

Figure 2: Control Plane Benchmark Framework

Efficiency of packet handling is the primary factor in performance of user plane functionality. On the control plane we have timeliness requirements of setting-up a service session, preparing accounting and charging, executing handoff, reserving and releasing resources, etc. The performance of control plane operations is affected by databases, authentication, communication between network elements, among others. Although processing efficiency is important, performance is not so critical on the management plane. The main concerns include correctness, reliability, security, and availability.

Application-level benchmarks exist, but only very few are applicable to benchmark control plane applications in telecom environments. SIPStone [31] is a benchmark for SIP proxy servers. The ETSI TISPAN working group has proposed a telecom benchmark for IP Multimedia Subsystem/Next Generation Networks (IMS/NGN) [9, 10, 11].

We have developed a new application-level benchmark specification for measuring the performance of different transport, operating system, database, and hardware configurations in control plane applications. In contrast to SIPStone, which measures different SIP proxy server implementations and treats the server as a black box, our benchmark has a fixed server core implementation, but requires the user to provide implementations of the other core components of the server, most importantly the signalling transport layer and the user location database. This allows us to change one component at a time and see the effects it has on total system performance in typical control plane usage. The benchmark framework is depicted in Figure 2.

Our benchmark implementation, which is available from SourceForge [19], uses two major software packages: OpenSER (<http://www.openser.org/>) and SIPp (<http://sipp.sourceforge.net/>).

The implementation has scripts for building, configuring, and running the benchmark. The implementation also includes tools for generating and monitoring server load, registering users to OpenSER, analysing the benchmark results, and producing graphs from the results.

The rest of the paper is organized as follows. In Section 2 we review related work including several networking related benchmarks, SIPstone and IMS/NGN Performance Benchmark among others. The specification of our control plane benchmark is given in Section 3. Suitable auxiliary benchmarks—Imbench, some database and XML benchmarks—are briefly summarized in Section 4. Finally, in Section 5, we describe our prototype implementation.

2 Related Work

Although the number of published benchmarks is high, very few target in learning the performance of control plane applications. In this section we briefly summarize some benchmarks that can be used to study performance of signalling systems. The benchmarks are SIPstone [31], IMS/NGN Performance Benchmark [9, 10, 11], CommBench [45], NpBench [17], NetBench [24] and benchmarks from Network Processing Forum (NPF) [25].

2.1 SIPstone

In a draft of the *SIPstone* benchmark [31], Schulzrinne, Narayanan, Lennox, and Doule propose a set of metrics for evaluating and benchmarking the performance of SIP proxy, redirect, and registrar servers. The goal of the benchmark is to measure the request handling capacity of SIP servers.

The benchmark environment consist of the *server under test* (SUT), which is a SIP proxy, redirect or registrar server, *load generators*, which generate the request load, *call handlers*, which simulate user-agent servers, and a *central benchmark manager*, which coordinates the execution of the benchmark. The load generators are, in essence, SIP user-agent clients and the call handlers SIP user-agent servers.

The coordinator is configured to know the names of the user-agent clients and servers. The test parameters are also fed to the user-agent clients and servers from the coordinator. The coordinator uses a remote invocation mechanism to start the user-agent clients and servers.

The user-agent clients are configured with the request rate to be generated, the request type (INVITE or REGISTER), transport protocol (UDP or TCP), and the number of requests to generate. The user-agent client gener-

ates SIP requests with a Poisson arrival distribution and records the response times.

The user-agent servers are the call receivers needed by some of the test cases. They answer an INVITE request with an immediate “180 Ringing” response, instantly followed by a “200 OK” message. The user-agent server must be able to respond within 100 ms.

The benchmark draft defines five test cases. Each test case is run using both UDP and TCP as the transport protocol. The test cases are:

- *Registration*: the load generator sends REGISTER messages using digest authentication to the SUT. The SUT should already have the user accounts established. The delay from sending the initial REGISTER message to receiving the final 200 response is measured. (There is one unauthorized registration attempt after each authorized attempt.)
- *Outbound proxy*: The load generator sends INVITE requests to the SUT acting as an outbound proxy.
- *Redirect*: The load generator sends INVITE requests to the SUT acting as a redirect server. The delay from sending an INVITE request to receiving the 3xx response is measured.
- *Proxy 480*: The load generator sends INVITE requests to the SUT acting as a redirect server. In this case, the destinations have not registered and the server returns a 480 (temporarily unavailable) response.
- *Proxy 200*: The load generator simulates a call, with a receiver that responds, by sending an INVITE with a BYE immediately following when the INVITE completes. The delay of the complete INVITE request transaction is measured.

Each user-name should only be used once per test and thus the user population should be large enough. The benchmark determines calls per second and registrations per second values for the server by increasing the request rate until transaction failure probability increases to 5%. The measurement period is 15 minutes. Finally, a composite number called SIPstone-A can be computed from the measurement results using weights.

2.2 IMS/NGN Performance Benchmark

ETSI Technical Committee TISPAN¹ is working on IMS/NGN (IP Multimedia Subsystem/Next Generation Networks) Performance Benchmark. Quite mature drafts of the benchmark standard were published in January 2007. Part 1 [9] covers *Core Concepts*, Part 2 [10] *Subsystem Configurations and Benchmarks*, and Part

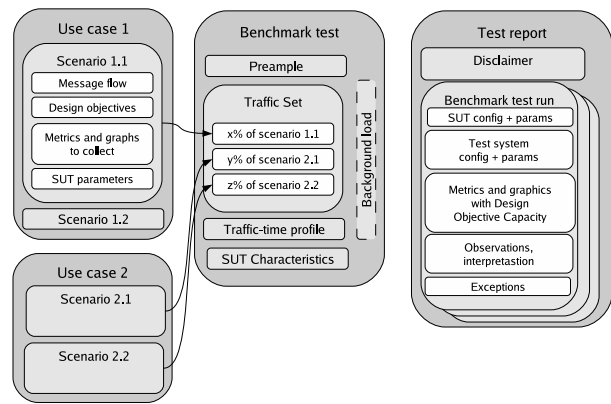


Figure 3: IMS Benchmark Information Model [ETSI TS 186 008-1]

3 [11] *Traffic Sets and Traffic Profiles*. The objective of the standard is to improve the data available for IMS or NGN deployment decision making since existing models—such as Erlang tables and “rule-of-thumb” values—cannot provide the data needed.

2.2.1 Benchmark Information Model

The benchmark information model, shown in Figure 3, takes the approach that any workload to a *System under Test* is derived from the behaviour of an individual user. When users interact with the system, they have particular goals, to make a call or to send a message, for example. The system may provide a variety of ways to accomplish the goals, but the number of common actions is small.

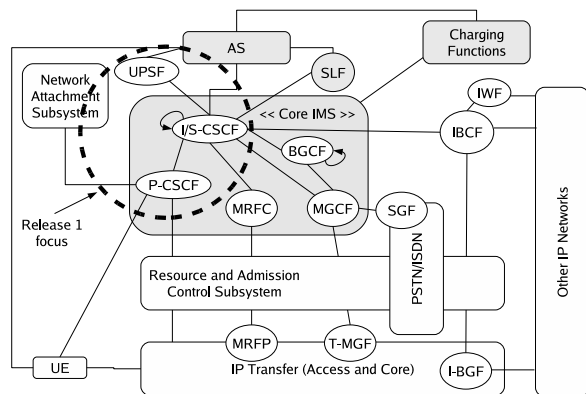
The information model consists of three primary elements:

1. *use-cases* that describe the behaviour of an individual user, which, in turn, defines *scenarios*;
2. *benchmark tests*, which generate a workload by aggregating the behaviour of individual scenarios in a controlled manner and collect logfiles of measurements done during the test; and
3. *benchmark test reports* that give metrics interpreted from the benchmark test logfiles.

2.2.2 Benchmark Test

A benchmark test defines:

- a *preamble*, the sequence of actions required to initialize a test system and the System under Test to perform a benchmark;
- a *traffic set*, a set of scenarios that simulated users perform during the test procedure together with rel-



- AS Application Server
- BGCF Border Gateway Control Function
- CSCF Call Session Control Function
- IBCF Interconnection Border Control Function
- I-BGF Interconnect-Border Gateway Function
- I-CSCF Interrogating CSCF
- IWF Inter-Working Function
- MGCF Media Gateway Control Function
- MRFC Media Resource Function Controller
- MRFP Media Resource Function Processor
- P-CSCF Proxy CSCF
- S-CSCF Serving CSCF
- SGF Serving Gateway Function
- SLF Subscriber Location Function
- T-MGF Trunk Media Gateway Function
- UE User Equipment
- UPSF User Profile Server Function

Figure 4: IMS Reference Architecture [ETSI ES 282 007]

ative frequencies of scenario occurrences during the test procedure;

- an *arrival distribution* describing the arrival rate of occurrences of scenarios from the traffic set; and
- the *traffic-time profile* that describes the evolution of the average arrival rate as a function of time over the duration of the test procedure.

Background load is a workload presented to the SUT in order to consume its resources. It may consist of a stream of traffic presented to the SUT by an external system apart from the test system. It may also be a workload presented to the processing elements, network, or storage subsystem of the System under Test.

