XTP implementation under Unix

Walid Dabbous, Christian Huitema

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

XTP Implementation Under Unix

Walid DABBOUS
Christian HUITEMA

N° 2102
Novembre 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

Rapport de recherche

1993

INRIA
SOPHIA ANTIPOLIS

XTP implementation under Unix

Walid Dabbous
Christian Huitema

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet RODEO

Rapport de recherche n°2102 — Novembre 1993 — 34 pages

Abstract: User level software protocol implementations are known to have more flexiblility. However, there ia a concern about the performance of such implementations. XTP is a new transport protocol with execution efficiency as inherent part of its design In this report we present a software implementation of the XTP protocol. Our work consisted in implementing this protocol as a set of user level library to be integrated within the application. We will present the implementation choices and some performance tests over both Ethernet and FDDI networks. We show that user level protocols implementation are not necessarily relatively slow: the untuned XTP implementation have comparable performance with kernel TCP. We also discuss the merits of the XTP protocol and its suitability for the the support high speed communication.

Key-words: Transport Protocols, High Speed Implementation

**Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65**

Implantation de XTP sous Unix

Résumé : Les implantations de protocoles au niveau utilisateur apportent une flexibilité pour l'intégration de ces protocoles dans les applications. Ceci dit, les implantations existantes ont des performances tellement médiocres que les protocoles de contrôle de transmission ont généralement été supporté par le système d'exploitation. XTP est un nouveau protocole de transport qui a été conçu afin d'avoir des hautes performances. Dans ce rapport, nous présentons une implantation en logiciel du protocole XTP. Nous l'avons implanté en une librairie de fonctionalités qui pourrait être intégrée à l'application. Nous allons décrire les choix d'implémentation et les résultats des tests de performance aux dessus des réseaux Ethernet et FDDI. Nous montrons qu'une implantation au niveau utilisateur n'est pas nécessairement pénalisante: l'implantation de XTP a des performances comparable à celle du protocole TCP implanté dans le noyau Unix. Nous discutons les mérites du protocole XTP ainsi que sa convenance pour le support de la communication à haut débit.

Mots-clé : Protocoles de transport, Implantation haute performance

Contents

List of Figures

List of Tables

0.1 Scope

This work is carried out as part of Workpackage A (Design of high-speed transport service and protocol) of the OSI95 ESPRIT project. The aim of this work was to assess the suitability of the XTP protocol for the support high speed communication. The work consisted in implementing this protocol as a set of user level functions to be integrated within the application. In this report we present the implementation choices and some performance tests over both Ethernet and FDDI networks. We also present general comments on the XTP protocol and (lack of) service definition.

Chapter 1

Introduction

1.1 High speed transport protocols

One important goal for many research projects is high speed operation of networking applications. As networks proceed to higher speeds, there is some concern that the existing transport protocols will present a bottleneck. Several alternatives have been proposed such as tuning of standard protocols [2, 4], outboard protocol processors [20] and new transport protocols design [3]. The design group of the Protocol Engine Project chosed a combination of the second and the third alternatives: the design of a new protocol with execution efficiency as inherent part of the design process and the production of a hardware implementation of this protocol on a chip-set. The resulting Protocol Engine should be able to perform real time packets processing initially over FDDI and in the future over gigabit networks. Real time packets processing means the ability to deliver back-to-back packets to the host at the same rate at which they arrive from the network.

1.2 Enhanced transport services

The OSI model implies a separation of functions in layers. The transport layer is responsible of the transmission control of application data units. There is only one connection oriented transport service. The application needs are expressed in terms of quality of service parameters such as the transit delay or the maximum throughput. However, this scheme is not sufficient to fulfill the requirements of a broad range of applications. On one hand, there is no enforcement of the quality of service parameters during the lifetime of the connection. One the other hand, applications need more than a set of QOS parameters to control the transmission. A set of requirements of the multimedia applications is cited in the ELIN-1 document [15]. The work in that document shows that applications need to be involved in the choice of the control *mechanisms* and not only the parameters of a unique transport service. This is to say that the transport service itself needs revision.

1.3 Integrated Protocol design and Implementation

The XTP protocol provides a set of mechanisms. The application selects the options that govern the data exchange. In other words, this approach means that the application will select the policy used for the transmission control. In fact, only the user has sufficient knowledge about the application to optimize the parameters of a data exchange. This approach is not consistent with the layered OSI model, where clear separation of data transmission control mechanisms (lower layers) and data processing functions (higher layer) is made. However, the integration of transmission control functions within the application has some advantages:

- the application can combine costly functions (byte oriented manipulations) in a more efficient way. Results reported in [5] indicated that a MIPS processor performing the copy and the checksum on data can run at 60 Mb/s if these functions were done separately and at 90 Mb/s if they were combined in a hand coded unrolled loop. The effect would be much more pronounced if several of functions necessary to the application (like e.g. presentation encoding, encryption and checksumming) were combined.
- the application can easily adapt to the network resources changes by appropriate steps: instead of reducing the window size at the transport level in response to a congestion indication[1], a videoconference application may chose to degrade either the quality or the frequency of the images in order to adapt to the available bandwidth.

In order to fulfill the application needs, two design principles should be adopted:

- to have a flexible architecture where the application can select and control the mechanisms needed for the data exchange,
- to have proper implementation techniques to combine the required functional modules, and to define the syntax used for the data exchange.

The OSI approach was to discharge the application from the transmission control functions by defining the transport service . The applications are *transport service users* according to the layered reference model. The integrated design and implementation approach is to let the application be inside the "control

[1]a packet loss or a duplicate acknowledgment

loop", but it still needs to define how the application level parameters will be mapped onto network parameters and control functions. According to the diversity of the applications two solutions are possible:

- either we define application classes and we select appropriate control functions to be integrated to these applications (i.e. we define typed transport services as proposed in [11]),
- or we use a suitable specification language to express the applications needs and we design a generic tool to derive the configuration of the protocol functions automatically.

We are currently investigating the feasibility of the second solution, and we will report on this work in a paper in preparation.

1.4 XTP a step towards integration

The interest of a user level implementation of XTP is to provide a support for the test of the architectural design described in the previous section. This work is a contribution to the assessment of the service provided by the XTP protocol. We will focus in this report on the description of the implementation. We will also compare the performance of user level XTP implementation with kernel supported TCP.

