# A high performance Streams-based architecture for communication subsystems.

Vincent Roca [a, b] and Christophe Diot [c]

[a] BULL S.A.
1, rue de Provence; BP 208; 38432 Echirolles cedex; France
e.mail: V.Roca@frec.bull.fr; fax: (33) 76.39.76.00

[b] LGI - IMAG
46, avenue Felix Viallet; 38031 Grenoble cedex; France

[c] INRIA
2004 route des Lucioles; BP 93; 06902 Sophia Antipolis; France
e.mail: christophe.diot@sophia.inria.fr; fax: (33) 93.65.77.65

**Abstract**
During the last few years several ideas have emerged in the field of communication stack architectures. Most of them question the use of the OSI layered model as a guideline to protocol implementation. In this paper we apply some of these ideas to a Streams-based TCP/IP stack. We define a lightweight architecture that aims at reaching high performances while taking advantage of Streams benefits. In particular, we introduce the notion of communication channels that are direct data paths that link applications to network drivers. We show how communication channels simplify the main data paths and how they improve both Streams flow control and parallelization. We also propose to move TSDU segmentation from TCP to the XTI library. It brings further simplifications and enables the combination of costly data manipulations. Early performance results and comparisons with a BSD TCP/IP stack are presented and analyzed.

Keyword Codes: C.2.2
Keywords: Network protocols

## 1 INTRODUCTION.

As networks proceed to higher speed, there is some concern that the Network to Presentation layers will present bottlenecks. Several directions of research are followed in order to make up the performance gap between the lower hardware layers and the upper software layers [Feldmeier 93b]. Some of them question the standard OSI model for protocol implementation.

This will be discussed in section 3.1.

Such considerations as portability, modularity, homogeneity and flexibility can lead to choose Streams as the environment of the communication subsystem. This is the choice that has been done for Unix System V. Yet this environment also adds important overheads. Our goal in this paper is to define design principles likely to boost the performances of Streams-based communication stacks while preserving as much as possible Streams assets. These principles will have to be beneficial both to standard mono-processor and to symmetric multi-processor (SMP) Unix systems. In order to support our proposals, we describe the changes we have introduced in a Streams-based TCP/IP stack. This choice of TCP/IP is not restrictive and the architecture presented here can be used for other protocol stacks.

This paper is organized as follows: because the Streams environment is central to our work we present its main concepts in the second section. In the third section we review experiments and ideas found in the literature that question layered implementations. We also analyze in depth a standard Streams-based stack and show its limitations. In the fourth section we present an architecture to solve these problems and discuss performance measurements. Then we conclude.

## 2 THE STREAMS ENVIRONMENT.

### 2.1 The basics of Streams.

We present here the main features of Streams. More details can be found in [Streams 90]. The notion of stream, or bidirectional (read and write, also called input and output) channel is central to Streams. A stream enables the user of a service to dialog with the driver that supplies that service. Communication is done through messages that are sent in either direction on that stream. In addition to drivers, Streams defines modules that are optional processing units. Modules can be inserted (pushed) then removed (popped) at any time on each stream. They are used to perform additional processing on the messages carried on that stream, without having to modify the underlying driver. Within a driver or module, messages can be either processed immediately by the interface functions that receive them (the put() routines), or queued in the queues associated with the stream (read and write queues) and processed asynchronously by the service() routines.

Figure 1 represents a basic Streams configuration where two applications dialog with a driver through two different streams. On one of them, an additional module has been pushed. In this example, the Streams framework is entirely embedded in the Kernel space of the Unix system. If this is usually the case, this is not mandatory: this environment can be ported to the User space of Unix, or to various different operating systems and intelligent communication boards. The Stream-Head component of Figure 1 is responsible of the Unix/Streams interface and in particular of the User space/Kernel space data copy.

Streams offers the possibility to link several drivers on top of each other. Such drivers are called multiplexed drivers because they multiplex several uppers streams onto several lower streams (see Figure 2). Streams can be used for many I/O systems, but in the case of communication stacks, a standardized message format is associated to each interface between adjacent layers of the OSI model. These normalized interfaces are called TPI (Transport Protocol Interface), NPI (Network Protocol Interface), and DLPI (Data Link Protocol Interface).

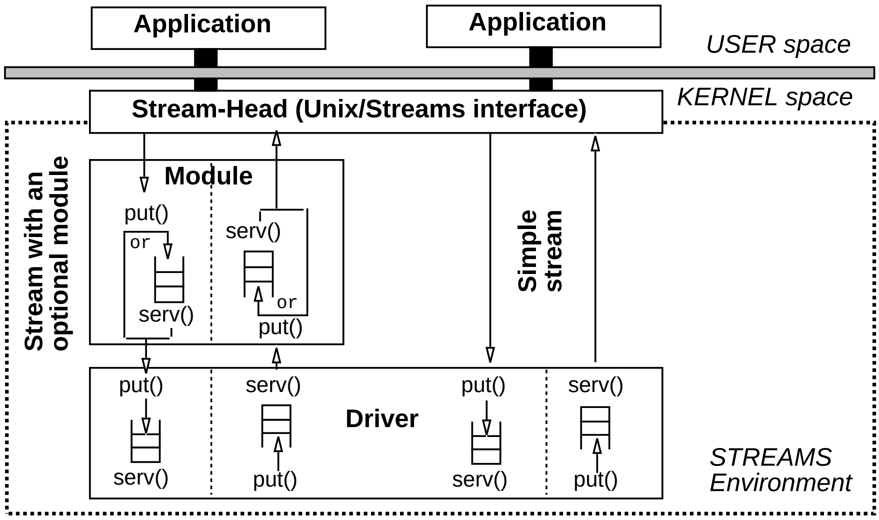


*Figure 1: A basic Streams configuration.*

Other implementation environments exist: the BSD style environment associated to the Socket access method is the most famous one but, unlike Streams, it does not impose a strict structure on protocol implementation. The x-Kernel [Hutchinson 91] is an experimental environment similar to Streams in the sense it provides a highly structured message-oriented framework for protocol implementations.