2.2.3 System under Test

Figure 4 shows the IMS Reference Architecture [8]. The components of the architecture are the primary building blocks that are defined either by the IMS standard or by external standards and referenced by IMS. The links between the primary building blocks represent reference

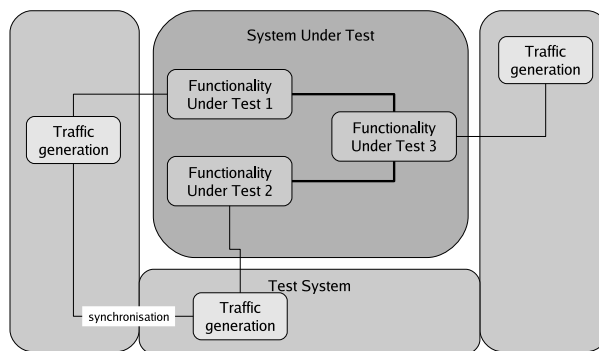


Figure 5: Interactions between Test System and System under Test [ETSI TS 186 008-1]

points over which the building blocks communicate with each other. The reference architecture is a logical architecture in the sense that no mapping of functional elements to hardware or software component is mandated.

In Release 1 of the benchmark standard the focus is on the Session Control Subsystem (SCS) that includes User Profile Server Function (UPSF) Subsystem of TISPAN NGN or Home Subscriber Server (HSS) of 3GPP IMS, Proxy Call Session Control Function (P-CSCF), and Interrogating/Serving Call Session Control Function (I/S-CSCF).

In order to proceed from a subsystem description to a benchmark test, a complete description of all aspects of the subsystem relevant to the performance of the benchmark must be present. This description is referred to as the *SUT configuration*. In the description all elements of the reference architecture and all reference points external to the subsystem are enumerated. The configuration requires a specification of hardware elements (servers, CPUs, network configuration and bandwidth, etc.) and software elements (operating system, database system, etc.).

2.2.4 Test System

The test system is used to generate the appropriate load on the System under Test. The standard does not mandate any specific test system to be used, but requires that the details of the test system must be reported in the benchmark report.

Figure 5 depicts test system connections and interactions with an SUT. The main functions of the test system are:

- *Traffic generation*: the test system must be able to execute scenarios of use-cases following the traffic-time profile. It must also be able to reproduce the appropriate traffic set, that is a mix of scenarios with a weight for each of them.

- *Network emulation*: optionally, network characteristics on the different interfaces must be emulated by the test system. Those characteristics are to be set separately for each direction so that non-symmetric interfaces can be emulated.
- *Synchronisation*: in the case where protocol information elements must be passed between an SUT interface and another and the test system is different for the interfaces, a synchronisation mechanism must exist to pass those information elements between the test systems.

2.2.5 Benchmark Metrics

The metrics reported by a benchmark test may be measured in real time during the execution of the test. An alternative is to compute them after completion of the test from event logs collected during the execution.

The metrics specified in Part 1 [9] of the standard are:

SAPS: (*Scenario Attempts Per Second*) The average rate in one second period at which scenarios are attempted (not necessarily successful).

TRT: (*Transaction Response Time*) Defined as the time elapsed from the first message sent to initiate the transaction until the message ending the transaction is received. Part 2 [10] defines exact points in the message sequence chart for each scenario. The maximum response times of adequately handled scenarios are also specified. In some scenarios there are separate thresholds for different segments in the scenario.

CPU: (*CPU usage ratio*) The ratio of used CPU to the total CPU available. This includes all CPUs available for the processing.

MEM: (*Memory usage ratio*) The ratio of used memory to the total memory available.

RETR: (*Retransmission Rate*) Applies to UDP transport. The retransmission rate is the ratio between the number of retransmissions and the number of messages sent.

SIMS: (*Simultaneous Scenarios*) Number of scenarios that are active at the same time.

%IHS: (*Percent Inadequately Handled Scenarios*) The ratio of inadequately handled scenarios to the total number of attempted scenarios. *Design Objective Capacity* (DOC) defines maximum total round-trip time (TRT) of the scenario. Under nominal load %IHS must be less than 1% and under stress condition (overload) less than 10%.

2.2.6 Use Cases

Part 2 [10] defines three use-cases: Registration/De-registration (9 scenarios), Session Set-Up/Tear-Down (25 scenarios), and Page-mode Messaging (2 scenarios).

Registration/De-registration Use-Case. Registration is the first use-case that must be employed when using an IMS network. During this operation the UE announces its contact location to the home domain registrar in order for the home network to route terminating messages towards it. Registration is performed by an UE when the device is turned on. De-registration is the last operation that an UE performs before it is turned off. De-registration is used to invalidate the registered contact information.

Because of security concerns, this operation has to be authenticated. The assigned S-CSCF challenges the UE using authentication vectors obtained from the HSS or UPSF.

Scenarios:

1. Successful Initial Registration without Synchronization
2. Successful Initial Registration with Synchronization
3. Re-Registration – User Currently Registered
4. Re-Subscription – User Currently Registered
5. Re-Registration – User Roaming
6. UE Initiated De-Registration
7. Network Initiated De-Registration
8. Network Initiated De-Registration upon Roaming or Expiration
9. Network Initiated Re-Authentication

Session Set-Up/Tear-Down Use-Case. This use-case corresponds to a normal two-party call, in which a multi-media communication session set-up is attempted between two users, of which at least one is an IMS user. During the set-up period a “ringing” delay is introduced. If the set-up is successful then the scenario is put on hold for the duration of the call and then it is terminated with a tear-down sequence.

This use case has 25 scenarios. They cover four types of calls: successful, abandoned (B party does not answer in time), rejected (B party immediately rejects the invitation), and failed call. There are eight scenarios for the first three call types:

- Four combinations of resource reservation (yes or no) when both sides are IMS users.

- Two combinations of resource reservation (yes or no) on originating side when the terminating side is non-IMS.
- Two combinations of resource reservation (yes or no) on terminating side when the originating side is non-IMS.

Page-mode Messaging Use-Case. Page-mode messaging, defined in 3GPP 24.247 and ETSI TS 183 041, is the simple use-case when a message is exchanged between two peers. This kind of communication is preferred when a small number of messages is exchanged between two peers. A normal call set-up and tear-down takes a minimum of five SIP messages and typically seven. If the application does not require relating between messages at the protocol level, simple messaging can be more efficient, since it employs just two SIP messages.

Scenarios:

1. Successful Message Exchange
2. Unsuccessful Message Exchange – Called User Not Found

2.2.7 Traffic Set

The Part 3 [11] defines the following initial benchmark traffic set.

- Scenario 1.1 (Successful Initial Registration without Synchronization) 1% of system load, Poisson arrival with mean selected by traffic-time profile
- Scenario 1.3 (Re-Registration – User Currently Registered) 1% of system load, Poisson arrival with mean selected by traffic-time profile
- Scenario 2.1 (Successful Call – Resource reservation on both sides) 12% of system load, Poisson arrival with mean selected by traffic-time profile, call hold time exponentially distributed with mean 120 sec
- Scenario 2.2 (Successful Call – No resource reservation on originating side) 12% of system load, Poisson arrival with mean selected by traffic-time profile, call hold time exponentially distributed with mean 120 sec
- Scenario 2.3 (Successful Call – No resource reservation on terminating side) 12% of system load, Poisson arrival with mean selected by traffic-time profile, call hold time exponentially distributed with mean 120 sec

- Scenario 2.4 (Successful Call – No resource reservation on either side) 12% of system load, Poisson arrival with mean selected by traffic-time profile, call hold time exponentially distributed with mean 120 sec
- Scenario 2.9 (Abandoned Call – Resource reservation on both sides) 3% of system load, Poisson arrival with mean selected by traffic-time profile, wait time exponentially distributed with mean 15 sec
- Scenario 2.10 (Abandoned Call – No resource reservation on originating side) 3% of system load, Poisson arrival with mean selected by traffic-time profile, wait time exponentially distributed with mean 15 sec
- Scenario 2.11 (Abandoned Call – No resource reservation on terminating side) 3% of system load, Poisson arrival with mean selected by traffic-time profile, wait time exponentially distributed with mean 15 sec
- Scenario 2.12 (Abandoned Call – No resource reservation on either side) 3% of system load, Poisson arrival with mean selected by traffic-time profile, wait time exponentially distributed with mean 15 sec
- Scenario 2.17 (Rejected Call – Resource reservation on both sides) 3% of system load
- Scenario 2.18 (Rejected Call – No resource reservation on originating side) 3% of system load
- Scenario 2.19 (Rejected Call – No resource reservation on terminating side) 3% of system load
- Scenario 2.20 (Rejected Call – No resource reservation on either side) 3% of system load
- Scenario 2.25 (Failed Call) 1% of system load
- Scenario 3.1 (Successful Message Exchange) 19% of system load, Poisson arrival with mean selected by traffic-time profile, message size (chars) uniformly distributed in [0 . . . 140]
- Scenario 3.2 (Unsuccessful Message Exchange – Called User Not Found) 5% of system load, Poisson arrival with mean selected by traffic-time profile, message size (chars) uniformly distributed in [0 . . . 140]

2.2.8 Traffic-time Profile

The Part 3 [11] defines the following initial benchmark traffic-time profile.