Chapter 2

XTP implementation

2.1 Introduction

This chapter describes the implementation of XTP under Unix. We implemented XTP from scratch as a user level library with kernel support via an extension of the sockets interface.

We do not aim to provide a complete and well tuned implementation of XTP. The goal of this work within task A.10 is to assess the XTP mechanisms, and to compare the protocol performance with TCP.

2.2 General Architecture

The implementation is based on an extension of the 4.3 BSD socket interface. The extension of the kernel protocols was done in order to add the support of XTP on top of IP. We ended with "XTP sockets" which will be described in the next section. The implementation architecture is depicted in figure 2.1.

The XTP Protocol Definition 3.6 mentions four components helpful in explaining the protocol operations. Figure 2.2 illustrates these components and their relationships to each other. We will indicate throughout the text the relation between this explanatory host architecture and our XTP implementation architecture.

2.3 Kernel support

The goal of the kernel support part of the implementation is to provide a "datagram" socket interface for applications using XTP. For this purpose, XTP has been defined as a new protocol on top of IP within the `AF_INET` family. The following lines have been added to the `in_proto.c` file:

```
{ SOCK_DGRAM,   &inetdomain,    IPPROTO_XTP,    PR_ATOMIC|PR_ADDR,
  xtp_input,    xtp_output,     0,              0,
  xtp_usrreq,                   ,
  xtp_init,         0,              0,              0,
},
```

2.3.1 Interface with IP

The definition of XTP as a new protocol above IP allows the switching of the IP datagrams having the `IPPROTO_XTP` value in the `protocol` field of the IP header. They will be passed to the `xtp_input` routine. On the other hand, packets from `xtp_output` will be passed to `ip_output`.

2.3.2 Socket level interface

User applications have access to the XTP socket interface. The following system calls are possible:

- `socket`,
- `bind`,
- `connect`,
- `send`,
- `sendto`,
- `writev`,
- `recvfrom`,
- `readv`,
- `select`.

Most of these system calls are mapped onto the corresponding routines implemented in the file `xtp_usrreq.c`. The `socket`, `sendto` and `recvfrom` systems calls have the same semantics as for UDP sockets [6]. The `bind` system call assigns a *name* to an unnamed socket. The socket address is identical to the TCP/UDP address structures: an IP address and a port number. The `connect` system call permanently specifies the peer to which datagrams are sent (by saving the address in the structure containing information about the socket: the `protocol control block` (or pcb). If a socket is connected, the `send` system call can be used to send a single datagram to the peer address. The `writev` system call attempts to gather an array of buffers of data and send it as a single datagram (connected socket). The `readv` system call attempts to read a datagram from a socket and scatters the input data into an array of buffers. The

select system call examines sockets whose addresses are specified by a mask for a maximum interval to see if some of their descriptors are ready for reading, ready for writing, or have an exceptional condition pending. It can be used to wait for either the arrival of a packet or for the expiry of a timeout.

xtp_input and xtp_output

The **xtp_input** routine is called when an IP datagram with a value in the protocol field of the IP header indicating XTP. The main protocol functions performed by the routine are the following:

- test the key field of the XTP header. If the value is a return key, then access directly pcb and socket. Otherwise, the **xtp_pcblookup** is called to perform a pcb lookup operation based on the pair **<srcipaddr, key>**[1].
- if invalid context reply a DIAG packet,
- copy the source address and port in order to pass them to the user (for FIRST and other packet types),
- update the socket buffer marks in order to reflect the receipt of the complete packet (XTP header included) and wakeup the socket.

The **xtp_input** routine corresponds to the **receiver** block in the XTP host architecture. The intermediate input queues are the kernel level **mbufs**.

The **xtp_output** routine is called when the user issues a send request (any packet type). The main protocol functions performed by the routine are the following:

- allocate and fill the fields of an IP header,
- if a FIRST packet initialize destination and source IP address and port number in the address segment,
- call **ip_output**.

2.3.3 Role of kernel support

We used a minimal kernel interface in order to simplify and optimize the part of the protocol functions performed by the kernel. The provision of XTP sockets enables the application to declare "network entry points". Transmission control procedures are performed at the user level and thus can be easily configured according to the application needs.

[1]We did not use the route field, it has always the value 0.

2.4 XTP control mechanisms

In this section we present the user level procedures. We will focus on the description of the implementation problems. The most important topics are: the programming interface, the context initialization, packet header coding, buffer management, connection management, data transfer and error control (retransmission strategies, timers, checksum).

2.4.1 Programming interface

The definition of the programming interface depends mainly on the *service* definition. As there are no standard XTP service(s) definition(s), we chosed to provide the following functions for the application:

- xtp_ctx_init(),
- xtp_bloc_init(),
- xoutput(),
- xf_input(),
- xd_input(),
- xtp_timer_init()

All these functions use the context data structure defined in the annex A. Most of the fields of this context structure are self explanatory. In the following sections, we explain how some of these state information are used by the control functions. The `xtpctx` structure corresponds to the `context records` defined in the XTP Protocol Definition. The control block field reflects the programmability of the XTP protocol. It allows the application to define the parameters that governs the data exchange (see annex B).

The `service` field designate the type of service requested by the application. It may have one of the values defined in the following:

```
/*
 * Service type values
 */

#define Connection       0x01
#define Transaction      0x02
#define UnackDgram       0x03
#define AckDgram         0x04
#define IsoStream        0x05
#define BulkData         0x06
```

These service types are used by the protocol machine to select appropriate control procedures to be applied. We have considered only the `Connection` and `UnackDgram` service types.

2.4.2 Context initialization

An application using XTP should initialize a context before sending or receiving data. This can be made using the **xtp_init()** function. After this call, the `cbloc` field points to a control block with default values. The application may overwrite some or all the fields of the control block . The kernel chooses a part of the context number to guarantee uniqueness on a given host . The structure of the key field is depicted in the figure 2.3.

The application may have access to the context number by doing an ioctl system call as follows:

```
ioctl(ctx.soc, SIOCGLKEYS, (char *)&ctx.my_key);
```

When an XTP packet with the key field identifying a context on the end system sending the packet (MSB = 0) is received, the complete context lookup will be performed by the kernel. The local key number (attributed by the kernel at the initialize time) will then replace the key field. This means that users always manipulate local key numbers.

