Testbed for header compression implementation

Mario Mornar*, Alen Pezelj*, Julije Ozegovic**
*HT-TKC Split
**University of Split, FESB Split
Split, Croatia
E-mail: mario.mornar@ht.hr, alen.pezelj@ht.hr, julije.ozegovic@fesb.hr

Abstract: **Early phases of EPIC-Lite compression scheme implementation required a flexible test platform. New hybrid testbed architecture was introduced, providing connection between user and kernel space. Changes to standard kernel and modules were spared. Experimental implementation was tested on Linux operating system platform.**

1. INTRODUCTION

The boom of wireless cellular networks, and the Internet phenomenon, pointed out the need for the next generation of cellular networks to offer information services available on the Internet. The stack of protocols used in Internet should preferably be used. Unfortunately, that stack does not perform well in the cellular wireless environment, because the average payload used is small in comparison to protocol headers. This situation is almost similar to slow speed serial link transfers [1,2]. Thus, compression schemes that reduce the size of the header have become of great interest.

During the project started at the University of Split, FESB Split [3,4,5], with a goal to implement the Epic-Lite header compression scheme [6], the need to realize a modular plug-in for Linux machines arose. As EPIC-Lite scheme had been proposed for standardization to the Internet Engineering Task Force (IETF), an early implementation could facilitate this process. Linux platform was chosen because of open source and widespread of Linux boxes.

The paper has the following organization. Chapter Two describes the general requirements that have been imposed, together with Linux kernel and networking specifics. Chapter Three details the issue from the protocol side of view. In Chapter Four, elementary and advanced sublayer architectures are proposed. Chapter five gives an overview of the experiment that proves the correctness of the model. Chapter six gives conclusions and guidelines for future work.

2. IMPLEMENTATION REQUIREMENTS

The EPIC-Lite implementation team posed the following requirements to be achieved.

Functionality. The application developed should have the functionality proposed by the EPIC-Lite compression scheme [6]. One profile one data flow should be supported, with a future goal to include one profile multiple flows, and finally multiple profiles multiple flows functionality.

Flexibility. Although EPIC-Lite specification language can be used to produce dedicated application, the possibility to change profiles on the fly should be supported.

Implementability. The application being developed had to be implemented easily on Linux machines, without great changes either to the existing kernel or any other part of the system.

Transparency. Flows not considered for compressing should flow transparently.

In order to focus better on the requirements listed, an overview of Linux network and kernel architecture is given.

2.1 Linux TCP/IP network architecture

Linux follows the general guidelines for the Internet architecture, but it has some specifics. The Linux network implementation architecture is shown on Figure 1.

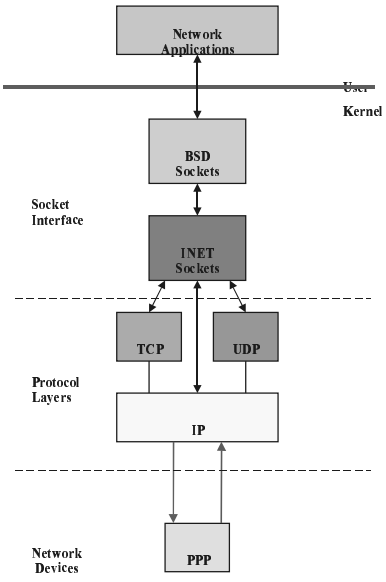

Figure 1 - Linux network architecture

The network application programs communicate via the protocol stack and the underlying actual network hardware using a special interface, sockets. The version of BSD sockets

has its implementation in many boxes around world, and is the predominant one. Underneath BSD sockets, there is a more detailed level, INET sockets, that operates efficiently due to its closer tie to the actual Linux core architecture and buffer management. The protocol processes hold the same layering of TCP/IP. These are connected to network device drivers that are on the bottom of the stack.

It should be noted that multiple network devices can be attached, and multiple protocol stacks can be supported by protocol software. Unique interface between kernel protocol stack and network device drivers provides standardized protocol identification. Ethertypes are actually used [10].