SystemLoad: Design Objective Capacity (DOC) given as Scenario Attempts Per Second (SAPS)

SimultaneousScenarios: Maximum 2 per UE

TotalProvisionedSubscribers: 100 000, XML Schema of subscriber data defined in Part 2

PercentRegisteredSubscribers: 40% in the beginning of a test

PercentRoamingSubscribers: None; no roaming in Release 1 of the benchmark

StepNumber: 3 steps: DOC underload, DOC, DOC overload

StepTransientTime: Maximum 120 seconds (warm-up period)

StepTime: Minimum 30 minutes

BackgroundLoad: None

SAPIncreaseAmount: Maximum 10; Reported results: step before, DOC, step after

MaximumInAdequatelyHandledScenarioAttempts: 0.1% (average over a test step)

2.3 CommBench

In the paper entitled “CommBench—a telecommunications benchmark for network processors,” Wolf and Franklin [45] present a benchmark suitable for the evaluation and design of network processors. The benchmark applications have been selected to represent typical workloads both for traditional routers, where the focus is on header processing, and active routers, which perform both header and payload processing. The SPEC integer benchmark [13], commonly used to measure processor performance, is used as a contrast to highlight the need for a specialized benchmark for the telecommunications domain.²

The header processing kernels consist of *lookup operations on tree data structure*, based on a radix-tree; *packet header modification and checksum computation*, based on an application called FRAG; *queue maintenance and packet scheduling for fair resource utilization*, based on deficit round-robin fair scheduling algorithm; and *pattern matching on header data fields*, based on the tcpdump application. The payload processing kernels consist of *encryption arithmetic*, based on the CAST-128 block cipher algorithm; *data compression*, based on the Lempel-Ziv algorithm as implemented in ZIP; redundancy coding, using the Reed-Solomon FEC; and *DCT and Huffman coding*, based on JPEG code. All of the

code used comes from freely available public domain programs.

Complexity of CommBench kernels is measured in instructions per byte. CommBench is then compared to SPEC with regard to static code size, code execution coverage, instruction set characteristics, and memory hierarchy characteristics. The static size of CommBench code is found to be smaller, and the ratio of code execution coverage to the static size even smaller when compared to SPEC. This has significant effect on what kind of memory and cache system should be selected for telecommunications tasks.

The header processing category of benchmarks is found to differ significantly from SPEC in terms of instruction mix, SPEC having less immediate loads and compares. Also, the header and payload processing differ from each other with regard to instruction mix.

The memory hierarchy characteristics differ most from SPEC, again, in the header processing kernel. Both CommBench kernels have less instruction cache misses, because of their small size. Data cache performance is found to be similar between SPEC and payload processing, but header processing has much less data cache misses.

Finally, the CommBench paper presents examples of using the gathered benchmark data in network processor design. The instruction per byte measure gives an estimate of required processor speed for a given link speed. The instruction mix results suggest possible optimizations in the processor instruction set, and the memory hierarchy data together with the instruction mix data may be used to calculate required memory bandwidth.

2.4 NpBench

Noting that not much has been written on control plane network processor benchmarking, Lee and John [17] present a set of benchmarks, called *NpBench*, representing both control plane and user (data) plane workloads. In the paper, network processor applications are divided into three groups: traffic-management and quality of service (QoS) group, security and media processing group, and packet processing group. Each application can be further categorized into control plane or user plane, or both.

The applications in the traffic-management and QoS group consist of the WFQ algorithm, the RED algorithm, SSL dispatcher, and multi-protocol label switching (MPLS). The security and media group has media transcoding, AES, MD5, and Diffie-Hellman key exchange as applications. The applications in the packet processing group are packet fragmenting (FRAG) and CRC calculation.

Similar to the CommBench paper, metrics are gath-

ered using the benchmark applications and results are compared to CommBench. With regard to instruction mix, control plane applications are found to use much more branch operations than user plane applications, whereas user plane is found to use much more load and store operations. Cache behaviour is found to be similar to CommBench, noting the especially poor data-cache performance of control plane applications for small cache sizes. Potential for instruction level parallelism is found more in control plane applications. Control plane applications are also much more complex, comparing the 180 to 32,000 instructions per packet of control plane to user plane's 330 to 440 instructions per packet.

2.5 NetBench

Network processor applications contain a large variety of tasks from traditional routing and switching tasks to much more complicated applications containing intelligent routing and switching decisions. Therefore, any benchmarking suite attempting to represent the applications on network processors should consider all levels of a networking application. Instead of using the traditional 7-level OSI model for categorizing the applications, a three-level categorization is used:

- Low- or micro-level routines containing operations nearest to the link or operations that are part of more complex tasks;
- Routing-level applications, which are similar to traditional IP level routing and related tasks; and
- Application-level programs, which have to parse the packet header and sometimes a portion of the payload and make intelligent decisions about the destination of the packet.

This categorization is performed by considering the complexity of the application, instead of the specific task it is performing.

1. Micro-Level Programs

CRC: The CRC-32 checksum calculates a checksum based on a cyclic redundancy check as described in ISO 3309.

TL: TL is the table lookup routine common to all routing processes. Radix-tree routing table, which is used in several UNIX systems, is used.

2. Routing-Level Programs

These programs make a decision depending on the source or destination IP address of the packet.

ROUTE: Route implements IPv4 routing according to RFC 2644.

DRR: Deficit-round robin (DRR) scheduling [32] is a scheduling method implemented in several switches today.

IPCHAINS: IPCHAINS is a firewall application that checks the IP source of each of the incoming packet and decides either to pass, to deny, to modify, or to reject the packet.

NAT: Network Address Translation (NAT) is a common method for IP address management.

3. Application-Level Programs

These programs are the most time-consuming applications in NetBench because of their processing requirements.

DH: Diffie-Hellman (DH) is a common public key encryption/decryption mechanism. It is the security protocol employed in several Virtual Private Networks (VPNs).

MD5: Message Digest algorithm (MD5) creates a signature for each outgoing packet, which is checked at the destination.

SNORT: Snort is an open-source network intrusion detection system (<http://www.snort.org/>), which is capable of performing real-time traffic analysis and packet logging on IP networks.

SSL: SSL or Secure Sockets Layer is the secure transmission package.

URL: URL implements URL-based destination switching, which is a commonly used content-based load balancing mechanism.

2.6 Network Processing Forum

The Network Processing Forum (NPF)—merged in 2007 to the Optical Internetworking Forum (OIF)—has produced standards and benchmarks for network processors. The benchmarks are presented as implementation agreements [25] that contain functional specifications of test configuration, variables, and metrics for the benchmarks.

The benchmarks are divided into task, application, and system levels. The task-level consists of tasks common to many networking applications, such as Longest Prefix Match table lookups, five-tuple table lookups, and string searches. The application-level benchmarks are intended to characterize the performance of well-defined application functions, such as IP forwarding, MPLS switching, and NAT. At the system-level, the performance of the

whole system is benchmarked in typical network processor application domains, such as firewalls and multi-service switches.

The NPF benchmarking work is continued in the Benchmarking Working Group of OIF. NPF benchmarking implementation agreements specify precisely how vendors and independent parties should measure the performance of subsystems based on network processors. These specifications will provide customers and vendors with objective performance data needed to properly compare and evaluate network processing components.

Network processing manufacturers that wish to certify the performance results of their benchmark tests must submit their products to a third party independent auditor and certification authority. Once testing is completed, the NPF provides the “NPF Certified” mark to the manufacturer validating that the benchmark results are in complete compliance with NPF benchmark specifications.

The NPF benchmarking implementation agreements include:

- TCP Proxy Application Level Benchmark (March 2006)
- IPsec Forwarding Application Level Benchmark (July 2004)
- Switch Fabric Benchmark (July 2003)
- IP Forwarding Benchmark (June 2003)
- MPLS Application Level Benchmark (January 2003)
- IPv4 Forwarding Benchmark (July 2002)

The **TCP Proxy Application Level Benchmark** specifies performance metrics and testing methodologies for the measurement of TCP Proxy applications on a network processor subsystem. TCP Proxies fully terminate TCP connections enabling full data scanning and modification. TCP Proxies serve as the foundation for applications such as L7 firewalls, load balancers, SSL accelerators and Intrusion Detection Systems. The benchmark defines standard testing procedures for TCP Proxy Goodput on existing connections, Goodput with connection setup/teardown, connection setup rate, connection setup and teardown rate, SYN/ACK latency, and connection setup latency. Standard workloads are defined including TCP object sizes, number of concurrent connections, traffic patterns and round trip times. Tester performance calibration is handled through the specification of both DUT and loopback tests.

The **IPsec Forwarding Application Level Benchmark** enables the objective measurement and reporting of the IPsec performance of any Network Processing Unit (NPU)-based device or set of components under

test. The benchmark provides both vendors and System OEMs with a quantitative analysis of how the system/part(s) will perform providing IPsec-based functions. It includes specific instructions for set-up and configuration, tests and test parameters and result reporting formats. The specification supports a number of different configurations including systems with multiple NPU's, PC Board's and/or daughter cards, and a backplane or switch fabric. Whichever configuration is used, all of the systems components must be documented. The benchmark includes three specific tests, each having its own procedure, test setup, frame sizes and reporting format. The three tests measure the IPsec forwarding rate, IPsec throughput and the IPsec latency.

The **Switch Fabric Benchmark** [7] is a compilation of five fabric Implementation Agreements: Switch Fabric Benchmarking Framework, Fabric Traffic Models, Fabric Performance Metrics, Performance Testing Methodology for Fabric Benchmarking, and Switch Fabric Benchmark Test Suites. Together, these agreements provide all of the background methodology and testing parameters needed for vendor independent switch fabric performance measurement. The tests are divided into three suites including hardware benchmarks, arbitration benchmarks and multicast benchmarks. Each suite includes multiple tests with their own test objective, arrival pattern, test procedure, and result presentation instructions. The three main performance metrics include latency, accepted vs. offered bandwidth and jitter. This benchmark will enable system design engineers to assess and compare different switch fabrics in an open and objective manner.

