# Analysis of Buffer Management Policies in Layered Software

*S. V.Raghavan and S.Krishnamoney,*
*Department of Computer Science and Engineering,*
*Indian Institute of Technology Madras, Madras 600 036, India.*
*{svr,kms}@iitm.ernet.in*

## Abstract

Use of shared memory in the transfer of protocol data units (PDUs) between layers of a network eliminates the need to physically copy the PDU, thereby saving considerable time and resources in the form of data buffers. In this paper we discuss three possible approaches to the management of shared memory between protocol layers, and analyze their relative merits and demerits. These strategies differ primarily on the compaction algorithms to cure memory fragmentation. The analysis of compaction algorithms is presented and the performance of the memory management strategies under various conditions of load are compared.

## Key words

PDU (Protocol Data Unit), SDU (Service Data Unit), Shared Memory, NIU (Network Interface Unit), Compaction, Segmentation.

## 1 INTRODUCTION

Layering of protocol functions and engineering the related software require that information in a coded form be exchanged across layer boundaries and between peer layers [Stallings84, Tanenbaum85, Zwaenopoel85, Raghavan90]. Protocol Data Units (PDUs) can be passed across protocol layers by physically copying the data structures or by allo-

S. V. Raghavan et al. (eds.), *Computer Networks, Architecture and Applications*
© Springer Science+Business Media Dordrecht 1995

cating these data structures in shared areas of their respective address spaces. These are referred to as the *C-scheme* and the *S-scheme* respectively in this work.

In a layered communication software, C-scheme denotes copying of PDUs across layer boundaries and S-scheme denotes the usage of *shared memory* to store PDUs and passing of pointers across layer boundaries. In C-scheme, each layer may add or remove a header (and possibly a trailer) and pass the corresponding pointer to the modified PDU to the next layer. The application layer will allocate enough buffer so that each layer simply copies its header in front of the incoming *Service Data Units* from the n + 1 layer and pass it to the n - 1 layer by giving a pointer to the PDU. The block schematic of the C-scheme and S-scheme are shown in Figure 1 and in Figure 2. *In this paper we analyze the performance of the S-scheme with special emphasis on compaction strategies.*

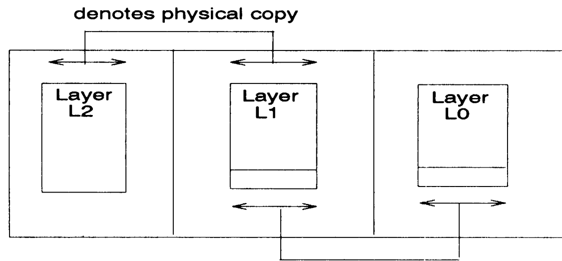


**Figure 1** Block Schematic of the C-Scheme.

The rest of the paper is organized as follows: Section 2 briefly discusses related work that has already been done in this area. Section 3 states the problems of buffer management in layered software and compares the C-scheme and S-scheme. Section 4 highlights the issues in implementation, while section 5 describes the implementation itself. Section 6 presents the temporal and spatial analysis in the S-scheme leading to the identification of the optimal block size. Section 7 describes the simulation experiments that are conducted to understand the effectiveness of the compaction strategies under different load conditions and with varying shared memory parameters. Finally section 8 summarizes the work by listing the salient features of our work, the significance of the results obtained and suggests possible future course of investigation.

## 2 RELATED WORK

Minimization of data copying is a major issue to be addressed in a protocol implementation. It would be ideal to support data transfer across protocol layers and the NIU in just one copy, but this is not possible because of the machine architecture or due to the security requirements of the host operating system. Thus, protocol modularization

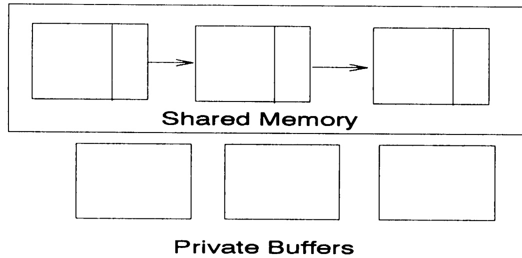


**Figure 2** Block Schematic of the S-Scheme.

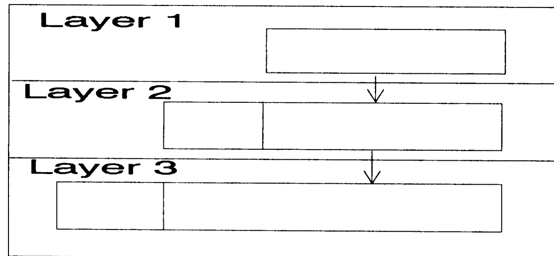


**Figure 3** Transmission Phase of PDU.

requires multiple copies of the PDUs [Zwaenopoel85].