2.2. Modular technology

Early operating systems had a monolithic central place, i.e. *kernel.* It was inconvenient, because it should have contained all functionality related to particular hardware. For Linux, use of a smaller but completely functional kernel was chosen. It should do the basic process, memory, scheduler and device management. For other services, especially hardware, plug-ins are developed and attached to the kernel.

In Linux, those plug-ins are called modules. Current Linux has strong support for modules, which the kernel knows how to activate and use. They are actually drivers for specific hardware. Modules are easy to attach while the machine is running, usually without power-on reset.

The module code is compiled differently than the usual program, because as part of the kernel it has some specific requirements. Being actually part of the kernel, module runs under the same privileges and restrictions kernel does.

2.3 Kernel and user space

Kernel is the core of Linux system, and it is differentiated from other processes running on the machine. As Linux is a multi-user, multi-tasking operating system, kernel as well as modules should be protected from user programs. The division of machine to user and kernel space is thus essential.

User applications usually run in the user space, one or more segments of memory allocated by the kernel. If the application needs something from specific device, it has to supply its request to the kernel. Kernel fulfills that request, called system call. Among them, there is a set of system calls related to network procedures.

On the other hand, the kernel must work efficiently, and be available to all user processes. That brings us to the problem related to kernel (protected) space. User applications can access complex functions that are usually stored in static or dynamic libraries. Such functions, even if made reusable, do not satisfy the need for efficiency that kernel has. So, in kernel space, the programmers can not access the functions available in user applications, and many basic ones have to be rewritten from scratch.

The same goes for developing modules, all functions have to be hard coded inside, with the only exception to some kernel interface calls. It is strongly recommended to keep module size below 64K bytes.

3. PROTOCOL ISSUES

The logical position of header compression function is somewhere between data link and network layers. Considering multiple possible data link layer devices, multiple network layer protocols, and applicability of compression to only limited number of protocol - device pairs, several possible positions can be found. The proper selection is influenced by the fact that header compression protocol (like ROHC [7]) is not part of original Internet architecture, and is thus not considered a full protocol.

Three possibilities exist: to integrate compression inside the protocol stack, put it between network and data link as specific sublayer, and to integrate it into device driver module, Figure 2.

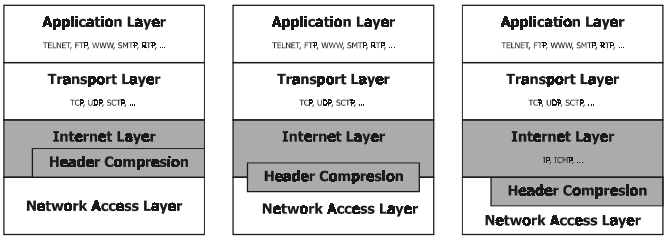

Figure 2 - Header compression integration

The device driver integration is the most logical solution, since compression takes place at single low capacity links only. End to end compression is impossible because full IP header is required for routing. Compression takes place at last hop links only. The excellent example is PPP protocol functionality, where header compression takes its role as data link protocol service. Usage of such service is negotiated among the communicating parties for a protocol stack, e.g. using PPP NCP [8]. With ROHC, this approach is proposed for standardization [9].

However, to be used for testing purposes of compression scheme, the complete new module (e.g. PPP module) must be enhanced and rebuilt, with all constrains of kernel programming being considered in this early phase.

To embed header compression functionality in protocol stack software has the only advantage that it is closely tied to that particular protocol stack. However, it is much easier for link driver to determine whether to compress particular PDU or not, because on network layer it also depends on routing decisions. The same implementation complexity remains.

The compressing sublayer approach provides a special piece of code that captures packets from and to particular link driver, and performs compression transparently.

Advantages of sublayer approach are that current protocol stack and link driver software remains untouched, that no complex negotiation procedures need to be implemented, various communications media can be exploited, and no conflict with routing function emerges.

Drawbacks are that at the moment sublayer implementation is not considered for network architecture standardization, no protocol identification is provided, and no standard way to attach such sublayer module exists.

It must be noted, that application level tunneling is often considered for protocol testing. In practice, UDP transport between test applications is established to carry compressed header packets generated from previously captured flows. The advantage of this approach is that measurements are conducted in user space easily. However, this approach does not provide real network experiments.