The **IP Forwarding Benchmark** [4] extends the scope of the IPv4 Forwarding Benchmark to include the IPv6 protocol and new IPv4 and IPv6 routing tables. This specification provides industry standard measures of the forwarding performance of network processing systems with native IPv4, native IPv6 and mixed IPv4/IPv6 traffic. IPv4 routing tables featuring 10k, 120k and 1M routes are included as are IPv6 routing tables of 400 and 1.2k routes. The implementation agreement details the terminology, test configurations, benchmark tests, routing tables and reporting formats needed to measure and publish the forwarding performance of network processing based systems.

The tests are grouped into three categories: user (data) plane tests, control plane tests, and concurrent user (data) plane and control plane tests. The user plane tests include measures of the aggregate forwarding rate, throughput, latency, loss ratio, overload forwarding rate, and system power consumption. Different traffic combinations are used including 100% native IPv4, 50% IPv4/50% IPv6, and 100% IPv6. The control plane tests include measures of forwarding table update rates. The concurrent

user plane and control plane tests include measures of concurrent forwarding table updates on the forwarding rate.

The **MPLS Application Level Benchmark** defines a methodology to obtain network processor MPLS application level benchmarks and describes the tests used to obtain MPLS performance metrics in Ingress, Egress and Transit configurations. It includes an Annex that describes a reference implementation of the benchmark, outlines the traffic streams required to run the benchmark tests, and provides the references and descriptions associated with the benchmark routing tables. An associated MPLS reporting template presents a sample report. The implementation agreement establishes consistent and objective measurement criteria that accurately assess the MPLS performance of Network processing products.

The **IPv4 Forwarding Benchmark** details the interfaces, configuration parameters, test set-up and execution details, including traffic mix and routing table contents, needed to measure the IPv4 forwarding performance of NPU-based systems. The IPv4 interface takes the methodology defined for network equipment by the IETF (RFC2544) and adapts it in a new framework that targets the NPU subsystem. Using a consistent methodology, it enables easy comparison between NP-based systems with widely varying architectures. Design engineers will now be able to assess and compare, in an open, objective, and reproducible way, the IPv4 forwarding performance of NP-based devices.

3 Benchmark Specification

We have designed a flexible benchmark for control plane applications to study their performance in telecom environments. Figure 2 depicts our benchmarking framework. Our main design objective was to specify a benchmark having a modular structure so that various subsystems are loosely coupled and easily exchangeable. The specification defines two performance tests: *Telephony Service* and *Echo*. The Telephony Service test is the actual control plane benchmark, whereas the Echo test measures latencies of the signalling transport layer.

The TPC benchmarks (see Section 4.2) was our driving example in the sense that we specified the benchmark as a set of numbered clauses. In addition, we applied the reporting requirements of TPC benchmarks.

We present a new application-level benchmark specification for measuring the performance of different transport, operating system, database, and hardware configurations for control plane applications. In contrast to SIPstone [31], which measures different SIP proxy server implementations and treats the server as a black box, our benchmark has a fixed server core implementation, but requires user to provide implementations of the other

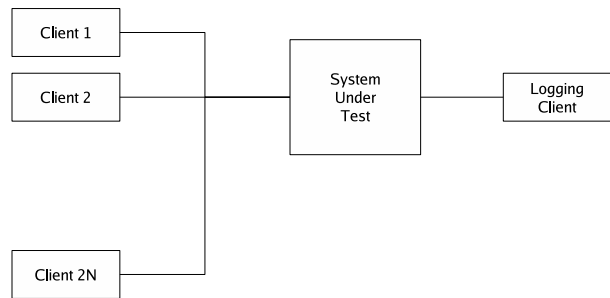


Figure 6: Control Plane Benchmark Architecture

core components of the server, most importantly the signalling transport layer and the user location database. This allows us to change only one component at a time and see the impact it has on total system performance in typical control plane usage. Below we describe the specification; more details can be found in [20].

3.1 General system architecture and requirements

The control plane benchmark architecture is shown in Figure 6. The internal organization of the SUT is restricted in Section 3.1.1.

2.0.0.1. Externally the SUT must connect to the driver using a physically separate network for each driver node.

2.0.0.2. If the logger is not run on the SUT, the SUT must connect to it using a network physically separate from any networks connecting to driver nodes.

2.0.0.3. The clocks in the nodes must be synchronized within 5 ms.

3.1.1 Requirements for individual components

The control plane benchmark measures the following components of the system: the signalling layer, the user location register, and call state tracking facility.

Signalling layer

2.1.1.1. The signalling layer is implemented in the SUT and in the driver.

Note: It is possible that the implementation of the signalling layer in the driver skews the results. This is mitigated by performing a simple request-response test whose results are used to define upper limits for SUT latencies.

2.1.1.2. The signalling layer must be able to perform the Echo test defined in Section 3.2.2.

2.1.1.3. The SUT and driver must be able to perform the call scenario presented in Section 3.2.1. All the functions must be implemented.

2.1.1.4. Optionally, a single benchmark operation may be implemented as several operations and messages between the driver and the SUT.

2.1.1.5. If a protocol is connection-oriented, a set of operations belonging to the same transaction may share the connection. Operations belonging to distinct transactions must not share a connection (for example, two distinct connect events in the driver may not use the same TCP connection).

Call state tracking

2.1.2.1. The call state tracking must be implemented to the level of detail that would allow time-based accounting in a real application.

2.1.2.2. The granularity at which the call connect and disconnect events are recorded must be equal to or shorter than 10 milliseconds.

User location register

2.1.3.1. The user location register must contain the following fields on every single user in the system: ID, ADDRESS.

2.1.3.2. The ID field must be unique within the system.

2.1.3.3. Optionally, the call state tracking may be implemented within the user location register.

Load generator

2.1.4.1. The generated background activity should have the same priority as the telephony server, and it should stress the processor, memory, and secondary storage resources.

2.1.4.2. The load generator load must be configurable by specifying how often and for how long per round it runs.

Logging

2.1.5.1. The telephony server must log all actions specified in Section 3.2.1.

3.2 Test Profiles

3.2.1 Telephony Service Test

The *Telephony Service* test simulates the usual telephone call scenario. The basic call sequence diagram is shown in Figure 7.

3.1.1.1. A test is carried out by performing the following steps in the specified order. Here R_i is the call rate at step i as specified in Clause 4.1.3.3 and N_i must be large enough to accommodate a test at the rate of R_i for the duration of 15 minutes without exhausting the population

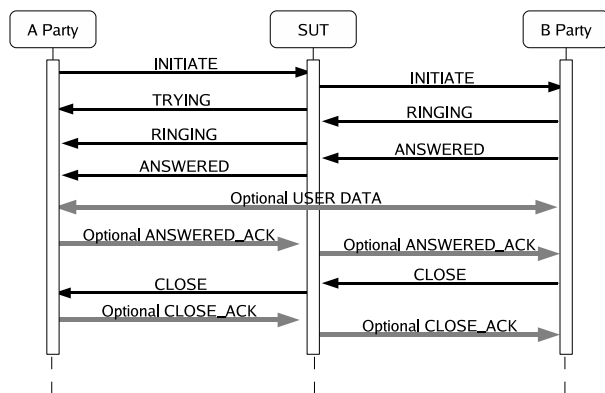


Figure 7: Basic Call Message Sequence Chart

(i.e. $900 * R_i < N_i$ when R_i is given in calls per second and 900 corresponds to 15 minutes).

1. Initialize the SUT (including the load generator)
2. Initialize the driver
3. Optionally, the driver primes the SUT by performing N'_i calls, where $N'_i = 0.01 * N_i$.
4. The driver waits until all priming calls are finished and there are no outstanding requests
5. The driver performs calls at rate R_i for at least 15 minutes
6. Report test results

3.1.1.2. The user location register in the SUT is populated with N_i user records at initialization. The ID fields of the records are of the form USERNAME. n , where n is a sequential number from 1 to N_i .

3.1.1.3. The user A chooses the recipient user B by selecting one of the USERNAME. n , where n is uniformly distributed random variable in the range $[1, N_i]$, excluding the value of the ID field of user A.

3.1.1.4. The user B answers x seconds after receiving the INITIATE message, where x is exponentially distributed with mean 7 seconds ($\lambda = 1/7$).

3.1.1.5. Optionally, users A and B may exchange data on the established connection, if that data is not transmitted through the SUT.

3.1.1.6. User A closes the connection y seconds after the ANSWERED message, where y is exponentially distributed with mean 30 seconds ($\lambda = 1/30$).

3.1.1.7. After receiving the first RINGING message, the A process will wait for ANSWERED message for 45 seconds. If the message is not received, A should report the failure.

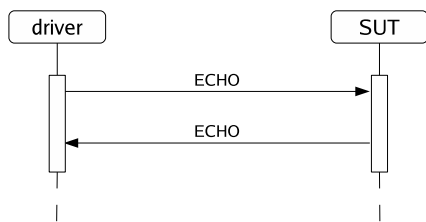


Figure 8: Echo Test Message Sequence Chart

3.2.2 Echo Test

The *Echo* test measures the performance of the signalling transport layer. It is meant as an auxiliary test case for isolating the effects of the signalling transport layer and layers below it.

The Echo test is performed between the driver and the SUT. The sequence diagram is shown in Figure 8.

