Interrupt Protocol Processing in the x-kernel

Mats Björkman

TR 94-19

Abstract

On many modern processors, context switches are costly in terms of latency. In this report, we argue that latency caused by kernel thread scheduling for incoming network messages can be avoided in a number of common situations, where instead the interrupt handler itself can perform the necessary protocol processing. For processors with high context switch costs, this can give a significant performance improvement. To support our argumentation, we present an implementation of the x-kernel communication protocol execution environment that exploits this execution paradigm. We present measurements on DECstations and on the Intel Paragon, showing the performance gain to expect when performing protocol processing in the interrupt handler.

June 27, 1994

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

1 Introduction

Short latency for network messages is an important performance measure for many applications. Sources for this latency are found both in the network and in the host computer systems. For local high-speed networks, most of the delay is found in the host systems. This intra-host delay includes protocol processing and operating system interaction delays.

Protocol processing is usually fairly inexpensive on the common execution path. Operating system interaction can be more time-consuming. Context switching is an operation in the host system that has more delay related than throughput related costs. For many modern processors such as RISC processors with large register sets, context switches are costly in terms of latency and should be avoided if possible. Even though the absolute time for a context switch might get smaller as processors and memory get faster, the number of processor cycles used (wasted) for context switches usually does not decrease.

The typical chain of actions when a network message arrives at a computer includes a number of context switches. First, the network interface receives the message and interrupts the host processor. This interrupt does itself not generally cause a full context switch, but the total cost of saving the processor registers and later restoring them is comparable to that of one context switch (not including the scheduling latency).

During interrupt handling for an incoming network message, the host processor typically does no or very little communication protocol processing. Rather, after some bookkeeping and maybe after copying the message from network interface specific buffers to more general memory buffers, the interrupt handler schedules a thread to take care of the protocol processing and then the interrupt handling ends. When the scheduled thread runs, it performs the protocol processing necessary for the message. The message is then delivered to the awaiting application, causing another context switch. Thus, a typical message delivery includes one interrupt and two context switches on the receiving side.

The context switch from the interrupt handler to the scheduled "network" thread can be avoided in some circumstances. If the protocol processing involved is short and does not require any critical resources, the required protocol processing may be done by the interrupt handler.

This report presents a number of scenarios where performance can gain from protocol processing during interrupt time, avoiding the context switch between the interrupt handler and a scheduled thread. In the rest of this report, we will use the term *interrupt protocol processing* for protocol processing performed by the interrupt handler.

We have performed a number of experiments with interrupt handler protocol processing in the x-kernel environment. The x-kernel is a protocol execution environment developed at the University of Arizona [HuPe91]. It is a self-contained execution environment in which protocols can be implemented in a uniform, yet efficient way. The x-kernel provides the protocol implementer with a set of uniform interfaces to other protocols and to operating system services likely to be used by a protocol implementer. The x-kernel also contains a set of tools that interface x-kernel services to those of the host operating system.

The x-kernel includes a Uniform Protocol Interface, which provides a protocol implementer with a uniform way of calling other protocols. The protocol implementer need not be aware of which protocols will be running on top of or below the protocol being implemented. Rather, by using a dynamic graph (see Figure 1), the binding of protocols into hierarchies is deferred until linking time.

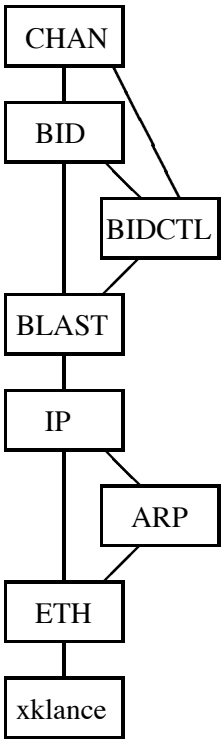

Figure 1: An x-kernel protocol graph similar to the one used in the CHAN performance measurements presented in section 4.

In the protocol graph above, there are a number of protocols designed at the University of Arizona. CHAN is a request-response channel protocol, modelled after Sprite RPC. BLAST is a fragmentation protocol, modelled after Sprite RPC fragmentation. BID and BIDCTL are Boot-ID protocols, handling reincarnations of hosts.