4. PROPOSED TESTBED ARCHITECTURE

Considering requirements and environment listed in Chapter Two, and possible protocol stack solutions from Chapter Three, one can conclude that for testing purposes sublayer implementation approach is optimal. It is therefore chosen for header compression testbed implementation. Regarding the fact that testbed is actually designed for EPIC-Lite scheme, it must satisfy two requirements: to intercept packets transparently, and to provide header compression flexibly, i.e. to allow profile changes.

4.1 Elementary sublayer architecture

To intercept packets transparently, sublayer module should be installed between standard protocol stack and the chosen network device module, Figure 3.

On boot-up time, kernel is loaded and predefined modules are attached. Other modules required can later be attached or detached manually. When attaching a module, the special module loader allocates memory in protected, kernel space. In module header, loader finds information needed to initialize entry points at kernel and module sides. This way, communication between module and kernel functions is established.

The specific feature of protocol stack interface is possibility to attach multiple modules. The same interface is reused, and list of net device structures is built. Itself, sublayer module is easily attached to this interface. It can normally access the kernel network functions, and have its own functions be called by kernel protocol stack.

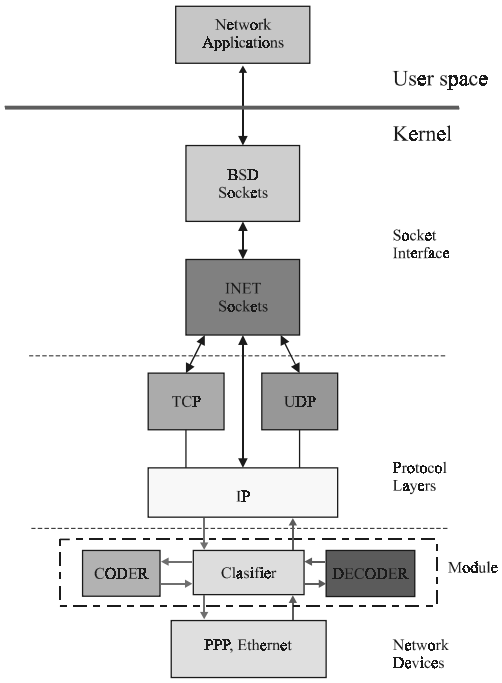

Figure 3 - Elementary sublayer architecture

Another problem is how to attach device driver module to already loaded sublayer module. In normal situation, subsequent network module would be just the next device in the device list. To the contrary, we want the subsequent network module to be attached directly to the sublayer module.

To achieve direct driver to sublayer module connection, a faked list of pointers is supplied to the module loader. Faked list is provided with information about particular driver module that is intended to be captured, so after capture has taken place, arbitrary combination of different network modules can be loaded directly to the kernel. However, this safety is not really required, because test modules will likely be loaded manually at the end of the sequence.

The proposed module sublayer solution provides elementary mechanism to integrate header compression functionality without need to modify kernel or network modules. However, to really operate header compression, module should have built in heuristics to recognize packets to be captured and compressed or decompressed.

Downstream, packets suitable for compression can be recognized using standard interface protocol identification. After compression, protocol ID should be changed to identify compressed header PDUs. Unfortunately, ROHC framework protocol is not assigned protocol identification. For testing purposes, unused or unassigned identifier can be inserted. At this point it is clear that, at least for testing purposes, protocol identifier for ROHC should be requested and assigned.

Upstream packets can be easily recognized if properly identified by compressor. However, taken the possibility that identifier chosen is used by some other source, additional testing for ROHC packets could be needed. After decompression, knowing the protocol stack to which packet belongs, decompressor can insert proper protocol identifier.

Note that PPP assigned ROHC identifiers can not be used, because they are intended to distinguish ROHC compressed packets on serial line, when ROHC compression is provided as PPP negotiated service [9].

4.2 Advanced sublayer architecture

One of distinguishing EPIC-Lite features is protocol independence, inherited from header compression definition ABNF input language. Flexibility required supposes that profiles can be change on the fly, even that multiple profiles can be supported simultaneously. To acquire new profile from the input language source, complex Huffman indicator flag and data structures generation is required. This functionality can hardly be implemented in restricted module environment. Moreover, some kind of profile source submission mechanism is needed.