3.2.1.1. The test is carried out by performing the following steps in order:

1. Initialize the SUT
2. Initialize the driver
3. Transmit 1 000 ECHO messages from the driver to the SUT
4. Measure response times of the received ECHO messages
5. Report test results

3.2.1.2. The ECHO messages are to be sent at the rate of one (1) message per second.

3.3 Performance metrics

3.3.1 Measurements

Driver

4.1.1.1. The driver should measure the time from sending the INITIATE request to the first server reply, which could be TRYING, RINGING or ANSWERED.

4.1.1.2. The driver should measure the time from the B party sending RINGING, ANSWERED or CLOSE to the A party receiving the message for each of the messages.

System under Test

4.1.2.1. The SUT should measure the time from starting a lookup to getting a successful lookup results.

4.1.2.2. The SUT should measure the number of failed lookup attempts.

4.1.2.3. The SUT should measure the number of answered calls.

4.1.2.4. The SUT should measure the number of failed calls.

4.1.2.5. The SUT should measure the number of Calls Serviced Late. A call that is serviced after one second has elapsed after it is received by the SUT is also considered failed.

4.1.2.6. The SUT should measure the length of every call.

4.1.2.7. The SUT should measure the CPU load every second.

4.1.2.8. The SUT should measure the memory usage every second.

Methodology

The methodology of measuring the maximum call throughput of the server is described by the following clauses. The methodology only applies to the test described in Section 3.2.1.

4.1.3.1. Every test run is separate and done according to the clauses in Section 3.2.1.

4.1.3.2. Every test run should run, at least, 15 minutes.

4.1.3.3. Starting from a call rate that the server can easily handle, the call rate is increased in each successive test run until a test run with 10% call failure rate is reached.

4.1.3.4. The maximum call rate the SUT can handle is derived by first plotting the successful and failed call attempts per second of all the test runs and then taking the maximum successful call rate from where the failure rate is below 10^{-4} .

4.1.3.5. *Busy Hour Call Attempts* (BHCA) is the number of attempted calls during a busy hour of the day. The BHCA value is calculated from the maximum call rate as specified in 4.1.3.4 by multiplying it by 3 600.

4.1.3.6. Whole tests, as described by the previous Clauses (4.1.3.3, 4.1.3.4, 4.1.3.5), should be run with background loads of 0% and 50% and a random load that varies between 0-50% and changes once per minute.

3.3.2 Calculated metrics

Transaction metrics

4.2.1.1. The maximum, average, and 90th percentile of the amount of calls per second should be reported.

4.2.1.2. The 10th, 25th, 50th, 75th, and 90th percentiles of the realised call lengths should be reported.

4.2.1.3. The variation of call processing times should be reported.

4.2.1.4. The amount of *Calls Serviced Late* in the last test run should be reported.

4.2.1.5. The *Busy Hour Call Attempts* as specified in Clause 4.1.3.5 should be reported.

System metrics

4.2.2.1. The maximum and average CPU load should be reported. The CPU load at the beginning and at the end of the benchmark should be reported.

4.2.2.2. The maximum and average memory usage should be reported. The memory usage at the beginning and at the end of the benchmark should be reported.

3.4 Benchmark Environment

3.4.1 Hardware Requirements

5.1.1.1. It is required that the SUT has enough memory; the benchmark result must not be affected by the slow-down caused by paging memory to and from secondary storage.

5.1.1.2. It is required that the connection between the SUT and the driver will provide enough bandwidth to not become a bottleneck in the system. With systems of today this requirement translates to a network connection of at least 100 Mbps.

3.4.2 Software Components

Signalling Layer

The requests, which the clients (user agents) make to the server, and the corresponding answers are carried by the signalling transfer layer. For example, the signalling transport layer could be an implementation of the SIP protocol. The minimum requirements for the signalling layer are specified in Clauses 2.1.1.1, 2.1.1.2, and 2.1.1.3.

The Driver

The driver generates the client-side test load for benchmark. It is composed of one or several applications that perform calls using the signalling layer. The driver applications can reside on one or several nodes.

Telephony Server

The main component of the SUT is the telephony server, which handles the incoming call requests made by the user agents simulated by the driver. The server uses the signalling transport layer to accept requests. It has a local or external database connectivity for performing lookups from the user location register (and possibly for recording state information.)

Load Generator

The load generator is part of the SUT. It is used to generate background load to the server, simulating activities

on the server that are not directly related to the control plane activity being benchmarked. This will provide a more realistic usage scenario.

3.5 Benchmark Implementation

3.5.1 Driver

The driver emulates the user-agents that connect to the SUT. The driver can be implemented using whatever approach as long as the implementation adheres to the requirements that are listed below.

6.1.0.1. The driver implementation must implement actions of user-agents in all of the test profiles as specified in Sections 3.2.1 and 3.2.2. It must be able to emulate a user specified amount of user-agents connecting to the SUT. It must also be able to emulate the user-agents at the receiving end.

6.1.0.2. The driver implementation must use a separate connection for each call attempt so that connection setup and tear-down will tax the SUT as they would in a real system.

6.1.0.3. Throughput and latency measuring may be implemented in the driver or using a separate network analyzer or by traffic capture and post-capture analysis tools.

6.1.0.4. Latency measurements must have a precision of at least 10 ms.

6.1.0.5. Latency measurements must be implemented at the points specified by Clauses 4.1.1.1 and 4.1.1.2.

3.5.2 Telephony Server

The Telephony Server software is composed using the framework described subsequently. The framework consists of the APIs for the components and two data structures holding the call and connection state. The server implementation itself is described below.

Component overview

The components that make up the telephony server are the signalling transport layer, the user location register, and the server event processor. The signalling layer and user location register should be implemented by the user, but the server event processor implementation is defined below.

Messages and Events

The abstract signalling layer uses the messages and events listed in this section. Also listed are events to be used by the user location database.

6.2.2.1. The abstract signalling layer uses the messages listed below. Any unsupported optional message can

be handled as a null operation in the signalling transport layer implementation. Later on, `enum MESSAGE` is used to refer to these messages.

ECHO Request immediate echo from the server (when sent by client) or server reply to the echo request.

INITIATE Connection initiation request between parties.

TRYING Server is trying to contact to a party requested in a previous **INITIATE** message. This message is optional.

RINGING The user-agent is contacted and ringing.

ANSWERED The user-agent has answered the incoming call.

ANSWERED_ACK The user-agent acknowledged an **ANSWERED** message from the other party. This message is optional.

CLOSE Connection close request.

CLOSE_ACK The user-agent acknowledged a **CLOSE** message from the other party. This message is optional.

ERROR There was an error and connection will be closed.

6.2.2.2. If the **ANSWERED_ACK** and **CLOSE_ACK** messages are not supported by the signalling layer, it should, however, pass these messages as events to the server event processor as soon as it gets **ANSWERED** or **CLOSE** message, respectively.

6.2.2.3. Timeouts in signalling must be communicated using the following events. These events are part of the `enum EVENT` enumeration.

INITIATE_TIMEOUT Connection initiation request sent by the server has timed out.

ANSWERED_ACK_TIMEOUT The user-agent acknowledgment to an **ANSWERED** message has timed out. This event is optional.

CLOSE_ACK The user-agent acknowledgment to a **CLOSE** message has timed out. This event is optional.

6.2.2.4. The user location database component must use the following events to communicate lookup results. These events are part of the `enum EVENT` enumeration.

LOOKUP_DONE Lookup was done successfully.

LOOKUP_FAILED Lookup failed for some reason other than timeout.

LOOKUP_TIMEOUT Lookup timed out.

Data structures

The following two data structures are used to hold call and connection state in the server. Additionally, a simple mutex structure is defined.

6.2.3.1. All of the following three structures should be implemented by the user. The contents of the structures will depend on the signalling transport protocol and the user location register implementations.

struct mutex A mutual exclusion device included by both of the structures below for implementing a multithreaded server.

struct connection contains the connection and dialog state between the server and one of the call parties. This structure must contain a `struct mutex mutex` field, a `struct connection *next_connection` field, and a `struct call *call` field, all of which will be needed by the server event processor.

struct call contains the call state. A call is established when both parties are contacted. This structure must contain a `struct mutex mutex` field, and a `struct connection *first_connection` field, both of which will be needed by the server event processor. This structure is mainly intended for the event processor's use.

Mutual exclusion API

6.2.4.1. The following two functions are used for mutual exclusion to the data structures. If implementation is fully single-threaded, mutual exclusion can be provided by empty operations. Both of these functions should be implemented by the user.

acquire_mutex(struct mutex) Acquires mutex.

release_mutex(struct mutex) Releases mutex.

Signalling API

6.2.5.1. The following two functions are relevant to the signalling layer.

post_event(enum EVENT, struct connection *) posts an event to the server. This function should be used by the signalling layer component implementation to post incoming messages and timeout events to the server. The `connection` field is used to identify the dialog and store the dialog state.

send_message(enum MESSAGE, struct connection *) sends a message using the signalling layer. This

function should be implemented by the signalling layer and it is called by the server to send messages. The connection field is used to identify the dialog and store the dialog state.

create connection(struct connection *) creates a new connection. This function should be implemented by the signalling layer and it is called by the server to create new connections. The connection field is used to identify the dialog and store the dialog state.

