

## INFORMATION SOCIETY TECHNOLOGIES (IST) PROGRAMME



A Scaleable Monitoring Platform for the Internet  
Contract No. IST-2001-32404

### D3.4 “Description of Experiment Results”

**Abstract:** This document provides description of evaluation tests performed on the SCAMPI architecture. This is the first release of D3.4 deliverable, which includes performance tests of the SCAMPI architecture software running on top of the Gigabit Ethernet SCAMPI adapter or commodity Intel Gigabit Ethernet adapter with monitoring functions implemented in software and first application tests. The final release of D3.4 will include also tests with the 10 Gigabit Ethernet SCAMPI adapter, tests with monitoring functions implemented in firmware and more application tests. These tests are yet to be performed, due to ongoing software, firmware and hardware development.

|                              |  |
|------------------------------|--|
| Contractual Date of Delivery | 31 May 2004                            |
| Actual Date of Delivery      | 4 June 2004                            |
| Deliverable Security Class   | Public                                 |
| Editor                       | Sven Ubik                              |
| Contributors                 | IMEC, FORTH, UNINETT, CESNET, FORTHnet |

The SCAMPI Consortium consists of:

|          |                      |                 |
|----------|----------------------|-----------------|
| TERENA   | Coordinator          | The Netherlands |
| IMEC     | Principal Contractor | Belgium         |
| FORTH    | Principal Contractor | Greece          |
| LIACS    | Principal Contractor | The Netherlands |
| NETikos  | Principal Contractor | Italy           |
| UNINETT  | Principal Contractor | Norway          |
| CESNET   | Principal Contractor | Czech Republic  |
| FORTHnet | Principal Contractor | Greece          |
| 4Plus    | Principal Contractor | Greece          |
| Siemens  | Principal Contractor | Germany         |

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>   | <b>3</b>  |
| <b>2</b> | <b>Configuration for the maximum load test</b>                | <b>3</b>  |
| <b>3</b> | <b>Configuration for packet processing overhead test</b>      | <b>3</b>  |
| <b>4</b> | <b>Maximum load test - Gigabit Ethernet</b>                   | <b>5</b>  |
| <b>5</b> | <b>Maximum load test - 10 Gigabit Ethernet</b>                | <b>6</b>  |
| <b>6</b> | <b>Packet overhead test - software processing</b>             | <b>7</b>  |
| <b>7</b> | <b>Packet overhead test - firmware processing</b>             | <b>9</b>  |
| <b>8</b> | <b>Application tests</b>                                      | <b>10</b> |
| 8.1      | Intrusion detection application . . . . .                     | 10        |
| 8.1.1    | Description . . . . .   | 10        |
| 8.1.2    | Performance . . . . .   | 10        |
| 8.2      | QoS Application Experiments . . . . .                         | 12        |
| 8.2.1    | Experimentation environment . . . . .                         | 12        |
| 8.3      | One-way delay and jitter measurements . . . . .               | 12        |
| 8.4      | Packet loss measurements . . . . .                            | 13        |
| <b>A</b> | <b>Sample program for the maximum load test</b>               | <b>15</b> |
| <b>B</b> | <b>Sample program for the packet processing overhead test</b> | <b>16</b> |
| <b>C</b> | <b>Initialization of PAPI performance analysis</b>            | <b>17</b> |
| <b>D</b> | <b>Using PAPI to read virtual counters</b>                    | <b>18</b> |

## 1 Introduction

The purpose of the workpackage WP3 “Experimental evaluation” is to do functionality and performance tests of the SCAMPI architecture in order to validate its applicability for passive network measurements.

This document is the first release of the deliverable D3.4. It includes the results of performance tests done on the SCAMPI software running on top of the Gigabit Ethernet SCAMPI adapter and part of the application tests. The objective of these tests was to find the maximum number of packets per second that can be processed by the SCAMPI architecture on a selected commodity PC in various scenarios, to measure the overhead of processing per packet, which is an indicator of the architecture effectiveness independent of the particular host computer and to evaluate certain applications developed as part of the project.

## 2 Configuration for the maximum load test

We used Spirent AX/4000 packet generator to produce a stream of packets with specified packet size and number of packets per second. The configuration is shown in Fig. 1. The SCAMPI architecture was installed on the PC on the left. It included the Gigabit Ethernet SCAMPI adapter, device driver, SCAMPI library, MAPI and simple packet capture application. The PC itself was P4/2.0 GHz with 256 MB RAM and Intel Gigabit Ethernet Adapter in a 64-bit/64 MHz slot. The operating system was Linux with kernel 2.4.22. The receiving buffer for each UDP socket was 1 MB. The way that packets passed through the SCAMPI architecture is illustrated in Fig. 2.