The solution is found in advanced sublayer architecture, Figure 4.

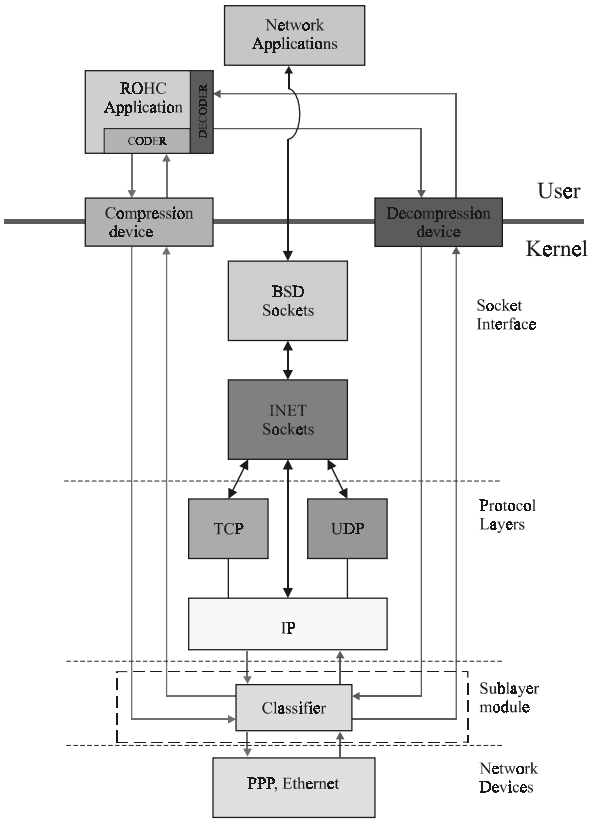

Figure 4 – Advanced sublayer architecture

Sublayer module is equipped with additional interface used to attach programs from user space. The interface is implemented to provide two additional Linux devices, using standard device access methods. One devices is used by compressor, and the other by decompressor. Character mode device is used, with advantage of simplified synchronization between the sublayer module and application program.

Using advanced architecture, only very limited functionality needs to be implemented in sublayer module. It should comprise basic packet capture, possibly with some packet filtering mechanism.

When profile preprocessing is finished, user space application is ready for compression and decompression. It opens sublayer device in standard way, triggering packet classifier operation. Proper packets are chosen for compression and decompression. Being performed in user space, this process can be traced, logged and measured easily.

4.3 Implementation issues

To perform complex connections that sublayer module is supposed to keep with protocol stack, network module and device interface, some standard attachment functions have to be workaround.

The modification was made to the *Init_module()* in order to intercept the appropriate *hard_start_xmit()* function. That function is part of the *netdevice* structure, which holds vital information about the network interface and is easily accessible from the network module. The actual low-level transmit function is patched to perform transmitter packet classification and redirection to compressor device interface. From the same interface, compressed packets are forwarded to network device calling the original *hard_start_xmit()* function.

The transmission path of the module is enhanced with FIFO buffer of *sk_buff* structures. This way, the problem of synchronization of kernel and user space is resolved. The packet intercepted is immediately acknowledged as sent, and actually is kept in FIFO for delivery to compressor device interface.

When interface card completes a frame, received from the data link, into a special memory structure called *sk_buff,* the *netif_rx()* function is called. Access to this function is done through the kernel symbol table, where names and pointers of all available functions are stored. The entries in that table can not be easily modified.

Being unable to directly change the address of *netif_rx()*, specific system call, namely *query_module()*, had to be modified. *Query_module()* is used in order to supply the kernel symbol table addresses, and is executed by the *insmod* command (module loader) that does the actual module loading into the kernel. The modification to *Query_module()* was made in order to submit a pointer to the new version of

netif_rx(). The patched version routes the packet to the decompressor device interface. Decompressed packet is returned, to be delivered to protocol stack software with a call to original *netif_rx()*.