## 2.2 The parallelization of Streams.

The Streams environment has been extended to facilitate the development of Streams components in SMP systems. This extension has been widely commented in [Campbell 91], [Garg 90], [Heavens 92], [Kleiman 92], [Mentat 92a] and [SunOS 93]. These extensions all define several levels of parallelism:

. ***Global level:*** only a single thread is allowed in the driver code. Drivers coming from non parallel systems will run with minimal changes.
. ***Queue pair level:*** only a single thread is allowed for a given queue pair (read and write queues of a stream). This is used for components that only share data between the input and output flows.
. ***Queue level:*** a separate thread is allowed in each queue. This is used by components that maintain no common data.

The simplest solution to implement these levels consists in using locks within the Streams framework [Kleiman 92]. A thread that wants to perform some work on a given queue must first acquire the associated lock. A better mechanism that maximizes CPU usage consists in using synchronization elements: when a thread cannot perform some work for a queue -

detected thanks to the associated synchronization element - the request is registered and the thread goes elsewhere. This work will be handled later by the thread that currently owns the synchronization element. An intelligent use of these parallelism levels should ideally remove the need of additional locks. In fact, standard Streams-based TCP/IP stacks make heavy use of private synchronization primitives to the detriment of efficiency (see section 3.2.1).

Two concepts are common in parallel Streams:

. ***Horizontal parallelism:*** this is the queue or queue pair parallelism. Several contexts of a given driver can be simultaneously active.

. ***Vertical parallelism:*** it is similar to the usual notion of pipeline parallelism. It requires that messages are systematically processed in service routines.

# 3 STATE OF THE ART.

## 3.1 Layered architectures and performance.

The use of the OSI model as a guide to protocol implementation has been more and more questioned during the past few years for several reasons:

. A layered protocol architecture as defined in the OSI model often duplicates similar functionalities. [Feldmeier 93a] identifies: error control, multiplexing/demultiplexing, flow control and buffering.

. Hiding the features of one layer to the other layers can make the tuning of the data path difficult. [Crowcroft 92] describes an anomalous behavior of RPC that comes from a bad communication between the Socket and TCP layers. The layered implementation principle is greatly responsible of this design error.

. The organization in several independent layers adds complexity that may not be justified [XTP 92] [Furniss 92].

. The ordering of operations imposed by layered architectures can prevent efficiency [Clark 90].

To remedy these problems, several solutions have been proposed:

. Adjacent layers may be gathered: This is the approach followed by the XTP protocol [XTP 92] that unifies the Transport and Network layers in a Transfer layer. Extending the transport connection to the network layer removes many redundancies and facilitates connection management. Another example is the OSI Skinny Stack [Furniss 92], a "streamlined" implementation of a subset of the Session and Presentation layers. It has initially been designed for an X Window manager over OSI networks. Upper layers are merged in a single layer, the invariant parts of protocol headers are pre-coded and the analysis of the incoming packet is simplified.

. [Feldmeier 90] and [Tennenhouse 89] argue that multiplexing should be done at the network layer. A first benefit is that the congestion control of the network layer, if it exists, can also provide flow control functionality for upper layers. Second, because the various streams are distinguished, the QOS specified by the application can be taken into account by the network driver. Eventually the context state retrieval is minimized when multiplexing is performed in a single place.

. [Mentat 92b] describes a lightweight architecture for Streams-based stacks where transport protocols are implemented as modules pushed onto streams. This feature greatly reduces the overhead due to layering (see sections 3.2.1 and 4.1).

. Starting from the fact that memory access overhead may become predominant over computation on RISC architectures, [Clark 90] proposes the ILP principle (Integrated Layer Processing) that aims at reducing data manipulations (encryption, encoding, user/ kernel copy, checksum...). Instead of doing them in separate loops in different protocols, each data word is read once and all the required manipulations are performed while data is held in the CPU registers.

## 3.2 A case study: a standard Streams-based TCP/IP stack.

We have first analyzed a standard Streams-based TCP/IP stack where each protocol is implemented as a multiplexed driver (see Figure 2).

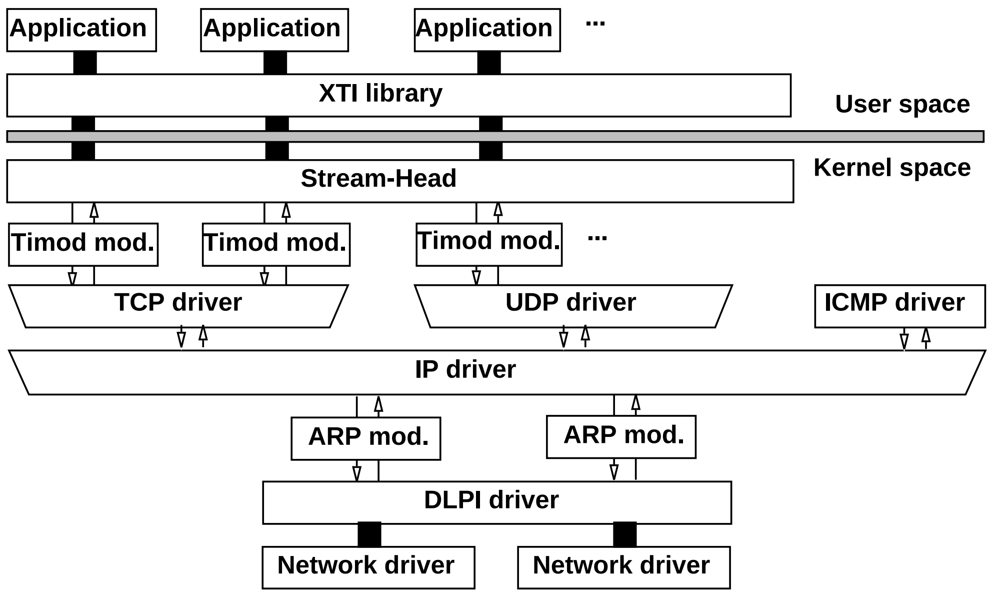


*Figure 2: A standard Streams-based TCP/IP stack.*

XTI is the transport library defined by the X/Open organization [XTI 90]; the Timod module works in close collaboration with XTI; the DLPI driver performs the interface between upper Streams-based components and the lower network drivers.

