TUTORIAL:
Alternatives in Efficient Protocol Implementation
for High Speed Networks

by

Per Gunningberg

Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, SWEDEN
TUTORIAL:
Alternatives in efficient protocol implementations for high speed networks

Per Gunningberg
per@sics.se
SICS
Box 1263
S-164 28 KISTA

June 1992
SICS Technical Report T92:08
Alternatives in efficient protocol implementations for high speed networks

Per Gunningberg
Swedish Institute of Computer Science
per@sics.se

Florida USA
June 22, 1992

Tutorial objective

To give some understanding of efficient protocol implementations with respect to:
- what protocol constructs yield efficient implementations
- where the performance bottlenecks are located
- the software and hardware implementation tradeoffs
Outline of tutorial

- Performance parameters
- From specification to implementation
- Performance bottlenecks in implementations
- Efficient packet processing
- Execution of data manipulation functions
- Putting it all together
- Some examples of protocol machines

Efficient implementation of protocols

- With respect to:
  - utilization of resources: CPU cycles, memory, interfaces, bandwidth
  - application user performance
  - implementation and maintenance efforts
Performance parameters

- latency or response time; the delay experienced by the user
- throughput; number of bits transferred per time unit
- delay variance or jitter
- connection set up time
- reliability; rate of packet and bit loss

From specification to implementation

Abstract Specification

- Decide on unresolved issues, e.g. acknowledgement strategies
- Add system dependent parameters, e.g. window size and time-out values
- Decide on service interfaces and interlayer communication
  - how Service Data Units are passed, e.g. via buffers
  - calling structures, from one layer to another
- Services from operating system, such as timers, memory management, interrupt handling, etc

Implementation

When are performance issues considered?
From specification to implementation....

in practice:

- Implementations are derived from existing implementations
- Typical example: TCP/IP of BSD Unix

Nice implementation, but performance was not a major concern

Performance specification?

- Specifications are written for:
  - clarity and non-ambiguity
  - proving correctness
  - independence of implementation environment
- Specifications give no guidance:
  - for performance requirements
  - how to implement for performance
  - how to test for performance
Alternatives to reach high performance 1

- More efficient protocols
  - imperfections in control algorithms,
  - flow and congestion control
  - delay caused by contention to shared resources

Alternatives to reach high performance 2

- Reduce redundant functionality from protocol stacks
  - Elimination of redundant services
  - Error control (layers 2-4)
  - Connection demultiplexing (layers 2-7)
  - Flow control (layers 4-7)
  - Reordering and reassembly (layers 3-7)
  - Select minimal functionality according to application need
  - Dynamic building of protocol stacks, e.g., xkernel from Arizona
  - Do not use general purpose protocol
Alternatives to reach high performance 3

- Efficient implementation of protocols
  - reduce number of CPU cycles
  - hardware implementations
  - concurrent execution

High Performance (Speed) Networks

- performance: High bit rate
- delay limited by propagation time of light in media
- 100 Mbit/s -> 1 Gbit/s -> 10 Tbit/s?
- => High bandwidth delay product – the amount of data which must be in transit to keep the pipe full, i.e., continuous transmission

HOW MUCH IS NEEDED TO KEEP A PIPE FULL? AN EXAMPLE:
ROUND TRIP DELAY = 30 msec
BIT RATE = 1 Gigabit/s
DELAY x BANDWIDTH = 30 Mbit

NOTE: 75-90% OF BULK TRANSFERS ON INTERNET ARE LESS THAN 10 KBYTE.
We will focus on host protocol implementations

- Routers and switches seems to be able to route packets in high speed networks
- Assume average packet size of 125 byte
- 1 Gigabit/s => A 125 byte packet every µsecond
- Assume 25 mips machine => 25 instructions/packet
- IP takes about 100-120 instructions plus device driver
- Virtual circuit set up – about 30 instructions
- ATM switching of cells - 53 bytes => less than 1 µsec
Measured performance on Sun 3/60

A simple protocol processing model

- Message processing time can be approximated with:

\[ T_{\text{message}} = T_{\text{packet}} + T_{\text{byte}} \cdot N; \]

where

- \( T_{\text{message}} \) is the total time,
- \( T_{\text{packet}} \) is the time to process the packet, independent of size of data part, eg header processing
- \( T_{\text{byte}} \) is the manipulation time for a byte, eg for checksum calculation
- \( N \) is the size of the data part in number of bytes