2.4.3 Packet processing

XTP header coding (decoding) is performed by the sender (receiver) process. When the sender calls `xoutput()`, the appropriate values of the header fields are derived from the context state variables. The cost (number of instructions) for a packet processing vary according to the requested type of service (datagram, connection, etc). However, the header processing is not the most cpu consuming protocol function as has been explained in [12] and [5].

2.4.4 Buffer management

The allocation of memory buffers is performed by the application. The sizes of the maximum receive or send buffers are specified in the corresponding fields in the xtpcbloc structure. These buffers are directly used by the application to compute/process data. There is no intermediate transport level buffers. When an XTP packet is received only the packet header is consulted. Only after the necessary control functions had been performed, the data is copied into the corresponding place within the user level buffer. The system calls **readv** and **writev** are used to accomplish the scatter/gather I/O. In some case, the received packet is scattered to four area buffers: the XTP header, a part of the data segment copied to the end of the user buffer, the rest of the data segment copied in the beginning of the user buffer (overwrite the processed information) and the trailer is read into a specified zone. When an `EOM` indication has been received, the application is invited to process the message. This is indicated by special return codes of the functions `xf_input` and `xd_input`.

2.4.5 Connection management

The connection is established by the implicit handshake mechanism. After the called side had performed a `xtp_ctx_init()` (resulting in an initialized context in the `Listening` state), the sender issues a call the `xoutput()`. This will put the sending context in the `Active` state, and will initiate `FIRST` and eventually subsequent packets. If the quality of service parameters match the values in the context at the called side, the connection is accepted (context state becomes `Active`) and data is copied to user level buffers (by the use of `xf_input` for this `FIRST` packet[2]). Otherwise, data is discarded and a `DIAG` packet is issued to inform the sender that the connection had been refused. Other negotiation mechanisms may also be implemented.

The application controls the data transfer by the use of the `SREQ` bit in the `Connection` service mode. When a `FIRST` ot `DATA` packet with the `SREQ` bit set is received, a control packet is returned to inform the application of the state of the receiver. During the data transfer window based flow control is performed through the use of the `alloc` field of the XTP control segments. When all the data has been transmitted, the application performs the abbreviated graceful close of the connection as detailed in XTP Definition 3.6 page 58.

Context scheduling

An application may have several active contexts in the same time (N point to point associations with different application entities). When a context becomes active, it will be referenced in the active context list (entry added in the end of the list), together with the date of the next meaningful event (timer expiry) on this context. After each processing of a received packet or a timer expiry, the application will issues a call to `xtp_timer_init()` to determine the next context to be processed and to initialize the wait timer value. A `select()` call with this timer value will be issued in order to wait for a packet receive in the meantime. The following simple algorithm is used with the `xtp_timer_init()` function to determine the next active context to be processed and the timer value: the active list is examined sequentially and the context with the earliest event date will be chosen. This technique allows to avoid complex sort procedures after each event as detailed in [7].

2.4.6 Error Control

The error control procedures within XTP are designed to facilitate reliable data transfer. The procedures provide error detection and recovery in several ways: (1) by removing old or duplicate packets, (2) by monitoring the data transfer for data loss, (3) by protecting against corrupted data, (4) by supporting resynchronization between endpoints. We will describe hereafter the implementation of the first three control mechanisms.

Old and duplicate packets

The detection of duplicate packets is performed by the inspection of sequence numbers. A special interest is given to the `FIRST` packet in order to avoid a duplication of the connection. The `sync` field is used to validate the `seq` field and to

[2]`xd_input` is used for subsequent data packets.

differentiate retransmitted **FIRST** packets. The local context variable **recv_sync** is initialized with the value of **sync** received in the first packet received from an end system during an association: i.e. a **FIRST**, **DATA**, or **CNTL** packet. Sync values received in subsequent packets are compared with **recv_sync**. If the **sync** is "less than"[3] the received value, it is an old or duplicate packet and is discarded. Another protection mechanism is provided by the socket instance part of the kernel generated key field: a process which has been killed and rerun cannot have a context with the same key value before the instance number wraps around. If context generation rate is not too high, the wrap time is greater than the maximum holding time for a context key after a network failure or termination of an association.

Timers

Certain aspects of the error control procedures rely on timers to signal when an expected event has failed to occur. Three timers are used: (1) the WTIMER (time to wait for a response to a **CNTL** packet, (2) the CTIMER limits the duration of an inactive association, (3) CTIMEOUT bounds the time spent trying to revive an inactive association (not implemented). The value of the WTIMER is static, we did not implement the update mechanism of the timeout value based on round trip times calculation. Time values are obtained using the **gettimeofday()** function. As the **time** and **techo** fields of the XTP control segment are only 4-bytes long, we adopted a representation of the time in units of $20ms$ as follows:
a_time_value = (date_sec - initdate_sec)*50 + date_µs/20000
where **initdate** is a initialized when the first context becomes active on a given host. This variable should then be updated at least once each 2.72 years.

Retransmission strategies

The selective ACK-upon-request[4] mechanism used by XTP facilitate the implementation of both well known retransmissions strategies, GoBack-N and Selective repeat. The mechanism works are described hereafter. When a sender wants to check the status of the receiver it will issue a packet with the bit **SREQ** set. The receiver will reply by a **CNTL** packet with the number of gaps in the received stream at the other end system and which spans of output sequence numbers have been received. The sender can then retransmit the gaps. The same procedure applies for GoBack-N: in this case, the receiver replies with a **CNTL** packet indicating one gap, and the number of the last octet received +1 (**hseq**). The

[3]In XTP Protocol Definition 3.6: A 32-bit unsigned number, A, is defined to be "less than" another such number, B, if (B-A)<0x80000000.

[4]The receiver may also issue a **CNTL** packet upon detecting an input data stream sequence gap, if the **FASTNAK** bit in the listen flags is set.

sender may then retransmit the "gap" (everything from the last byte received in sequence `rseq` to `hseq`).

Maintaining selective acknowledgement queues, tracking spans and gaps, can lead to considerable complexity and overhead (see the `update_spans()` routine in annex C). Both strategies are possible in this implementation. We will compare the performance of these strategies in future work. During the tests presented in the next chapter of this report, the receiver was implementing the selective repeat strategy.