G.S. Poo et al. [G.S.Poo91] gives an efficient algorithm for passing PDUs across layers as linked lists. Their method registered a maximum improvement of 90% for bulk file transfer without software checksum. Saltzer et al. [Saltzer et al.84] discuss the implications of choosing an appropriate buffer management strategy in implementations of protocols. Buffer management strategies are also discussed in Lantz [K.A.Lanz et al.85]. Analogous to buffer management strategies are the memory management strategies and disk management strategies in Operating Systems. An overview of these strategies is presented in Peterson [Peterson et al.85]. Clark [Clark82] presents the modularity and efficiency criteria in protocol implementations, which include the issue of data copying across layers. Watson and Mamrak [Watson et al.87] also discuss this issue in detail and provide a review of previous work in the same area. *The considerable importance given to the memory management strategies in protocol implementations and the desire to implement a clean memory management strategy that eliminates multiple copies of PDUs, especially in the context of high speed networks and high performance network interface adapters, are the motivations for this work.*

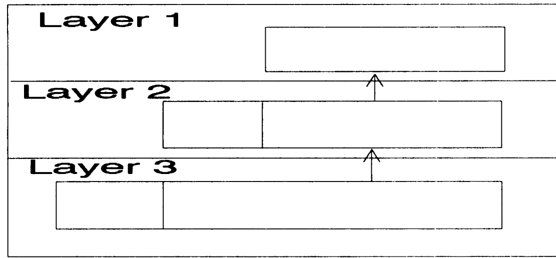


**Figure 4** Reception Phase of PDU.

# 3 BUFFER MANAGEMENT PROBLEMS IN LAYERED SOFTWARE

In a shared memory implementation, allocation of buffers, deallocation of buffers, and management of external or internal fragmentation may be done in several ways. One simple approach is to allocate and deallocate only fixed sized buffers thereby avoiding external fragmentation and compaction. The price paid for this is extra memory due to internal fragmentation. The implemented protocol stack should handle single character packets, normal sized packets which do not need any fragmentation and large packets which do require fragmentation in an efficient manner. Therefore it becomes necessary to identify an optimal buffer management policy that is appropriate for different load conditions. In this paper we discuss both C and S schemes, compare their performance and establish through simulation, the utility and appropriateness of C and S schemes under light, medium and heavy loads.

The metric for comparison of the performance of the two schemes is *the time taken to transmit and receive one PDU across a layer. The significant operation considered in the calculation of transmit/receive time is a block copy.* The transmission phase and the reception phase are shown in Figure 3 and in Figure 4.

Let $H_L$ be the header size of layer L in blocks.

Let $D_L$ be the data size of layer L in blocks.

Let T be the time function of a block copy.

[ Note: T is a linear function of the form $T(X) = aX$, where $X$ is the size of the buffer and $a$ is a constant. Therefore, $T(X_1) + T(X_2) = T(X_1 + X_2)$ ]

We consider a layer $L_2$ with user layer $L_3$ and service layer $L_1$.

Table 1 gives the ratio of C-scheme and S-scheme data copying time. This ratio establishes the S-scheme as a very attractive alternative to the C-scheme. Also it is to be noted that the gain in efficiency is directly proportional to the PDU size. However the buffer management cost has not been considered in this analysis. Processing overhead depends on the buffer management strategy used. External and internal fragmentation

**Table 1** Calculation of the ratio of C-scheme to S-scheme time for data copying

| Transmission time for C-scheme | |
|---|---|
| Copying time of $L_2$ SDU to local buffer | $T(D_{L2})$ |
| Adding $L_2$header | $T(H_{L2})$ |
| Total C-scheme transmission time | $T(D_{L2} + H_{L2})$ |
| **Reception time for C-scheme** | |
| Copying time of $L_1$data to local buffer | $T(D_{L2} + H_{L2})$ |
| Transmission and Reception time across layer $L_1$ in C-scheme | $2 * T(D_{L2} + H_{L2})$ |
| **Transmission time for S-scheme** | |
| Pointer to $L_2$data passed from $L_3$ to $L_2$ | 0 |
| Header prefixed to $L_2$data | $T(H_{L2})$ |
| Pointer to $L_2$data passed from $L_2$ to $L_1$ | 0 |
| Total S-scheme transmission time | $T(H_{L2})$ |
| **Reception time for S-scheme** | |
| Pointer to $L_1$data passed from $L_1$ to $L_2$ | 0 |
| Pointer to $L_2$data passed from $L_2$ to $L_3$ | 0 |
| Total S-scheme reception time | $T(0)$ |
| Transmission and Reception time across layer $L_2$ in S-scheme | $T(H_{L2})$ |
| The ratio of the time taken by C-scheme to S-scheme $2 * T(D_{L2} + H_{L2})/T(H_{L2}) = 2 * (D_{L2} + H_{L2})/H_{L2}$ | |

depends on the organization of the buffer pool and the size of one block. The above mentioned factors vary for different environments. In this paper it is our aim to solve these problems and to show that buffer management cost does not affect the ratio mentioned in Table 1 significantly.

