# Demultiplexed Architectures: A Solution for Efficient STREAMS Based Communication Stacks

Vincent Roca[a,1], Torsten Braun[b], Christophe Diot[b]

[a]INRIA Rhône-Alpes; ZIRST; 655 avenue de l'Europe; 38330 Montbonnot S$^{t}$ Martin; France
e.mail: vincent.roca@inrialpes.fr; fax: (33) 76.61.52.52. http://sirac.inrialpes.fr/~roca/

[b]INRIA - 2004 route des Lucioles; BP 93; 06902 Sophia-Antipolis cedex; France
e.mail: {torsten.braun | christophe.diot}@sophia.inria.fr; fax: (33) 93.65.77.65. http://www.inria.fr/rodeo/

***Abstract:*** This paper analyzes the efficiency of various high performance implementation techniques for the communication system of UNIX workstations. Using an Open System implies that a certain compatibility level is required from the protocol, user interface, and implementation framework. These constraints limit the opportunities to design a high performance communication system. We have designed an experimental platform around the TCP/IP protocol suite, using the STREAMS environment. A BSD TCP/IP stack and a classic STREAMS based TCP/IP stack serve as reference implementations for performance comparisons. We explain why the efficiency of some high performance implementation techniques we applied to this platform is limited. The impacts of the hardware architecture, of the operating system, and of the communication stack architecture on performances are analyzed. It is shown that the efficiency of data transmission would benefit from more simplicity and more synchronism in the communication environment, from direct data paths between the applications and the device drivers, and from a limited ILP integration.

***Keywords:*** communication stack, STREAMS, high performance, implementation techniques.

## 1 INTRODUCTION

The communication system of a workstation is at the meeting point of three trends: the evolutions of the physical network technology in terms of higher performance, lower error rates, and new kinds of services; the evolutions of the application needs, since new multimedia applications often have real-time constraints; and the evolutions of the workstations from the hardware and software points of view. The need to take advantage of advances in technology and to offer to applications the best possible services, has motivated much research in the field of high performance communication stacks.

This paper focuses on standard workstations, i.e. mono and multiprocessor machines running a UNIX operating system.This paper shows how several high performance techniques can be applied to such a system. The experimental platform we built includes several TCP/IP communication stacks and uses the STREAMS environment. A BSD TCP/IP stack serves as a reference implementation for performance comparison. This platform allows us to draw several conclusions on the efficiency of the high performance implementation techniques in Open Systems, in particular from the hardware, operating system, and communication architecture viewpoints.

This paper is organized as follows: Section 2 identifies the major problems encountered when designing a high performance STREAMS based communication system. Section 3 describes our experimental platform. Section 4 presents the performance experiments carried out on this platform. Finally, Section 5 discusses the efficiency of the high performance techniques implemented.

1. This work has been done thanks to a CIFRE contract between LGI-IMAG and BULL S.A.1, rue de Provence; BP 208; 38432 Echirolles cedex; France.

# 2 EFFICIENT COMMUNICATION IN OPEN SYSTEMS

The design of high performance STREAMS based communication stacks on a UNIX workstation arises three classes of problems: the constraints related to the use of an Open System, the limitations of classic communication stacks using the STREAMS environment, and the limitations of current workstations. We discuss each of these points, and introduce several high performance implementation techniques that are applicable in the discussed context.

## 2.1 Constraints related to the use of an Open System

The main goal of an Open System is to ensure the *portability and interoperability* of software components across platforms. The POSIX (Portable Operating System Interface for Computing Environments) standards and the X/Open Portability Guide Issue 4 are two examples. They define a comprehensive set of standards to ensure the portability of applications at the source code level. For instance XTI (or X/Open Transport Interface) [XTI93] is a standardized API to access networking services. Another goal of an Open System is to promote the *independence* of the components thanks to standardized interfaces (portability and flexibility are increased, and development is made easier).

Therefore, using an Open System has great implications on performance improvement techniques:

- the independence principle contradicts the techniques that rely on the protocol layer integration.
- the standardization, is not always suited to high performance techniques. Flexibility is essential to adapt to new situations, needs, and application requirements.
- the solutions requiring the use of a non standard protocol family are of little interest as they compromise interoperability.

## 2.2 Limitations of classic STREAMS stacks

This section focuses on classic TCP/IP stack implementations in the STREAMS environment. After a fast introduction to STREAMS, we identify and discuss the limitations of these implementations.

### 2.2.1 STREAMS versus BSD

Two major network implementation and execution environments exist for UNIX: BSD, the environment found on the initial TCP/IP stacks, and STREAMS, introduced later by AT&T. They both provide basic system functionalities and architectural concepts that simplify the development of communication stacks. This section does not describe them (see [Stevens90] and [STREAMS90]), but compares their features. This comparison is interesting because they rely on two different communication concepts:

- *function calls* for BSD, and
- *message passing* for STREAMS.

Their goals are also different (Table 1). If BSD favors the integration and performance of the components, STREAMS promotes the independence, portability and flexibility aspects. Several standardized interfaces (TPI, NPI, and DLPI) between the various protocol layers are defined for STREAMS communication stacks. At the opposite, BSD does not impose any structure to the communication subsystem. Therefore the configuration of a BSD stack is mainly hard-coded in kernel tables, while a STREAMS stack is configured dynamically by inserting or removing drivers/modules (this configuration is usually described in an ASCII file that is processed by a dedicated tool to issue the appropriate `ioctls`).

Then, if the use of BSD is restricted to communication systems, STREAMS is used by several I/O systems like the TTYs (historically the first use of STREAMS). This specialization of BSD is the reason why its system calls (the socket layer) are specifically designed for transport protocol services. On the contrary, STREAMS system calls are generic. The notions of API (TLI/XTI or socket) and of access method (STREAMS system calls) are well distinguished.

| BSD environment | STREAMS environment |
|---|---|
| - favor the integration of components | - favor the independence, portability and flexibility |
| - static configuration | - dynamic configuration, loadable drivers/modules |
| - does not impose any internal structure | - standardized internal interfaces |
| - used solely for the communication system | - used for numerous I/O systems |
| - transport protocol oriented access method | - generic access method |

*Table 1: BSD and STREAMS comparison.*

### 2.2.2 Description of a classic STREAMS stack

Figure 1 shows a classic STREAMS TCP/IP stack. From top to bottom we identify:

- the XTI library: A cooperating module, TIMOD, is required because most of the API processing is performed at user-level. This module retains sufficient information to synchronize after a `fork`. Similarly, the socket library and SOCKMOD module implement the socket API (another architecture exists that interfaces the BSD kernel Socket layer with STREAMS).
- the TCP, UDP, IP, ICMP and RawIP multiplexing drivers as well as the ARP module.
- the DLPI driver which provides a DLPI interface between the network device drivers and the IP driver. There is one stream per logical network interface.