User location register API

6.2.6.1. The following two functions are relevant to the user location register.

post_event(enum EVENT, struct connection *) posts an event to the server. This function should be used by the user location register component implementation to post lookup results and timeout events to the server. The connection field is used to identify the dialog and store the dialog state.

lookup(struct connection *) looks up the receiver of the call based on information in the connection structure and fill the connection information from the database to the structure. This function should be implemented by the user location register component. The connection field is also used to identify the dialog and store the dialog state.

Logger API

6.2.7.1. The server logging mechanism is implemented using the following function call.

log_action(time_t timestamp, char *message) logs an event using the server logging mechanism. Timestamp is the time the event happened.

Event processor

State-machine for the server event processor is shown in Figure 9.

6.2.8.1. The event processing will be done as defined by the handle_event function given in appendix. Although not explicit, allocations should be freed at any event that tears down the connection. Also, mutexes needed solely by logging statements are not explicitly marked.

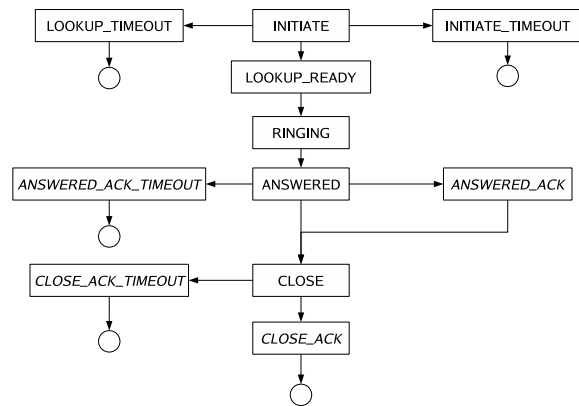


Figure 9: Telephony Server State Machine

3.6 Reporting

3.6.1 Report Format

The report has an executive summary on the front page, followed by the report details and, optionally, an appendix.

Executive Summary

7.1.1.1. The report must start with an executive summary consisting of the *Busy Hour Call Attempts* and *Calls Serviced Late* figures, as specified in Clauses 4.2.1.5 and 4.2.1.4.

Report Details

7.1.2.1. The network and node organization must be reported. For each node: number and type of processors, memory, and disk must be reported. For each network connection: endpoint interface devices, media type, and maximum bandwidth must be reported.

7.1.2.2. All metrics calculated from the benchmark run as specified in Section 3.3 must be reported.

7.1.2.3. The driver implementation must be disclosed in detail.

7.1.2.4. The driver measurement system implementation must be disclosed in detail.

7.1.2.5. The implementation of the API functions as specified in Clauses 6.2.4.1, 6.2.5.1, 6.2.6.1, and 6.2.7.1 must be disclosed.

7.1.2.6. Implementation specific parameters and settings, such as number of threads and memory settings, must be disclosed.

7.1.2.7. Software components used in the system must be reported. This is limited to software used in deployment (including compilation of the benchmark) and execution of the test.

7.1.2.8. Additional information about the system configuration or benchmark details may be added in an appendix at the end of the report.

4 Additional Benchmarks

As Figure 1 depicts, a modern telecommunication system is quite complex. Therefore, additional benchmarks to the Telephony Service test and Echo test are needed to get insight to performance of control plane applications. Such benchmarks include *lmbench*, *TPC-C*, and *Open Source Telecommunications Database Benchmark*. The role of XML is increasing also in control plane applications. In this section we briefly summarize a few suitable auxiliary benchmarks.

4.1 lmbench

In the Unix world the *lmbench* [23, 34] is the best known operating system benchmark. It is a collection of microbenchmarks that measures processor speed and various operating system functionalities. The first version of *lmbench* was criticised in [3]. As a reaction an obvious error was corrected.

The *lmbench* test suite is self-adjusting. It runs replications until it believes that the results are stable. In addition, the summary report contains warnings if the test package believes that the results are not reliable. As practical arrangements, the benchmark SHOULD be run from command line, mouse and network cable unplugged. In particular, X-server MUST not be running.

The *lmbench* collection of microbenchmarks is wide. The tests can be divided into six groups:

processor: integer operations, floating point operations (both single and double precision), memory latencies

OS-1 processes: null call, null I/O, stat, open/close, slct TCP, sig inst, sign hndl, fork proc, exec proc, sh proc

OS-2 context switch: 2p/0K, 2p/16K, 2p/64K, 8p/16K, 8p/64K, 16p/16K, 16p/64K

OS-3 local communication latencies: 2p/0K ctxsw, pipe, AF Unix, UDP, RPC/UDP, TCP, RPC/TCP, TCP conn

OS-4 file and virtual memory latencies: 0K create, 0K delete, 10K create, 10K delete, mmap latency, prot fault, page fault, 100fd select

OS-5 local communication bandwidths: pipe, AF Unix, TCP, File reread, Mmap reread, Bcopy (libc), Bcopy (hand), Mem read, Mem write

The results provided by *lmbench* are very useful in gaining insight to performance of a system.

Warning: Benchmarking textbooks and literature warns of using out-dated and inadequate benchmarks, such as *Dhrystone* [42, 43]. Nevertheless, many processor manufacturers are still quite widely using *Dhrystone*. First of all, *Dhrystone* is too small: it fits most L1-caches of today. A good analysis of *Dhrystone*'s inadequacy is in the white paper by Alan R. Weiss [44].

4.2 Database Benchmarks

The best known database benchmark is the on-line transaction processing (OLTP) benchmark by Transaction Processing Performance Council (TPC) [40]. The current version is called *TPC-C*, which is the successor of *TPC-A* and *TPC-B*. Three other database benchmarks are specified by TPC:

- Decision-support benchmark called *TPC-H* that replaced the *TPC-D* and *TPC-R* benchmarks.
- Application server and Web Services benchmark called *TPC-App* that replaced the *TPC-W* benchmark.
- *TPC-E* is a new OLTP workload that uses a database to model a brokerage firm with customers who generate transactions related to trades, account inquiries, and market research.

The TPC benchmarks specify a realistic database schema and operations that resemble those typically executed in business scenarios. The benchmarks also measure the cost to performance ratio by taking into account product, hardware and maintenance costs.

TPC-C [39] is an OLTP benchmark that specifies certain transaction types that are executed on populated tables. The benchmark models a wholesale supplier and its business transactions related to order processing. The transactions are concurrent and the record and population sizes are varied in the benchmark.

Hsu et al. [14] have compared version 3 of the *TPC-C* and now obsolete *TPC-D* benchmark against several peak workloads in world's largest corporation production databases. They criticize the *TPC-C* benchmark for not modeling the I/O patterns of the real workloads accurately enough, claiming that it does not sufficiently exercise the caching, pre-fetching and write buffering features.

The **TPC-E** benchmark uses a database to model a brokerage firm with customers who generate transactions related to trades, account inquiries, and market research. The brokerage firm, in turn, interacts with financial markets to execute orders on behalf of the customers and updates relevant account information.

The benchmark is scalable, meaning that the number of customers defined for the brokerage firm can be varied to represent the workloads of businesses of different sizes. The benchmark defines the required mixture of transactions the benchmark must maintain. The TPC-E metric refers to the number of Trade-Result transactions the server can sustain over a period of time.

Although the underlying business model of TPC-E is a brokerage firm, the database schema, data population, transactions, and implementation rules have been designed to be broadly representative of modern OLTP systems.

The **Open Source Development Labs** (OSDL), now merged to Linux Foundation, has published four database performance tests. The tests are fair use implementations of TPC tests, but as they employ different workloads, the results are not comparable. OSDL's DBT-1 corresponds to TPC-W, DBT-2 to TPC-C, DBT-3 to TPC-H, and DBT-4 to TPC-App. The OSDL benchmarks are released under the Artistic License.

The **Open Source Telecommunications Database Benchmark** [36] developed by Toni Strandell [35] defines a database benchmark that mimics database usage in telecommunication applications.

Seven different transactions are employed in a random mix that has 80% read operations and 20% write operations. Both operation types ultimately affect only one row but may use others during evaluation. The operations are performed on a server machine separate from the client that generates the workload by using *Open Database Connectivity* (ODBC).

The benchmark has three different population sizes: one that is smaller than main memory size, one that is about twice the main memory size and one that is about five times the main memory size. The populations closely model the data that is in a Home Location Register (HLR) element in a mobile (GSM) network.

4.3 XML Benchmarks

The use of XML is increasing also in telecom domain. Nowadays XML starts to be the defacto standard of presentation layer, that is the presentation format of message payload, in Internet. Therefore, XML processing and messaging efficiency can be an important factor in performance of distributed applications.

An XML system may have several logically separable parts. At least the following tasks may be identified: XML document parsing, document tree manipulation, XML query parsing and processing, XML document database management. Products do not necessarily address all the tasks, and for any task several alternative implementation strategies exist (for example DOM and SAX for XML document parsing). Due to the nature of

XML it is used in many different application domains in widely different ways.

XML document database management has been studied in several papers. A framework for XML database benchmarking is introduced in [30]. The paper identifies ten challenges in XML processing, aiming at covering all performance critical aspects: bulk loading, reconstruction, path traversals, casting, missing elements, ordered access, references, joins, construction of large results and full-text search.

Böhme and Rahm [1] proposed the **XMach** benchmark as a scalable multi-user benchmark for evaluating the performance of XML data management systems. The XML database defined by their benchmark contains both structured data and text documents. Two benchmark variants were designed to test schema-based and schemaless content. The benchmark was also designed to test single/multiple DTD(s) for all documents. The workload defined in the benchmark consists of a total of eleven XQuery queries: eight retrieval and three update queries. All retrieval queries are designed to handle regular path expressions.