A sample program written on top of MAPI to test the maximum number of packets processed per second is shown in Appendix A. PKT\_COUNTER function was applied to a flow and `mapi_get_result()` was called each 100 ms to get the number of processed packets. This was indicated as the number of packets from the previous call to `mapi_get_result()` for the SCAMPI adapter and as cumulative number of packets for the Intel Gigabit Ethernet adapter.

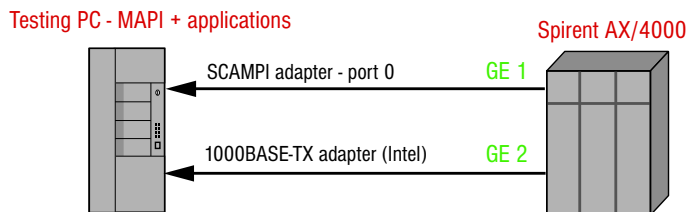


Figure 1: Configuration for the maximum load test

## 3 Configuration for packet processing overhead test

We used the PAPI (Performance API) [1] version 3.0 beta 2 to find the number of instructions and cycles required to process one packet in various scenarios. The configuration is shown in Fig. 3. The SCAMPI architecture was installed on the PC on the left. The PC on the right was used to generate packets satisfying required criteria, such as including a specified text string.

A sample program written on top of MAPI to try various scenarios of packet processing is shown in Appendix B. First, a new stream of packets was created. Then, various functions were applied to this stream one at a time or several of them together. Finally, part of the code in the sample program and inside the MAPI were wrapped around by calls to PAPI to find the overhead of the particular piece of code. We will describe these PAPI calls in more detail below.

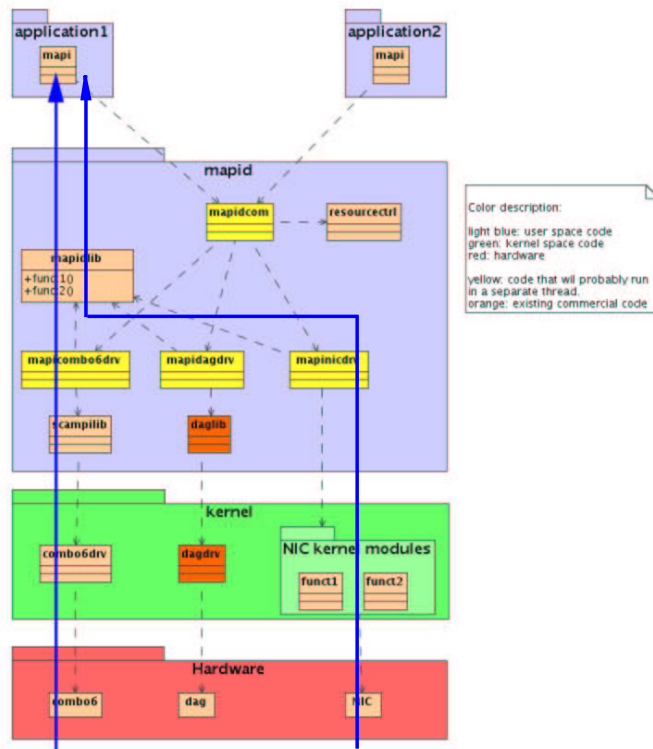


Figure 2: Packet passing through the SCAMPI architecture

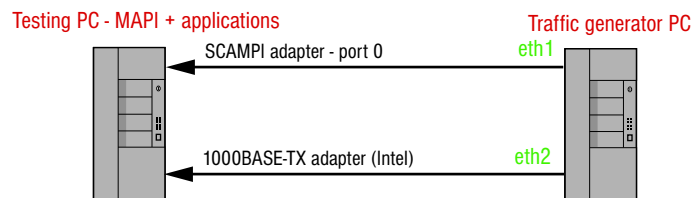


Figure 3: Configuration for the packet overhead test

## 4 Maximum load test - Gigabit Ethernet

We measured the maximum number of 1500-byte and 64-byte packets that can be processed per second when using the SCAMPI adapter or the Intel Gigabit Ethernet adapter and a sample program written on top of MAPI shown in Appendix A. The program applies only PKT\_COUNTER to the flow that is packets are just counted.