### 3.2.1 Limitations of this architecture.

***Importance of access method and interface overheads.***

In [Roca 93] we have shown that the access method overhead (XTI, Stream-head) and the driver's upper and lower interfaces overhead[1] are extremely costly. In a Streams-based XTP stack working in a loopback configuration, only 25 to 45% of the total processing time is spent for core protocol processing.

***An analysis of buffer management.***
A detailed analysis of buffer management on the output path shows that:

. Old versions of the access method (XTI) and of the Stream-Head impose a limitation to the application TSDU size. Large TSDUs are first segmented by XTI to 4 kilobytes and each segment is sent independently to the transport provider.
. Because TSDUs are copied in kernel buffers by the Stream-Head without any regard to the packet boundaries (not yet known), it is not possible to reserve room for the protocol headers. A separate buffer must be allocated and linked to the data segment.
. As mentioned in section 2, the format of Streams messages are normalized at each interface. A TSDU received by TCP consists in a buffer containing a transport data request (or T_DATA_REQ) TPI primitive followed by the data buffers. A packet created by TCP must be preceded by a buffer containing a network datagram request (or N_UNITDATA_REQ) NPI primitive. Because the TSDU boundaries differ from the packet boundaries, there is no possibility to reuse the T_DATA_REQ buffer to initialize the N_UNITDATA_REQ primitive.
. As packets may overlap several TSDUs or be only part of a TSDU, every operation in TCP outgoing list is based on expensive offset calculations. Overlapping also increases the number of buffers allocated to hold (through pointers and offsets to avoid data copy) duplicated data.

***Data buffering in Streams-based transport protocols.***
A BSD transport protocol implementation has direct access to the sending and receive data lists (sockets). This is not the case with a Streams-based stack where there are two receive data lists: one of them consists in the read queue of the Stream-Head. A Streams-based TCP driver cannot know what amount of data, waiting to be given to the receiving application, is present in this read queue. The second list is an internal TCP list used to store received data when the Stream-Head read queue is full. The receive window is estimated by examining this local TCP input queue, now often empty. This feature can increase the number of acknowledgments: window updates may now be sent for each packet received (every two packets if we take TCP optimizations into account) instead of every application buffer filled.

Another consequence is that data waiting to be sent to the application can be greater than the receive window! This is the case when both the Stream-Head read queue and the internal TCP list are full.

By default, and unlike the Socket strategy, the Stream-Head does not try to optimize the application receive buffer filling. A second consequence of giving received data immediately to the Stream-Head is that application buffers often contain only one packet worth of data.

---

1. By upper interface overhead we mean the time spent to identify the message type and queue/remove this message. By lower interface overhead, we mean the time spent to allocate and initialize a message block that identifies the message type, and then send it. Some of these operations may be by-passed in some cases; message blocks may be reused from one driver to the next one and messages may be processed immediately without queuing them.

***Streams flow control.***
Streams flow control is based on the examination of the next queue of the stream. If the queue is saturated, the upstream driver/module is informed of it and stops sending messages. Because a multiplexed driver multiplexes several upper streams on several lower streams, upper and lower streams are not directly linked. Therefore Streams flow control cannot see across a driver. If the DLPI driver is saturated, applications won't be blocked until all the intermediate queues are saturated (see Figure 3[1]). In the meanwhile, a lot of memory and CPU time re sources will be devoted to non-critical tasks to the detriment of DLPI. Similar remarks can be done on the input path.

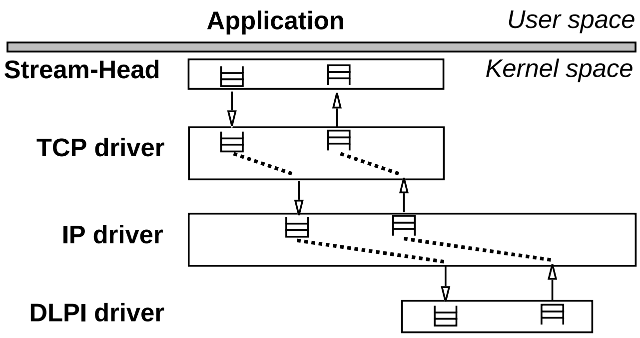


*Figure 3: Streams flow control in a standard Streams based stack.*

***Parallelization of the stack.***
The parallelization of multiplexed drivers with few access points (DLPI, connectionless protocols such as IP) creates problems because queued message processing are serialized (Figure 4 - left). In order to preserve parallelism, the standard solution consists in multiplying the access points to those drivers, namely to open several streams (Figure 4 - right). The problem can be solved but at the expense of additional complexity.

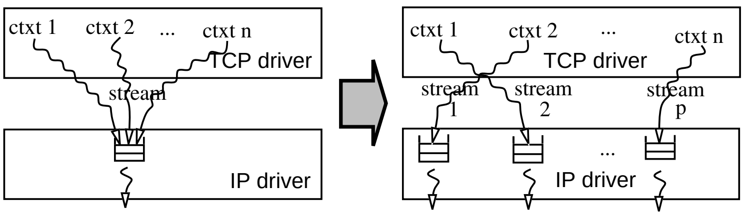


*Figure 4: Parallelization of the IP driver.*

Another problem is the important use of locks within the stack. [Heavens 92] describes the various locks used by the TCP driver: each control block structure (or TCB) is protected by a "mutex" lock, the chain of TCB is protected by a readers/writer lock, and the hash table used to demultiplex incoming packets is protected by a write lock. This solution does not take advantage of the synchronization facilities offered by Streams (see section 2.2).

1. Figure 3 is a simplified vision of reality. The actual situation is yet close to it.

## 4 AN IMPROVED ARCHITECTURE FOR STREAMS-BASED COMMUNICATION STACKS.

We have shown in the previous chapter why standard Streams-based communication stacks are inefficient. We now present two design principles that reuse some of the ideas of section 3.1, and that may shatter the myth that "Streams-based stacks are slow":

. the communication channel approach, and
. an evolution to the XTI library.

