 | 26 | 32 | 47 | 80 | 137 | 193 |
| **4. Pupa one way delay** | **198** | **208** | **241** | **253** | **268** | **318** | **410** | **492** |
| **5. TCP one way delay** | **405** | **411** | **422** | **438** | **451** | **482** | **564** | **645** |

Table 1: *Breakdown of one-way latency for messages of different sizes in* Pupa *and its comparison with TCP latencies with the NO_DELAY option. Message sizes are in bytes and times are in $\mu$secs.*

messages *Pupa* 's latency is less than half of TCP's latencies[1]. Table 1 also presents the breakdown of *Pupa* latencies into three main components, viz., the sender side delay, the receiver side delay and the transmission and DMA overhead. The sender side delay refers to the time between the user making a *Pupa* library call to send a message and the device driver handing the packet containing the message over to the network card. The receiver side delay represents the time from the invocation of the Ethernet device driver at the receiver until the data in the packet is available to the user program. The time listed under *transmission and DMA* includes the time to transmit the message, the propagation delay, the time to DMA the packet into memory at the receiver, the time taken by the Ethernet card at the receiver to interrupt the CPU, the context switch overhead and the time to invoke the interrupt handler. The total one-way delay was computed as half of the measured round-trip delay. The sender end delay and the receiver end delay were measured along the message transfer path as the messages traversed it. The *transmission and DMA* time was computed as the rest of the one-way delay not accounted for by the sender end delay and the receiver end delay.

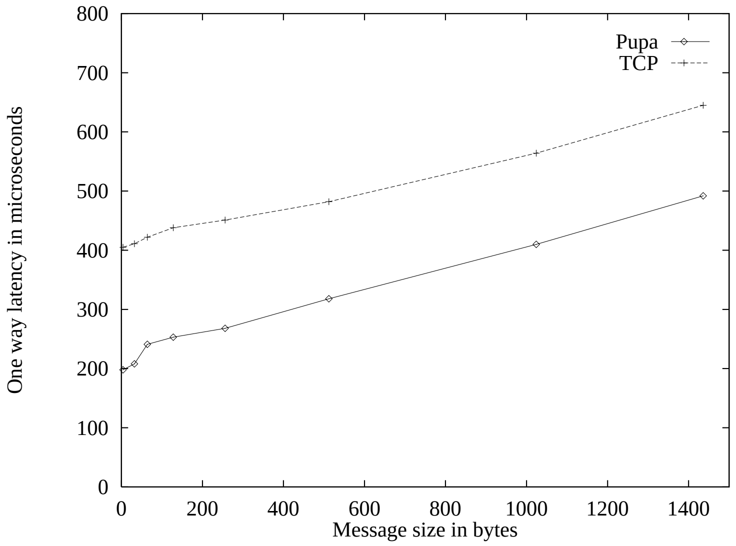


Figure 3: Pupa *and TCP one-way latencies as functions of message size.*

The *transmission and DMA* component is miniscule for small messages. However, it becomes the dominant component for larger messages. As a result, use of faster networks and faster memory buses is expected to reduce the latencies of large messages. However, it will have only a small impact on the latencies seen by smaller messages. The software overhead at the two ends is significant for both small and large messages. Use of faster processors will reduce the software overhead and benefit both small and large messages equally. Since the *transmission and DMA* component is common to both *Pupa* and TCP, the difference between the latencies in the two cases comes from the software overheads at the two ends. Consequently, faster networks

[1]The numbers for TCP connections were obtained with the TCP_NODELAY option on. The TCP_NODELAY option disables the Nagle's optimization. Nagle's optimization makes TCP wait for a minimum number of bytes to be available in the send queue before attempting to transmit.

are going to increase the gap between the relative latencies of the two system, whereas faster processors will bring them closer. A graphical representation of the comparative *Pupa* and TCP latencies for various message sizes is shown in Figure 3. We observe that initially the gap between the *Pupa* and TCP latencies decreases as the message size increases. The main reason for this behavior is that as the message size increases, *Pupa* adopts a more complicated buffer management policy.