Different from XMach, where XML data in the benchmark is document-oriented, the **XMark** is a benchmark where the data models an auction Web Site [29]. XMark was designed to concentrate on the core ingredient of the XML benchmark: the query processor and its interaction with the data store. Network overhead, communication costs, transformations, and multi-user setting were not taken into consideration. The workload in XMark specifies twenty XQuery queries including exact match, ordered access, casting, regular path expressions, chasing references, and so on.

XBench is an XML database benchmarking application [47, 48]. It is based on a set of queries that are performed on differently populated document collections. XBench has been compared to other XML database benchmarking systems in [18] and [22]. Other XML benchmarks include **X007** [2] and **mbench** or Michigan benchmark [27, 28]. The mbench introduces a microbenchmark, which consists of a set of queries, a XML database population mechanism and a set of tests.

XMLTest [37, 38] is an XML processing test developed at Sun Microsystems. It simulates a multi-threaded server program that processes multiple XML documents in parallel. This is very similar to an application server that deploys web services and concurrently processes a number of XML documents that arrive in client requests. XMLTest is a standalone multi-threaded program implemented in Java. To avoid the effect of file I/O, the documents are read from and written to memory streams.

XMLTest measures the throughput of a system processing XML documents. For streaming parsers it just involves parsing through each document without any

writing or serialization. XMLTest reports one metric: *throughput*, that is the average number of XML transactions executed per second. It can be configured using the following parameters:

- Number of threads: This is tuned to maximize CPU utilization and system throughput.
- PullParserFactory: Implementation of parser used to parse through the document.
- StreamUsage: Whether stream parsers are being tested.
- RampUp: Time allotted for warm-up of the system.
- SteadyState: The length of interval when transaction throughput is measured.
- RampDown: Time allotted for ramp down, completing transactions in flight.
- XmlFiles: The actual XML documents used by XMLTest.

XMLTest reads these properties at initialization into an in-memory structure that is then accessed by each thread to initiate a transaction as per the defined mix. To keep things as simple as possible, XMLTest is a single-tier system where the test driver that instantiates an XML transaction is part of each worker thread. A new transaction is started as soon as a prior transaction is completed (there is no think time). The number of transactions executed during the steady state period is measured. The throughput, transactions per second is calculated by dividing the total number of transactions by the steady state duration. The average response time for each transaction is also calculated.

The **XML Benchmark** [46] provides the following separate benchmarks:

- Non-Validating Parsing with Native,SAX,DOM Engines Benchmark
- Creating + Serializing DOM tree Benchmark
- Schema Validation Benchmark
- XSL Transformation Benchmark
- XML Security (Signature, Encryption) Benchmark

Recently W3C's **Efficient XML Interchange** (EXI) working group [41] has published a measurement framework [26] to evaluate compactness and processing efficiency of various XML wire-formats. The XML document set is derived from use cases that might benefit from alternative wire-formats. The use cases include 1) Metadata in Broadcast Systems, 2) Floating Point Arrays in

the Energy Industry, 3) X3D Graphics Model Compression, Serialization and Transmission, 4) Web Services for Small Devices, 5) Web Services within the Enterprise, 6) Electronic Documents, 7) FIXML in the Securities Industry, 8) Multimedia XML Documents for Mobile Handsets, 9) Intra/Inter Business Communication, 10) XMPP Instant Messaging Compression, 11) XML Documents in Persistent Store, 12) Business and Knowledge Processing, 13) XML Content-based Routing and Publish Subscribe, 14) Web Services Routing, 15) Military Information Interoperability, 16) Sensor Processing and Communication, 17) SyncML for Data Synchronization, 18) Supercomputing and Grid Processing. For details, see [5].

The **EXI measurement framework** is a testing framework developed by the W3C EXI working group for the purpose of obtaining empirical data about properties (processing efficiency and compactness) of several XML and binary XML candidates. The EXI framework is built on top of another framework called Japex [16], which provides basic functionality for drawing charts, generating XML and HTML reports, etc. To simplify the creation of new drivers, the EXI framework includes additional functionality on top of what is provided by Japex that allows the framework to be executed in memory or over the network.

4.4 Embedded Microprocessor Benchmark Consortium

The Embedded Microprocessor Benchmark Consortium (EEMBC) [6] was founded by Markus Levy accompanied by several large semiconductor vendors as a non-profit industry-standard organization. The initial goal was to create a better way of evaluating embedded processor performance than the Dhrystone benchmark. EEMBC uses an outside entity—EEMBC Certification Laboratories (ECL)—to ensure that the published benchmark scores are repeatably and fairly produced.

The EEMBC benchmarks consist of suites of benchmarks for different application domains. The domains are typical usage areas of embedded processors: automotive, consumer, digital entertainment, mobile Java, networking, network storage, office automation, telecom, and power/energy. In version 1, each suite consist of a number of small processor benchmark kernels that present typical workloads in the specific domain. Second generation of EEMBC benchmarks (will) provide system-level benchmarks. The benchmark kernels are defined in freely available data sheets, but the source code of the benchmarks requires EEMBC membership.

The results of EEMBC—particularly *NetBench*, *StorageBench*, and *TeleBench*—can provide useful information to understand performance of control plane applica-

tions.

5 Implementation Notes

In this section we discuss the prototype implementation of our control plane benchmark. More detailed description can be found in [21].

Our prototype implementation of the control plane benchmark uses two major software packages: OpenSER (<http://www.openser.org/>) and SIPp (<http://sipp.sourceforge.net/>). The implementation has scripts for building, configuring, and running the benchmark. The implementation also includes tools for generating and monitoring server load, registering users to OpenSER, analysing the benchmark results, and producing graphs from the results.

OpenSER is used as the telephony server and SIPp as the user-agent clients (callers) and user-agent servers (recipients). OpenSER is instrumented to output time stamped events to system log from which the benchmark results are derived. A load generator and load monitor are also run alongside the telephony server on the SUT.

Control Plane Benchmark Package include:

README instructions

alog.c server log analysis

benchmark.sh benchmarking script

build-sut.sh SUT build script

build.sh tools build script

calibrate.sh benchmarking first stage

demo.cfg OpenSER configuration

duac.xml SIPp scenario

gencallgraph.octave results formatting

gencallstats.octave results formatting

iters.c load generator

loadan.sh server log analysis

loadmon.sh log monitor script

ml.c load generator

openser-1.1.0-notls_combined_ts-v3.patch OpenSER patch

options.xml SIPp ECHO scenario

register.sh user registration

run-sut.sh SUT run script

sip_reg.c user registration

sippan.py SIPp log analysis

vman.py server log analysis

5.1 Build tools

Building the benchmark is automated by two bash scripts: `build.sh` and `build-sut.sh`. The scripts are used to build everything required for the benchmark. Building the benchmark could be done by hand as well, but the scripts make building the benchmark convenient.

The `build-sut.sh` script first fetches OpenSER sources from the Internet using `wget`. It then patches the sources with the patch included in the benchmark package and builds OpenSER and installs it under the `sut` directory from which the script was invoked. The benchmark configuration (`demo.cfg`) for OpenSER is installed under the same directory hierarchy, in which OpenSER is installed. The script patches (using `sed`) absolute paths into the OpenSER configuration file and thus the OpenSER install directory is not movable. The script also builds the load generator executables, `iters` and `ml`, and places them to the directory from which the script was invoked. The script requires `bash`, `sed`, `wget`, GNU `tar`, C compiler, and GNU `make`.

Both scripts have variables that can be configured by the user. There are C compiler and make variables in both scripts, and `build.sh` also includes the SIPp sub-version revision to be used.

5.2 OpenSER

OpenSER is used as the main SUT software. The benchmark implementation instruments OpenSER to output time stamps of events that are needed for performance analysis. OpenSER is configured to output events to `syslog` so that its performance can be later analysed from the log. The version of OpenSER used by the benchmark is 1.1.0-notls.

5.2.1 Instrumentation

The OpenSER instrumentation is in the OpenSER patch file. The patch inserts time stamps in log events from the accounting module. It also makes the registrar module log timing information (time stamp and lookup time) of all lookups. Time stamps are derived using the standard `gettimeofday()` library function and expressed in microseconds.

5.2.2 Configuration

The default configuration of the benchmark uses the standard OpenSER configuration, but adds accounting

and options modules. The thread and memory usage of OpenSER are made configurable by the benchmark scripts. The default is to use a main memory database for the user register.

The accounting module is needed to store the call events into the system log. By default the accounting module logs the INVITE answered, ACK received, and BYE answered events. For proper call length analysis, the configuration has logging added at the reception of INVITE in the routing script.

The options module is used in the ECHO test. The SIP options request is basically answered statelessly, and as such it fits well as a signalling transport test.

5.3 Load generator

The load generator consists of two C language programs, `iters` and `ml`. The main load generating algorithm has an unrolled loop that does summation and writing over a memory area. The `iters` program runs the loop for 1 000 000 rounds, times the execution, and finally calculates how many rounds the underlying hardware can execute the loop in 10 ms. The `ml` program then uses the results of `iters` and creates a user-specifiable load to the system by creating a new process every second that executes the loop for as many rounds as the specified load requires.

5.4 Load monitor

The load monitor is implemented as a simple bash script in `loadmon.sh`. It repeatedly runs `date` and `vmstat` and sleeps for one second. The idea is to create a log of SUT's CPU and memory usage information with dates. The script is started by the `run-sut.sh` script along with OpenSER.