### 4.1 The Communication Channel approach.

In this approach all the protocols are implemented as Streams modules instead of Streams multiplexed drivers. When an application opens a transport endpoint a stream is created and the adequate protocol modules are automatically pushed onto this stream. We call "Communication Channel" the association of a stream and its protocol modules, because we create a direct path (i.e. without any multiplexing/demultiplexing operation) between an application and the data link component. Note that there are as many communication channels as there are transport level endpoints. Figure 5 illustrates this approach in case of a TCP/IP stack. It must be compared to Figure 2.

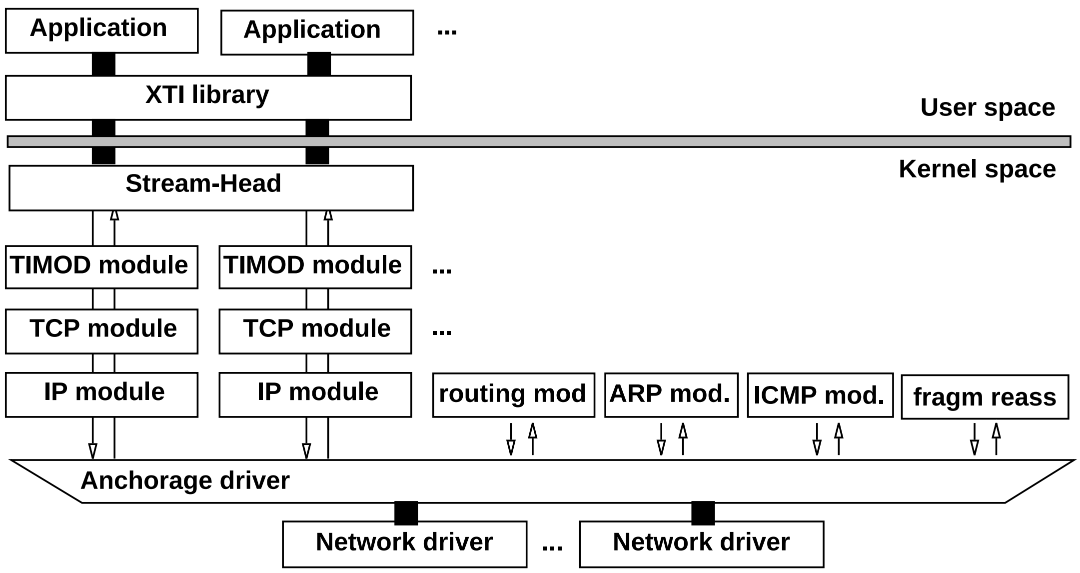


*Figure 5: Architecture of a TCP/IP stack using the Communication Channel approach.*

The only driver now used is the bottom Anchorage driver. Its goals are:

. to be an anchorage point for the communication channels,
. to serve as common interface to the lower network interfaces,
. to determine the destination channel for each incoming packet.

This third point has to be developed. In a standard TCP/IP stack, packets received from an

Ethernet network are demultiplexed in three steps: first the Ethertype field of the Ethernet header enables a switch onto IP or ARP. Then IP packets are demultiplexed according to protocol field of the IP header and given to the corresponding transport protocol. Finally, the transport protocol assigns the packet to the connection concerned. This mechanism is incompatible with our approach where a communication channel is created per transport connection; inserting an incoming packet on a given channel means it is destined to the associated connection. The search for the right channel requires that the initial three stage demultiplexing is gathered and moved to the Anchorage driver (see section 4.3.1).

The general problem of demultiplexing incoming packets could be greatly simplified when using new networks like ATM or AN1 [Thekkath 93]. They have fields in their link-level headers that may be directly associated to the destination channel.

Tasks that are not related to a given transport connection (i.e. forwarding of packets coming from one LAN and destined to another LAN, ARP and ICMP message processing, IP fragment reassembly) cannot be associated to a communication channel. They are handled by dedicated modules pushed onto special streams, out of the main data path. Multiprotocol configurations are also possible. Incoming packets are demultiplexed by the Anchorage driver in the same way and handed to the appropriate channel.

This architecture is an extension to [Mentat 92b]. In that case, IP is still a multiplexing/demultiplexing component. An advantage is to enable the use of standard DLPI drivers and to avoid the problems we have with packets not related to a local transport connection (see above). Yet this is also a limitation in the sense that it does not bring as far as possible the concept of communication channel i.e. of direct communication path. In particular the DLPI and IP components are still linked by a single stream which may prove to be a bottleneck in a SMP implementation.

### 4.1.1 The benefits of this approach.

We can identify three kinds of benefits to this approach:

***Simplified data path.***

The main data path is greatly simplified by this architecture: a module is much simpler than a driver since multiplexing is no more required. Protocols needed by an application are selected once and for all.

Another asset is the reduced use of Streams queues: there is now at most one queue pair per module, whereas two queue pairs (upper and lower) can be used in case of a driver. Anyway, the use of queues and service procedures is optional. When the outgoing and incoming flows are not flow-blocked, messages are never queued within a channel. It saves several queueing/service procedure scheduling/dequeueing operations. It also minimizes process switching: all outgoing or incoming packet's processing are performed by the same kernel thread. This is similar to the x-Kernel [Hutchinson 91] strategy that attaches processes with messages rather than protocols.

Then, as each path is distinguished, they can be optimized separately. For instance, the routing component can work on raw buffers containing the incoming packets that need routing. It saves the need of formatting and decoding messages in accordance with the DLPI interface. At the same time, communication channels can still work on well formatted DLPI messages.

Another point is that ARP (used to perform Internet to Ethernet address translation) has been removed from the main data path. On the outgoing side, the Anchorage driver calls a functions that performs the address translation if necessary (it depends on the network nature). On the incoming side, the Anchorage driver automatically identifies ARP packets and sends them to the ARP module. On the contrary, in standard stacks, each incoming packet needs to cross ARP (see Figure 2).

Then, the demultiplexing of incoming packets destined to well established connections uses a hashing algorithm [Kenney 92]. Other packets still use the original linear lookup algorithm of TCP (see section 4.1.2).