A detailed delay breakdown of one way message latency is shown in Table 2. to paragraph. The numbers in parentheses are the latency components expressed as percentages of the total one way latency.

| Message size (bytes) | 4 | 32 | 64 | 128 | 256 | 512 | 1024 | 1436 |
|---|---|---|---|---|---|---|---|---|
| 1. Protocol processing | 46 (23%) | 46 (22%) | 46 (19%) | 46 (18%) | 46 (17%) | 46 (14%) | 46 (11%) | 46 ( 9%) |
| 2. OS and driver overhead | 53 (27%) | 53 (25%) | 53 (22%) | 53 (21%) | 59 (22%) | 59(19%) | 59 (14%) | 59 (12%) |
| 3. Transmission & DMA | 20 (10%) | 23 (11%) | 26 (11%) | 32 (13%) | 47 (18%) | 80 (25%) | 137 (34%) | 193 (39%) |
| 4. Data copy | 4 ( 2%) | 9 ( 5%) | 14 ( 6%) | 20 ( 8%) | 22 ( 8%) | 40 (13%) | 74 (18%) | 101 (21%) |
| 5. Buffer and header management | 75 (38%) | 77 (37%) | 102 (42%) | 102 (40%) | 94 (35%) | 93 (29%) | 94 (23%) | 93 (19%) |
| **6. Total** | **198** | **208** | **241** | **253** | **268** | **318** | **410** | **492** |

Table 2: *A more high level breakdown of one way latency for messages of different sizes in* Pupa . *Message sizes are in bytes and times are in microseconds. Numbers in parentheses are the percentages of the latencies accounted for by the corresponding components.*

It is clear that OS and device driver overhead is one of most dominant costs for small messages. Even for the largest message (1436 bytes), they account for as much as 12% of the total latency. As long as the network of computers are based on off-the-shelf hardware and are also expected to support general purpose time shared computing, reducing this delay component is a difficult problem. The overheads associated with kernel-user boundary crossing, interrupt scheduling, device driver overheads etc. can not be eliminated. The protocol processing overhead, which in itself is quite significant for small messages, is less than the OS and device driver overheads.

The largest part of the small message latency is contributed by the buffer management and header manipulation overhead. There is a significant difference between the buffer management overhead incurred by small messages and that incurred by large messages. This difference supports the design decision of keeping the buffer management simple for small messages and by keeping the message bodies in primary buffers. The buffer management overhead for messages of sizes 64 bytes and 128 bytes is larger than those for messages larger than 256 bytes. This is because in the current implementation 64 and 128 byte messages are copied into the secondary buffers whereas larger messages are retained in the *mbufs* where they have been put by the card.

Our measurements show that data copy overheads are insignificant for small messages but they account for up to 21% of the total overhead for large messages. Currently our implementation requires a data copy between user program buffers and the *Pupa* buffers at each end. The transmission and DMA overheads are strictly dependent on hardware performance and communication software has no control over them. Our measurements show that while they constitute only a minor part of the total overhead for small messages, they become the most dominant overhead component for larger messages. Faster networks and faster memories will bring this overhead down. But since CPU speed is expected to increase at a faster rate than network and memory speed, this component is expected to continue to dominate the latency for larger messages.

## 3.2 Bandwidth Measurements

In this section we present the bandwidth characteristics of *Pupa* as compared to the bandwidth characteristics of TCP. We have measured the maximum bandwidth achieved between a pair of user programs on a pair of nodes communicating with each other using messages of varying sizes. For each message size, a constant stream of 100000 messages was sent from a user program at one node to another user program at another node. Total number of bytes transferred were divided by the time between the arrival of the first message and the arrival of the last message at the receiver to compute the realized bandwidth. The TCP bandwidth was measured using 16 Kbyte send and receive socket buffers which are the default parameters on our system. Increasing the socket buffer sizes did not have any impact on the realized bandwidth. There was no other network activity while these experiments were conducted. Experiments with *Pupa* were conducted in two modes. In one mode every message was individually acknowledged. In the other mode the receiver sent acknowledgments only when the sender requested it or when the acknowledgment timer expired. Note that since data communication was only in one direction there were no piggy-back acknowledgments. The bandwidth achieved for each message size for three different experiments are plotted in Figure 4. *Pupa* can achieve up to 62Mbps bandwidth while operating in the sender-controlled acknowledgment mode. In contrast, TCP can achieve only up to 44Mbps.