5. EXPERIMENTAL HYBRID SOLUTION DEPLOYMENT

In order to test the presented concept, a compact sublayer module was developed. It contains device module registering and packet interception code as described before.

Two test Linux machines were setup, namely *A* and *B*, both running Redhat 7.2. The machines were connected via a serial cable, and the protocol used was PPP. A simple PING utility was intended to provide the data flow between the stations. Figure 5 shows the logical layout of the test.

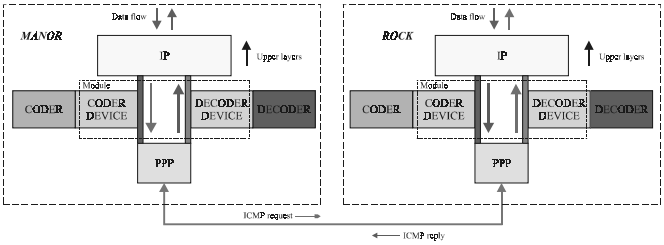

Figure 5 – Test layout

On both machines the developed sublayer module was started before loading the PPP deamon on *A* and the PPP client on *B*. The module intercepted the initialization routines of those modules, and exchanged the usual network routines with expanded ones.

The coder and decoder devices were initialized, and two simple user applications, *coder* and *decoder* were provided. Both were developed to run in user space. The *coder* application was programmed to shorten the packets by 20 bytes, whilst the *decoder* was written to do the opposite, to expand those missing 20 bytes.

On *A* the *coder* and *decoder* applications were run. The PING utility was started on *A* and the ICMP request packets (ICMP uses TCP/IP stack for delivery) were sent to *B*. There was no reply from *B* as all the packets were discarded by the upper layers because they were of incorrect size. Then, the *coder* and *decoder* programs were started on *B*, and immediately the PING utility started to receive the ICMP replies from *B*. The PING was then executed from the other side to test the reverse functionality.

6. CONCLUSION

The proposed hybrid solution successfully conforms to the implementability and transparency requirements, and is especially feasible for initial, real network protocol tests. In early phases of protocol development such an interface is invaluable, allowing quick adaptation to the changes in specification, and a quick way to validate and test new premises and functionality.

The functionality implemented can support the EPIC-Lite compression scheme testing without a need to provide complex negotiation procedures, which should otherwise be needed. These procedures would make real network testing more complicated.

Running test application in user space, various compression profiles can be submitted and used without any burden about kernel data transfers or in-band and out-band signaling.

Finally, the EPIC-Lite functionality can be approved in file to file compression and decompression of captured flows, and then quickly attached to the testbed for real network experiments.

The current sublayer module implementation has only rudimental packet classifying possibilities. In future, these should be tailored to EPIC-Lite needs. However, the best solution could be to assign full protocol identifier to ROHC framework PDUs, at least for testing purposes. That could provide freedom to apply the header compression as service, or as full protocol.

REFERENCES

[1] Jacobson, V. "Compressing TCP/IP Headers for Low-Speed Serial Links", RFC 1144, 1990.

[2] Casner S, Jacobson V, "Compressing IP/UDP/RTP Headers for Low-Speed Serial Links", RFC 2508, 1999.

[3] Ozegovic, J. "Optimal syntax for packet header compression specification", SOFTCom 2002.

[4] Cizmic, M., Vodopija, T.,' Ozegovic, J.: "EPIC Lite offline processing", SOFTCom 2002.

[5] Stula, M., Vidjak, L.,' Ozegovic, J.: " Program structures for EPIC-LITE experimental implementation ", SOFTCom 2002.

[6] R. Price, R. Hancock, S. McCann, A. Surtees, P. Ollis, M. West: "Framework for EPIC-LITE", draft-ietf-rohc-epic-lite-01.txt, February 2002.

[7] Bormann C, et al, "Robust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed", RFC 3095, 2001.

[8] Simpson, W., " The Point-to-Point Protocol (PPP)", RFC 1661, 1994.

[9] Bormann, C, "Robust Header Compression (ROHC) over PPP", RFC 3241, 2002.

[10] Linux cross reference, http://lxr.linux.no