Finally we implemented a mechanism to solve the problem of independence between transport protocols and their input data lists located in the Stream-Head (see section 3.2.1). By default TCP now retains data until the Stream-Head tells it, with an M_READ message, that the application wants to read data, and until enough data has been received to fill the application buffer. Then TCP sends the required amount of data to the Stream-Head which in turn sends it to the application. Of course if the PUSH flag is set in the TCP header, data is immediately sent to the Stream-Head.

***Extended Streams flow control.***
Because the protocol modules are pushed onto a single stream, the Streams flow control is now extended to the whole communication stack. Before doing any processing, we first check if a component within our channel is saturated. If yes we immediately take appropriate measures, i.e. we stop any processing on the current message or we throw this message away. For instance, an incoming UDP datagram can be freed in the Anchorage driver, as soon as the application has been identified, if this latter is saturated. On the contrary, in a standard stack this datagram would cross DLPI and IP, be demultiplexed by UDP and then freed. Having a Streams flow control extended to the whole stack insures that saturation situations are quickly solved. A direct consequence is that more connections can be simultaneously handled.

***Better parallelization.***
The multiplication of access points to the IP and Anchorage components is a natural consequence of this approach. The horizontal parallelism available at transport level is now extended to the whole communication stack. This is true for the outgoing side as well as the incoming side where processing is parallelized as soon as packets have been affected to the right communication channel. The second benefit is that the use of Streams synchronization mechanisms is now possible in transport protocols. Because the demultiplexing of incoming packets has been moved elsewhere, transport modules can take advantage of the queue pair synchronization level (see section 2) which makes output and input processing for each transport context mutually exclusive.

### 4.1.2 The technical problems arisen.

***Demultiplexing of incoming packets in the Anchorage driver.***
As the Anchorage driver is now in charge of demultiplexing incoming packets, this latter:

. must know the protocol header formats of the transport, network and physical layers.
. must know the transport connections in order to compare the identifiers of the packets (Ethertype, local and foreign addresses, transport protocol identifier, local and foreign port numbers) with that of the connections.

The transport protocol informs the Anchorage driver of the local and foreign addresses/port numbers used on a connection as soon as possible, i.e. when the communication channel is established and the foreign address known:

. In case of a TCP active open this is done during the connection request.
. In case of a passive open TCP needs to wait until the connection is accepted. Before the connection is accepted, there is no channel associated to this embryonic connection. Incoming messages are then oriented to a common default TCP/IP channel and are demultiplexed by the default TCP module. These packets are demultiplexed two times: by the hashing algorithm of the Anchorage driver, then by the linear lookup algorithm of the default TCP module. This is penalizing during the setup stage but it favors well established communications.

A similar problem may occur during a graceful connection close: the communication channel can be released whereas there remains unsent or unacknowledged data. In that case the default channel is used.

The case of connectionless protocols like UDP is simpler. Because the foreign address can change at any time, the UDP context search only relies on the destination port number. This piece of information is known as soon as the application has bound itself and is immediately communicated to the Anchorage driver.

***Reassembly of IP fragmented packets.***
The demultiplexing of incoming packets requires the analysis of all the protocol headers. In case IP has fragmented the packet, the transport header may not be present and if the fragment has reached destination, its reassembly is required before it can be affected to the right channel. This is handled by a special module (see Figure 5). But it is well known that IP fragmentation is costly and should be prohibited [Kent 87]. [RFC 1191] describes a mechanism to find the Maximum Transmission Unit (MTU) along an arbitrary Internet path.

***Parallelization of the routing tasks.***
This approach naturally parallelizes communication channels, but other management channels are not parallelized. This is not a problem for ARP or ICMP that have little traffic, but it may be serious if this system is used as a router. The simplest solution consists in multiplying the access points to the routing module in the same way as IP is parallelized in standard stacks (see section 3.2.1).

### 4.1.3 Performance analysis.

***Test methodology.***

In this section we present performance measurements of our Streams-based stack when doing bulk data transfers over a single TCP connection and compare it with a BSD TCP/IP stack. We work in loopback mode with both the sending and receiving applications on the same machine which is a RISC monoprocessor system. Because the relative priority of the sending and receiving processes has an important impact on throughput when working in loopback mode, we set all process priorities to the same fixed and favored level. Both stacks use 16 kilobytes windows, no TCP or IP checksum, the receiving application buffer size is set to 5888 bytes, and the MTU of the loopback driver is set to 1536 bytes.

***Configuration setup time.***

In our stack the creation of a transport access point requires a stream to be opened and modules to be pushed onto this stream. With a traditional Streams-based or BSD stack, the opening of a transport access point is local to the application and to the protocol. Experiments yielded a ratio 24 between these two solutions.

*Table 1: Transport access point creation time.*

| | |
|---|---|
| BSD TCP/IP stack (socket system call) | 0.16 ms |
| Our Streams-based TCP/IP stack (t_open system call) | 3.91 ms |

***TCP/IP throughput.***

Figure 6 represents the performances of our Streams-based compared to that of the BSD stack and table 2 the behavior above 8 kilobytes. In spite of the additional overhead created by the message-based communication of Streams (see next section), performances are very similar.

The BSD curve shows a sharp throughput increase after 935 and 5031 (i.e. 4096 + 935) byte TSDUs. This value of 935 is the boundary between "small" buffer requests satisfied by allocating up to four 256 byte mbufs and "large" buffer requests satisfied by allocating a single 4096 byte cluster. In order to optimize buffering, the socket layer tries to compress mbufs: if the last mbuf of a socket is not filled and if the new mbuf is small enough to fit there, then data is copied and the new mbuf freed.

Streams has a better memory management since it maintains a pool of buffers of several sizes. The smallest suitable buffer is allocated for each request. Memory is used optimally and performances are higher.

Note that these tests only highlight the data path simplification asset of our solution. As TCP already regulates data transfers, the extended Streams flow control does not intervene here.