# 4 ISSUES IN THE IMPLEMENTATION OF S-SCHEME

1. The shared memory is treated as a ***sequence of blocks*** of size BLOCK_SIZE, a compile time parameter, typical values being 256 bytes, 512 bytes and 1K bytes. The blocks are treated as individual entities, the number of blocks in the shared buffer being $\lceil$ size of buffer requested / BLOCK_SIZE $\rceil$.

2. The order in which the PDUs are released from the shared buffer is not predicatable. Hence, over a period of time, such unordered freeing of blocks leads to *fragmentation* of the shared buffer, creating a spurious insufficiency of the buffer space and necessitating compaction. Memory compaction requires processing overhead.

3. Compaction involves the physical shifting of the buffer blocks and it will affect the validity of the pointers passed, thereby rendering passing of pointers impossible. To overcome this difficulty, another level of indirection ought to be introduced. This level is provided by the *index table*. When a layer allocates buffer for an incoming PDU, an entry is allocated in the index table. This entry contains the number of contiguous blocks, a block pointer to the first of these blocks and links (next and previous) to index entries. The combination of array and list data structures in this model ensures minimum overhead in index table maintenance.

## 5 IMPLEMENTATION DETAILS OF S-SCHEME

Buffer pool is organized as an array of fixed sized blocks and initially free pointer points to the beginning of the buffer pool. When an allocate request comes, the buffer manager will check whether enough free space is available and if available, free pointer is advanced by the size of the allocate request and the newly allocated block is returned to the calling routine.

### 5.1 Data Structures

1. Buffer : Buffer of user requested size. Number of blocks allocated = $\lceil$ size/BLOCK_SIZE $\rceil$
2. Index map of buffer : An array of integers, corresponding to the blocks in the buffer pool. An Index map entry holds -1 if the corresponding block is free, else holds the index value to which it is attached.
3. Index table : A table of structures. The structure contains
   (a) Number of contiguous blocks attached to the corresponding table entry.
   (b) Block pointer to the first of these blocks.
   (c) Links to a free list. This list is required because index entries can be released in arbitrary order.
4. Current_index is the head of the free list of indices.
5. Current_buffer_block is the pointer to the next free area in the shared memory.

### 5.2 Initialization

1. Creation of shared buffer space.
2. Initialization of Index map.
3. Initialization of Index table. The Index table is initialized as follows : For each entry :
   (a) No. of blocks = 0.
   (b) Block_ptr = -1.
   (c) Indices are chained circularly. Initially they are in sequential order.
   (d) Current index is set to 0.

The initialized data structures are shown in Figure 5 and initial buffer state is shown in Figure 6.

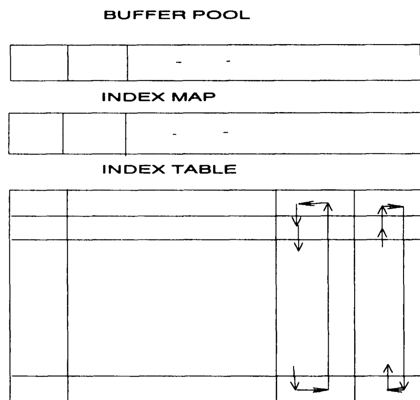


**Figure 5** Initialized Data Structures.

### 5.3 Allocation of buffer

When a request to allocate a buffer of size SIZE is made, if the *end run* ( the last run of free blocks ) is greater than or equal to SIZE,

1. Blocks are allocated.
2. An index entry is removed from the free index list and values are entered for the number of blocks and block pointers.

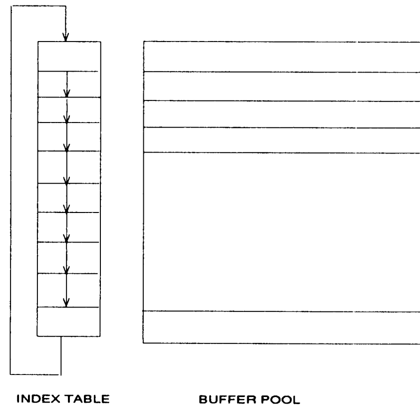


**Figure 6** Initial Buffer Pool State.

3. The corresponding index map values for blocks are set to the index entry values.

4. Index entry value is returned and current index set to the next free index.

If the end run is less than SIZE and the buffer is not already compacted, the buffer is compacted and the end run is compared to SIZE again. If the end run is still less than SIZE, a REQUEST_OVERFLOW is reported. The final buffer state after some allocations and deallocations are shown in Figure 7.