Checksum calculation

The 32-bit XTP check function called CXOR, is defined as the catenation of two 16-bit functions: XOR which is a straight "vertical" exclusive-or of each 16-bit short word in a block of information, and RXOR a rotated "spiral" exclusive-or of each 16-bit short word. The algorithm to compute the checksum is given in page 78 of the XTP Protocol Definition 3.6. However, the performance of the checksum algorithm may vary in a factor of 12 according to implementation techniques. We implemented the following enhancements of the checksum calculation:

- Unroll loops,
- Calculate XOR's on words that would have the same RXOR and saving the rotate operations until the end,
- Calculate XOR's over 32-bit words instead of 16-bit words and "folding" the results down to 16 bits.

The optimized code is given in annex D. Results of the performance test show that we can run the checksum calculation at 173 Mbps on a Decstation 5000/200.

2.4.7 Conclusion

In this chapter we presented a brief description of the control mechanisms that we implemented with task A10. Within this task, the goal of this exercise is to compare the performance of this user level implementation with the well tuned TCP kernel implementation. The next chapter contains the performance evaluation of this prototype implementation. Several XTP mechanisms have not been implemented e.g. rate control, route management, multicast procedures. Some of these mechanisms (rate control) may be added to the XTP implementation in a further work concerning application driven flow and rate control.

This implementation was done in parallel with the work by Bull: *Implementation of XTP under Streams*. The Bull work consisted in porting the KRM implementation of XTP in the STREAMs environment with the goal of comparing the performance of XTP and TCP. Our user level implementation relies on a minimal kernel interface in order to test the integrated protocol implementation approach described in section 1.3.

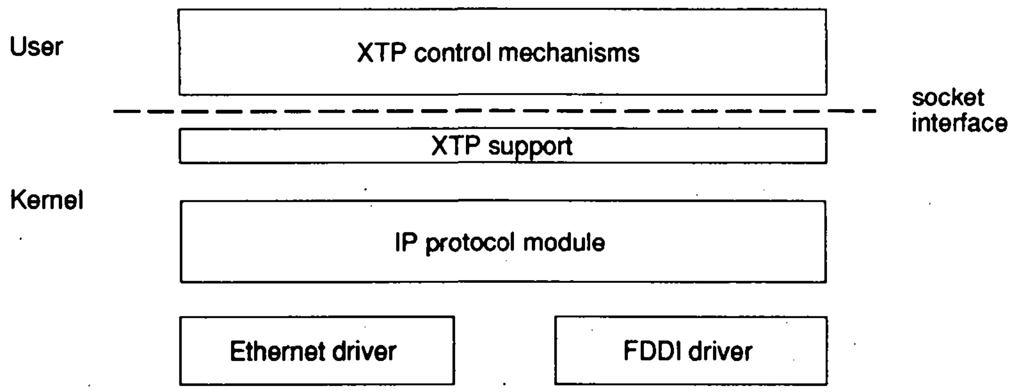

Figure 2.1: General architecture of the XTP implementation

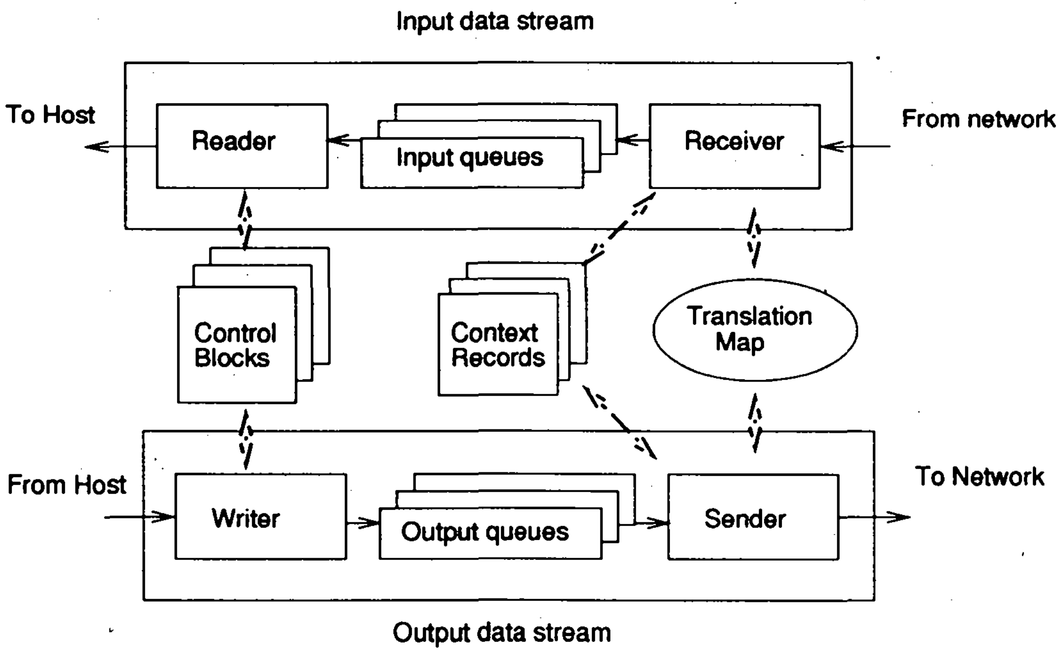

Figure 2.2: XTP host architecture as in XTP Protocol Definition

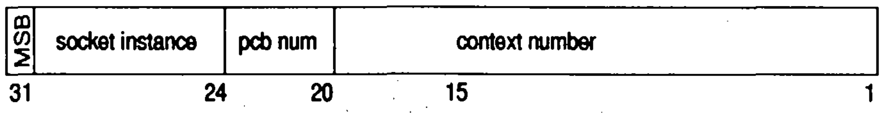

Figure 2.3: Key field

Chapter 3

Performance tests

3.1 Introduction

In this we present results of performance tests of both prototype implementation of XTP and kernel implementation of TCP. The figures should be taken very carefully: the performance of a protocol depends on implementation tunings and enhancement technique as demonstrated in [14], and the XTP implementation is not as well "tuned" as the TCP kernel implementation. Thus, it is not meaningful to make precise comparison of both protocols based on the performance figures we have. However, the figures gives an idea on the possible performance of XTP when implemented in the user level.