We increased the packet rate on the Spirent AX/4000 generator in a logarithmic scale and observed packet losses. When packet losses started to occur, we tried to find the maximum rate without losses by adjusting sending rate in steps of 500 packets per second. The result is summarized in Table 1. It should be noted that the test was done with the first release of SCAMPI adapter driver, which is not yet optimized for maximum performance.

|            | SCAMPI adapter |         | Intel adapter |         |
|------------|----------------|---------|---------------|---------|
|            | 1500-byte      | 64-byte | 1500-byte     | 64-byte |
| 4000 p/s   | 0%             | 0%      | 0%            | 0%      |
| 5000 p/s   | 0%             | 0%      | 0.004%        | 0.003%  |
| 10000 p/s  | 0%             | 0%      | 0.016%        | 0.017%  |
| 17500 p/s  | 0%             | 0%      | 0.019%        | 0.021%  |
| 25000 p/s  | 0%             | 0.004%  | 0.031%        | 0.030%  |
| 37500 p/s  | 0%             | 0.005%  | 0.055%        | 0.050%  |
| 50000 p/s  | 10.3%          | 0.017%  | 0.039%        | 0%      |
| 75000 p/s  | 40.1%          | 0.031%  | 0.230%        | 0.002%  |
| 100000 p/s | NA             | 9.6%    | NA            | 0.057%  |
| 150000 p/s | NA             | 39.4%   | NA            | 59.1%   |

Table 1: Packet loss rate depending on sending rate and adapter type

We can observe the following points:

- The maximum rate that could be processed without any loss by the Intel Gigabit Ethernet adapter was 4000 packets per second for both 1500-byte and 64-byte packets.
- The maximum rate that could be processed without any loss by the SCAMPI adapter was 37500 packets per second for 1500-byte packets and 17500 packets per second for 64-byte packets.
- When using the Intel Gigabit Ethernet adapters, packet losses started much earlier than when using the SCAMPI adapter, which was able to process much more packets without losses.
- The maximum IP layer bit rate that could be processed when using the Intel Gigabit Ethernet adapter was 898 Mb/s for 1500-byte packets and the rate of 75000 packets per second.
- The maximum IP layer bit rate that could be processed when using the SCAMPI adapter was 538 Mb/s.
- With the current development version of the SCAMPI adapter driver, the maximum processed bit rate was lower than with the Intel Gigabit Ethernet adapter. A more effective data transfer should be available with future driver versions.

## **5 Maximum load test - 10 Gigabit Ethernet**

This test will be done when the 10 Gigabit Ethernet SCAMPI adapter is completed and when we acquire 10 Gb/s packet generator. The adapter development and packet generator procurement are ongoing. The results will be presented in the final release of D3.4.

## 6 Packet overhead test - software processing

We used a set of calls to PAPI shown in Appendix C to initialize PAPI library and start counting of virtual instruction and cycle counters in the client application written on top of MAPI, as well as inside the mapid daemon. We then used a set of calls to PAPI shown in Appendix D to read virtual instruction and cycle counters in various places inside the code.

The number of instructions and cycles measured as required for processing of one packet when various functions were applied to a flow is given in Table 2. The packet processing overhead includes resources spent in MAPI internal function `mapid_process_pkt()` and all functions applied to a packet from this function.

The number of instructions and cycles measured as required for certain MAPI calls, which are the most important calls from the performance viewpoint, are given in Table 3.

The number of additional instructions measured as required for processing of one packet when more flows were opened and the particular function were applied to each of these flows are indicated in Table 4. For example, when three flows are opened, the particular MAPI function is applied to all three flows and an arriving packet also belongs to all three flows, then the processing overhead for this function per packet will be the sum of the number indicated in Table 2 and twice the number indicated in Table 4.

The numbers in these tables include the overhead of PAPI itself. In order to establish the overhead of PAPI itself we used two consecutive calls to individual PAPI functions, with no intervening program code between them, and summarized overhead of those functions used in the PAPI code fragment described in Appendix D. One set of calls to PAPI required 265 instructions or 700 cycles. The PAPI overhead should be deducted from the packet and MAPI function overhead tables.