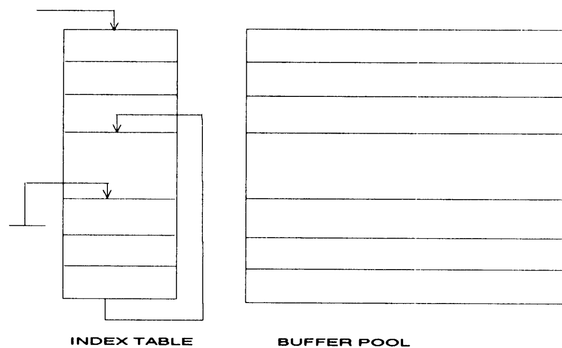


**Figure 7** Final Buffer Pool State.

### 5.4 Freeing of buffer

To free a run of blocks, the index value is passed to the function *free_buffer.*

1. The run of buffer blocks is identified because the block pointer and the number of blocks are available.
2. The corresponding index_map entries are set to -1.
3. The index entry is added on to the free list.
4. If the run of blocks was the last occupied run, the current buffer block is set back by no_of_blocks, to increase the end run.

Because freeing blocks may occur anywhere, as per request, the buffer may be fragmented over time. This creates an artificial scarcity of space. When a request for buffer blocks cannot be granted, the buffer manager resorts to compaction. This physically moves the blocks and this is the reason for the added level of indirection. The fragmentation in this implementation is taken care of by compaction *on demand.* This corresponds to strategy *S2* in our implementation.

## 6 ANALYSIS OF COMPACTION STRATEGIES

In this section, three compaction strategies are investigated and analyzed. Variations of these strategies are introduced at the end of this section. The trade-off between less space consuming strategies that require compaction and more space consuming strategies with no compaction is presented. Compaction coalesces all the free runs into one contiguous end run. This is an $O(N)$ operation where N is the number of busy blocks in the buffer pool. The different compaction strategies analyzed are:

*S1* : When a sequence of blocks is deallocated, compaction is performed.

*S2* : When a sequence of blocks is deallocated, no compaction is performed. When a buffer request requires a run of blocks of the size greater than the end run, then compaction is done.

*S3* : When a buffer (consists of a run of blocks) is deallocated, if the length of the last run of assigned blocks is less than the length of the deallocated run, then this last run of assigned blocks is fitted into the deallocated run and the end run is increased. This operation is recursively done on the remaining run of blocks. This process is called *stuffing.* When a buffer request requires a run of blocks of size greater than the size of the end run, a compaction is done.

| | | |
|---|---|---|
| *C(S1)* | deallocation<br>compaction<br>allocation | 0<br>*O(N)*<br>$\sum_{i=1}^{n}[p(i) \times O(i)] + \sum_{i=n+1}^{M} p(i) \times F$ |
| *C(S2)* | deallocation<br>if size <= $f$ allocation<br>if size > $f$ compaction<br>+ allocation | 0<br>$\sum_{i=1}^{f}[p(i) \times O(i)]$<br>$\sum_{i=f+1}^{n}[O(B-n) + O(i)] \times p(i)$<br>$+\sum_{i=n+1}^{M}[O(B-n) + F] \times p(i)$ |
| *C(S3)* | deallocation<br>stuffing<br>if size <= $f+k$ allocation<br>if size > $f+k$ compaction<br>+ allocation | 0<br>O(k)<br>$\sum_{i=1}^{f+k}[O(i) \times p(i)]$<br>$\sum_{i=f+k+1}^{n}[O(i) + O(B-n)] \times p(i)]$<br>$+\sum_{i=n+1}^{M}[O(B-n) + F] \times p(i)$ |
| where, | | |
| Cost of request overflow | | $F$ |
| Probability of the request of size $i$ | | $p(i)$ |
| Maximum request size | | $M$ |
| Buffer pool size in blocks | | $B$ |
| Number of filled blocks after deleted run | | $N$ |
| Total number of free run | | $n$ |
| End run size | | $f$ |
| Number of block copies for stuffing | | k |

**Table 2** Comparison of the three strategies.

The cost of performing a deallocation of a run of blocks followed by an allocation of a run of blocks is now evaluated for the three strategies. The significant operation in the analysis of the cost of a deallocation followed by an allocation is a block copy. We denote the cost of one deallocation followed by one allocation in scheme S by *C(S)*. Table 2 gives the cost of the buffer management overhead for the three strategies *S1, S2* and *S3*. Therefore,

$$C(S1) = \sum_{i=1}^{n}[O(i) \times p(i)] + \sum_{i=n+1}^{M}[F \times p(i)] + O(N) \quad (1)$$

$$C(S2) = \sum_{i=1}^{n}[O(i) \times p(i)] + \sum_{i=n+1}^{M}[F \times p(i)] + \sum_{i=f+1}^{M}[O(B-n) \times p(i)] \quad (2)$$

$$C(S3) = \sum_{i=1}^{n}[O(i) \times p(i)] + \sum_{i=n+1}^{M}[F \times p(i)] + \sum_{i=f+k+1}^{M}[O(B-n) \times p(i)] + O(k) \quad (3)$$

Let $\mathcal{F} = \sum_{i=1}^{n}[O(i) \times p(i)] + \sum_{i=n+1}^{M}[F \times p(i)]$. Physically, $\mathcal{F}$ is composed of two components, *allocation cost* and *Failure cost.* It may be noted that $\mathcal{F}$ is common to all the strategies.