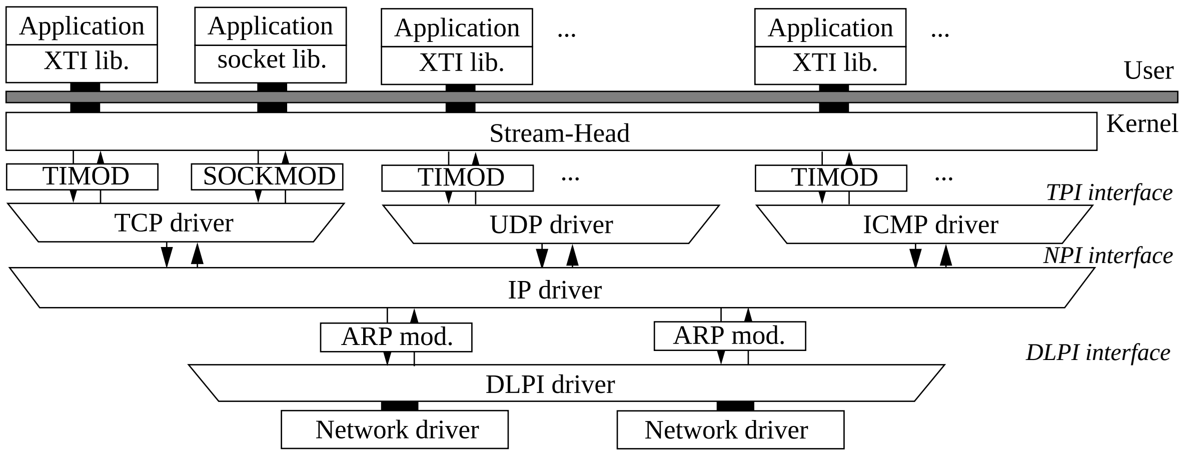

*Figure 1: Architecture of a classic STREAMS based TCP/IP stack.*

### 2.2.3 Limitations of this stack

The limitations directly associated to this classic architecture and to STREAMS are numerous.

***Use of queues and service routines***

Transport protocols usually queue data transmission requests in their write queue and process them in the associated service routine. This is an easy way to flow control the application: when TCP transmission window is full, the write queue turns full too (its high water mark is set equal to the window size), and the application is blocked. But there are two problems:

- Queueing and dequeueing a message is costly and occurs in the main data path. A dedicated queuing facility would be more efficient than STREAMS general purpose facility.
- The scheduling of the service routine adds latency (implementation specific). This is penalizing as it delays the (possible) transmission of the packet.

***STREAMS local flow control***

Because a multiplexing driver multiplexes several upper streams on several lower streams, the upper and lower streams are not directly linked. Therefore STREAMS flow control cannot see across a driver, which creates problems. For instance, if the DLPI driver is saturated and if the intermediate drivers queue messages, the application will not be blocked until all the intermediate queues are saturated. If an intermediate driver does not queue messages (e.g. IP), the saturation information cannot be propagated upwards. It means that a lot of memory and CPU resources can be spent in non-productive tasks (packets lost or queued) to the detriment of the components that do need these resources.

***Buffering at the receive side: a functional limitation of STREAMS***

Experiments have shown that the poor performance of a classic STREAMS stack is largely due to the Stream-Head strategy for filling applications receive buffers, with either `getmsg` or `read` system calls. Unlike the socket layer that by default always tries to optimize application buffer filling, the Stream-Head returns what data there is in the read queue at that time. [Streams90] does not propose any other strategy. This behavior increases the number of receive system calls (`getmsg`/`read`) and context switches. Most STREAMS stacks do not care about this problem.

***Inefficient vertical parallelism***

The parallelization of this classic STREAMS TCP/IP stack reveals several limitations: the layering concept promoted by STREAMS and the idea that doing work in service routines is a good way to parallelize a system, have lead to the concept of vertical parallelism. It relies on the systematic use or service routines and is similar to layer-based parallelism (Section 2.4). Unfortunately, layering and vertical parallelism are at the opposite of efficiency.

***The limitations of STREAMS or SOCKET access methods***

Eventually, the access method limits performance. A stream-oriented protocol like TCP sees TSDUs as a succession of bytes that can freely be segmented. The kernel buffers (i.e. `mblk` for STREAMS or `mbuf` for BSD, with their associated data buffer) are allocated without any consideration for the future segment boundaries (Figure 4 on page 9). Consequently:

- a segment usually straddles several buffers or is only a part of a buffer. It makes buffer related operations complex (e.g. duplication and liberation in the TCP list, copy to the device driver).
- a segment can start anywhere in the buffer and no room is available for the protocol headers.
- as the segment boundaries are not known during the user-to-kernel copy, this latter can not be integrated with the checksum calculation (ILP, Section 2.4).

These limitations are not related to STREAMS (the situation is the same with a BSD stack). They derive from the late knowledge of the segment boundaries. It is different with a datagram protocol like UDP that do not segment application datagrams. Integrating copy and checksum calculation is then easy.

## 2.3 Limitations of current workstations

The limitations of current workstations in regard to the communication system efficiency have two origins: the hardware and system environments.

***The hardware architecture aspect***

The performance limiting factor is no longer the physical medium nor the processor, but the memory in spite of elaborated cache techniques. [Mosberger95] reports that a fast RISC processor is never busy more than 37% of the time during a TCP/IP traffic. Protocol processing is either instruction-bandwidth or data-bandwidth limited, but never CPU limited.

The network adapters are also far from perfection. The use of a DMA requires that an interrupt is generated on completion, which adds a significant overhead [Druschel94], the system buffers must be pinned and their address mapped to the bus address space, the system memory area from/to which data is moved must be contiguous in physical or virtual memory [Druschel94, IBM92] which is rarely the case (protocol headers, segmentation, etc.), and cache lines are selectively invalidated before or after a DMA transfer to guaranty memory coherency [Thekkath93].

Because of these constraints, the IBM Ethernet driver defines a few static, page-aligned, pinned and mapped transmit and receive buffers in the host memory [Tracey94]. An outgoing Ethernet frame is first copied to such a buffer and then moved by DMA to the adapter. This additional copy results in higher performance than pinning and mapping an arbitrary `mbuf` chain [IBM92].

***The system environment aspect***

System related tasks have always been known as being costly. Early research identified that 80% of the costs are caused by system tasks, leaving only 20% for the protocol tasks [Chesson91]. With the advent of fast networks, the operating system impacts on performance will become more and more obvious.

Another aspect is the growing complexity of both the operating system and the machine architecture. [Montz94] introduces the notion of "*system entropy*". When both are mixed, unpredictable and sometimes unexplained artifacts can result (which makes optimizations difficult).

Finally, the increased need for QoS support is not only confronted to the communication protocol limitations, but also to the lack of QoS based resource management capabilities within the OS. Such services as appropriate thread scheduling and buffer allocation schemes, extended APIs, management and monitoring of QoS among protocol layers are essential and should be managed by the operating system.

## 2.4 Design approaches for high performance communication systems

Several kinds of approaches have been proposed to improve the communication system [Roca96]. This section essentially focuses on those that comply with Open Systems constraints.

***Demultiplexing the communication architecture***

[Feldmeier93] underlines that demultiplexing at the lowest possible level enables each data flow to be isolated, and makes per-flow processing possible (e.g. for QoS, or Quality of Service, support). Two levels of demultiplexing must be distinguished: in the communication protocols or network, and in the communication system implementation.

For the protocol or network aspect, the flow identifier of IPv6 and the VCI (Virtual Channel Identifier) of ATM (if allocated on a connection basis) are two solutions that facilitate this demultiplexing. For the implementation point of view, the classification technique [Wakeman95] enables to demultiplex incoming packets but requires all the protocol headers to be analyzed (see also Section 3.1).

***Optimization of the communication system within its hardware environment***

The memory bottleneck in modern workstations leads to many research efforts on the optimization of data manipulations (data copy, checksum, encryption, compression, marshaling, etc.).

The *ILP technique* (or Integrated Layer Processing) [Clark90] aims at avoiding memory accesses and using caches efficiently. It consists in integrating all the data manipulations into one single processing loop (the ILP loop). These manipulations are thus performed while data are still in fast registers or cache memory, saving memory accesses. The practical efficiency of ILP is discussed in Section 5.1.2.

Another way to avoid memory accesses is to *reduce the number of data transfers*. Several techniques exist: certain are based on virtual page remapping [Druschel93], on dual-port memory [Jacobson92, Edwards94], or on a direct transfer between the adapter and a user-level communication stack [Edwards95]. But these techniques have limitations: they require a complete reorganization of the communication stack, sometimes rely on user-level communication stacks or on specific networks.