3.2 Performance of the checksum algorithm

The speed of checksum calculation is known to be one of the important factors that determine the performance of a transport protocol [5], [13]. That's why it is often done by the kernel. The designers of the Protocol Engine proposed to perform this task in hardware [1]. The following results show that it is possible to optimize the software implementation of this routine thus removing of the bottlenecks for the protocol performance.

The performance of TCP and ISO TP4 checksum algorithms are given in the first two rows of the table 3.1 for three different machine hardwares. These are byte or short word oriented calculations. The "standard" implementation of the XTP checksum as a loop of short word oriented exclusive-OR and rotate operations was too slow (maximum throughput of 13.67 Mbps).

We analyzed the cost of the basic operations needed for the the checksum routine in order to determine the most costly functions. The results are presented in table 3.2. The `rotate(x,n)` macro is defined as `n` circular shifts on the short word **x**. This is a relatively costly operation and should be saved as much as possible.

A first optimization (op1) consisted in doing the XORs on words with the same RXOR rotation and saving the rotate operations until the end. For this algorithm we used an array of 16 short words to store the XOR values. The (op2) optimization consisted in declaring 16 short words instead of the array in order to save indirections (unroll the loop). In (op3), the number of XOR operations is divided by 2 by XORing the 16 words at the end instead of doing it on the fly. In (op4) the same optimizations as (op3) are applied with long words calculations and then folding the results down to 16-bits. The (op4) algorithm is given in appendix D.

Checksum	Sun 3/60	Sparc IPX	DEC 5000/200
TCP	2.61 Mbps	13.3 Mbps	30.52 Mbps
ISO	1.21 Mbps	6.50 Mbps	10.69 Mbps
XTP (std)	1.36 Mbps	7.02 Mbps	13.67 Mbps
XTP (op1)	1.46 Mbps	6.83 Mbps	10.93 Mbps
XTP (op2)	2.95 Mbps	16.54 Mbps	36.96 Mbps
XTP (op3)	5.78 Mbps	31.25 Mbps	66.06 Mbps
XTP (op4)	22.59 Mbps	172.05 Mbps	173.53 Mbps

Table 3.1: Maximum throughput with different checksum algorithms

Operation	Sun 3/60	Sparc IPX	DEC 5000/200
rotate(x,1)	6.24 μs	1.00 μs	0.687 μs
Short XOR	4.00 μs	0.88 μs	0.445 μs

Table 3.2: Cost of the basic checksum operations (in a loop)

Operation	Packet length	Sun 3/60	Sparc IPX	DEC 5000/200
buffer catenation	10000 o	60 μs	12 μs	4 μs
buffer update	10000 o	80 μs	14 μs	4 μs

Table 3.3: Checksum of assembled or modified buffers

3.3 Throughput figures

Tests have been performed over both Ethernet and FDDI networks. We implemented transport level UDP, TCP and XTP clients and servers. For UDP the test scenario is the following: (1) declare and bind socket then loop on recvfrom at the server, (2) declare socket and send numbered packets at the client, (3) exit at the server when the last packet is received. For TCP the scenario is as follows: (1) declare, bind, listen, accept, then loop on recv at the server, (2) declare, connect, send from the client side, (3) exit when the last packet is received. For XTP the scenario is: (1) xtp_ctx_init(), loop on recvfrom at the server, (2) xtp_ctx_init(), then call xoutput(), (3) exit when the last packet is received correctly at the server.

The throughput is defined in the three cases as the number of received bits divided by the time interval between the reception of the first and the last packet at the server.

The following tables give the results of the tests over both Ethernet and FDDI. The machines running the client and the server are both DECstations 5000/200.

Packet size	Ethernet	FDDI
1000 o	6.90 Mbps	7.02 Mbps
1400 o	9.57 Mbps	10.32 Mbps
2000 o	9.52 Mbps	13.22 Mbps
3000 o	9.66 Mbps	16.66 Mbps
4000 o	9.47 Mbps	29.03 Mbps
4600 o	9.32 Mbps	24.53 Mbps
9000 o	9.21 Mbps	24.66 Mbps

Table 3.4: Maximum throughput for UDP

3.4 Results Analysis

One may be tempted to say that the UDP figures give the raw performance that may be obtained on either Ethernet or FDDI. However, this assertion should be taken with care: packet losses at the ipqueue level due the high number of packet transmitted during a test (100 to 500) reduce the throughput as defined hereabove. The best performance for UDP are obtained for small number of transmitted packets (15 or 20). Over Ethernet, there is no bottleneck at the UDP level: the throughput is limited by the speed of the Ethernet interface. Over FDDI, the maximum raw UDP throughput is also limited by the performance of the FDDI interface. Note that the MTU is 1460 and 4312 octets

Packet size	Ethernet	FDDI
1023 o	7.25 Mbps	8.43 Mbps
1024 o	8.35 Mbps	19.05 Mbps
1400 o	9.91 Mbps	20.74 Mbps
2000 o	9.82 Mbps	18.86 Mbps
3000 o	9.16 Mbps	18.04 Mbps
4000 o	9.00 Mbps	17.60 Mbps
5000 o	8.71 Mbps	15.73 Mbps

Table 3.5: Maximum throughput for TCP

Packet size	Ethernet	FDDI
2000 o	9.62 Mbps	18.01 Mbps
4000 o	9.65 Mbps	19.72 Mbps
8192 o	9.71 Mbps	20.38 Mbps
16384 o	9.77 Mbps	21.24 Mbps

Table 3.6: Maximum throughput for XTP

over Ethernet and FDDI respectively. 9000 octets packets are transmitted as 7 Ethernet packets and 3 FDDI frames and reassembled by the kernel at the UDP/IP layer.

The TCP figures are more interesting: no bottleneck over Ethernet, TCP can easily saturate the 10 Mbps CSMA-CD LAN. However, the performance over FDDI are limited by the control mechanisms of TCP. The highest throughput (20.74 Mbps) is lower than the best UDP performance. The socket receive buffers have been set to 65535 octets in order to increase the window of TCP. TCP Packets shorter than 1024 octets show poor performance even over FDDI. The limitation comes from the buffering mechanism within the kernel: small buffers of 128 octets are used, thus limiting the overall performance of the transmission. For packets longer than 1024 octets, the performance are quite similar with a tendency to decrease.