The extra buffer management costs involved in each strategy are :

- *O(N)* in *S1* for compaction done always.
- $O(B-n)$ in *S2* for compaction done on demand.
- $O(B-n)+O(k)$ in *S3* for compaction done on demand and stuffing done always.

Therefore,

$$C(S1) = \mathcal{F} + O(N)$$

$$C(S2) = \mathcal{F} + \sum_{i=f+1}^{M}[O(B-n) \times p(i)]$$

$$C(S3) = \mathcal{F} + \sum_{i=f+k+1}^{M}[O(B-n) \times p(i)] + O(k)$$

We now analyze the conditions under which each strategy turns out to be the most cost effective one.

**Theorem 1** *When the probability distribution, p(i) is uniform:*

1. *under light or moderate load conditions S2 is better than S1.*
2. *under light or moderate load conditions S2 is better than S3 and under heavy load conditions S3 is better.*

*Proof:* $C(S1) > C(S2)$ if $O(N) > \sum_{i=f+1}^{M}[O(B-n) \times p(i)]$
that is $C(S1) > C(S2)$ if

$$O(N) > O(B-n) \times (M-f)/M \tag{4}$$

Under light or moderate load, $f > M$. So *S1* is more expensive than *S2*. Under heavy load, $f$ is comparable to $M$. So *S1* and *S2* are comparable, *S1* being positionally dependent on the deallocated run of blocks.

$C(S2) > C(S3)$ if

$\sum_{i=f+1}^{M}[O(B-n) \times p(i)] > O(k) + \sum_{i=f+k+1}^{M}[O(B-n) \times p(i)]$ that is

$$\sum_{i=f+1}^{f+k}[O(B-n) \times p(i)] > O(k) \tag{5}$$

that is $(B-n) > M$.

Under heavy load $n$ is negligible. So *S2* is more expensive than *S3*. Under light or moderate load, *S3* is more expensive.

**Theorem 2** *When probability distribution, p is peaked at M/2 (Peaked bell shaped curve with no skewness):*

1. *under light or moderate load S1 is more expensive than S2 and under heavy load, S1 is better than S2.*

2. *under light or moderate load S3 is more expensive than S2 and under heavy load both S2 and S3 are comparable.*

*Proof :* Under light or moderate load, $f > M$. So *S1* is more expensive than *S2*. Under heavy load $f > M/2$. Hence *S1* is better than *S2*.

E(k) = M/4 , where E is the expectation function.

$C(S2) > C(S3)$ if

$\sum_{f+1}^{f+M/4}[(B-n) \times p(i)] > M/4$

Under light or moderate load, $f > M/2$. So *S3* is more expensive than *S2*. Under heavy load, *f* is comparable to *M*. So *S3* and *S2* are comparable.

From the above analysis, the conclusions reached are : When probability distribution is uniform, under *heavy* load, *S3* is most cost effective. Under *light* load, *S2* is most cost effective. When probability distribution is a peaked bell shaped curve with no skewness, (peaked at M/2), under *light* to *moderate* load, *S2* is most cost effective. Under *heavy* load, all strategies are comparable. These results hold for a deallocation followed by an allocation. A variant of *S3* is the modification of the *best fit* algorithm. In this algorithm, a list of fragmented free runs, ordered in descending size is maintained. When a run of blocks is deleted, the strategy *S3* is applied on the largest run in the ordered list. The remaining fragment is then inserted *in order* in the list. This reordering cost is insignificant when compared to the significant operation of block copy. We now provide a spatial and temporal analysis of the three strategies. We divide time into discrete intervals and consider $p_a$ and $p_d$ to be the probability of an allocate request and a deallocate request in a time interval, respectively. The probability of a deallocate request before an allocate request is :

$$P = (1-p_a)^i * (1-p_d)^{(i-1)} * p_d$$

Let the state of the system be denoted by the triple $<$ *work done, length of end run, total free space* $>$. Initial state of the system is $< 0, B, B >$ where B is the total shared memory

space. After $i$ releases, the state of the system corresponding to each strategy is :

*S1* : $< O(n_1 + n_2 + .. + n_i), f + n_1 + .. + n_i, f + n_1 + .. + n_i >$

*S2* : $< 0, f, f + n_1 + .. + n_i >$

*S3* : $< O(k_1 + .. + k_i), f + k_1 + .. + k_i, f + n_1 + .. + n_i >$

where $n_i$ denotes the $i$th run of deleted blocks and $k_i$ denotes the $i$th run of stuffed blocks. Let the work done for allocation with end run of $f$ and free space of n be $C(f, n)$. Then, work done for one allocation after $i$ deallocations is,

*S1* : $\sum_{j=1}^{i} O(N_j) + C(f + \sum_{j=1}^{i} n_j, f + \sum_{j=1}^{i} n_j) = C_{S1}(i)$.

*S2* : $C(f, f + \sum_{j=1}^{i} n_j) = C_{S2}(i)$.