The x-kernel also provides the implementer with a set of tools that implement commonly used routines in a machine-independent or portable way. The most important tools are the Message Tool which implements efficient handling of message buffers; the Process Tool which implements thread execution and handling; the Event Tool which implements asynchronous events, and the Map Tool which is used to map identifiers to x-kernel objects.

Whenever we need to distinguish the implementation of the x-kernel that does interrupt protocol processing from the regular x-kernel, we will call the former *IPP x-kernel* (for Interrupt Protocol Processing x-kernel), and the latter *Regular x-kernel*. In the IPP x-kernel experiments, incoming messages are processed partially or fully within the interrupt handler. Our results show that a performance gain is possible when protocol processing during interrupt handling removes the need for a context switch.

The rest of the report is outlined as follows; Section 2 contains further motivation why protocol processing during interrupt handling is an issue, and where it may be exploited. Section 3 describes the general mechanisms and handoff policy necessary to implement interrupt protocol processing, and how this is done in our implementation of IPP x-kernel. Section 4 presents measurement results for a number of experiments. Finally, section 5 contains a summary of our conclusions.

2 Motivation

Context switches are costly in terms of latency on many modern processors. Table 1 gives context switching times as they have been measured on some of the machines where the x-kernel currently runs. Note that saving the floating point registers is *not* included in the numbers presented below. Some systems may require this.

Machine, OS	Context Switch time (cycles)
DECstation, Mach3 MK82	425 – 500 cycles
DEC Alpha, OSF/1 1.3	121 – 1158 cycles
Intel Paragon i860, OSF1	2300 – ?? cycles

Table 1: Context switch times for some popular processors

From the table we can see that, in most cases, hundreds or even thousands of CPU cycles are used to perform a context switch. For applications that require low message delivery latency, a context switch might be saved for each message reception if protocol processing is performed in the interrupt handler. For a communication-intensive machine, like a server that receives frequent requests from many clients, the compound cost of context switches can impact overall performance of the server.

There are several scenarios where interrupt protocol processing may be applied. Below we present three examples of situations were interrupt protocol processing can increase performance.

2.1 Servers

In applications with RPC type communication between the clients and a server, the server sometimes does only a small amount of processing for an incoming request, e.g. a request for the value of a variable or a sequence number. If this server processing can be done during interrupt time, such systems would be suitable for interrupt protocol processing, even though the server processing is not protocol processing in the communication protocol sense. Of course, the sending of a reply should not block if it is to be performed during interrupt handling.

Active messages is an example of this kind of communication. Active messages is a communication mechanism that allows for efficient protocol processing [ECGS92]. The basic idea is that every incoming message carries a pointer to the handler that should be executed as a result of the message reception. This is a form of direct early demultiplexing where the message carries the pointer to the handler to be executed, rather than that handler being pointed out by a table indexed by the value of some function applied to particular fields in the message header(s).

Active messages requires the sender to know the address of the receiver's handler. Active messages is envisioned to be implemented in a system where all hosts have the same core image, thus allowing the sender to supply a pointer to the handler without first asking the receiver for it. Active messages tries to avoid context switches by having short, non-blocking handlers that may be executed at interrupt time. Demultiplexing is done simply by calling the handler pointed to by the supplied pointer.

The kind of interrupt protocol processing presented in this subsection requires server code to be executed during interrupt handling, which might be a security issue. On the other hand, executing server code inside the operating system kernel is a security issue even if the processing does not take place during interrupts. We recognize the potential problem, but see the security issues as beyond the scope of this report.

2.2 Acknowledgments

In many protocols, the reception of an acknowledgement triggers only a limited amount of processing. Typically, only some (small) amount of housekeeping is performed in one of the protocols. This type of acknowledgement processing may often be performed during interrupt handling.

In the case where an incoming acknowledgement triggers the sending of a new data packet, execution may have to be handed off to a scheduled thread, since the sending of a data packet may include blocking, either because of resource shortages, or because of the network being busy.

2.3 Fragment reassembly

Some networks have smaller Maximum Transfer Unit (MTU) sizes than the message sizes upper layer protocols support. Fragmentation (and subsequent reassembly) then has to be performed in order to get a large message over the network.

IP is an example of a protocol that performs fragmentation and reassembly. When running over an Ethernet, IP on the sending side fragments larger messages from upper layers into fragments not larger than the Ethernet MTU size. IP on the receiving side reassembles the incoming fragments into larger-sized messages.