For the XTP tests, one should distinguish between two notions: ADU and NDU as they are defined in [16]. ADU or *Application Data Unit* is the basic unit of data exchange between application entities. NDU or *Network Data Unit* is the unit of data processing and switching within the network. For the purpose of these tests NDUs over Ethernet are 1400 bytes long and NDUs over FDDI are 4000 bytes long. The packet sizes in table 3.6 are ADU sizes. This table also shows that XTP can saturate Ethernet and operate up to a speed of 21 Mbps over FDDI. This was made possible because of the following factors:

- minimize data copying by the use of writev and readv,
- optimize the checksum calculation,
- minimal overhead by return key based context look up in the kernel

This, in fact, optimize the input processing in the normal case (no errors). This optimization is a capital condition to enhance protocol performance.

3.5 Conclusion

This chapter shows that it is possible to have high speed communication with user level implementation of the XTP transport protocol. The throughput figures should be taken as an indication of the performance of the XTP protocol with all control mechanisms implemented. However, the good performance of the implementation of the error and flow control procedures is an encouraging step in the direction of software implementation of protocols which facilitate profiling of transmission control procedures by the application.

Chapter 4

On XTP

This chapter contains general evaluation of the XTP protocol mechanisms and services. The XTP protocol is a step in the right direction for the definition a new high performance communication protocol. XTP proposed a set of well founded protocol enhancement mechanisms. However, there is no service(s) definition(s) for the XTP protocol. In the next sections, we analyze these two aspects.

4.1 The mechanisms

XTP includes several performance enhancement mechanisms like (1) fixed-length and position fields, (2) efficient 1-way implicit handshake connection setup, (3) efficient context lookup with a key-based scheme, (4) data alignment on 4-byte boundaries, (5) selective retransmission, (6) acknowledgment control (7) "on-the-fly" checksum calculation when the protocol is implemented in hardware, and other mechanisms concerning routing. These mechanisms contribute to the enhancement of the performance of the protocol. To give only one example, note the important speed increase factor for checksum calculation using 4-byte long words oriented operations. Some of these mechanisms are algorithmic enhancements like (2), (3) and (4) and some other (5, 6, 7) are driven by an important design philosophy of XTP: the programmability of the protocol mechanisms by the end systems.

4.2 The services

The programmability of XTP facilitates the profiling of the transmission control protocol by the transmitter. XTP provides a set of functional enhancements like rate control, priorities based scheduling, reliable multicast, No-error mode. The application selects the desired functions or policies via the `control block` structure.

However, this feature does not only have advantages. The lack of service definition complicate the problem of the selection of protocol profiles or policies by the application. Only the names of the five service types are cited in the XTP Protocol Definition. There is no definition of service interface. The problem of the service selection is left to the application. For this purpose, there has been considerable work within the OSI95 project [17] and elsewhere [18], [19] to define high speed transport service(s). Another possible way to solve the profiling problem is to have automatic generation of communication profiles based on a set of enhanced mechanisms as described in section 1.3. This corresponds to an horizontal approach to the layered architecture i.e. applications select and combine control functions based on the service parameters and on the cost of these functions. Some tools are required to implement this approach: the service description and specification language, the profiling tool that configures the appropriate implementation based on the service specification and on the generic protocol mechanisms. We plan to develop such tools as a continuation of our work on XTP.

4.3 Conclusion

This XTP implementation exercise helped us to deeply understand the mechanisms of XTP. The performance of the software implementation of XTP may be comparable to the performance of well tuned kernel implementation of TCP. Software implementations are not necessarily unefficient. They are, however, more easily programmable. This motivated us to consider the work about integrated protocol design and implementation and about automatic profiling of communications services for network applications.

Appendix A

Context structure

```
typedef struct
{
  char * recv_buf;  /* pointer to application data area */
  u_long saved_sync; /* see page 23 XTP Protocol Definition 3.6 */
  u_long recv_sync;
  u_long should_xkey;
  u_long my_key; /* local key number, to be put in the xkey field */
  u_long rem_key; /* remote key number to be used for reply */
  u_long sort; /* priority of this context */
  u_long hseq; /* highest seq number ever received + 1*/
  u_long rseq; /* next in seq byte to be received = hseq if no holes */
  u_long dseq; /* seq num of next byte to be delivered to application */
  u_long bseq; /* seq number of first unacked byte (at sender side) */
  u_long last_sent; /* last consumed sequence number := eseq-1 */
  u_long xseq; /* first octet "currently" being trx (!= bseq if retrx) */
  u_long lseq; /* one past the last octet "currently" being transmitted */
  long first_seq; /* for duplicate First packet check */
  u_long init_seq; /* start seq numbers from this value */
  u_long nspans; /* number of spans for this context*/
  struct xtpspan * span; /* pointer to the spans structure */
  xtpcbloc * cbloc; /* pointer to the control block */
  int      cstate;  /* state of the context */
  int      soc; /* attached socket num */
  struct sockaddr_in paddr;  /* peer address (used for connect before send) */
  struct timeb t_start, t_end; /* for throughput calculation */
} xtpctx;
```

Appendix B

Control block structure

```
/*
 * Definition of the communication structure
 * This structure represents the control block (interface)
 * between the applications (XS-users) and the
 * protocol machine.
 */

typedef struct {
    u_long      mode_flags;        /* specifies XTP options flags */
    u_long      sort_value;        /* sort value for context */
    u_long      listen_flags;      /* specifies LISTEN semantics */
    u_long      field_flags;       /* enables writing the remaining fields */
    char        service;           /* service type requested/provided */
    u_int       rbufsize;          /* read buffer size */
    u_int       wbufsize;          /* write buffer size */
    u_int       maxdata;           /* maximum output segment size */
    u_int       rwindow;           /* default allocation at the receiver */
    u_int       retry_count;       /* max number of handshake attempts */
    long        input_burst;       /* input burst size */
    long        input_rate;        /* input rate */
    long        output_burst;      /* output burst size */
    long        output_rate;       /* output rate */
    u_int       ctimeout;          /* ctimeout value */
    u_int       ctimer;            /* ctimer value */
    u_int       wtimer;            /* wtimer value */
    adrseg    address_segment;   /* connect, listen or FIRST address segment*/
    u_short     mc_drops;          /* number of multicast drops */
    u_short     mc_sreqs;          /* number of multicast sreqs */
    u_short     mc_rttgain;        /* multicast RTT gain */
    u_short     mc_rttvgain;       /* multicast RTT var gain */
    u_long      code;              /* DIAG code */
    u_long      value;             /* DIAG value */
} xtpcbloc;
```