*S3* : $O(k_j) + C(f + \sum_{j=1}^{i} k_j, f + \sum_{j=1}^{i} n_j) = C_{S3}(i)$.

where $C_{si}(i)$ denotes the work done for one allocation after i deallocations. The probability of the occurrence of such an event is :

$P^i * (1 - P) * C_{Sn}(i)$, where n = 1,2,3.

This is the generalized mathematical formulation of the compaction strategies. We carried out simulation studies of the three strategies under various conditions of load, mean packet size and shared memory size. We assumed a poisson arrival rate for request of allocation, an exponential distribution for the duration that a PDU is in the shared memory and an exponential distribution for packet size. We find that the simulation results generalize the results obtained from our restricted analysis of the case of a deallocation followed by an allocation. The work done in memory management due to each of the strategies is measured in terms of the number of block copies done per blocks allocated. The details of the simulation experiments are described in the next section. A strategy that totally eliminates block copies can also be implemented. In this strategy, every block has associated with it, a structure that contains the index with which the block is associated, and a pointer to the next block. When a sequence of 'n' blocks is requested, the free list is traversed down n links, and the corresponding blocks are allocated. This strategy involves an *O(B)* space overhead, where B is the size of the shared memory, and an *O(n)* time overhead in terms of pointer traversals. It however, does away with compaction. The principal advantage of splitting the shared memory into blocks is the reduction of the size of the index map and speeding up of memory management. It is based on the *principle of segmentation*, where resources of the system are divided into units, the size of which depends on the application and the particular resource. The disadvantage of this approach is a potential wastage of space in the form of internal fragmentation. A large buffer would speed up compaction and facilitate buffer management but could cause more

internal fragmentation, while a very small block defeats the purpose of this scheme. We now derive the optimal block size $B$, for a given maximum packet request size $P$, and a given probability distribution $p$.

**Theorem 3** *An optimal block size, B can be obtained form the equation* $d/dB(F(B) + C(E(w))) = 0$ *where* F(B) *is the cost of block operations for a size B,* $E(w) = (\lceil (i/B) \rceil \times B - i)/\lceil (i/B) \rceil \times B \times p(i)$*, the expected fraction of wastage and* $C(x)$ *is the cost of buffer of space x.*

*Proof:* If the block size B is increased then $F(B)$, the cost of block operations will decrease and $C(E(w))$, the cost of internal fragmentation will increase. The total cost is the sum of the above two factors and hence to minimize the total cost we have to solve the equation d/db $(F(B) + C(E(w))) = 0$, the cost of space of a block B be C(B). Under the conditions of uniform probability of packet request size,

$p(i) = 1/P$,

$E(w) = (1/P) \times_1^p \Sigma_{(n-1)B+1}^{nB} \Sigma(nB - i)/B$

# 7 SIMULATION RESULTS

The buffer management overhead variation with reference to three important simulation parameters, PDU size, buffer pool size and Residence time of the PDU in the buffer pool has been studied. Simulation was carried out in two steps. In the first step effect of varying PDU size is studied by keeping the buffer pool size constant where as in the second step, the effect of varying buffer pool size is studied by keeping the mean PDU size constant.

In Figure 8 (a) the PDU size is varied from 32 bytes to 1KB keeping the buffer pool size at 512 bytes. We assumed Poisson distribution for the arrival rate with a mean of $\lambda = 15$ and negative exponential distribution for the residence time with a mean of $\mu = 10$. From the figure we can see that out of the 3 methods *S2* has a clear advantage. When the PDU size is in the range 32 bytes to 64 bytes, there is a gradual increase in the overhead for *S2* and *S3*. Overhead of *S2* remains almost stable when the PDU size is in the range 64 bytes to 256 bytes. As the average PDU size is greater than or equal to 512 bytes, for all strategies overhead decreases dramatically well below 0.2. This is because the Buffer Manager is unable to satisfy most of the requests and hence the actual work done is quite small. Even at a low PDU size of 32 bytes there is a high failure probability of around 0.7. This indicates that even at light loads buffer pool size of 512 bytes is inadequate.

Next a series of simulation runs were conducted by varying the parameter buffer pool size to 2048 bytes, 4096 bytes, 8192 bytes, 16384 bytes and 65536 bytes. This is shown in Figure 8 (b), Figure 9 (b), Figure 10 (a), Figure 10 (b) and Figure 11 (a). When the

average PDU size is 32 bytes the failure probability is as low as 0.3 (Figure 8 (a)). At this conditions overheads at a low PDU residence time are 0.47, 0.12 and 0.23 respectively for *S1*, *S2* and *S3*. Total buffer pool size is increased to 4096 bytes and the results are plotted in Figure 9 (b). When the PDU size is 32 bytes failure probability is as low as zero. In the second step of experiments the mean PDU size is kept constant and buffer pool size vs overhead is plotted for $\mu$ =15, 20, 25 and a mean arrival rate of $\lambda$ = 15. (Figure 12 (a)). In the case of *S1* overhead is almost constant and well above *S2* and *S3*. In Figure 12 (a) as the buffer size = 16385 bytes for a mean PDU size of 32 bytes, overhead becomes zero. Here *S1* has a constant overhead while for *S2* and *S3* overhead decreases as buffer size increases. Comparing this results with PDU size of 128 bytes and 256 bytes, we can see that as the PDU size is small the gap between *S1* with the other two strategies become large. From Figure 12 (b) and 13 (a), we can see that all the three curves diverges from a point as we go away from origin along X axis. The conclusion drawn from the above observations are that *'Under heavy load conditions all the three strategies become comparable'*.