When receiving message fragments, protocol processing for each fragment is limited to the protocols up to and including the reassembly protocol. Only when all fragments have arrived, the reassembled message is delivered to the upper layer protocols. The processing of a fragment that is not the last arriving fragment in the message being reassembled, is a short-duration processing suitable for execution during interrupt handling.

2.4 Potential drawbacks

There are a number of potential drawbacks that make interrupt protocol processing a not very easy task to design, implement and perform.

If executing an application protocol during interrupt time, for example on behalf of a server, the code executed must be trusted in the sense that it is permitted to be run inside the kernel. This is true for all protocol code run inside the kernel.

If execution in the interrupt handler goes on for too long, either new incoming messages or other activities causing interrupts may be delayed longer than necessary, and this may upset the behavior of the host operating system.

The executed code must be modified to do a handoff wherever it risks to block. With the x-kernel, this can be done almost entirely in the x-kernel framework and does not affect the protocol implementations very much.

With careful design and proper implementation it is possible to get around these drawbacks. They shall however not be underestimated.

3 Mechanisms and policy for interrupt protocol processing

This section presents the mechanisms and policy needed to implement interrupt protocol processing. We also present our implementation of interrupt protocol processing in the x-kernel.

The basic idea of interrupt protocol processing is as follows: When a network interrupt occurs, protocol processing starts in the interrupt handler. Only if and when it is determined that a handoff to a thread is

necessary is a handoff performed. In the case of a handoff, the current context (i.e. the contents of the interrupt stack and the current register contents) is handed off to a thread, and the interrupt handler returns. The scheduled thread then completes the execution started by the interrupt handler.

In order to implement this, we need mechanisms for context (stack and register) handling and handoff, and a policy for determining when to hand off to a scheduled thread.

3.1 Handoff mechanisms

We need a mechanism that allows us to perform a fast switch from the interrupt handler to a thread whenever the need to hand off occurs. In order for this to be fast, we need to do some preparation when starting the interrupt handler.

3.1.1 Startup preparation

To be able to speculatively start protocol processing during interrupt handling, there must be a fall-back, should the need for lengthy processing or blocking occur. This means that at the start of the processing, we must to some extent prepare for the potential case of handoff. We do this by allocating to each instance of the network interrupt handler a detachable stack on which the interrupt handler executes. In the anticipated case where no handoff is necessary, the detachable stack is simply freed as the interrupt handler is about to exit.

3.1.2 Handoff if necessary

If execution hits a point where it is determined (by the handoff policy) that execution no longer can continue in the interrupt handler, the detachable stack is handed off to a waiting thread, and that thread is scheduled for execution. This will cause a scheduling latency similar to that caused by the thread scheduling in a regular network interrupt handler. The interrupt handler then returns from the interrupt.

3.2 Handoff policy

There are three situations where handoff to a scheduled thread should be performed. The first situation is when the execution reaches a point in the protocol graph where it has been determined that a handoff should be performed, this is called an *explicit handoff policy*. The second situation is when handoff is necessary because the execution may block when executing a certain procedure or system call, this is called an *implicit handoff policy*. The third situation is when protocol processing exceeds some time limit, this is called a *time based handoff policy*.

3.2.1 Explicit handoff policy

The protocol graph may be annotated with points where protocol processing should stop being performed by the interrupt handler. There can be several reasons for annotating the protocol graph with such a barrier. The execution of the protocol(s) above the barrier may require lengthy processing, or the protocol(s) above the barrier may try to require require some shared resources and thus risk to block. Also, the protocol or protocols above the barrier may cross domain boundaries that make continued processing in the interrupt handler impossible.

This type of handoffs are called explicit handoffs because they are explicitly marked in the protocol graph.

3.2.2 Implicit handoff policy

When a system resource is not immediately available, blocking occurs. Blocking might occur in situations like dynamic memory allocation, device requests and semaphore waits. Blocking during interrupt handling must be avoided because the system is most likely to deadlock if the requested resource is held by a lower priority process that does not get to run until the interrupt handler has finished. Also, we must limit the time spent in the interrupt handler. When blocking, the time until unblocking is unknown. Therefore, a handoff should be performed before calling any procedure that might block.