Appendix C

Spans update

```
/* Spans array processing
 *  -- Receiver actions
 * to be called only if NOERR is 0
 */
update_spans(xh, ctx)
xtphdr * xh;
xtpctx * ctx;
{
  int reorder = 0;
  struct xtpspan * sp;
  unsigned long seq, last;

  seq = htonl(xh->seq);
  last = seq + htonl(xh->dlen) + htonl(xh->offset);

  if (seq > ctx->rseq){
    if (seq > ctx->hseq){
      struct xtpspan lspan;
      lspan.low = seq;
      lspan.high = last;
      lspan.prev = lspan.next = (struct xtpspan *)0;
      insertlast(&lspan, ctx);
      ctx->hseq = last;
    } else {
      if (ctx->nspans && seq < ctx->span->low){
/* falls before the first span, process separately */
if ( (seq == ctx->rseq) && (last == ctx->span->low)){
  /* fills in the first gap */
  ctx->rseq = ctx->span->high;
  remq(ctx->span, ctx);
} else if (seq == ctx->rseq){
  /* append to in sequence data */
  ctx->rseq = last;
} else if (last == ctx->span->low){
  /* append to first span */
  ctx->span->low = seq;
} else {
  struct xtpspan fspan;
  fspan.low = seq;
  fspan.high = last;
  fspan.prev = fspan.next = (struct xtpspan *)0;
  insertfirst(&fspan, ctx);
}
      } else for(sp=ctx->span;sp->next;sp=sp->next){
if ((seq >= sp->high) && (last <= sp->next->low)){
  /* within this gap */
  if ((seq == sp->high)
      && (last == sp->next->low)){
    /* data fills in gap */
    sp->high = sp->next->high;
    remq(sp->next, ctx);
    break;
  } else if (seq == sp->high) {
    /* append to this span  */
    sp->high = last;
    break;
  } else if (last == sp->next->low){
    /* append to next span */
    sp->next->low = seq;
    break;
  } else {
    /* in the middle of this span and the next one */
    struct xtpspan newspan;
    newspan.low = seq;
    newspan.high = last;
    newspan.next = newspan.prev = (struct xtpspan *)0;
    insertafter(&newspan, sp, ctx);
    break;
  }
}
      }
    }
  } else {
    ctx->rseq = last;
  }
}
```

Appendix D

Checksum Calculation

```
#define rotate(x,n) ((((x)<<(n))&0xffff)|(((x)&(0xffff<<(16-(n))))>>(16-(n))))
#define revrotate(x,n) rotate((x),(16-(n)))


xtp_cksum(buf, len, res1, res2)
unsigned char * buf;
int len;
unsigned short * res1, * res2;
{
        int i, halfbuf, quartbuf, align, lastp;
        register unsigned long *lp, *lp0, lrx0, lrx1, lrx2, lrx3,
            lrx4, lrx5, lrx6, lrx7, lmax;
        register unsigned short *p, xor1, rxor1, rx0, rx1, rx2, rx3, rx4,
            rx5, rx6, rx7, rx8, rx9, rx10, rx11, rx12, rx13, rx14, rx15;

        if (len == 0) return(0);
        align = (len - len%32);
        if (align ==0) align = 32;
        lastp = (len - align);
        halfbuf = len>>1;
        quartbuf = len>>1;

        lp0 = (unsigned long *)buf;
        xor1 =0;
        rxor1 = 0;
        lrx0=lrx1=lrx2=lrx3=lrx4=lrx5=lrx6=lrx7=0;
        for(i=0,lp=lp0;i<align;i=i+32,lp=lp+8){
            lrx0 ^= *lp;
            lrx1 ^= *(lp+1);
            lrx2 ^= *(lp+2);
            lrx3 ^= *(lp+3);
            lrx4 ^= *(lp+4);
            lrx5 ^= *(lp+5);
            lrx6 ^= *(lp+6);
            lrx7 ^= *(lp+7);
        }
```

```
#if defined(vax) || defined(MIPSEL)
        rx0 = lrx0&0xFFFF;
        rx1 = (lrx0>>16)&0xFFFF;
        rx2 = lrx1&0xFFFF;
        rx3 = (lrx1>>16)&0xFFFF;
        rx4 = lrx2&0xFFFF;
        rx5 = (lrx2>>16)&0xFFFF;
        rx6 = lrx3&0xFFFF;
        rx7 = (lrx3>>16)&0xFFFF;
        rx8 = lrx4&0xFFFF;
        rx9 = (lrx4>>16)&0xFFFF;
        rx10 = lrx5&0xFFFF;
        rx11 = (lrx5>>16)&0xFFFF;
        rx12 = lrx6&0xFFFF;
        rx13 = (lrx6>>16)&0xFFFF;
        rx14 = lrx7&0xFFFF;
        rx15 = (lrx7>>16)&0xFFFF;

#else
        rx1 = lrx0&0xFFFF;
        rx0 = (lrx0>>16)&0xFFFF;
        rx3 = lrx1&0xFFFF;
        rx2 = (lrx1>>16)&0xFFFF;
        rx5 = lrx2&0xFFFF;
        rx4 = (lrx2>>16)&0xFFFF;
        rx7 = lrx3&0xFFFF;
        rx6 = (lrx3>>16)&0xFFFF;
        rx9 = lrx4&0xFFFF;
        rx8 = (lrx4>>16)&0xFFFF;
        rx11 = lrx5&0xFFFF;
        rx10 = (lrx5>>16)&0xFFFF;
        rx13 = lrx6&0xFFFF;
        rx12 = (lrx6>>16)&0xFFFF;
        rx15 = lrx7&0xFFFF;
        rx14 = (lrx7>>16)&0xFFFF;
#endif

        xor1 = rx0^rx1^rx2^rx3^rx4^rx5^rx6^rx7^
             rx8^rx9^rx10^rx11^rx12^rx13^rx14^rx15;
        rxor1 ^= rx0;
        rxor1 = rotate(rxor1,1);
        rxor1 ^= rx1;
        rxor1 = rotate(rxor1,1);
        rxor1 ^= rx2;
        rxor1 = rotate(rxor1,1);
        rxor1 ^= rx3;
        rxor1 = rotate(rxor1,1);
        rxor1 ^= rx4;
        rxor1 = rotate(rxor1,1);
        rxor1 ^= rx5;
        rxor1 = rotate(rxor1,1);
        rxor1 ^= rx6;
        rxor1 = rotate(rxor1,1);
        rxor1 ^= rx7;
        rxor1 = rotate(rxor1,1);
```

```
rxorl ^= rx8;
rxorl = rotate(rxorl,1);
rxorl ^= rx9;
rxorl = rotate(rxorl,1);
rxorl ^= rx10;
rxorl = rotate(rxorl,1);
rxorl ^= rx11;
rxorl = rotate(rxorl,1);
rxorl ^= rx12;
rxorl = rotate(rxorl,1);
rxorl ^= rx13;
rxorl = rotate(rxorl,1);
rxorl ^= rx14;
rxorl = rotate(rxorl,1);
rxorl ^= rx15;

if (lastp){
    p = (unsigned short *)(buf+align);
    rxorl = rotate(rxorl,lastp>>1);
    switch(lastp>>2){
    case 7:
        xorl ^= *(p+12)^*(p+13);
        rxorl ^= *(p+13);
        rxorl = revrotate(rxorl, 1);
        rxorl ^= *(p+12);
        rxorl = revrotate(rxorl, 1);
    case 6:
        xorl ^= *(p+10)^*(p+11);
        rxorl ^= *(p+11);
        rxorl = revrotate(rxorl, 1);
        rxorl ^= *(p+10);
        rxorl = revrotate(rxorl, 1);
    case 5:
        xorl ^= *(p+8)^*(p+9);
        rxorl ^= *(p+9);
        rxorl = revrotate(rxorl, 1);
        rxorl ^= *(p+8);
        rxorl = revrotate(rxorl, 1);
    case 4:
        xorl ^= *(p+6)^*(p+7);
        rxorl ^= *(p+7);
        rxorl = revrotate(rxorl, 1);
        rxorl ^= *(p+6);
        rxorl = revrotate(rxorl, 1);
    case 3:
        xorl ^= *(p+4)^*(p+5);
        rxorl ^= *(p+5);
        rxorl = revrotate(rxorl, 1);
        rxorl ^= *(p+4);
        rxorl = revrotate(rxorl, 1);
    case 2:
        xorl ^= *(p+2)^*(p+3);
        rxorl ^= *(p+3);
        rxorl = revrotate(rxorl, 1);
        rxorl ^= *(p+2);
```

```
                rxorl = revrotate(rxorl, 1);
            case 1:
                xorl ^= *p^*(p+1);
                rxorl ^= *(p+1);
                rxorl = revrotate(rxorl, 1);
                rxorl ^= *p;
            case 0:
                break;
            default:
                break;
           }
           rxorl = rotate(rxorl, (lastp>>1)-1);
        }
*res1 = xorl;
*res2 = rxorl;
}
```