The simulation results confirm our conclusions reached in section 6. The conclusions drawn from the above analysis are

1. *Under light or moderate load conditions S2 is the most cost effective strategy .*
2. *Under heavy load conditions all the three strategies become comparable.*
3. *Different combinations of the values of the three simulation parameters Buffer pool size, PDU size, and Residence time has no effect on the conclusions 1 and 2 above.*

## 8 CONCLUSION

In this paper, various shared memory schemes have been investigated and their relative merits and demerits under various conditions of load, shared memory size and maximum packet size have been presented. The various strategies of memory management can be characterized by allocation of buffer space, deallocation of buffer space and their compaction strategies.

Under different conditions of load and shared memory size, different strategies prove to be optimal. However, since strategies *S1, S2* and *S3* differ only in their method of compaction of the fragmented shared memory, depending on the load condition, it is possible to choose the optimal one dynamically. Optimal performance may be facilitated by allowing decisions to be taken at the application level.

## References

[Clark82] Clark D.D. Modularity and Efficiency in Protocol Implementation. *Internet Protocol Implementation Guide Network Information Center SRI International Menlo Park, California*, August 1982.

[G.S.Poo91] G.S. Poo, W.Ang. Cut-Through Buffer Management Technique for OSI Protocol Stack. *Computer Communications*, 14(3):166 – 177, April 1991.

[K.A.Lanz et al.85] K.A.Lanz, W.I.Nowiki, M.M.Theimer. An Empirical Study of Distributed Application Performance. *IEEE Transactions on Communications*, SE-11(10):1162 – 1173, October 1985.

[Peterson et al.85] Peterson J.L, Slibershatz. Operating System Concepts. *Addison-Wesley Publishing company*, 2nd Edition, 1985.

[Raghavan90] Raghavan S.V. *Solutions to Local Area Networks, The Indian Context.* Tata McGraw-Hill Publishing Company, 1990.

[Watson et al.87] R.W. Watson, S.A. Mamark. Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices. *ACM Transactions on Computer Systems*, 5(2):97 – 120, May 1987.

[Saltzer et al.84] Saltzer J.H., D.P.Reid, D.D.Clark. End to End Arguments in System Design. *ACM Transactions on Computing Systems*, 2(4):277 – 288, November 1984.

[Stallings84] Stallings W. Local Networks. *ACM Computing Surveys*, 16(1):3 – 42, March 1984.

[Tanenbaum85] Tanenbaum A. and R.V Reness. Distributed Operating Systems. *ACM Computing Surveys*, 17(4):479 – 490, December 1985.

[Zwaenopoel85] Zwaenopoel W. Protocol for Large Data Transfers over Local Networks. *Proceedings of the 9th Data Communication Symposium, Whistler Mountain, British Columbia, Canada, ACM, New York*, pages 22 – 32, 1985.

## 8 AUTHOR INFORMATION

**S.V. Raghavan** is on the faculty of the Department of Computer Science and Engineering, Indian Institute of Technology, Madras. He is also the Chief Investigator of the project on *Education and Research in Computer Networking* jointly sponsered by the Department of Electronics and Telecommunication Engineers. He is presently serving on the

Board of Editors of the journal of IETE for computers and control. He is also a member of the Editorial Advisory Borad for Computer Communications, Butterworth-Heinemann Ltd. His research interests are networks, protocols, multimedia systems and performance.

**S Krishnamoney** received his B.Tech degree in Computer Science from the Government Engineering College Trichur India in 1991 and is currently working towards his MS degree at the Department of Computer Science, Indian Institute of Technology Madras, India. His main research interests presently lie in computer networks and distributed systems.

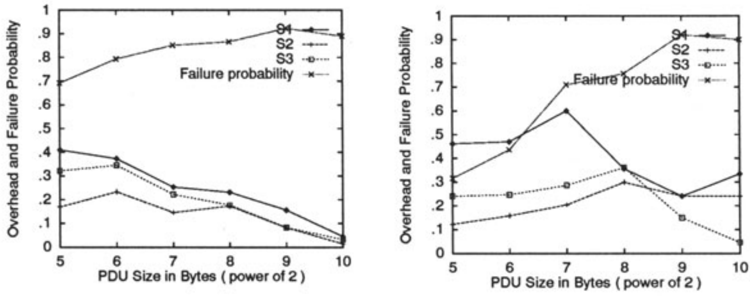


Figure 8: (a) Buffer pool size = 512 bytes $\lambda = 15$ $\mu = 10$ (b) Buffer pool size = 2048 bytes $\lambda = 15$ $\mu = 15$.