NOTES
Byte manipulation example: Checksum Calculation

```
While (count > 0) {
    sum = sum + Data[count];
    count = count - 1;
}
```

Let us look at the numbers

- Client execution time = Server execution time
- Divide measured time by 4 to get per message processing time
- 2 Mips machine

\[ T_{message} = T_{packet} + T_{byte} \times N \]

where

\[ T_{packet} = \frac{6000\mu s}{4} = 1500\mu s \Rightarrow 3000 \text{ instructions} \]

\[ T_{byte} = \frac{3 \mu s}{4} = 0.750\mu s/\text{byte} \Rightarrow 6 \text{ instructions/word} \]
Performance bottlenecks: What happens when a packet arrives?

- interrupt, packet is in interface
- allocate buffer
- copy/DMA into buffer
- look-up for connection state
- process header
- set/reset timer
- possibly calculate checksum
- protocol state machine transition
- possibly schedule a response
- copy data part from buffers to application address space
- schedule application process

An execution time profile example

HOST 1 --> interface board --> interface board --> HOST 2

20
Build message in host

60
Pass message to interface board over VME bus

15
Transport and link protocol processing

5
Optical fiber transmission time

12
Interrupt

18
Data link and transport

8
Build message

10
Pass message to host over VME Bus

20
Read message

Time (microseconds)
Total: 168 microseconds

Adapted from measurements on CMU Nectar Communication Processors (SIGCOMM 90)
List of bottlenecks: Packet oriented

- interrupt handling for each packet
- header parsing
- timer handling
- buffer management
- context switching, both for connections as well as processes

List of bottlenecks: Byte manipulation

- Encoding/decoding
- Checksum calculation
- Data compression/decompression
- Encryption/decryption
- Copy from user address space to system address space
- Alignment to page/word to/from buffers
- Copy from system address space to interface
The 80/20 experience

- "About 80% of processing involved is non-protocol related code and roughly 20% is protocol dependent"
- => Focus on operating system issues
- => Protocols designed for efficient implementation
- => Big Packets

Packet process optimizations: interrupt handling

- 125 byte @ Gigabit/s => interrupt every μsecond
- faster interrupt handling:
  - reduce size of saved context - switch between register banks
- Fewer interrupts - use:
  - big packets and buffers
- "outboard processing" of interrupts when small packets are used. Example: Belcore's ATM interface board which does segmentation and reassembly of ATM cells (size 53 bytes) to/from big packets
Packet process optimization: header parsing

- Fixed size and position of header fields - no need to parse each field in header
  - Example: eXpress Transfer Protocol
- Header prediction - based on locality in the packet stream. The next packet header is assumed to be identical to the previous one except for some few predictable fields, e.g. sequence number. Predicted header is compared with received.
  - Example: Unix BSD Reno, Van Jacobson
- Hardware for header parsing. When (complicated) parsing is needed
  - Example: The AT&T VLSI Engine PROVE

Packet process optimizations: timer handling

- Two operations: (1) set a time-out interval and, (2) reset timer when not needed
- For networks with high delay-bandwidth products we may need to organize several thousand timers per connection. These timers are organized in a data structure.
- Time-consuming to traverse big structures in order to find timer to reset or position to do an insert. We need an organization that scales -> constant search for set and reset
- For most protocols, time-out interval do not need to be exact => a timer can cover several packets
- Hardware timer - Columbia’s PROMPT, AT&T VLSI Engine
- Software timer - timing wheel of Varghese, hash tables

NOTES
Packet process optimizations: buffer management

- A buffer: a preallocated data structure in a reserved area in memory. Holds packet headers and user data.
- Fixed execution time to allocate and deallocate buffer structures
  - Trade-offs in buffer size:
    - One big size -> poor utilization for small packets. But memory tend to be relatively cheap.
    - Small and chained buffer -> higher cost in allocation/deallocation than for big packets
- Optimizations
  - Use buffer templates, e.g. for known header sizes
  - Match buffer size with application data units - reduces copying
  - Use buffer size == page sizes