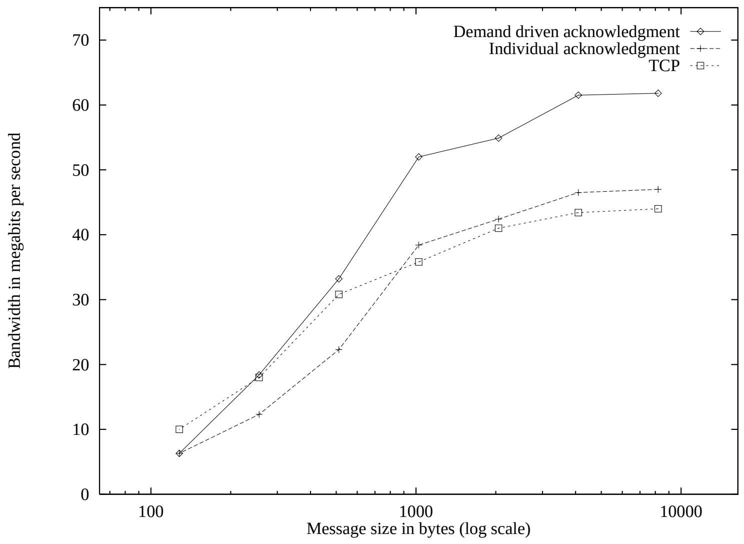


Figure 4: *Bandwidth as a function of message sizes.*

As expected, the sustained bandwidth initially increases in all three cases with increasing message size. As message size increases the fixed per-message overhead is amortized over a larger number of data bytes, and thus leading to better performance. The sustained bandwidth saturates when the message size reaches 4 Kbytes, because larger messages are split into message fragments less than or equal to 1500 bytes (Ethernet MTU) before transmission. The throughput difference between 2 Kbyte packets and 4 Kbyte packets in the case of *Pupa* is much higher than that in the case of TCP. This is because *Pupa* is a message boundary preserving system and it splits each message into sizes manageable by Ethernet individually. As a result with 2 Kbyte messages every alternate Ethernet packet carries only a small amount of actual data. More concretely, a 2 Kbyte message is sent in two Ethernet packets: one carries 1436 and the other 612 bytes. On the other hand, TCP is a byte stream protocol and coalesces all pending messages together and thus can pack as much real data into each Ethernet packet as available. Hence TCP bandwidth reaches close to peak bandwidth with 2 Kbyte packets. However, even with TCP bandwidth being close to its peak, both *Pupa* modes achieve better bandwidth than TCP with 2 Kbyte messages. The peak for *Pupa* is reached with a message size of 4 Kbytes.

*Pupa* in both modes lags behind TCP when messages of size less than 256 bytes are used. This is because TCP is a byte stream protocol and can coalesces all available data into one unit before sending. However, *Pupa* is a message boundary preserving protocol and it must incur per message processing overhead for every message sent even if some of the messages are transmitted in the same packet. As message size increases, TCP loses this advantage and *Pupa* catches up with it.

The performance impact due to acknowledgments is shown through the performance difference between the two *Pupa* modes. The *Pupa* mode with individual acknowledgments lags behind even TCP in performance until message size becomes 1 Kbytes. This is because *Pupa* in the individual acknowledgment mode sends more acknowledgments than TCP. But as the message size increases, *Pupa* sends fewer and fewer acknowledgments whereas the number of acknowledgments for TCP remains more or less the same, and the bandwidth gap closes down. Eventually *Pupa* with individual acknowledgments overtakes TCP because of reduced per-message processing overhead.

## 3.3 Message Passing Programs Statistics on *Pupa*

In this subsection we present statistics gathered from production runs of four parallel program to examine the validity of certain design decisions made during the development of *Pupa* and to investigate ways of further improving *Pupa* . The main focus is on the performance tradeoff of FIFO buffer management, optimistic flow control, and sender-controlled acknowledgement.

The four programs were Laplace equation solver (LES), Gaussian Elimination (GEL), Long range particle dynamics simulation (LRS), and Short range particle dynamics simulation (SRS). They are running on a 10-node *Pupa* cluster. The results of these measurements have been presented in Table 3. The communication statistics presented here are approximate as they were extracted from the kernel just before the termination of the programs when the connections were not yet closed.