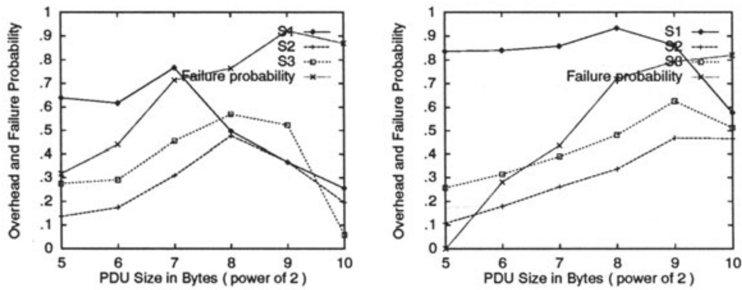


Figure 9: (a) Buffer pool size = 2048 bytes $\lambda = 15$ $\mu = 25$ (b) Buffer pool size = 4096 bytes $\lambda = 15$ $\mu = 25$.

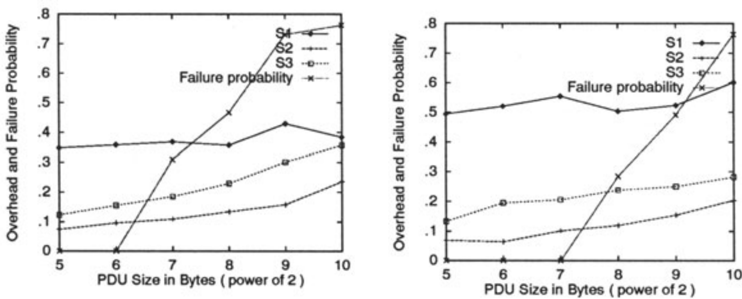


Figure 10: (a) Buffer pool size = 8192 bytes $\lambda = 15$ $\mu = 10$ (b) Buffer pool size = 16384 bytes $\lambda = 15$ $\mu = 15$.

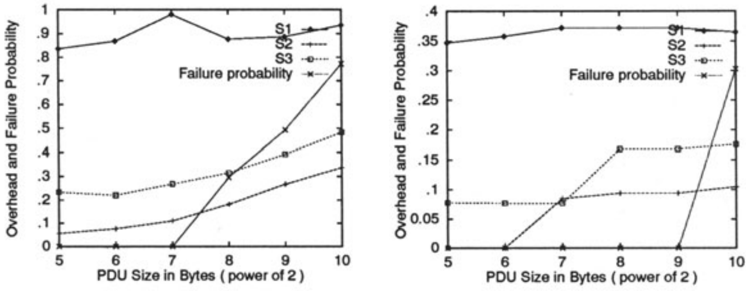


Figure 11: (a) Buffer pool size = 16384 bytes $\lambda = 15$ $\mu = 25$ (b) Buffer pool size = 65536 bytes $\lambda = 15$ $\mu = 10$.

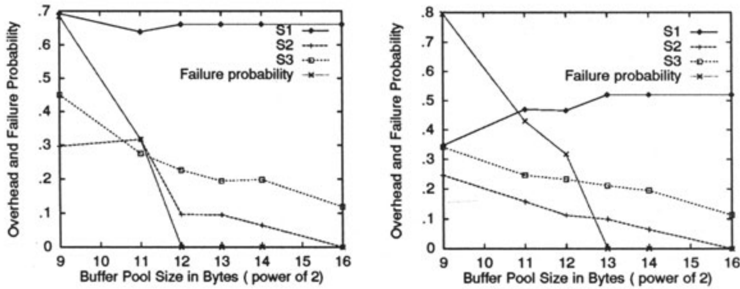


Figure 12: (a) Average PDU size = 32 bytes $\lambda = 15$ $\mu = 20$ (b) Average PDU size = 64 bytes $\lambda = 15$ $\mu = 15$.

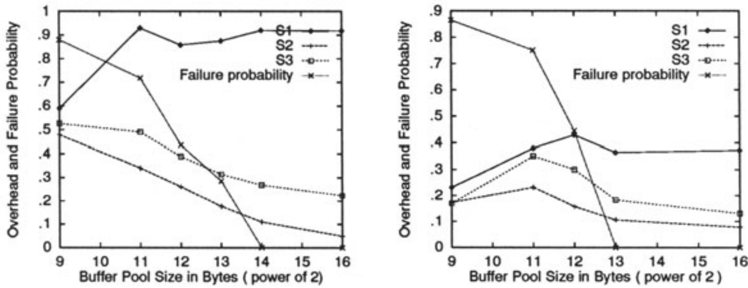


Figure 13: (a) Average PDU size = 128 bytes $\lambda = 15$ $\mu = 25$ (b)Average PDU size = 128 bytes $\lambda = 15$ $\mu = 10$.