Packet process optimizations: context switching

- Per connection
  - A table lookup per connection
  - Loading of connection state from memory
  - Optimizations:
    - Locality - by caching state of last send/received packet
    - Hardware lookup and state loading, e.g. Columbia's PROMPT connection processor
- Operating system process context switch
  - Try to avoid them
  - Use lightweight processes
Some instruction counts for an optimized TCP implementation (Clark, et al estimation)

- A stripped production TCP - Unix BSD 4.3
- Recode the essential parts - "common path"
- Exclude mbufs, checksum & control state

```
| SEND 235    | RECEIVE 191-235 |
| MESSAGE     | ACK             |
| RECEIVE 186 | SEND 235        |
```

- Assume 25 mips RISC machine

<table>
<thead>
<tr>
<th>Packet size</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 byte</td>
<td>100 Mbit/s</td>
</tr>
<tr>
<td>4Kbyte</td>
<td>1.33 Gbit/s (2 MESSAGES to 1 ACK)</td>
</tr>
</tbody>
</table>

Where is the problem with data manipulation functions?

*One answer: Memory Bandwidth*

- Relative high cost in RISC machines to read/write from primary memory
  - => Reduce memory access
  - => Do not copy data

```
RISC CPU   | Cache | Memory Bus | Primary Memory
-----------|-------|------------|----------------
```
Measured memory access time on a Sun SPARCstation 2

- Write/Read sequential bytes (ldbyte, inc_counter, test, branch, nop)
- SPARC has "write through" cache
- 1 cycle instruction time: 25 nanoseconds

<table>
<thead>
<tr>
<th></th>
<th>Primary Memory</th>
<th>Cache</th>
<th>Register</th>
<th>Double Word</th>
<th>Primary memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td>320</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td>190</td>
<td>160</td>
<td>125</td>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

Do not copy data – unnecessarily!

- nonDMA design - copy by CPU direct to application address space if CPU needs to manipulate data
- buffer area in dual port memory, mapped cacheable
- need to invalidate cache for areas written to by controller
Reduce memory access: Collapse Reads and Writes

\( R = \) Read from primary memory  
\( E = \) Evaluate manipulation function  
\( W = \) Write to primary memory

\[ R \rightarrow E \rightarrow W \rightarrow R \rightarrow E \rightarrow W \]

Manipulation Functions

Time(Read Message) \( \gg \) Time(Eval Message)  
Time(Write Message) \( \gg \) Time(Eval Message)

\( \Rightarrow \)

Collaps Reads and Writes to primary memory

Estimated evaluation times  
(\texttt{SPARCstation 2})

<table>
<thead>
<tr>
<th>Data manipulation function</th>
<th>Memory access time per byte</th>
<th>Evaluation time per byte (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP checksum, half word (R)</td>
<td>200</td>
<td>10</td>
</tr>
<tr>
<td>Insertion of NL in ASCII stream (RW)</td>
<td>500</td>
<td>210</td>
</tr>
<tr>
<td>Byte ordering (RW)</td>
<td>500</td>
<td>-</td>
</tr>
<tr>
<td>Word alignment (RW)</td>
<td>500</td>
<td>-</td>
</tr>
<tr>
<td>DES encryption (software) (RW)</td>
<td>200</td>
<td>56300</td>
</tr>
</tbody>
</table>

\( R = \text{READ} \)
\( W = \text{WRITE} \)
\( RW = \text{READ+WRITE} \)
Interdependencies restrict parallelism

- Reduce dependencies - out of order processing, see for example Bellcore's transport protocol TP++

Integrated Implementation for collapsing

- Several layers implemented within one module
- Must follow the specification
- Desirable to keep some layer abstraction for structuring
- Lightweight layer interfaces

=>
- Manipulation functions can be computed in one step
Pipeline evaluation

- data unit or fragment of unit is pushed through ordered set of manipulation functions
- evaluation is postponed until it can be done within one message read and one message write to memory
- software pipeline - data in registers or cache. Data manipulation code is serially applied. Example: SICS delayed evaluation
- hardware pipeline - encryption and checksumming, eg Stanford Network Adapter Board

Multiprocessing paradigms

- Paradigms:
  - A process/processor per connection - simple and efficient. Typically used in multiprocessor servers, eg NFS servers.
  - A process/processor per layer - eg transputer based solutions
  - A process/processor per PDU - eg xkernel
  - A process/processor per function - eg timer, buffer manager, receiver, sender etc
Host and Multiprocessors

- **Outboard processor** - all, or most protocol processing is done on an attached board with one or several CPUs.
  - It is connected via a bus to the host.
  - Will off-load the host.
  - Example: Protocol Engine
- **Dedication of processors** in a multiprocessor machine to protocol processing.