| Functions applied  | Packet size | Packet processing overhead |                       |
|--|-------------|----------------------------|-----------------------|
|  |             | SCAMPI adapter             | Intel adapter         |
| No function  | any         | 321 ins. 1500 cyc.         | 318 ins. 900 cyc.     |
| COUNTER  | any         | 321 ins. 1500 cyc.         | 371 ins. 700 cyc.     |
| BPF_FILTER   | any         | 601 ins. 2000 cyc.         | 490 ins. 2800 cyc.    |
| SAMPLE_PACKETS   | any         | 379 ins. 800 cyc.          | 376 ins. 3700 cyc.    |
| TO_BUFFER  | any         | 537 ins. 8800 cyc.         | 8316 ins. 14700 cyc.  |
| BPF_FILTER + TO_BUFFER   | any         | 817 ins. 11900 cyc.        | 8491 ins. 15500 cyc.  |
| COUNTER + BPF_FILTER +<br>SAMPLE_PACKETS + STR_SEARCH +<br>TO_BUFFER | 1500 bytes  | 18850 ins. 26500 cycles    | 8541 ins. 3500 cyc.   |
| STR_SEARCH   | 64 bytes    | 1393 ins. 3900 cycles      | 1388 ins. 2900 cyc.   |
| STR_SEARCH   | 128 bytes   | 2149 ins. 3400 cycles      | 2144 ins. 2800 cyc.   |
| STR_SEARCH   | 256 bytes   | 3697 ins. 4400 cycles      | 3692 ins. 3800 cyc.   |
| STR_SEARCH   | 512 bytes   | 6757 ins. 7500 cycles      | 6752 ins. 6200 cyc.   |
| STR_SEARCH   | 1024 bytes  | 12913 ins. 14500 cycles    | 12908 ins. 11300 cyc. |
| STR_SEARCH   | 1500 bytes  | 18459 ins. 20300 cycles    | 18452 ins. 15600 cyc. |

Table 2: Packet processing overhead in instructions and cycles

| Functions applied      | <code>map_get_next_packet()</code> |               | <code>map_get_result()</code> |               |
|------------------------|------------------------------------|---------------|-------------------------------|---------------|
|                        | SCAMPI adapter                     | Intel adapter | SCAMPI adapter                | Intel adapter |
| COUNTER                | NA                                 | NA            | 7152 ins.                     | 553 ins.      |
| TO_BUFFER              | 513 ins.                           | 1246 ins.     | NA                            | NA            |
| BPF_FILTER + TO_BUFFER | 522 ins.                           | 1246 ins.     | NA                            | NA            |

Table 3: MAPI function overhead in instructions and cycles

| Functions applied              | Additional packet overhead |               |
|--------------------------------|----------------------------|---------------|
|                                | SCAMPI adapter             | Intel adapter |
| COUNTER                        | 21 ins.                    | 74 ins.       |
| TO_BUFFER                      | 237 ins.                   | not supported |
| BPF_FILTER (the same filter)   | 86 ins.                    | 82 ins.       |
| BPF_FILTER (different filters) | 274 ins.                   | 191 ins.      |
| STR_SEARCH (the same filter)   | 95 ins.                    | 96 ins.       |
| STR_SEARCH (different filters) | 18156 ins.                 | 4295 ins.     |

Table 4: Additional packet overhead for each additional flow opened

Comments and observations:

- Unlike numbers of instructions, numbers of cycles differed in different test runs. We investigated this behaviour and thanks to advice from PAPI developers in PAPI mailing list [2] we managed to reduce this effect by proper sequence of PAPI calls. Numbers of cycles in tables are averages from 5 test runs rounded to 100 cycles.
- Packet processing overhead when just the function COUNTER was applied to a flow and the SCAMPI adapter was used was the same as when no function was applied to a flow. The reason is that packets and bytes were counted inside the SCAMPI adapter driver and this counting is performed all the time.
- We can see that the packet processing overhead when using the SCAMPI adapter is comparable to overhead when using the Intel Gigabit Ethernet adapter. The reason is that all packet processing is now done in software, rather than in firmware on the SCAMPI adapter. We will make a new round of tests when development of processing in firmware is completed.
- The only case when the packet processing overhead depends on the packet size is then the function STR\_SEARCH is applied to a flow. Obviously, the longer the packet, the more resources are required to search its payload.
- The overhead of `mapi_get_result()` function for PKT\_COUNTER is bigger when the SCAMPI adapter is used, because the counter

## **7 Packet overhead test - firmware processing**

This test will be done when processing in firmware becomes available. Several firmware components are already developed and tested, but the whole firmware design and its support in driver are not yet suitable for performance tests. The results will be presented in the final release of D3.4.

## 8 Application tests

The purpose of application tests is to demonstrate the practicability of the SCAMPI architecture for network monitoring. Development of most applications is ongoing. The following sections present preliminary results. The complete results will be presented in the final release of D3.4.