A completely different approach to minimize the memory access costs, that does not alter the communication stack architecture, consists in *improving the instruction cache hit ratio*, either with a compiler oriented technique [Mosberger95, Castelluccia95] or run-time linker oriented technique [Montz94].

***Optimization of the communication system software environment***

Two optimized communication system frameworks have been designed: x-Kernel [Hutchinson91] and an improved BSD-derived architecture [Jacobson92, Jacobson93]. x-Kernel, like STREAMS, relies on a message-based communication paradigm. The improved TCP/IP stack architecture relies on the integration of the various components (protocol layers, buffer management, device drivers, data manipulations). We will not investigate this direction as we have already chosen to work with STREAMS.

***More processing thanks to parallelism***

Using a multiprocessor to increase the communication system performance requires that the speedup obtained from parallelism outweighs the associated synchronization and context switching overheads.

The CPU support is important. The `load-and-reserve` and `store-conditional` instructions have replaced the traditional `test-and-set` [Chesson94, Talbot95]. They can be used to build all the classical synchronization facilities, lock-free list manipulations, TCP/IP statistic counters, etc. An appropriate use reduces the need of locks and improves the communication system performance.

The parallelism model is also essential. [Schmidt94] identifies:

- *task-based parallelism*: It can be refined into *layer parallelism* where each protocol layer is attached to a dedicated thread, and *function parallelism* where protocols are decomposed in several functions (header processing, acknowledgment, etc.) attached to separate threads.
- *message-based parallelism*: It can be refined into *connection parallelism* where all the messages destined to a given context are handled by the same thread (thread-per-connection), and *message parallelism* where a message is processed by any available thread (thread-per-message).

Task-based parallelism creates a high level of synchronization and context switch overhead. Message-based parallelism is more efficient [Schmidt94]. Because it relies on dynamic elements (connections or messages), it adapts to the number of processors. Then, a message is usually processed by a single thread. [Chesson94] argues that the connectional parallelism has the advantage of minimizing synchronization problems (at most one thread per context). On the contrary, with the message parallelism, each connection can benefit from all the processors, at the expense of increased synchronization needs.

Anyway, experiments show that a TCP/IP stack has a limited scalability, even with a message-based type of parallelism [Bjorkman93]. The TCP scalability (3.4 ratio on an octoprocessor) is well behind the optimal speedup ratio, i.e. the number of processors. The situation is different with UDP which requires less synchronization due to its connectionless nature.

# 3 IMPROVING STREAMS BASED COMMUNICATION SYSTEM

In this section, we first analyze the benefits of a demultiplexed architecture. Then we identify the possible improvements to the STREAMS environment and its use, we analyze the opportunity of a parallelized implementation, and we see how the access method to the transport protocols.

In order to comply with the Open System constraints (Section 2.1), great care has been paid to preserve the upper API which is the warrant of application portability. The other internal interfaces are sometimes modified or bypassed when required or when they are of little benefit. The question of the independence between the various components is more controversial. We kept a clear separation whenever possible, but we removed them when they seriously compromised performance.

## 3.1 A demultiplexed STREAMS stack

The limitations of classic STREAMS stacks (Section 2.2.3) and the theoretical benefits of demultiplexing (Section 2.4) led us to design a “demultiplexed” architecture (Figure 2), where all the (de)multiplexing tasks are performed at a very low level, in the “anchorage driver”. When an application opens a transport endpoint, a stream is created and the appropriate protocol modules are automatically pushed onto this stream. We call “Communication Channel” (CC) the association of a stream and its protocol modules. *A dedicated CC is thus associated to each application flow*.

The goals of the anchorage driver are to be an anchorage point for the various streams, to serve as a common DLPI-like interface to the lower network interfaces, and to demultiplex incoming packets only once for the upper layer protocols.

Tasks not related to a given transport protocol (packet forwarding, ICMP processing, IP fragment reassembly) are handled by dedicated modules. They are pushed onto special streams during configuration setup. The default TCP/IP stream is used in several circumstances, in particular during a graceful connection close when an application closes its transport endpoint while there remains packets to exchange. In that case the TCP control block is moved to the default TCP module and all the subsequent packets are exchanged on this stream.

STREAMS is essential here because its *flexibility* is absolutely required to this demultiplexed architecture. The notion of stream and the possibility of dynamically inserting various processing modules onto these streams are essential, and no equivalent exists in BSD.

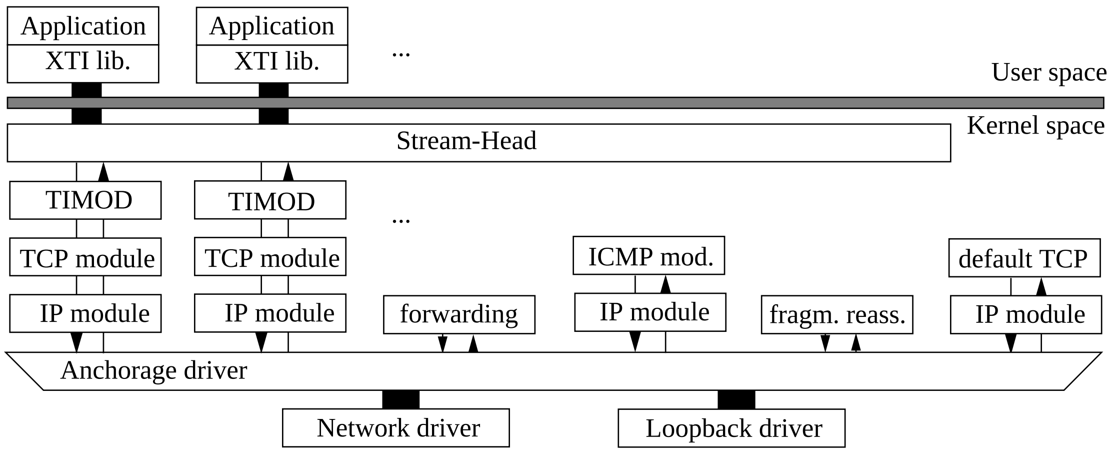

*Figure 2: Demultiplexed STREAMS TCP/IP stack.*

***Benefits of this approach on the local flow control and QoS management aspects***

This demultiplexed architecture does not significantly simplify the main data path: functionalities are relocated rather than being simplified. We now consider the local flow control and QoS aspects.The STREAMS flow control is significantly enhanced. Because this flow control is stream-wide, having a single stream from the application down to the anchorage driver means that a greedy application can immediately be blocked if the network device driver is congested. The algorithm is the following: if the device driver indicates a congestion, the anchorage driver inserts the packet in the write queue of the CC. This CC turns full and is inserted in the linked list of blocked CCs. The associated transport module detects it immediately and prevents itself from sending new packets. Finally, a watchdog timer is set in the anchorage driver in order to reschedule the blocked CCs later.

The algorithm used to decide when to block or unblock a CC can be based on congestion information as we have just seen. Otherwise it can be based on a periodic unblocking of a CC when a rate control is required on a connectionless (UDP) data flow. It is also possible to use a priority based mechanism. From this point of view, the anchorage driver is privileged: it has the knowledge of all the CC and can easily be extended to know the QoS needs of each of them. A Class Based Queuing [Wakeman95] access control mechanism can easily be implemented in the anchorage driver.

The demultiplexed architecture does more than just controlling the access to the medium: all the applications that have not the highest priority at a given time are automatically blocked. *The possibility of blocking a CC is an easy and efficient way to control the allocation of the CPU, memory and network* *resource on a per-flow basis.*

## 3.2 Additional high performance techniques

Due to the demultiplexed architecture, other improvement techniques (except ILP) became easy to implement and let us expect a significant gain. These techniques are described in this section.