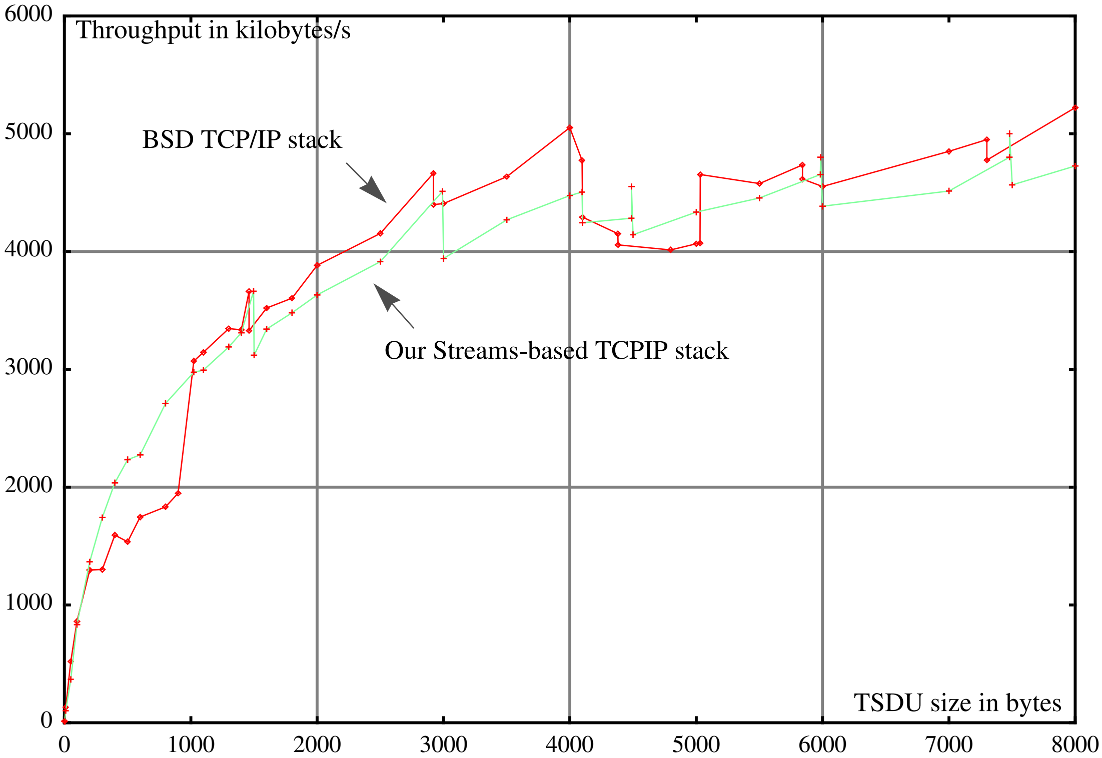


*Figure 6: Performance comparison.*

*Table 2: Performance comparison above 8 kilobytes.*

| TSDU size (in bytes) | BSD TCP/IP stack (in kilobytes/s) | Our TCP/IP stack (in kilobytes/s) |
|---|---|---|
| 12 000 | 5 045 | 4 988 |
| 16 000 | 5 274 | 5 424 |
| 24 000 | 5 243 | 5 372 |
| 32 000 | 5 170 | 5 452 |

## 4.2 Doing segmentation at User level.

Segmentation of TSDUs is usually performed by transport protocols. We have shown in section 3.2.1 that it creates several complications. We propose here to move this functionality to the XTI library.

### 4.2.1 The benefits of this approach.

Performing TSDUs segmentation in the XTI library, before crossing the User/Kernel boundary has two kinds of advantages.

First, the main data path of TCP is simplified:

. We can take advantage of the reorganization of buffering imposed by the User/Kernel data copy to reserve room for the future protocol headers.
. The T_DATA_REQ buffers associated to the segments can be reused to initialize the N_UNITDATA_REQ.
. Data manipulations in TCP outgoing list (duplication and retransmission) are now based on complete buffers (or on list of buffers if the MSS is greater than the maximum buffer size). We have the relation:

one segment <=> one buffer [1]

. because the segment boundaries are known, it is possible to combine the physical data copy with the checksum calculation as proposed in [Clark 90].

The second kind of advantage is the reduced number of User/Kernel interface crossings when dealing with small TSDUs. Indeed TCP coalescing feature is also moved to XTI; XTI now waits until enough data is received before sending anything to TCP. With applications like FTP that use one byte TSDUs when working in ASCII mode, this is particularly benefic. The case of isolated TSDUs is handled by the PUSH mechanism: if a small TSDU must be sent immediately, [RFC 793] specifies that the application needs to set the PUSH flag. Our XTI recognizes this flag[1] and sends data to TCP at once.

#### 4.2.2 The technical problems arisen.

We have supposed so far that TSDUs' segments of size MSS (Maximum Segment Size) will never be resegmented by TCP. This is usually the case. Yet, it may be required to send fewer bytes than initially expected (when its sending window is zero, TCP sends a 1 byte packet to probe the peer window). In that case, we allocate a new buffer, copy data to send in that buffer and insert it in the outgoing data list. Relation [1] is thus respected.

Because data can be stored in XTI and the control returned to the application, a strict buffer management policy must be adopted to prevent data corruption. There are several possible solutions: either the application systematically works on new buffers, or XTI copies not transmitted data into an internal buffer. The drawback of the first solution is to require the modification of applications buffer management which is contrary to our goal. If the last solution is more satisfactory, it also leads to a third copy of some data (the other two copies are at the User/Kernel and Kernel/device boundaries). Experiments (see section 4.2.3) have shown it is not a problem.

This evolution of XTI compels TCP to inform XTI of the MSS negotiated during connection opening. This is done on the first transmission request. XTI informs TCP it will perform TSDU segmentation and in the acknowledgment TCP returns the MSS in use.

As mentioned above, efficiency requires that applications tell XTI when data transfer is

1. [XTI 90] specifies that the PUSH flag found in the Socket library cannot be used through the XTI interface. Instead of adding this facility we slightly modified the semantics of the T_MORE flag that delimitates the TSDU boundaries to make its absence implicitly equivalent to PUSH. This is possible because TCP is stream oriented and does not use the notion of TSDUs.

finished. Because we can't only rely on this mechanism, we have added a timer based mechanism in XTI to force the sending of data after a certain idle time.