### 8.1 Intrusion detection application

#### 8.1.1 Description

MAPI offers all the functionality needed to build a fast intrusion detection system. We have implemented such an intrusion detection system, called *sids*. *Sids* takes as an input simple rules describing potential attacks and translates them into the appropriate MAPI functions. Each rule is mapped into a flow and can perform header analysis as well as packet payload inspection. The format of rules given as input to *sids* is fixed. As an example

```
count tcp and dst port 80; content: /bin/perl.exe; id:100;
```

describes a rule that searches for “/bin/perl.exe” pattern inside TCP packets that are destined to port 80. The first keyword describes what action should be done if a packet matches the rule. So far, *sids* supports counting the packets matching the rule (keyword “count” in the previous example) and copying them into files (keyword “copy”). The second argument is a BPF filter, describing the header we search for. Furthermore, using keyword “content” *sids* can perform pattern matching. One rule can contain more than one patterns to be searched.

As MAPI supports flow cooking, *sids* packet inspection can become stateful so as not to miss attacks on packet borders. Furthermore, packets can be read from trace files either on *tcpdump* or *dag* format. For practical purposes, we have developed a tool, *snort2sids* that converts precisely Snort rulesets into *sids* rules. This tool saves users from writing rules with hand as Snort<sup>1</sup> rulesets are large, frequently updated and publicly available.

#### 8.1.2 Performance

We have measured the performance of *sids* under various conditions. The results presented here are preliminary as both MAPI and *sids* are in development stage but are representative of MAPI’s processing capability. In our experiments, we used live Ethernet traffic at 100Mbps rate. Two machines connected back to back were involved in our measurements, a sender and a receiver. The intrusion detection system and MAPI daemon were running on the receiver, a Pentium 4 3GHz with 1GB main memory and 256KB L1 cache. All packets were uniform (tcp and destination port 5001) and were generated with *ttcp*<sup>2</sup> tool.

First, we examined how the processor load scales with the number of rules. Each rule was describing an attack on a specific port and more precisely attacks on a Web server. We tested two implementations of MAPI: an unoptimized one, where all flows are traversed linearly and an optimized one, where only the flows needed are examined (optimization is based on destination port). In Figure 4, we observe that *sids* can examine up to 500 rules without packet loss for the unoptimized version but the load remains stable for the optimized one. In the case of optimized version, no rules matched the traffic examined as we had no web traffic and consequently redundant flow traversal was avoided.

Our second experiment was focused on measuring the performance of *sids* in case where rules search for patterns inside the packet. We used the unoptimized version of MAPI as no header description was provided by the rules, thus no optimization could be performed. In Figure 5, we see that *sids* can examine up to 150 rules containing patterns. The payload of each packet was constant, as generated by the *ttcp* tool (-abcde...-). Optimization on pattern matching is in progress and major performance gains are expected.