### 3.2.1 Improvement of the STREAMS environment and its use

The experience gained with classic STREAMS implementations (Section 2.2.3) lead us to define several techniques. First of all, the use of queues and service routines have been banished from the main data flow. In our implementation, data messages are always processed in the `put` routines in order to avoid any additional delay. The STREAMS flow control is performed by inserting a dedicated message in the write queue of the transport module (message of size larger than the queue's high-water mark) and removing it when the blocking condition is removed.

Secondly, in order to solve the buffering problem at the receive side, we have used an option (available in our STREAMS environment) telling the Stream-Head to fill the application's buffer when possible. But this solution is not portable (not defined in [STREAMS90]), and in case of interactive traffic where it is not possible to wait till the end of the buffer filling, an additional message must be sent upward to inform the Stream-Head. This working mode is now automatically set by the XTI library.

### 3.2.2 Parallelization of the demultiplexed STREAMS stack

STREAMS has been extended to facilitate the development of components on a symmetric multiprocessor platform [Garg90, Kleiman92, Saxena93]. These extensions, which are proprietary, define several levels of parallelism (Figure 3), according to the span of the mutual exclusion section (e.g. the module level allows a single thread to execute in a driver or module). Non parallelized components will run with minimal changes if a module level is used, those that only share data between the input and output flows can benefit from the queue pair level, those that have no common data can use the queue level, and if one wants to minimize lock manipulations while preserving parallelism between the various streams, the stream level is adequate.

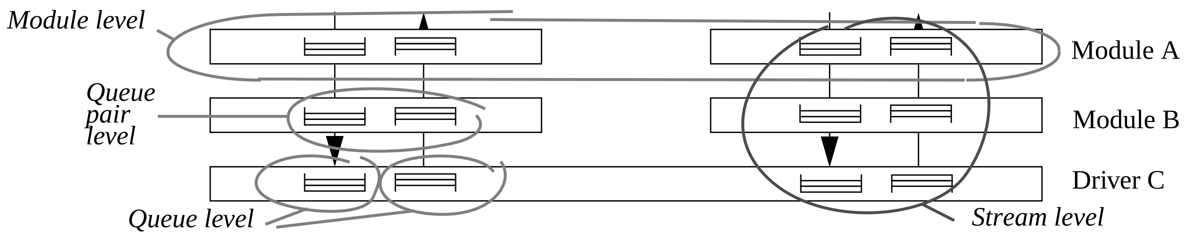

*Figure 3: The main parallelism levels found in STREAMS.*

These parallelism levels can be implemented in two ways:

- Use *locks* within STREAMS [Kleiman92, Saxena93]: A thread that wants to perform some work on a given queue must first acquire the associated lock.
- Associate *Synchronization Elements* (SE) to the queues: When a thread cannot perform operations on a queue because its SE is already owned, the request is registered. The work will be handled later by the thread that owns the SE. This mechanism maximizes the CPU utilization under heavy load (fewer context switches), but it also increases the amount of processing when no conflict occurs. The STREAMS framework we use follows this approach.

***Extension of the parallelism to the whole stack***

The demultiplexed STREAMS stack naturally multiplies the access points to the protocol components. The "per-context" parallelism (called "horizontal" in STREAMS jargon) available in TCP and UDP is extended to the whole communication stack. The parallelism is now of "*connection*" type while the BSD stack parallelism is of "*message*" type.

***Reduced use of locks***

With a classic STREAMS stack, the "queue pair" level cannot be used to synchronize the TCP input, output, and timer processing, because work is done in multiple locations. It has been made possible with the demultiplexed stack by the early demultiplexing of packets (packets are now received on the right queue), and by the integration of timer processing within TCP (if a time-out routine identifies some work to be done on a TCP context (e.g. delayed ACK), a message is created and sent to its queue). Thanks to STREAMS synchronization, the use of private locks is reduced to its minimum. Within TCP, it mainly consists in protecting the shared list of contexts during its traversal or modification (context creation or removal).

Then we have integrated the two TCP time-out routines (fast and slow) in a single one. The resulting routine is now scheduled 4 times per second. Delayed ACK processing is always done, whereas timer processing is done every other time. So, there are fewer interrupts (4 IT/s instead of 5+2), and fewer lock manipulations (the list of TCP contexts is crossed 4 times/s instead of 5+2).

STREAMS synchronization facilities proved to be costly and not always required. So we have defined an additional level that imposes no synchronization. It is up to the developer to add its private synchronization when required. This level is now used by TIMOD, IP and the anchorage driver.

### 3.2.3 Improvement of the access method

We have also improved the access method to the demultiplexed stack. Section 2.2.3 has shown that the limitations of the access method derive from the late knowledge of the TCP segment boundaries. We thus propose to *move the TSDU segmentation functionality from TCP to the API library* (XTI in our case). This technique consists in identifying in the application data flow the (full sized) segment boundaries, and controlling the way the kernel buffers are allocated and data copied into them (Figure 4).

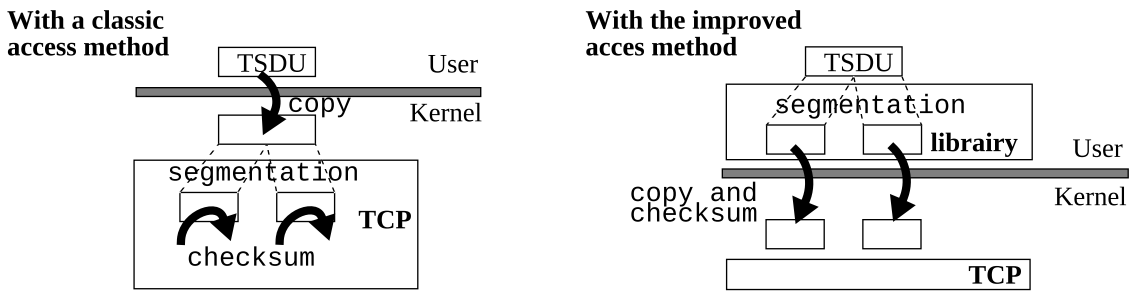

*Figure 4: Copy and checksum calculation without/with the improved access method.*

The advantages are numerous:

- user-to-kernel data copy and checksum calculation can now be integrated (ILP),
- a segment is always contained in a single kernel buffer, thereby simplifying the lookup, duplication and freeing operations in TCP outgoing list, as well as the copy to the device driver buffer,
- room can be reserved for the future protocol headers which saves a buffer allocation, and
- the number of system calls can be significantly reduced when an application uses small TSDUs and is not concerned by latency, because the library waits until it can fill a full sized segment.

On the other hand, this access method creates certain difficulties. For instance, TCP can resegment the anticipated segments of the library in case of a window probe. In that case, a new buffer is allocated and data is copied to this buffer to preserve the "one segment, one buffer" rule. Also, the necessity to return control of its buffers to the application often requires the copy of unsent data to an internal library buffer.

This technique has the advantage to *be "almost" transparent to applications* since everything is hidden in the access method (library, Stream-Head, and TCP). Yet there is an exception: by default the library accumulates data. A flushing mechanism is provided to send off data to TCP when the application has no more data to send or when it is concerned by latency. This is notified by a "`PUSH`" flag, set by the application, and only known by the library. Doing so, the API is slightly modified, but as this access method is optional, legacy applications keep on working.

Another point is that a new STREAMS system call, `putextmsg`, has been designed. It recognizes the boundaries of the data blocks (i.e. segments identified by the library), allocates the kernel buffers as required, and performs the copy and checksum. In order to favor performance, several segments are processed in a single system call (not shown in Figure 4). Creating a new system call obviously compromises the portability of the solution, but we believe that it highlights the inadequacy of the traditional access methods and system calls to new needs (Section 5.2.2).