Multiprocessing issues to consider

- Granularity of parallelism
  - fine grain, like pipelining
  - coarse grain, for outboard processor
- Communication between processors
  - shared memory for fine grain and coarse grain
- Locking of shared data in shared memory and cache invalidation
- Load balancing and scheduling

NOTES
Putting it all together: Software approach 1

According to Van Jacobson’s WITLESS approach

- For TCP/IP – will appear in Unix BSD 4.4?
- dual port memory with simple network interface
- CPU controlled bus transfer - nonDMA (see slide 32)
- one size (big) buffers - accessible by higher layer protocols
- collapsing checksum with address space copying
- header prediction and header templates
- principle applied to HP 700 series => throughput > 40 Mbit/s over FDDI

Putting it all together: Software approach 2

Experiments at Cray Research

- Run TCP/IP on Cray Research over Hippi-channel
- increase to 64 kbyte packets
- increased sequence number space
- header prediction
- optimized buffers, perfect alignment, write/read 1512 Kbyte at each time
- On Cray Y-MP: 795 Mbit/s
Putting it all together: Software approach 3

The x-kernel from University of Arizona

- software execution environment for protocol implementation (actually a complete OS-kernel)
- provides efficient and general interlayer communication
- a lightweight process per PDU
- software library of commonly used objects for protocol implementations – "microlayers"
- performs competitively with less structured operating systems

Putting it all together: Outboard processor

Example: The Nectar processor from CMU

- Outboard processor – off-loads the host
- Useful when host has specific architectures, like for supercomputers
- Connected to host via VME-Bus and shared memory with common address space
- Board-to-board measured throughput over TCP without checksum: 90 Mbit/s (8kbyte packets)
- Host-to-host throughput over TCP: 24 Mbit/s
Putting it all together: Multiprocessor

The transputer approach of Karlsruhe

- OSI CNLP and OSI TP4
- pipelining of network and transport protocols
- concurrent processing of receiver and sender entities, a transputer each
- throughput estimation with software monitor and 8 transputers: 32 Mbit/s with 1950 byte packets.
- Transputer to transputer communication limits the throughput

Outboard approach, parts in silicon: Protocol Engine

- Developed by Greg Chesson, SGI, for an industrial consortium
- Will implement XTP, TCP and FDDI - scheduled for 1992
- 4 RISC chips + special VLSI chips
  - In silicon:
    - checksum
    - special purpose ALU
    - byte order swapping
    - priority checking
    - address parsing
- Estimated performance > 100 Mbit/s
Outboard approach, parts in silicon: The AT&T PROVE

Programmable Protocol VLSI Engine from AT&T Murray Hill

- Set of programmable VLSI chipset
  - standard protocols, level 2, 3 and 4 of OSI-model
  - protocols are formally specified as communicating FSMs
  - compiler generates microcode to RAMs
- 4 Units using microcode:
  - Message parser and Message assembler
  - Control Unit and User interface
- for LAPD estimated performance 50 000 pkt/s

Conclusions, or rather some opinions

- *What to implement in hardware*
  - data manipulation functions, eg ASN.1 encoding/decoding, checksum, etc
  - support for moving data, eg dual port memory, wide buses, DMAs, cache control
  - state lookup tables
  - protocol state machine
- *What to put on dedicated processors*
  - timer, interrupt and buffer managers, segmentation and reassemble
- *What to implement in software*
  - The rest -- or -- all of above!
- *What else -- design protocol with performance in mind*

NOTES
Some selected references

- Bjorkman, M. Designing Hierarchical Hardware for Efficient Timer Handling, 2nd Workshop on Future Trends in Distributed Computing, Cairo 1990
- Svobodova, L., "Implementing OSI Systems", IEEE Journal on selected areas in communication, Vol. 7., No. 7, 1989
- Traw, B. & Smith, J., "A High-Performance Host Interface for ATM Networks", ACM SIGCOMM91