This type of handoffs are called implicit handoffs because they are not marked in the protocol graph, rather they happen as the result of some procedure being called.

3.2.3 Time based handoff policy

During protocol processing, we may reach a point where we know that the processing is going to take too much time, and we therefore are forced to hand off. Depending on how much incoming data the network interface can receive before over-running its input buffers, the peak time limit may vary. However, the operating system is more likely to have the necessary resources to accommodate a temporary burst of messages than the network interface. Therefore, the average interrupt handling time limit for one message should not be longer than the time for a message reception by the network interface.

In reality, there should also be some time left in the system for activities other than the execution of interrupt handlers.

In the discussion on fragmentation in the previous section, we saw that only for the last fragment in a message will the execution continue with upper layer protocols. The detection of a completely reassembled message is one example of a point where a time limit handoff is suitable. The continuing execution might involve several protocol layers and potentially complex and time consuming processing.

3.3 The IPP x-kernel implementation

We have implemented interrupt protocol processing in the x-kernel environment. We call this new implementation *IPP x-kernel*.

This far, IPP x-kernel has only been fully implemented on DECstation 5000s, running Mach 3.0 (MK82/UX41), using the x-kernel v 3.2 December 1993 release as a base for the implementation of IPP x-kernel, and as a pilot implementation running on an Intel Paragon, under OSF/1. We have run a number of performance measurement experiments on these systems. The results of these experiments are presented in section 4 below.

The IPP x-kernel implements the mechanisms and handoff policy described above as follows:

3.3.1 Handoff mechanisms

The Mach we used for our DECstation implementation does not supply a separate interrupt stack, but uses the stack of whatever kernel thread that happened to execute when the interrupt occurred. Therefore, we implemented a pool of detachable stacks to be used by the interrupt handler. When a network interrupt occurs, a detachable stack is allocated from the pool and used as the stack for the interrupt handler.

When a handoff occurs, the detachable stack is assigned to a waiting thread, the thread is scheduled for execution, and the interrupt handler exits. Later, when the thread finishes its execution, the detachable stack is returned to the pool of stacks.

3.3.2 Handoff policy

In the IPP x-kernel implementation, the protocol graph can be augmented with annotations of where a handoff is suitable (explicit handoffs), and a handoff call can automatically be included in the x-kernel interface code. Thus, no changes has to be done to the protocol code. The x-kernel framework also includes points where implicit handoffs are performed. Also, specific protocols can have specific points where handoffs occur (implicit or time based handoffs).

4 Performance of IPP x-kernel

This section presents performance results from a number of experiments conducted on the IPP x-kernel implementation presented in the previous section. If nothing else is stated, the experiments were conducted on the aforementioned DECstations.

We have micro-benchmarked the routines used for handoff and scheduling in IPP x-kernel and Regular x-kernel, respectively. The results are shown in Table 2.

Routine (version)	Time in μs
Stack startup (IPP)	3.6
Thread handoff (IPP) includes scheduling latency	32.2
Thread handoff (Regular) includes scheduling latency	30.7

Table 2: Execution times for handoff routines in IPP x-kernel and Regular x-kernel

As can be seen, using IPP x-kernel introduces a penalty of 5.1 μs in the case where a handoff is required. Should a handoff be avoided, however, 27.1 μs is gained from using IPP x-kernel on the DECstations.

Below is presented three experiments performed in the IPP x-kernel environment on the DECstations. The first experiment presents the performance of an RPC style protocol (CHAN) when using IPP x-kernel to increase server performance. The second experiment presents the performance of the fragmentation/reassembly functions of IP when using IPP x-kernel. Because of the unexpected results obtained in the second experiment, a third experiment was performed. This third experiment shows how variations in the number of threads used to service incoming messages affect performance.

After these three experiments, one experiment on the Intel Paragon is presented. This experiment shows the performance gain possible on a system with a very expensive context switch.

4.1 RPC measurement

The RPC measurements simulate a network server that is able to handle incoming requests and reply to them during interrupt time. The RPC measurements were made on a CHAN protocol graph, the graph being similar to the one in shown in figure 1. Message sizes were 1 byte, 1000 and 2000 bytes. The results are presented in table 3.