Finally, an evolution of the putmsg() Streams system call is needed to tell the Stream-Head to create several Streams messages, one per segment, within a single system call. This is some kind of writev() system call that, in addition, recognizes message boundaries.

### 4.2.3 Performance analysis.

***TCP/IP throughput.***

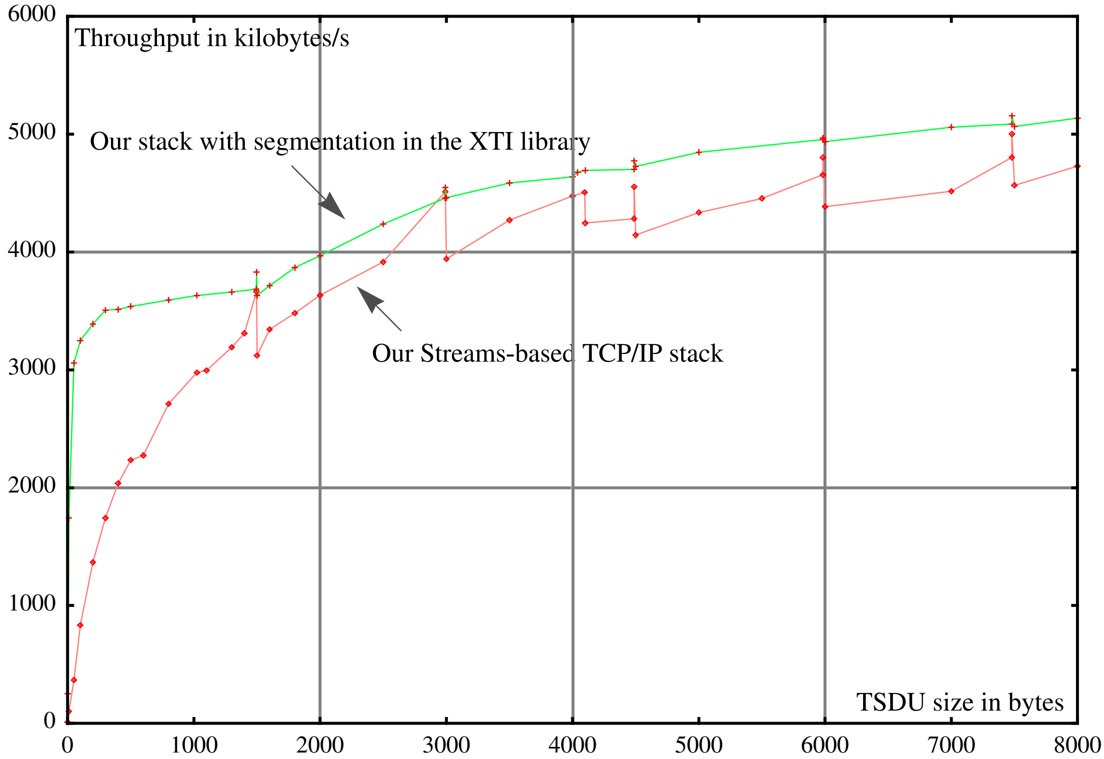


*Figure 7: Traditional vs. XTI-based segmentation of TSDUs.*

Performance results are presented in Figure 7. We see that doing segmentation in the XTI library makes the throughput curve radically different: instead of a steady increase of throughput with performance degradations after multiples of MSS, the curve reaches immediately a first knee, and then looks rather smooth with peaks for TSDU sizes equal to a multiple of MSS. We never experience sudden performance degradations after particular values of the TSDU size. There are two reasons for it:

. because XTI retains data until it can fill a packet, using small TSDUs only increases the number of calls to the XTI sending primitive. This is less expensive than doing a system call, hence the first knee.
. because we chose to return the control of TSDU buffers to the sending application, it is required to copy data that can not be sent immediately in an internal buffer of XTI. The

only TSDU sizes that do not yield additional data copies are the multiple of MSS, hence the peaks. The size of these peaks gives an idea of the overhead induced by these additional data copies: relatively small.

Note that protocol checksums are still disabled. The possibility of gathering the data manipulation loops (copy and checksum) is neither reflected in Figure 7 nor in Table 2.

***Detailed measures.***
Table 3 shows the processing time distribution of our TCP module for data transmission requests. We compare the two TSDU segmentation policies. With XTI segmentation, room has been reserved for protocol headers which saves a buffer allocation/liberation. It amounts to 8 µs (allocation, third row) plus 9 µs (liberation, not shown in Table 2). The simplification of data duplication adds another 13 µs saving. This saving increases when the segment overlaps several buffers in TCP outgoing list: two or more new buffers must be allocated to hold the additional segment parts. This often occurs if segmentation is performed in TCP, never when segmentation is performed in XTI. A total of 29 µs are saved (in comparison, the sending of a 1024 bytes TSDU requires 260 µs from the application to the loopback driver when disabling checksums). This estimation takes into account neither the reduction of the number of system calls nor the additional work required in the XTI library and Stream-head.

*Table 3: TCP performance analysis when doing TSDU segmentation in TCP vs. XTI.*

| | | Segmentation in TCP | | Segmentation in XTI | |
|---|---|---|---|---|---|
| Upper TPI interface | reception and analysis of the T_DATA_REQ message | 5 µs | 6% | 5 µs | 9% |
| TCP processing | insertion of the TSDU in TCP output list | 6 µs | 73% | 7 µs | 63% |
| | control and initialization of TCP header | 25 µs | | 17 µs | |
| | duplication of data to send | 25 µs | | 12 µs | |
| Lower NPI interface | creation of an N_UNITDATA_REQ | 13 µs | 21% | 13 µs | 28% |
| | sending of the message to IP | 3 µs | | 3 µs | |
| | | total: 77 µs | | total: 57 µs | |

The additional data copy required in the XTI library when data cannot be sent to TCP immediately amounts to 22 µs for a 1024 byte TSDU. This figure compares very favorably with the 147 µs required to send a small TSDU to TCP when this latter has not enough data to generate a packet.