---

<sup>1</sup><http://www.snort.org>

<sup>2</sup><http://www.pcausa.com/Utilities/pcattcp.htm>

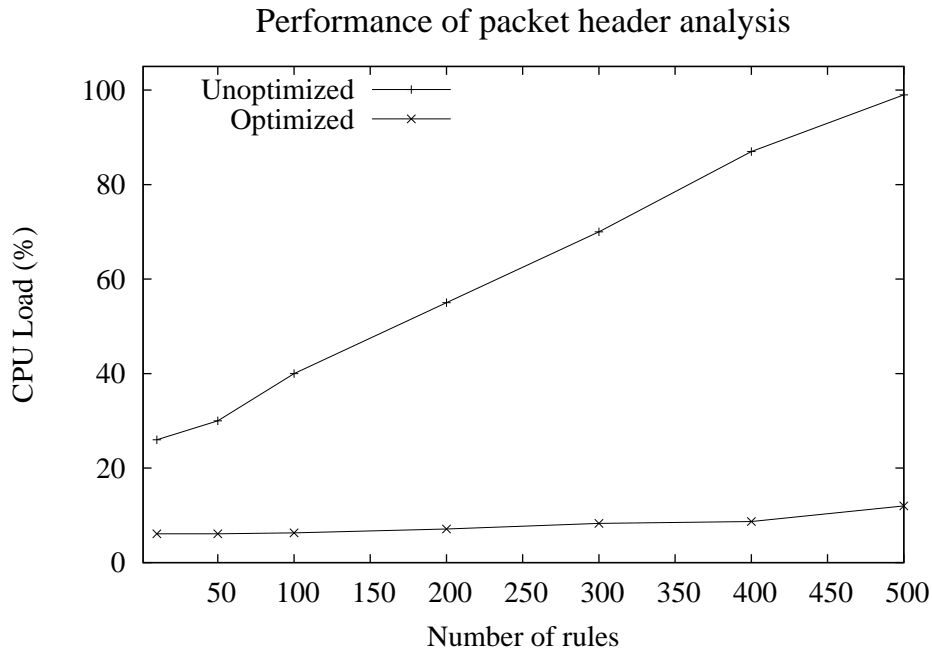


Figure 4: Performance analysis of header-only rules

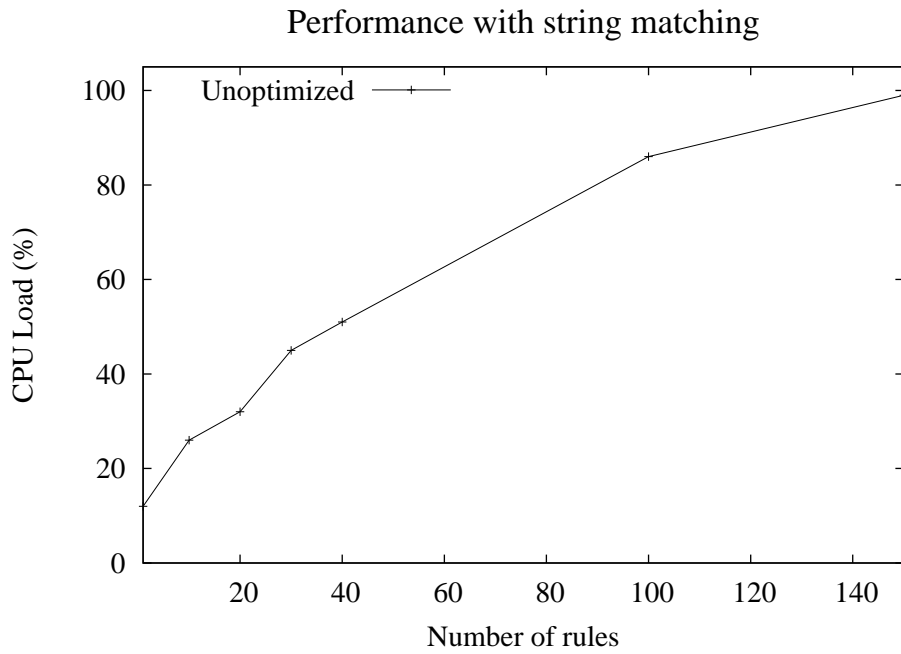


Figure 5: Performance analysis when rules contain patterns

## 8.2 QoS Application Experiments

In order to evaluate the functionality of the Quality of Service (QoS) monitoring application, some tests were conducted. These tests will validate the correctness of the measured QoS characteristics (one-way delay, jitter and packet loss) in a controlled environment.

### 8.2.1 Experimentation environment

To introduce delay and loss in the network, a Click-based[3] impairment node was configured in the network. To verify the measured QoS characteristics, they were compared to the results obtained by a Smartbits[4] network performance analysis system. This same Smartbits was used to generate the traffic. The used topology is illustrated in Fig. 6. The Smartbits system is connected to two access routers (Leucothea170 and Leucothea171). These routers contain a SCAMPI monitor running the QoS monitoring application. The application is configured to sample part of the captured packets and write useful data to the database. The core of the network only consists of one router (Leucothea173), which runs the impairment software.

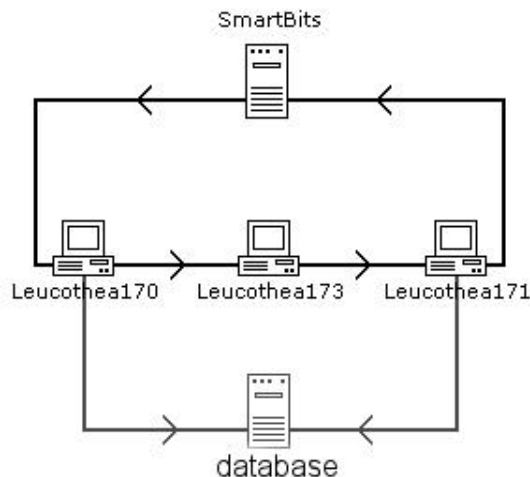


Figure 6: Experiment topology

The artificially introduced traffic generated by the Smartbits hardware contains random payload. This allows the SCAMPI QoS monitor to compute a unique hash over the individual packets. For each test a 10Mbit stream was generated during a period of 5 minutes. The frame size of each packet was 256 bytes.

### 8.3 One-way delay and jitter measurements

When comparing the delay and jitter obtained by the SCAMPI monitor and the Smartbits system, we varied the amount of introduced delay by the impairment node. The introduced delay was progressively increased from 0 to 0.5ms in steps of 0.05ms. Figure 7 shows the results obtained when using a sampling rate of 5% in the SCAMPI application.