## 4 PERFORMANCE EVALUATION OF THE DEMULTIPLEXED STACK

This section quantifies the impacts of the various high performance techniques we applied. We used a tool that performs statistical performance evaluations at the application level. The transfer is either of bulk type for throughput measurements, or of echo type for latency measurements (the client sends data and waits for an echo). The machines are a 42 MHz-Power DPX/20 and a quadri-processor 66 MHz-PowerPC601 ESCALA. The operating system is either AIX/325, a monoprocessor system, or AIX/4.1 [Talbot95], a multiprocessor system that also exists in a monoprocessor flavor.

### 4.1 Comparison of the BSD and classic STREAMS stacks

Figure 5 shows the performance of the BSD TCP/IP stack versus that of a classic STREAMS TCP/IP stack. There is one TCP connection over a loopback interface (1460 byte MSS). The machine is a DPX/20 running AIX/3.2.5.

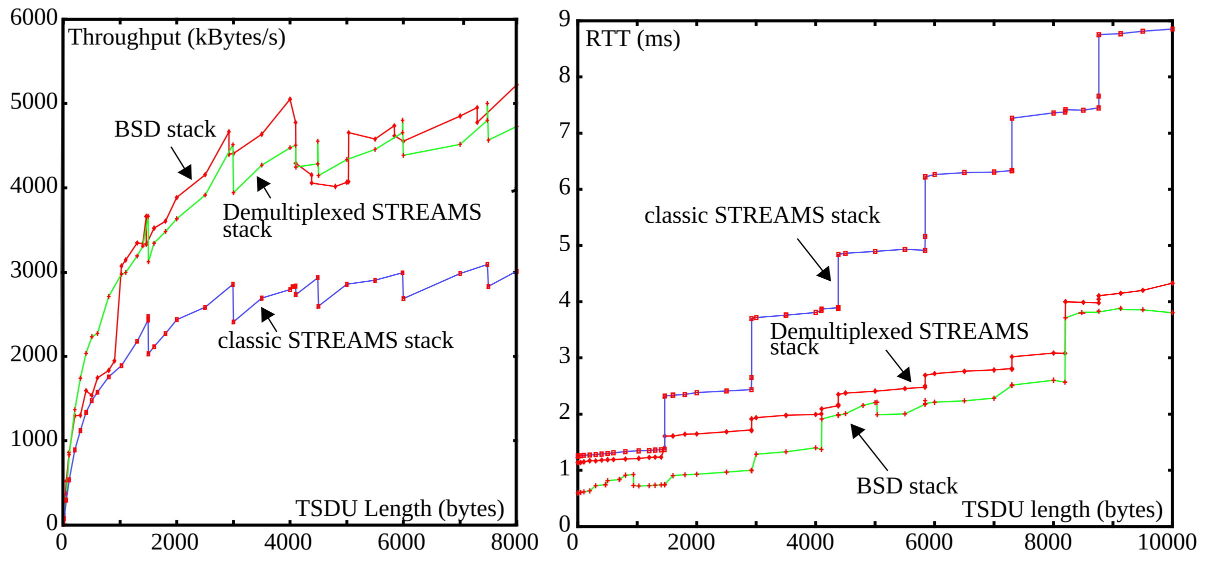

*Figure 5: Throughput (left) and RTT (right) comparisons of TCP/IP stacks.*

*These experiments highlight the limited performance of the classic STREAMS stack both from a throughput and latency point of view.* This is mainly due to the Stream-Head buffering strategy at the receive side (Section 2.2.3). It is visible in Figure 5-right at MSS multiples. The steps are much larger with the classic stack because a `receive`/`send` cycle is required for each incoming data packet.

## 4.2 The demultiplexed stack

In order to asses the demultiplexed stack performances, we have performed two kinds of experiments: basic ones that endeavor to globally evaluate the performance level of this stack, and experiments that exhibit the advantages related to its demultiplexed architecture.

***Basic experiments***

Figure 5-left shows that a carefully implemented STREAMS stack nearly reaches the same throughput level as a BSD stack on a monoprocessor. This is essentially the result of a good use of the STREAMS environment (Section 3.2.1) rather than of the demultiplexed architecture. But the demultiplexed stack results during latency tests are less good (Figure 5-right) because of the strategy used for application buffers filling.

***Flow control experiments***

This experiment is meant to appreciate the efficiency of the local flow control mechanism that blocks the application in case of an adaptor or medium saturation (Section 3.1). It consists in juxtaposing a UDP flow to a TCP connection. If TCP can adapt to the available medium bandwidth, UDP cannot.

| **TSDU size and protocol** | | **BSD stack** | | **Demultiplexed stack** | |
|---|---|---|---|---|---|
| | | **sender (kbyte/s)** | **receiver (kbyte/s)** | **sender (kbyte/s)** | **receiver (kbyte/s)** |
| 512 | TCP | 0 | 0 | 244 | 244 |
| | UDP | 2244 | 511 | 597 | 470 |
| 1024 | TCP | 0 | 0 | 165 | 165 |
| | UDP | 4703 | 964 | 780 | 780 |

*Table 2: Sending and receiving throughputs in kByte/s over Ethernet.*

Table 2 shows that when there is no local flow control (BSD), UDP can easily consume all the medium bandwidth while losing a significant portion of its packets. These losses are due to the saturation of the Ethernet driver transmission queue. This phenomenon is avoided with the demultiplexed STREAMS stack where the local flow control gives TCP and UDP a share of the network resources each.

This experiment is perhaps artificial: UDP is not suited to bulk data transfers unless the application implements a feedback mechanism [Diot95]. We believe that local flow control and end-to-end flow (or rate) control are complementary. For instance, local flow control enables a good response to transitory congestions due to bursty streams on an already loaded workstation. Because of the small transfer duration, there is no feedback mechanism. In that case, the local flow control prevents packets from being lost, reacts immediately, and is protocol independent.

But the main point is to *validate the communication channel blocking mechanism*. It proved to be both simple and efficient.

## 4.3 Parallel processing

In order to evaluate the parallelization efficiency of the stacks, tests have been performed on a quadriprocessor machine running AIX/4.1.2, in loopback mode. Processors are always 100% busy.

Figure 6 shows the performance of the BSD and demultiplexed STREAMS stacks during bulk data transfers. *The BSD stack has better performance*, especially with small TSDUs. This behavior is related to the high overhead imposed by the parallelized STREAMS environment.

The corresponding speedup with four and eight connections is shown in Table 3. We see that *the demul-*

*tiplexed stack has a better scalability* than the BSD stack. However, the speedup values remain always far below the optimal values, i.e. the number of CPUs.

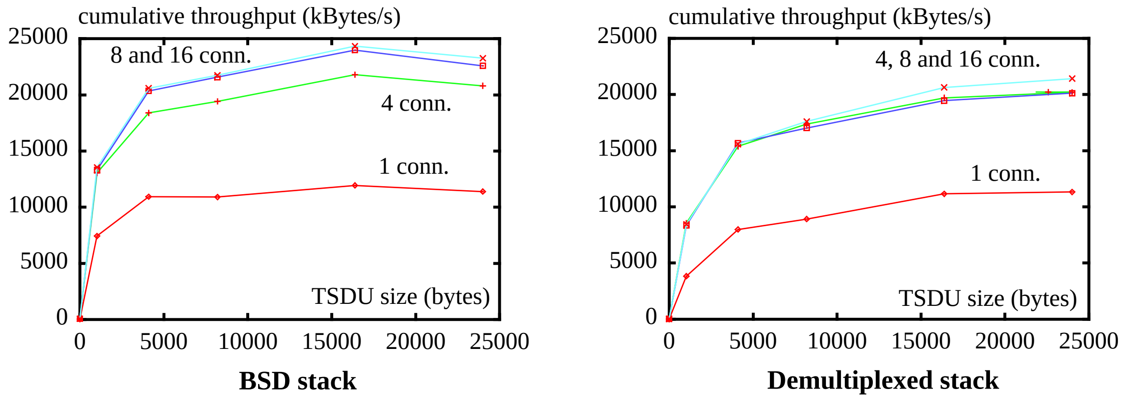

*Figure 6: Cumulative throughput comparison during bulk data transfers.*

| | BSD stack | Demultiplexed stack |
|---|---|---|
| 1 to 4 CPUs | speedup of 2.0 with 4 connections<br>speedup of 2.2 with 8 connections | speedup of 2.2 with 4 connections<br>speedup of 2.6 with 8 connections |