Version	RTT in μs, 1 byte	RTT in μs, 1000 bytes	RTT in μs, 2000 bytes
Regular	891	4862	8431
IPP, client handoff	845	4821	8384

Table 3: Performance results for CHAN test, message size 1, 1000 and 2000 bytes

CHAN has a very explicit handoff on the client side in that CHAN uses a semaphore to awaken the client thread when the reply has arrived. This handoff could not be avoided without changing the semantics of CHAN. This is the reason why we only present numbers for client handoff for IPP x-kernel.

As can be seen from the numbers, the performance gain is between 41 and 47 μs, and independent of message size.

4.2 Fragmentation measurements

Fragmentation measurements were performed to show the gain of allowing fragments of larger messages to be processed up to (and including) the reassembly protocol during interrupt time. When the last fragment in a message has arrived, execution is handed off to a scheduled thread.

Fragmentation measurements were made on a UDP protocol graph. In these experiments, IPP x-kernel was used on either side of the communication and on both. Message sizes were 1 byte, 1000, 2000 and 4000 bytes. The results are presented in table 4.

Version	μs RTT, 1 byte	μs RTT, 1000 bytes	μs RTT, 2000 bytes	μs RTT, 4000 bytes
Regular	762	4647	7857	11735
IPP, client handoff	728	4627	7840	11709
IPP, server handoff	728	4628	7845	11711
IPP, no handoff	664	4561	7809	11647

Table 4: Performance results for udptest, message size 1, 1000, 2000 and 4000 bytes

What can be seen from the table above is that the gain is rather independent of the message size. This may appear surprising, since the positive performance effects of IPP x-kernel should show up for messages larger than the Ethernet MTU (for message size 2000 and 4000 bytes).

A possible explanation is that Regular x-kernel as used in the measurements was configured to employ only one input thread. Protocol processing for each fragment took longer time than the interval between fragment arrivals. Thus, when the first fragment had been processed, the next was already awaiting processing. The network interrupt handler was written so that if no thread was waiting for an incoming fragment, the fragment was queued on one of the active threads. When this thread finished processing the previous fragment, it could immediately start processing the next. Thus, in the typical case, no context switch would occur in Regular x-kernel for additional fragments, since the only active thread could process all additional fragments without delays. We suspected this to be the reason behind the performance results presented in table 4. In order to verify this, the next experiment was conducted.

4.3 Input thread measurements

We performed a set of measurements to confirm our suspicions that clever implementation of Regular x-kernel made it possible to avoid some context switches. We measured the number of context switches performed during 10000 message test runs, see Table 5.

# input threads	# context switches, 1 byte msg (1 fragment)	# context switches, 1000 bytes msg (1 fragment)	# context switches, 2000 bytes msg (2 fragments)	# context switches, 4000 bytes msg (3 fragments)
1	10001 of 10004	10004 of 10014	10011 of 20023	10019 of 30042
2	10002 of 10002	10005 of 10005	20017 of 20017	30027 of 30027

Table 5: Number of context switches and received messages during a 10000 message test

The table presents the number of context switches performed and the total number of received messages during the test run. The spurious extra messages are due to some background activities on the machines and network. As can be seen from the table, for one input thread, only one (the first) fragment of a large message causes a context switch. The rest of the fragments get to be handled by the same thread without context switches. For two (or more) input threads, each incoming fragment cause a context switch.

As we saw in the previous experiment, we did not gain anything in the fragmentation experiment, because we had only one input thread in Regular x-kernel. Thus we only had one context switch per large message in Regular x-kernel as well as in IPP x-kernel.

However, having only one input thread is an unlikely scenario, because if that only thread blocks for some reason, no further incoming messages can be handled until the input thread is unblocked and ready to handle a new message.

In Table 6, the performance for different number of threads for Regular x-kernel and udptest is presented.

# input threads	μs RTT, 1 byte (1 fragment)	μs RTT, 1000 bytes (1 fragment)	μs RTT, 2000 bytes (2 fragments)	μs RTT, 4000 bytes (3 fragments)
1	762	4647	7857	11735
2	761	4651	7911	11829
3	762	4650	7925	11847
4	767	4664	7933	11866
8	775	4682	7964	11929

Table 6: Performance of different number of threads for udptest, Regular x-kernel

Here we see that going from one to more than one thread causes a drop in performance for messages larger than the Ethernet MTU (for message sizes 2000 and 4000), that is, messages that are sent as more than one fragment. For each additional fragment, the performance drop when having more than one input threads is about 50 μs per additional fragment, representing the total performance cost for having more than one input thread.