The delay measured by the Smartbits software is generally 0.12ms higher than the delay measured with the SCAMPI application. This is due to the extra delay over the two links between the Smartbits and the access routers. Ignoring this additional constant delay experienced by the Smartbits, we see that the measured delay of the QoS application is exactly the same as the Smartbits results.

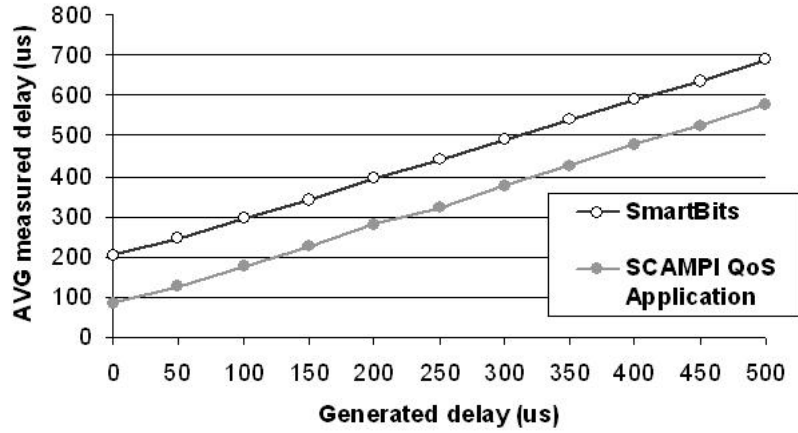


Figure 7: Delay measurements

#### 8.4 Packet loss measurements

Instead of delaying packets in the impairment node, in these tests random packets were dropped, generating packet loss. Using a probabilistic dropper, we progressively introduced more packet loss, ranging from 0% to 100% in steps of 10%. Figure 8 shows the comparison of the measured packet loss. In case of the QoS application, different sampling rates were used.

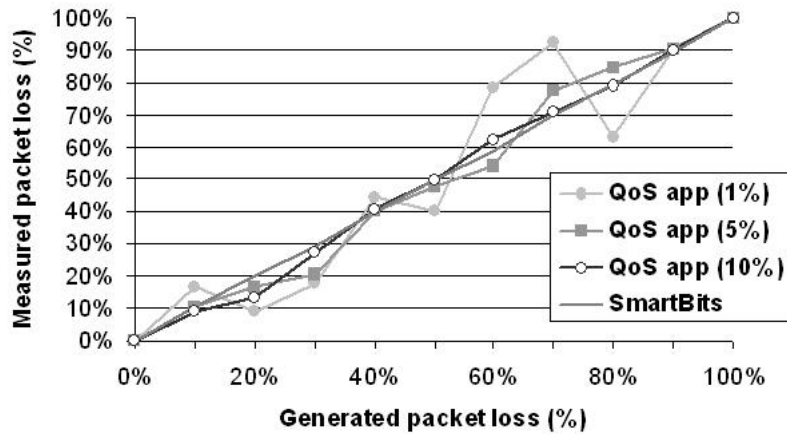


Figure 8: Packer loss measurements

The measurements with the Smartbits system show the correct packet loss. Using the QoS application, we try to approximate this correct result by choosing an appropriate packet sampling rate.

Comparing the results to the previously measured accuracy of the delay measurements, the packet loss deviates much more from the correct result as obtained by Smartbits. This is because, in case of the delay measurements, each sampled packet contributes to the average delay. Even though there is a high delay variation, the average will stay about the same. A subset of sampled packet will have the same average delay as the average of the entire set. This however is not the case with packet loss. A subset of sampled packets does not necessarily contain the same number of dropped packets as the entire flow. It greatly depends on which packets are sampled in both access points.

Figure 8 illustrates that the results improve when raising the number of sampled packets. For a sampling rate of 1%, we obtain very poor results. 5% is much better, while 10% already gives a good approximation of the real packet loss. Depending of the amount of precision needed by the application and the processing power of the SCAMPI monitor, an appropriate sampling percentage can be chosen.