*Table 3: BSD and demultiplexed stack scalability for 16 kByte TSDUs.*

Lock contention experiments during bulk data transfers on a 4-CPU SMP [Roca96] have shown that *memory lock contention dominates, especially in case of the demultiplexed stack*. The reason is that a STREAMS stack makes heavy use of small buffers. They are used to hold the TPI, NPI and DLPI primitives associated to each data message, to hold time-out indications (Section 3.2.2), etc. The second source of contention for the BSD stack is the socket locks below 4 kBytes and the process locks above. In case of the demultiplexed stack, the sources of contention other than memory related are negligible. This is both due to the notion of SE and to the connectional model used (Section 3.2.2).

In all cases, it can be observed that *OS related synchronization is the main source of contention*.

## 4.4 The improved access method

We now evaluate the benefits of the improved access method. Tests are performed with the demultiplexed stack over the loopback interface, on a DPX/20 running AIX/4.1.2. Because user-level segmentation is sensitive to the MTU (Maximum Transmission Unit), two sizes have been used: 1500 bytes (like Ethernet) and 3500 bytes.

Figure 7-left shows that this access method enables *high throughput gains, up to 400% with small TSDUs*. This is due to the reduction of the number of system calls thanks to data accumulation in the library. *The asymptotic throughput gains with large TSDUs are smaller*, from 13% with a 1500 byte MTU to 28% with a 3500 byte MTU. These gains are directly related to the ILP integration and memory management improvements since data accumulation is no longer significant.

*Improvements are more limited from the latency point of view* (Figure 7-right). During these tests, data transmission requests are immediately sent to TCP (PUSH flag). Therefore the gains only arise from ILP and from the simplification of TCP processing. The tests show a 6%-8% RTT improvement with a 1500 byte MTU above 5 kBytes, and a 13%-16% gain with a 3500 byte MTU. These experiments also show that segmentation at user-level becomes *interesting when the TSDU is larger than 800 bytes*.

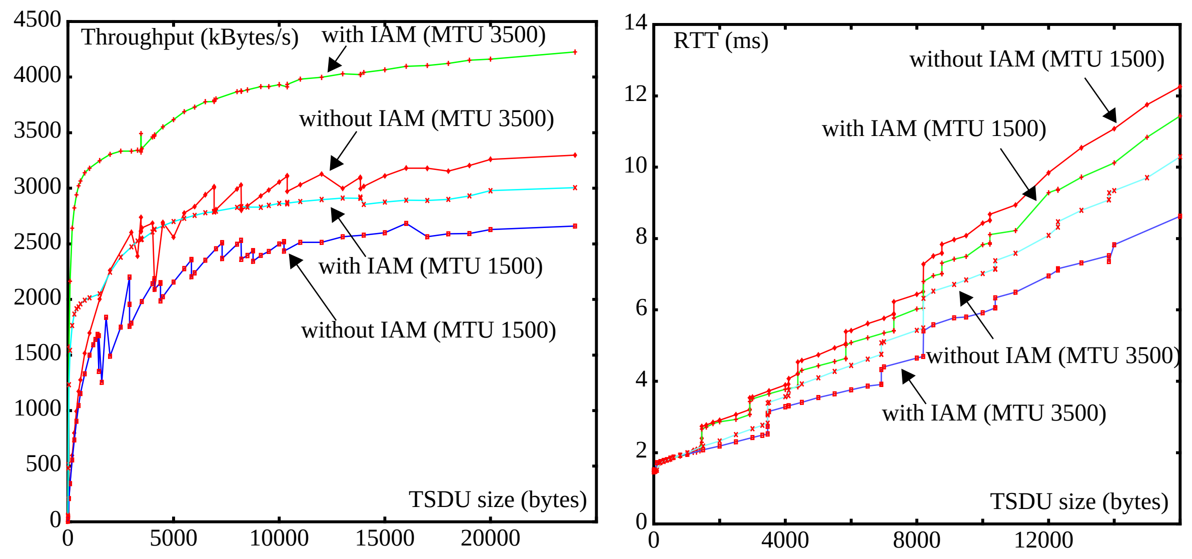

*Figure 7: Throughput (left) and RTT (right) comparisons with and without the Improved Access Method (IAM).*

***Machine-language instruction count experiments with the improved access method***

We have also analyzed the machine-language instruction distribution while sending a 1024 byte TCP segment over Ethernet with the demultiplexed stack. The machine is a DPX/20 running AIX/4.1.2.

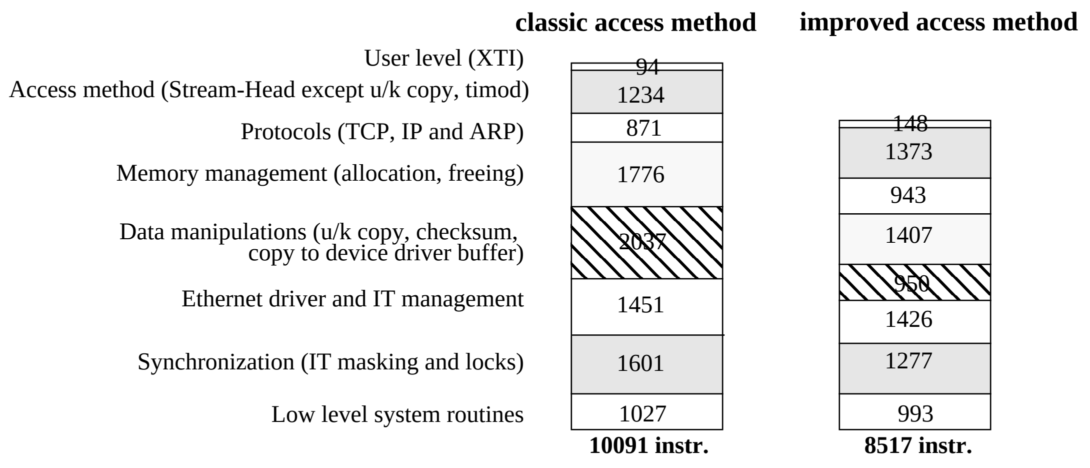

*Figure 8: Instruction count during the sending of a 1024 byte TCP segment by the demultiplexed stack.*

Figure 8 shows that *the main contribution of the improved access method comes from the integration of data manipulations* (1087 instructions). It also confirms that *improving the memory management has an important impact*. The possibility to reserve some room for the protocol headers and to have exactly one buffer per segment improves both the memory management (fewer allocations/freeings) and the synchronization (memory management heavily relies on synchronization, Section 4.3). These two aspects lead to a 693 instruction saving, i.e. roughly half that of the integration saving.

By comparison, the processing increase within the access method is limited. The necessity to control precisely data transfers during a `putextmsg` adds 265 instructions. This nearly constant overhead justifies the fact that the improved access method benefits increase with the TSDU and MTU sizes.

## 5 DISCUSSION

The previous section has quantified the efficiency of various high performance implementation techniques and has shown that their gains are sometimes far from that expected. This section discusses the impacts of the hardware architecture, operating system, and communication system architecture.

### 5.1 The hardware architecture aspect

Various characteristics of the hardware architecture influence the performance of the communication system. We have identified two directions: its parallelization and the integration of data manipulations.