| **Program** | **LES** | | **LRS** | | **SRS** | | **GEL** | |
|---|---|---|---|---|---|---|---|---|
| **Comm. buffer size** | 64K | 48K | 64K | 48K | 64K | 48K | 64K | 48K |
| 1. Packets sent | 91008 | 91755 | 76771 | 105371 | 258176 | 261659 | 448811 | 492081 |
| 2. Packets lost | 0 | 0 | 5 | 33 | 4 | 10 | 91 | 150 |
| 3. Ack packets sent | 29173 | 29921 | 35434 | 61470 | 63115 | 66242 | 209998 | 249863 |
| 4. Ack packets lost | 0 | 0 | 5 | 9 | 4 | 8 | 26 | 53 |
| 5. Total valid acks | 32590 | 30198 | 16593 | 19888 | 77964 | 77443 | 165779 | 171804 |
| 6. Total valid nacks | 0 | 0 | 11 | 23 | 35 | 119 | 486 | 578 |
| 7. Messages sent | 61557 | 61557 | 38377 | 38389 | 194742 | 194846 | 231704 | 231721 |
| 8. Messages rexmitted | 0 | 0 | 20 | 32 | 104 | 156 | 617 | 709 |
| 9. Messages received | 75130 | 75130 | 41145 | 41153 | 210153 | 210220 | 369819 | 368908 |
| 10. Overflow count | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 11. Out-of-order count | 0 | 0 | 15 | 23 | 92 | 139 | 541 | 615 |
| 12. Message in primary | 28120 | 28120 | 4120 | 4120 | 23161 | 23131 | 202343 | 201358 |
| 13. Message in secondary | 0 | 0 | 0 | 0 | 129291 | 129167 | 14390 | 14390 |
| 14. Message in *mbuf* | 47010 | 47010 | 37010 | 37010 | 57659 | 57783 | 151106 | 151106 |

Table 3: *Communication statistics for the four parallel programs.*

### 3.3.1 Lost Packets and Acknowledgments

First we ran the programs with the default communication system parameters. The measurements from these runs are presented in the first two columns for each program. The first four rows in the table present statistics about the number of packets put on the network and the number of packets carrying only acknowledgments. The assumption about packets not getting lost in the network is validated, at least for these applications. Among the packets that are lost, the fraction of acknowledgment packets is rather high. Despite implementing a sender-controlled acknowledgment scheme, the number of packets carrying only acknowledgments is still quite high. In the worst case (the LRS application) half of the packets carry only acknowledgments, and in the best case (the SRS application) one fourth of the packets carry only acknowledgments. The large number of acknowledgments are one of the reasons for the dropped packets due to excessive collisions on the Ethernet.

The cause for this behavior is that the secondary send buffer pool of 64 KBytes at each node is not large enough to accommodate outgoing messages destined to all other nodes. Consequently, this buffer is always close to overflow and the senders keep on sending acknowledgment requests to the receivers. This explains why the fraction of packets carrying only acknowledgments is the highest for the LRS application, since in this application every node exchanges its entire data set with every other node in every time step. The kernel within which *Pupa* is implemented limits the size of the memory regions that are mapped to both user space and kernel space to be 64 KBytes. To test this above conjecture, we changed the secondary send buffer pool to 48 KBytes, and re-run the experiments. The measurements for these runs have been reported in the second of the two columns in Table 3. The number of lost packets and the total number of acknowledgment packets have indeed increased, thus confirming the theory that small secondary send buffer pool is the culprit of excessive acknowledgement traffic.

### 3.3.2 Buffer Overflows and Out-of-order Arrivals

Rows 7-11 in Table 3 present statistics about the number of messages sent and received, the number of messages retransmitted and the number of messages that were rejected at the receiver due to either buffer overflow or out-of-order arrival. The number of messages received is greater than the number of messages sent in all runs because some of the messages are multicast. We note that buffer overflow never occurred for any of the four programs – thus supporting the the use of optimistic flow control as a default.

There were out-of-order arrivals. However, the messages arriving out-of-order were less than 0.2% of the total number of messages in all runs. Messages arrive out-of-order if a previous message was lost in the network or if messages that have already reached the receiver are retransmitted due to a lost acknowledgment. If the total number of acknowledgments are reduced by using larger secondary send buffer pool – thus reducing the collisions on the network – the number of out-of-order message arrivals should be further reduced.

## 4 Conclusion

This paper presents the design and implementation details of *Pupa*, and the results of a comprehensive performance study on an operational *Pupa* prototype and their analysis. Despite the use of off-the-shelf networking hardware and the constraint that *Pupa* has to co-exist with existing communication systems, we have shown that *Pupa* is at least twice as fast as TCP/IP in latency and 1.5 times better in throughput for large message transfers.