## A Sample program for the maximum load test

```
#include <stdio.h>
#include <unistd.h>    /* sleep */
#include "mapi.h"

int main(void)
{
    int fd;
    int counter_all;
    int total_packets=0;
    unsigned long long *cr;

    fd=mapi_create_flow("/dev/scampi/0");
    // fd=mapi_create_flow("eth1");
    counter_all=mapi_apply_function(fd, "PKT_COUNTER");
    mapi_connect(fd);

    while(1) {
        usleep(100000);
        cr=mapi_read_results(fd, counter_all, 1);
        total_packets+=*cr;
        printf("pkts all: %lld\n", total_packets);
        // printf("pkts all: %lld\n", *cr);
    }

    return 0;
}
```

## B Sample program for the packet processing overhead test

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <sys/time.h>
#include "mapi.h"

int main(void)
{
    int fd;
    int counter_all, counter_bpf, counter_str;
    unsigned long long *cr, *cr2, *cr3;
    int bufid;
    struct mapipkt* pkt;
    int i, j;
    unsigned char *p;
    char *fce="testcombo6()";

    fd=mapi_create_flow("/dev/scampi/0");
    counter_all=mapi_apply_function(fd, "PKT_COUNTER");
    mapi_apply_function(fd, "BPF_FILTER", "src port 2005");
    mapi_apply_function(fd,"SAMPLE_PACKETS","5", "PROBABILISTIC");
    counter_bpf=mapi_apply_function(fd, "PKT_COUNTER");
    mapi_apply_function(fd,"STR_SEARCH", "www", 0, 1500);
    counter_str=mapi_apply_function(fd, "PKT_COUNTER");
    bufid=mapi_apply_function(fd, "TO_BUFFER");
    mapi_connect(fd);

    while(1) {
        sleep(1);

        cr=mapi_read_results(fd, counter_all, 1);

        pkt=mapi_get_next_pkt(fd, bufid);

        printf("size: %d\n", pkt->caplen);

        cr2=mapi_read_results(fd, counter_bpf, 1);
        cr3=mapi_read_results(fd, counter_str, 1);
        printf("pkts all:%lld bpf:%lld (%.2f%%) str:%lld (%.2f%%)\n",
            *cr, *cr2, ((double)(*cr2)/(double)(*cr))*100.0, *cr3, ((double)(*cr3)/(double)(*cr))*100);
    }

    return 0;
}
```

## C Initialization of PAPI performance analysis

```
#include <papi.h>

#define NUMBER_OF_EVENTS 3

int event_set = PAPI_NULL;
long_long values[NUMBER_OF_EVENTS], values2[NUMBER_OF_EVENTS];
long_long start_cycles, end_cycles, start_usec, end_usec;
long_long start_cycles2, end_cycles2, start_usec2, end_usec2;

int main(void)
{
    int retval;

    /* Initialize the PAPI library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);
    if (retval != PAPI_VER_CURRENT) {
        fprintf(stderr, "PAPI library init error!\n");
        exit(1);
    }

    /* Create event set */
    if (PAPI_create_eventset(&event_set) != PAPI_OK) {
        fprintf(stderr, "PAPI_create_eventset() failed\n");
        exit(-1);
    }

    /* Add events */
    if ((retval=PAPI_add_event(event_set, PAPI_TOT_INS)) != PAPI_OK) {
        fprintf(stderr, "PAPI_add_events(PAPI_TOT_INS) failed: %s\n", PAPI_strerror(
retval));
        exit(-1);
    }
    if ((retval=PAPI_add_event(event_set, PAPI_TOT_CYC)) != PAPI_OK) {
        fprintf(stderr, "PAPI_add_events(PAPI_TOT_CYC) failed: %s\n", PAPI_strerror(
retval));
        exit(-1);
    }

    /* Start counting */
    if (PAPI_start(event_set) != PAPI_OK) {
        fprintf(stderr, "PAPI_start() failed\n");
        exit(-1);
    }

    /* . . . */
}
```

## D Using PAPI to read virtual counters

```
if (PAPI_read(event_set, values) != PAPI_OK) {  
    fprintf(stderr, "PAPI_read()\n");  
    exit(-1);  
}  
start_cycles=PAPI_get_virt_cyc();  
start_usec=PAPI_get_virt_usec();  
start_cycles2=PAPI_get_real_cyc();  
start_usec2=PAPI_get_real_usec();
```

## References

- [1] *PAPI - Performance Application Programming Interface*, <http://icl.cs.utk.edu/papi>.
- [2] Philip J. Mucci. Discussion in PAPI mailing list, <http://icl.cs.utk.edu/papi/custom/index.html?id=50&slid=67>.
- [3] R. Morris, E. Kohler, J. Jannotti and M. F. Kaashoek, “The Click modular router”, In Proc. 17-th Symposium on Operating Systems Principles, pages 217-231, 1999.
- [4] SmartBits network performance analysis system, Spirent communications, “<http://www.spirentcom.com/>”.