The difference observed when going from two threads to more than two is small; each additional thread adds a small amount to the total execution time, roughly 5 μs per thread per fragment.

From tables 5 and 6 we draw the conclusion that if the number of input threads are more than one, context switches are performed, and then performance can gain from interrupt protocol processing. To confirm this, we measured the performance of IPP x-kernel when not handing off, having 3 input threads. In table 7, the results are compared to the numbers from 1 input thread, and to the numbers for 1 and 3 input threads for Regular x-kernel.

x-kernel/# threads	μs RTT, 1 byte (1 fragment)	μs RTT, 1000 bytes (1 fragment)	μs RTT, 2000 bytes (2 fragments)	μs RTT, 4000 bytes (3 fragments)
Reg/1 thread	762	4647	7857	11735
Reg/3 threads	762	4650	7925	11847
IPP/1 thread	664	4561	7809	11647
IPP/3 threads	662	4564	7811	11636

Table 7: Comparison of Regular and IPP x-kernel for 1 and 3 input threads

As can be seen from table 7, our conclusions above were correct. The performance difference for IPP 1 and 3 threads are almost none, while comparing IPP/3 threads with Regular/3 threads shows a performance difference that increases with the number of fragments per message.

4.4 The Paragon

The above reported experiments show that interrupt protocol processing on the DECstations gives only a modest performance gain. In order to verify that interrupt protocol processing can give better performance gains on systems with more expensive context switches, a pilot version of IPP x-kernel has been implemented on an Intel Paragon, in the OSF/1 operating system. The Intel Paragon is known to have a very expensive context switch time, see table 1, and avoiding a context switch could help much to improve performance.

In this experiment, we executed a ping-pong test for a BLAST stack. For IPP x-kernel, no handoff was performed. The results are shown below in table 8.

Version	RTT in μs 8 bytes	RTT in μs 1K bytes	RTT in μs 2K bytes
Regular	500	625	741
IPP, no handoff	237	353	492

Table 8: IPP performance on the Intel Paragon compared to Regular x-kernel

As can be seen from table 8, the performance gain on the Intel Paragon when using IPP x-kernel is significant. The average RTT reduction is about 260 μs. For small messages, this means that the round-trip time is reduced to less than half if performing protocol processing in the interrupt handler.

5 Conclusions

We have presented arguments for protocol processing during interrupt time. We have shown a number of scenarios where protocol processing during interrupt time can yield a performance gain as compared to more common implementations, where protocol processing is done by a scheduled thread.

We have presented an implementation of the x-kernel environment, the IPP x-kernel, where interrupt time protocol processing is performed. Performance comparisons between IPP x-kernel and Regular x-kernel show that performance gains are achieved in the measured scenarios when context switches are avoided by allowing protocol processing to take place in the interrupt handler. For machines with high costs for context switches, like the Intel Paragon, the performance gain is significant. For small messages on the Paragon, the round-trip time when using IPP x-kernel is cut to less than half of that for Regular x-kernel.

We conclude that interrupt protocol processing can be used in a number of common scenarios to increase performance.

Interrupt protocol processing is a generalization of Active Messages in that processing is performed at interrupt time, but any protocol graph can be executed since no early demultiplexing has to be performed.

6 Acknowledgements

I would very much like to thank Professor Larry L. Peterson at the University of Arizona for his help with the preparation of this report.

References

[BALL90] B. Bershad, T. Anderson, E. Lazowska and H. Levy. Lightweight Remote Procedure Call, In *ACM Transactions on Computer Systems*, vol 8 no 1, pp 37-55, February 1990.

[DBRD91] R. Draves, B. Bershad, R. Rashid and R. Dean. Using Continuations to Implement Thread Management and Continuation in Operating Systems, In *Proc. 13th ACM Symposium on Operating Systems Principles*, pp 122-136, October 1991.

[ECGS92] T. von Eicken, D. Culler, S.C. Goldstein and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation, In *Proc. of ACM XXX*, pp 256-266, 1992.

[HuPe91] N. Hutchinson and L. Peterson. The *x*-kernel: An Architecture for Implementing Network Protocols , In *IEEE Trans. on Software Engineering*, Jan. 1991.