The *Pupa* prototype has been operational for over a year, and we are porting it to a 16-node PentiumPro-200 MHz cluster using a Fast Ethernet switch as the underlying interconnection fabric. Two significant systems have been developed on top of *Pupa*. One is a compiler-directed distributed shared virtual memory system called *Locust* [18], which exploits the flexibility of *Pupa*'s buffer organization to match the need of a software-controlled cache consistency protocol. The other is a parallel multimedia index server called *PAMIS* [6], which is a straightforward message passing program that achieves a linear speedup on the *Pupa* cluster. The experience we had with these two applications and other message passing programs developed on *Pupa* is that *Pupa* is surprisingly robust, and delivers a consistently good communication performance close to the micro-benchmarking results as reported in the Performance section.

## Acknowledgement

This research is supported by an NSF Career Award MIP-9502067, NSF MIP-9710622, NSF IRI-9711635, and a contract 95F138600000 from Community Management Staff's Massive Digital Data System Program.

## References

[1] Anindya Basu, Vineet Buch, Werner Vogels, and Thorsten von Eicken. U-Net: A user-level network interface for parallel and distributed computing. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[2] Donald Becker, Thomas Sterling, Daniel Savarese, John Dorband, Udaya Ranawake, and Charles Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of International Conference on Parallel Processing*, 1995.

[3] Marshall Kirk McKuisck Keith Bostic, Michael J Karels, and John Quarterman. *The Design and Implementation of the 4.4BSD UNIX Operating System.* Addison-Wesley, 1996.

[4] Andrew Chen, Vijay Karamcheti, and John Plevyak. The Concert system - compiler and runtime support for efficient fine-grained concurrent object-oriented programs. Technical Report UIUCDCS-R-93-815, Department of Computer Science : University of Illinois at Urbana, June 1993.

[5] T. Chiueh and M. Verma. A compiler-directed distributed shared memory system. In *Proceedings of the 9th International Conference on SuperComputing*, July 1995. Also available at http://www.cs.sunysb.edu/~manish/locust.

[6] Tzi cker Chiueh, Dimitris Margaritis, and Srinidhi Varadarajan. Design and implementation of a parallel multimedia index server. In *Proceedings of Visual 97 Conference*, December 1997.

[7] David Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, pages 23–39, June 1989.

[8] FreeBSD Documentation. *http://www.freebsd.org*.

[9] Vijay Karamcheti and Andrew Chen. Software overhead in messaging layers: Where does the time go? In *Proceedings of the Sixth Symposium on Architectural support for Programming Languages and Operating Systems*, 1994.

[10] Vijay Karamcheti and Andrew Chien. Do faster routers imply faster communication? In *Proceedings of the Parallel Computer Routing and Communication Workshop*, pages 1–15, 1994.

[11] Jonathan Kay and Joseph Pasquale. A performance analysis of TCP/IP and UDP/IP networking software for the DECstation 5000. Technical report, Department of Computer Science and Engineering : University of California at San Diego, 1992.

[12] Samuel Leffler, Marshall Kirk McKuisck, Michael Karels, and John Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

[13] R. Martin. HPAM: An active message layer for a network of HP workstations. In *Hot Interconnects II*, August 1994.

[14] Scott Pakin, Maurio Lauria, and Andrew Chien. High performance messaging on workstations: Illinois fast messages (FM) for myrinet. In *Proceedings of Supercomputing*, 1995.

[15] Robert D Russell and Philip J Hatcher. Efficient kernel support for reliable communication. Technical Report http://www.cs.unh.edu/Personal/pjh.html, University of New Hampshire.

[16] Mark Swanson and Leigh Stoller. Low latency workstation cluster communications using sender-based protocols. Technical Report UUCS-96-001, Department of Computer Science: University of Utah, 1996.

[17] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating data and control transfer in distributed operating systems. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[18] M. Verma. *A Compiler-Directed Shared Memory System.* PhD thesis, Computer Science Department, State University of New York at Stony Brook,, ECSL-TR-33, ftp://ftp.cs.sunysb.edu/Pub/TechReports/chiueh/TR33.ps.Z, December 1996.

[19] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Low-latency communication over fast ethernet. In *Proceedings of Euro-Par*, pages 27–29, August 1996.