#### 5.1.1 The limitations of parallelism

The advent of general purpose multiprocessor machines makes parallelism support of prime interest. Experiments have shown that three conditions are required to improve scalability:

- *An appropriate CPU support.* An appropriate use of the `load-and-reserve` and `store-conditional` instructions significantly reduces the use of locks (Section 2.4).
- *An appropriate operating system support.* Because lock contention is essentially operating system related (Section 4.3), great care must be paid when parallelizing the system functionalities. Another example is STREAMS synchronization (Section 3.2.2). It is convenient (everything is hidden) but its implementation with SE adds a prohibitive overhead.
- *An appropriate communication system architecture.* Task-based architectures must definitively be avoided (Section 2.4). The two message-based architectures (connectional for the demultiplexed stack, and message for the BSD stack) exhibited similar scalability, with a slight advantage for the connectional architecture.

In spite of these techniques, experiments have exhibited *poor scalability* results with TCP, be it a BSD or STREAMS stack. The communication system of a quadriprocessor is only 2.0 to 2.6 times faster than that of a monoprocessor.

The generalization of multiprocessor machines had a negative side effect: it has been done *to the detriment of mono-processor* performance. Indeed financial costs imply that all the platforms of a given constructor share the same operating system. The operating system is thus designed for a multiprocessor. If a few minimal adjustments are made on a monoprocessor platform, performance remain lower than with a monoprocessor operating system[2].

In conclusion we can say that:

- multiprocessors are perhaps valuable for processing-intensive applications that make limited use of shared system resources. But *their use in order to speedup communications is questionable.*
- even if the scalability is limited, a parallelized communication system remains, by lack of a better solution, the best practical solution on a multiprocessor.

#### 5.1.2 Opportunity of the improved access method

ILP is one solution to the memory bottleneck problem. But several constraints limit its benefits:

- Data manipulation functions must not be ordering constrained, i.e. the algorithm must not require data to be processed in a serial order (e.g. the CRC calculation is not suited).
- Data manipulations must remain simple. If the locality of data accesses is improved (all the manipulations are performed while the data are in the cache), this is at the expense of the locality of the data manipulation functions (the code and sometime the pre-calculated tables may be too large to fit in the cache). [Braun96] shows that ILP can lead to higher cache miss ratios.

2. This phenomenon is sometimes hidden by the fact that several optimizations, which did not exist on the old monoprocessor OS, have been implemented to compensate the performance drop.

These problems justify that the throughput gains obtained in [Braun96] by the integration of four data manipulations (marshalling, encryption, checksumming and copying) are only in a 5% to 15% range.

We have focussed on a less ambitious integration, i.e. user/kernel copy and checksum calculation. Because they are two elementary manipulations, the previous limitations do not apply. If the idea is not recent [Clark89, Clark90], we have shown how to apply it to an Open System. We can say that:

- Results greatly depend on the application nature. Interactive applications that use small TSDUs and are latency concerned are not suited.
- If using large TSDUs is beneficial, it is anyway limited by the MTU (or Path MTU). Gains are more attractive with new networking technologies using large MTUs.
- The ILP integration aspect is just one cause of performance gains. This access method has very beneficial impacts on buffer management and on synchronization. The reduction of the number of system calls is also responsible, under certain conditions, of large gains.
- Finally, the engineering implications of ILP have always been a concern. In our case, it has required a deep modification of the library, Stream-Head, and TCP components. Nevertheless the improved access method is almost transparent to the application.

In conclusion, if the improved access method is profitable, it must remain an alternative: some applications cannot take advantage of it, and a small adaptation of the application is required. Also, this optimization is not restricted to a STREAMS stack and could be applied to a BSD stack too.

## 5.2 The Operating System aspect

Our work has exhibited the importance of the communication system environment. This section analyzes its impacts and proposes several directions for its improvement.

### 5.2.1 Critic of the STREAMS environment

The current STREAMS environment suffers from many limitations:

- *Intrinsic limitations.* Many features of STREAMS are naturally costly: its message based communication, the use of standardized interfaces, its layered access method composed of several independent components split across the user/kernel boundary and using various abstractions (library function call, system call and TPI message). But this is the cost to pay for more flexibility and portability.
- *Functional shortcomings.* The fact that no standardized mechanism is provided to control and optimize the application receive buffer filling is clearly a serious deficiency. It is probably an inheritance of its initial use for TTY drivers where only few bytes are transferred at a time.
- *Implementation specific issues.* With parallelized STREAMS versions, scheduling and synchronization are complex tasks that add overhead. In particular the use of SE (Section 3.2.2) turned out to be prohibitive. The situation is less acute with a mono-processor restricted STREAMS framework.
- *Bad usage.* The fact that SREAMS promotes the concept of independent components with well defined interfaces has led to inefficient layered implementations making heavy use of service routines. On multiprocessors, a dedicated name (vertical parallelism) has been given to the kind of parallelism that derives from their use. Unfortunately, layering and vertical parallelism are source of inefficiencies.

The integrated nature of the BSD environment naturally limits these problems. But their lack of flexibility is source of other problems. For instance, because of the difficulty to compose the protocol suite that best suites the application needs, “optional” protocols like RPC are moved out of the kernel which limits their efficiency [Hutchinson91]. This lack of flexibility also prevents the design of a demultiplexed architecture which heavily relies on STREAMS advanced features.

### 5.2.2 Propositions for the improvement of the communication system environment

If the STREAMS environment is not perfect, it can be improved.

***Improving synchronism in the STREAMS environment***

Our work has shown that STREAMS offers a good potential for efficient communication systems. Its intrinsic limitations are not the most expensive features, and the other factors (functional shortcomings, implementation issues and bad usage) can be solved. One way to improve this environment and its use is to go towards *more synchronism* and *more simplicity*.

More synchronism means that data transmission request and incoming packets must be processed by a single thread. It means that the service routines, the complex synchronization mechanisms, and the idea that a message can be processed by any thread, must be avoided. The advantage is to reduce the number of context switches, to improve the cache behavior and the parallelism support. At the same time, the environment gains in simplicity.

***More flexibility in the access method***

Another aspect is the lack of flexibility in the access method. STREAMS already distinguishes the API from the generic system calls. But the system calls remain traditional and the application lacks control on the kernel data flows. Our access method has already exhibited this deficiency. [Edwards94] proposes a method to build customized APIs. A similar approach could be used, based on:

- an elementary and generic building block level (e.g. the allocation of kernel buffers, the copy between kernel and user buffers, a copy-and-checksum routine, the set up of a stream between two kernel endpoints, etc.)
- a mechanism to build customized system calls on top of these building blocks.

If these facilities are standardized and powerful enough, an application (or library) having special requirements can easily build its set of customized system calls and API.

## 5.3 The communication system architecture aspect

The last aspect we have probed is that of demultiplexed communication stack architecture. We have shown that the notion Communication Channels (CC) has many benefits: it complies with the synchronism principle (Section 5.2.2), and CCs can be associated with STREAMS flow control. The CCs thus become the basis of local flow control and QoS support and enable an easy and efficient management of the three main resources: memory, processor and communication medium.

The QoS support we propose has a local scope (i.e. at the operating system level). A general end-to-end solution requires of course the use of additional application and protocol mechanisms. It has therefore a more limited scope than the QoS-A system [Campbell94], meant to provide an extensive QoS support, including strict guaranties. But doing so requires a complex architecture. On the opposite, our demultiplexed approach provides a simple and efficient way of resource allocation sufficient for most applications that do not need strict guaranties.

Concerning the adequacy of the demultiplexed stack with the Open System constraints, our solution brings intelligence to dumb components (anchorage driver), and requires a slight modification of the DLPI interface (between the IP modules and anchorage driver). But these points are internal to the communication stack and do not compromise the upper (TP) and lower (CDLI, below DLPI) interfaces.