## 5 CONCLUSIONS AND WORK IN PROGRESS.

In this paper we have presented a lightweight architecture for Streams-based communication stacks. We applied various ideas that have emerged during the last few years and that intend to solve some of the problems created by standard layered architectures. We have shown that a condition to reach a good performance level is to avoid designing a multiplexed driver for each protocol, as the OSI model and Streams may urge us to do. Using Streams modules instead of drivers:

. simplifies the main data path and increases performances,
. improves the Streams flow control within the stack; memory and CPU resources are affected to important tasks. A direct consequence is that more connections can be supported.
. and improves the parallelization of the stack; we take full advantage of Streams synchronization facilities, save the need of additional locks in transport protocols and increase the parallelism available at the network layer.

These improvements required to reorganize some of the protocol tasks and in particular the demultiplexing of incoming packets. We also modified the XTI library in order to make it perform the TSDU segmentation, a task usually done by TCP. This solution:

. enables some more simplifications on the output path,
. reduces the number of crossings of the User/Kernel boundary, and
. enables the combination of the User/Kernel data copy with checksum calculation.

One could think to apply these enhancements to a BSD TCP/IP stack. The concept of communication channel cannot be easily applied to a BSD stack which lacks the notions of stream and of pushable processing components. On the contrary, moving the TSDU segmentation in the Socket library is possible and does not yield other problems than those mentioned in section 4.2.2.

Portability of applications has become an important issue. In this regard if the modifications we made to our TCP/IP stack are important, they are also totally hidden to applications. Applications still open, use and close transport endpoints *exactly* as they used to do.

For the present we have shown that our Streams-based stack performs as well and sometimes better than a BSD stack. This is encouraging when considering that the message oriented aspect of Streams creates large overheads. On the contrary the BSD stack is an "integrated" stack; its function-call based communication between protocols induces a small overhead. The performance gains obtained so far are not decisive enough. Future work will unable us to go further in this quest for performances. It includes the integration of data manipulation functions, experiments highlighting the impact of our extended Streams flow control, and performance studies on multiprocessor systems. We will also compare our work with other implementation techniques, in particular with user level protocol libraries.

## ACKNOWLEDGEMENTS.

The authors thank the people at Bull S.A. who helped us and provided us a working environment. Special thanks to Michel Habert from Bull and Christian Huitema from INRIA for the interest they showed in that project and for their advice.

## REFERENCES.

[Boykin 90] J. Boykin, A. Langerman, "Mach/4.3 BSD: a conservative approach to parallelization", USENIX, Vol 3, No 1, Winter 1990.

[Campbell 91] M. Campbell, R. Barton, J. Browning, D. Cervenka & ali, "The parallelization of UNIX System V Release 4.0", USENIX, Winter'91, Dallas, 1991.

[Clark 89] D. Clark, V. Jacobson, J. Romkey, H. Salwen, "An analysis of TCP processing overhead", IEEE Communication Magazine, June 1989.

[Clark 90] D. Clark, D. Tennenhouse, "Architectural considerations for a new generation of protocols", ACM Sigcomm '90, Philadelphia, September, 1990.

[Crowcroft 92] J. Crowcroft, I. Wakeman, Z. Wang, "Layering considered harmful", IEEE Network, Vol 6 No 1, January 1992.

[Feldmeier 90] D. Feldmeier, "Multiplexing issues in communication system design", ACM SIGCOMM'90, September 1990.

[Feldmeier 93a] D. Feldmeier, "A framework of architectural concepts for high-speed communication systems", IEEE Journal on Selected Areas of Communication, May 1993.

[Feldmeier 93b] D. Feldmeier, "A survey of high performance protocol implementation techniques", Research Report, Bellcore, February 1993.

[Furniss 92] P. Furniss, "OSI Skinny stack", draft paper, February 1992.

[Garg 90] A. Garg, "Parallel Streams: a multiprocessor implementation", USENIX, Vol 3, No 1, Winter 1990.

[Heavens 92] I. Heavens, "Experiences in fine grain parallelization of Streams based communication drivers", Technical Open Forum, Utrecht, Netherlands, November 1992.

[Hutchinson 91] N. Hutchinson, L. Peterson, "The x-Kernel: an architecture for implementing network protocols", IEEE Transactions on Software Engineering, Vol 17, No 1, January 1991.

[Kay 93] J. Kay, J. Pasquale, "The importance of non-data touching processing overheads in TCP/IP", ACM Sigcomm'93, New-York, September 1993.

[Kenney 92] P. McKenney, K. Dove, "Efficient demultiplexing of incoming TCP packets", Computing Systems, Vol 5 No 2, Spring 1992.

[Kent 87] C. Kent, J. Mogul, "Fragmentation considered harmful", ACM SIGCOMM'87, August 1987.

[Kleiman 92] S. Kleiman, J. Voll, J. Eykholt, A. Shivalingiah & ali, "Symmetric multiprocessing in Solaris 2.0", COMPCON, San Francisco, Spring 1992.

[Mentat 92a] "Mentat Portable Streams", Mentat Inc., commercial document, 1992.

[Mentat 92b] "Mentat TCP/IP for Streams", Mentat Inc., commercial document, 1992.

[RFC 793] "Transmission Control Protocol", Request For Comments, September 1981.

[RFC 1191] J. Mogul, S. Deering, "Path MTU discovery", Request For Comments, November 1990.

[Roca 93] V. Roca, C. Diot, "XTP versus TCP/IP in a Unix/Streams environment", Proceedings of the 4th Workshop on the Future Trends of Distributed Computing Systems, Lisbon, September 1993.

[Streams 90] "Streams Programmer's Guide", Unix System V Release 4, 1990.

[SunOS 93] "Multi-threaded Streams", SunOS 5.2, Streams Programmer's Guide, May 1993.

[Tennenhouse 89] D. Tennenhouse, "Layered multiplexing considered harmful", Proceedings of the IFIP Workshop on Protocols for High-Speed Networks, Rudin ed., North Holland Publishers, May 1989.

[Thekkath 93] C. Thekkath, T. Nguyen, E. Moy, E. Lazowska, "Implementing network protocols at user level", ACM Sigcomm '93, New York, September 1993.

[XTI 90] "Revised XTI (X/Open Transport Interface): Developers' specification", X/Open Company, Ltd., 1990.