Bibliography

[1] W. Timothy Strayer, Bert J. Dempsey, Alfred C. Weaver, *XTP: The Xpress Transfer Protocol,* Addison-Wesley, 1992.

[2] Richard W. Watson, Sandy A. Mamrak. Gaining efficiency in transport services by appropriate design and implementation choices. *ACM transactions on computer systems,* Vol 5, No. 2, May 1987, pp 97-120.

[3] *XTP Protocol Definition, Revision 3.6,* PEI 92-10, January 1992, xtp-request@pei.com, Protocol Engines Incorporated, Santa Barbara, CA 93101, USA.

[4] R. Colella, R. Aronoff, K. Mills. Performance Improvements for ISO Transport. Ninth Data Communication Symposium, *ACM SIGCOMM, Computer Communication Review,* Vol. 15, No. 5, September 1985

[5] David D. Clark and David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. *Proceedings ACM SIGCOMM '90,* September 24-27, 1990, Philadelphia, Pennsylvania, pp. 200-208.

[6] Unix manual pages.

[7] G. Varghese, T. Lauck, Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. *Proceedings of the* 11th *ACM symposium on Operating Systems Principles,* ACM Operating Systems Review, Austin, TX, November 1987.

[8] Cheriton D. R., VMTP: a transport protocol for the next generation of communication systems, *Proceedings ACM SIGCOMM '86,* Stowe, Vermont, pp. 406-415, August 1986.

[9] Chesson G., XTP/PE Design Considerations, in *Protocols for High-Speed Networks,* H. Rudin, R. Williamson, Eds., Elsevier Science Publishers/North-Holland, May 1989.

[10] Clark D., Lambert M., Zhang L., NETBLT: a high throughput transport protocol *Proceedings ACM SIGCOMM '87,* Stowe, Vermont, pp. 353-359, August 1987.

[11] Dabbous W., *Etude des protocoles de contrôle de transmission à haut débit pour les applications multimédias,* PhD Thesis, Université de Paris-Sud, March 1991.

[12] David D. Clark, Van Jacobson, John Romkey, Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine,* June 1989, pp. 23-29.

[13] Marshall T. Rose. *The Open Book, a Practical Perspective on OSI.* Prentice-Hall, 1990.

[14] W. Dabbous, al. "Applicability of the session and the presentation layers for the support of high speed applications". *Rapport Technique RT 144,* Institut National de Recherche en Informatique et en Automatique, Octobre 1992.

[15] H. Leopold, al. "Distributed Multimedia Communication System Requirements". *Deliverable ELIN-1,* OSI 95 project, May 1992.

[16] James R. Davin. "ALF Protocol Specification", Internal draft, M.I.T. Laboratory for Computer Science, February 1992.

[17] L. Léonard. "Enhanced Transport Service Specification". *Deliverable ULg-4*, OSI 95 project, October 1992.

[18] ANSI X3S3.3. Document 91-265: High Speed Transport Service definition. Expert Contribution to ISO/IEC JTC1 SC6/WG4, September 1991.

[19] Christophe Diot, Patrick Coquet and Didier Stunault. "Specifications of ETS the Enhanced Transport Service". *Rapport de Recherche RR 907-I*, LGI-Institut IMAG, May 1992.

[20] Hemant Kanakia, David R. Cheriton. The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors. *Proceedings of the SIGCOMM'88,* Stanford, CA, 1988, pp. 175-187.

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399

★ RR - 2102 ★