# 6 CONCLUSION

In this paper we have studied how high performance implementation techniques can be applied to the communication system found on current workstations. First of all, we have shown that using an Open System (i.e. UNIX) creates numerous constraints that are often opposed to many high performance techniques. We can mention the independence between components principle, the necessity to preserve a certain compatibility level from the protocol, user interface, and implementation framework points of view. Secondly, the impacts of the hardware architecture, of the operating system, and of the communication stack architecture on the efficiency of the communication system have been analyzed. We found that these two kinds of factors seriously compromise the communication system efficiency and can lead certain high performance techniques to have a lower efficiency than expected.

This paper has shown how efficient data transmission are nevertheless possible. In particular, we have studied what benefits can be drawn from more simplicity and more synchronism in the communication environment, from direct data paths between the applications and the device drivers, and from a limited ILP integration. The notion of direct data paths, i.e. of demultiplexed communication stack architecture, associated to the flow control functionalities of STREAMS, enabled an easy but efficient way to control the allocation of the CPU, memory and medium resources on a per data-flow basis. Therefore, it provides the basic mechanisms for a system support of QoS management. We have also shown how to improve the access method of the demultiplexed stack and integrate data manipulations. If this improved access method is beneficial, it must remain an alternative.

Globally, the STREAMS environment, once solved a few functional limitations and used correctly, proved to be attractive. Indeed, the implementation environment must be seen from a general standpoint that goes well beyond its sole use for a TCP/IP stack. A flexible environment enables the building of complex communication systems that easily interoperate with other processing components thanks to an easy interconnection. The advent of multimedia systems will rapidly stress the need for flexibility.

## BIBLIOGRAPHY

[Biersack94] E. Biersack, E. Rütsche, T. Unterschütz, “Demultiplexing on the ATM adapter: experiments with Internet protocols in user space”, HIPPARCH’94, Sophia-Antipolis, France, December 1994.

[Bjorkman93] M. Bjorkman, P. Gunningberg, “Locking effects in multiprocessor implementations of protocols”, ACM SIGCOMM’93, September 1993.

[Braun96] T. Braun, C. Diot, “Performance evaluation and cache analysis of an ILP protocol implementation”, IEEE/ACM Transaction on Networking, June 1996.

[Campbell94] A. Campbell, G. Coulson, D. Hutchinson, “A Quality of Service architecture”, ACM SIGCOMM Computer Communication Review, Vol. 24, Number 2, April 1994.

[Castelluccia95] C. Castelluccia, P. Hoschka, “A Compiler-Based Approach to Protocol Optimization”. Third IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems, August 1995.

[Chesson91] G. Chesson, “An introduction to the XTP/PE project”, Slides presented at Interop’91, 1991.

[Chesson94] G. Chesson, “Experiments with lock-free data structures, parallel communication software”, slides presented during HPN’94, Grenoble, France, June 1994.

[Clark89] D. Clark, V. Jacobson, J. Romkey, H. Salwen, “An analysis of TCP processing overhead”, IEEE Communication Magazine, pp. 23-29, June 1989.

[Clark90] D. Clark, D. Tennenhouse, “Architectural considerations for a new generation of protocols”, ACM SIGCOMM '90, Philadelphia, pp. 200-208, September, 1990.

[Diot95] C. Diot, C. Huitema, T. Turletti, “Multimedia applications should be adaptative”, HPCS workshop, Mystic (CN), August 1995.

[Druschel93] P. Druschel, L.L. Peterson, “Fbufs: A high-bandwidth cross-domain transfer facility”, Technical Report TR 93-5, University of Arizona, also appears in the Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, December 1993.

[Druschel94] P. Druschel, L. Peterson, B. Davie, “Experiences with a high speed network adaptor: a software perspective”, SIGCOMM’94, London, pp. 2-13, September 1994.

[Edwards94] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Calamvokis, C. Dalton, “User-space protocols deliver high performance to applications on a low cost Gb/s LAN”, SIGCOMM’94, pp. 14-23, September 1994.

[Edwards95] A. Edwards, S. Muir, “Experiences implementing a high performance TCP in user-space”, SIGCOMM’95, Cambridge, pp. 196-205, August 1995.

[Feldmeier93] D. Feldmeier, “A framework of architectural concepts for high-speed communication systems”, IEEE Journal on Seclected Areas in Communications, May 1993.

[Garg90] A. Garg, “Parallel STREAMS: a multiprocessor implementation”, USENIX, Vol 3, No 1, pp. 163-176, Winter 1990.

[Hutchinson91] N. Hutchinson, L. Peterson, “The x-Kernel: an architecture for implementing network protocols”, IEEE Transactions on Software Engineering, Vol 17, No 1, January 1991.

[IBM92] “AIX version 3.2 for RISC System/6000$^{TM}$: Kernel extensions and device support programming concepts”, IBM technical documentation, SC23-2207-01, pp. 3-31, January 1992.

[Jacobson92] V. Jacobson, “Design changes to the kernel network architecture for 4.4BSD”, slides presented during a class, http://ftp.ee.lbl.gov/talks/vj-nkarch.ps.Z, May 1992.

[Jacobson93] V. Jacobson, “TCP receive packet processing in 30 instructions”, reposted by C. Partridge, Usenet, comp.protocols.tcp-ip Newsgroup, Message-ID <1993Sep8.213239.28992@sics.se>, ftp://ftp.ee.lbl.gov/email/vanj.93sep07.txt, September 1993.

[Kleiman92] S. Kleiman, J. Voll, J. Eykholt, A. Shivalingiah & ali, “Symmetric multiprocessing in Solaris 2.0”, COMPCON, San Francisco, Spring 1992.

[Montz94] A. Montz, D. Mosberger, S. O’Malley, L. Peterson, T. Proebsting, J. Hartman, “Scout: a communications-oriented operating system”, Research Report TR 94-20, University of Arizona, June 1994.

[Mosberger95] D. Mosberger, L. Peterson, S. O’Malley, “Protocol latency: MIPS and reality”, Technical Report TR 95-02, University of Arizona, Tucson, 1995.

[Roca96] V. Roca, “Architecture hautes performances pour systèmes de communication”, PhD dissertation (in French), Grenoble, France, January 1996.

[Saxena93] S. Saxena, J. Peacock, F. Yang, V. Verma, M. Krishnan, “Pitfalls in multithreading SVR4 STREAMS and other weightless processes”, USENIX Winter’93, San Diego, pp. 85-96, January 1993.

[Schmidt94] D. Schmidt, T. Suda, “Measuring the impact of alternative parallel process architectures on communication subsystem performance”, Fourth IFIP Workshop on Protocols for High-Speed Networks (PfHSN’94), Vancouver, Canada, pp. 123-138, August 1994.

[Stevens90] W. R. Stevens, “UNIX network programming”, Prentice Hall, ISBN 0-13-949876-1, 1990.

[STREAMS90] “STREAMS Programmer's Guide”, UNIX System V Release 4, 1990.

[Talbot95] J.Talbot, “Turning AIX operating system into an MP-capable OS”, USENIX, New Orleans, January 1995.

[Thekkath93] C. Thekkath, T. Nguyen, E. Moy, E. Lazowska, “Implementing network protocols at user level”, ACM Sigcomm '93, New York, pp. 64-73, September 1993.

[Tracey94] J. Tracey, A. Banerji, “Device driver issues in high-performance networking”, USENIX, High-Speed Networking, California, pp. 31-43, August 1994.

[Wakeman95] I. Wakeman, A. Ghosh, J. Crowcroft, V. Jacobson, S. Floyd, “Implementing real time packet forwarding policies using STREAMS”, USENIX, New Orleans, pp. 71-82, January 1995.

[XTI93] “X/Open Transport Interface (XTI) version 2”, X/Open Company, Ltd., September 1993.