5.5 SIPp

SIPp is used as the user agents in the benchmark implementation. Two call scenario files are provided for SIPp by the benchmark implementation: `duac.xml` and `options.xml`. The user agent server uses the SIPp default UAS scenario. The used SIPp version is subversion revision 57, but it can be configured in the `build.sh` script.

The `duac.xml` scenario is a standard SIP user agent scenario. It send INVITE message with user information from a database of users (created by the `register.sh` script), waits for the response, sends ACK, pauses for the call length, sends BYE, and waits for the response.

The `options.xml` scenario is for the client side of the ECHO test. It sends a SIP OPTIONS request and waits for the answer. SIPp's response time view can be

used to see the ECHO latencies. The scenario has a response time partition tuned for modern hardware and low latency network. In high latency networks the partition should probably be changed.

5.6 Benchmark scripts and tools

The benchmark implementation uses several scripts to run the benchmark. The bash script `register.sh` registers a number (default is 1000) of users to OpenSER using the `sip_reg` program and creates a user database csv file for SIPp. The `calibrate.sh` and `benchmark.sh` bash scripts are used to run the benchmark and the `run-sut.sh` script is used to start the SUT.

The user registration is done by the `sip_reg` program written in C. The program uses GNU `libosip` and `libxosip` libraries for a SIP protocol implementation. The program sends SIP REGISTER messages to a server and waits for the responses. The user names have user specified name formatting.

The `calibrate.sh` script runs SIPp for a given number of seconds using the benchmark call scenario. Starting at a given calls per second rate, the script uses `sippan.py` to analyse the SIPp log after the run has ended. If the realised call rate corresponds to the one requested, the call rate is increased and the call scenario re-run. The cycle is repeated until a call rate that the server can barely sustain is found.

The `benchmark.sh` script performs the benchmark by running SIPp for a given number of seconds using the benchmark call scenario and a given call rate. After the benchmark the script prints the realised call rate, using `sippan.py` to analyse the SIPp results log.

5.7 Log analysis tools

The benchmark implementation includes two tools for server log analysis and two octave scripts to produce graphs and statistics from results of the log analysis. There is also one tool for analysing SIPp logs.

5.7.1 Server log analyser

The `alog` program, written in C, is the primary tool of analysing the server logs. The function of `alog` is to read and analyse the log that OpenSER writes to the system log, that is the statements output by the accounting and registrar modules.

The program reads standard input line by line, optionally ignoring lines that do not fall inside a given span of times. All log lines that are output by the "openser" process are further analysed, rest of the lines are omitted.

Log lines from OpenSER are parsed and events with their timestamp are created into singly linked lists. Separate lists are kept for lookups, incomplete calls, and complete calls.

Lookup events are added to a list in increasing time stamp order. Call events are first added to the incomplete calls list, and then, when all call events (INVITE, INVITE answered, ACK, BYE) are encountered, the call is moved to the complete calls list. Call events are not kept in timestamp order. A call may have multiple events of the same kind; for INVITE the latest INVITE before the answer to INVITE is kept, for other events, the first encountered event is kept. After EOF, the lookup and call structures are analysed and optionally output to Octave or CSV files.

5.7.2 Server load log analyser

The server load log can be analysed using the `loadan.sh` bash script. The script uses `sed` to filter out log lines that do not fall into a given time interval. It then feeds the results to the `vman.py` Python script, which calculates and outputs CPU and memory usage statistics.

5.7.3 Graph and statistics output

The Octave scripts `gencallgraph.octave` and `gencallstats.octave` can be used to output graphs and statistics from `alog` output. They are implemented using GNU Octave and also require `gnuplot`. The former script outputs a graph of calls per second, including completed calls, deadlined calls, and failed calls as separate curves. The latter script outputs statistics from call data that is more detailed than the statistics output by `alog`.

The `gencallgraph.octave` script reads a file (`-cpu.octave`) output by `alog` into a matrix. It uses Octave interface to `gnuplot` to plot the complete, deadlined, and failed calls to a single graph.

The `gencallstats.octave` reads a file (`-calls.octave`) output by `alog` into a matrix and prints timing values for all call events. Printed values include minimum, maximum, mean, median, variance, and 10th, 25th, 50th, 75th, and 90th percentiles of lengths between events. The measured events are the full call from INITIATE to CLOSE, and all the intervals between two successive events. For failed calls, different event types are counted and printed out.

6 Acknowledgments

The authors want to thank Mika Kukkonen and Tapio Tallgren from Nokia Networks (now Nokia Siemens Net-

works) for fruitful discussions and valuable comments during the development of the benchmark. Kare Piekkola joined the UHe research team in 2007. He and Feetu Nyrhinen from Tampere University of Technology carried out testing and finalization of our prototype implementation.

7 Availability

The prototype implementation, technical documentation and specification are available from <http://sourceforge.net/projects/control-plane/>.

References

- [1] BÖHME, T., AND RAHM, E. Multi-user evaluation of XML data management systems with XMach-1. In *Proceedings of VLDB 2002 Workshop EEXTT (2002)*, Springer Verlag, pp. 148–159. Lecture Notes in Computer Science, Vol. 2590.
- [2] BRASSAN, S., LEE, M. L., LI, Y. G., LACROIX, Z., AND NAMBIAR, U. The XOO7 benchmark. In *Proceedings of VLDB 2002 Workshop EEXTT (2002)*, Springer Verlag, pp. 146–147. Lecture Notes in Computer Science, Vol. 2590.
- [3] BROWN, A. B., AND SELTZER, M. I. Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the Intel x86 architecture. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (Seattle, Washington, 1997)*, ACM, pp. 214–224.
- [4] CASTELINO, M., GUNTURI, R., FILAURO, V., VLANTIS, G., CAMPMAS, M., AND COPPOLA, A. Benchmark for IP forwarding tables. In *Proceedings of the IEEE International Conference on Performance, Computing, and Communications (2004)*, IEEE, pp. 123–130.
- [5] COKUS, M., AND PERICAS-GEERTSEN, S. XML binary characterization use cases. Working Group Note 31 March 2005, W3C, Mar. 2005.
- [6] EEMBC. The embedded microprocessor benchmark consortium. <http://www.eembc.org/>, 2007.
- [7] ELHANANY, I., CHIOU, D., TABATABAEE, V., NORO, R., AND POURSEPAJ, A. The network processing forum switch fabric benchmark specifications: An overview. *IEEE Network* 19, 2 (Mar. 2005), 5–9.
- [8] ETSI. Telecommunications and Internet converged Services and Protocols for Advanced Networking (TISPAN); IP Multimedia Subsystem IMS; Functional architecture. Technical Specification ETSI ES 282 007, V 1.1.1, European Telecommunications Standardisation Institute, June 2006.
- [9] ETSI. Telecommunications and Internet converged Services and Protocols for Advanced Networking (TISPAN); IMS/NGN Performance Benchmark. Part 1: Core Concepts. Draft Technical Specification ETSI TS 186 008-1, V 0.0.98, European Telecommunications Standardisation Institute, Jan. 2007.
- [10] ETSI. Telecommunications and Internet converged Services and Protocols for Advanced Networking (TISPAN); IMS/NGN Performance Benchmark. Part 2: Subsystem Configurations and Benchmarks. Draft Technical Specification ETSI TS 186 008-2, V 0.0.98, European Telecommunications Standardisation Institute, Jan. 2007.

- [11] ETSI. Telecommunications and Internet converged Services and Protocols for Advanced Networking (TISPAN); IMS/NGN Performance Benchmark. Part 3: Traffic Sets and Traffic Profiles. Draft Technical Specification ETSI TS 186 008-3, V 0.0.98, European Telecommunications Standardisation Institute, Jan. 2007.
- [12] HENNING, J. L. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer* 33, 7 (July 2000), 28–25.
- [13] HENNING, J. L. Special issue: SPEC CPU2006 analysis. *ACM SIGARCH Computer Architecture News* 35, 1 (Mar. 2007), 63–134.
- [14] HSU, W. W., SMITH, A. J., AND YOUNG, H. C. I/O reference behavior of production database workloads and the tpc benchmarks — an analysis at the logical level. *ACM Transactions on Database Systems* 26, 1 (Mar. 2001), 96–143.
- [15] HUBER, J. F. Mobile next-generation networks. *IEEE MultiMedia* 11, 1 (Jan. 2004), 72–83.
- [16] Japex Project. <https://japex.dev.java.net/>, 2007.
- [17] LEE, B. K., AND JOHN, L. K. NpBench: A benchmark suite for control plane and data plane applications for network processors. In *Proceedings of the 21st International Conference on Computer Design (ICCD'03)* (San Jose, Calif., Oct. 2003), IEEE, pp. 226–233.
- [18] LI, Y. G., BRESSAN, S., DOBBIE, G., LACROIX, Z., LEE, M. L., NAMBIAR, U., AND WADHWA, B. XOO7: Applying OO7 benchmark to XML query processing tool. In *Proceedings of the Tenth International Conference on Information and Knowledge Management* (Oct. 2001), ACM, pp. 167–174.
- [19] LINDHOLM, H., PIEKKOLA, K., AND VÄHÄKANGAS, T. Control plane benchmark software, Apr. 2007. Available from <http://sourceforge.net/projects/control-plane/>.
- [20] LINDHOLM, H., AND VÄHÄKANGAS, T. Control plane benchmark specification. Technical report, University of Helsinki, Department of Computer Science, Nov. 2006.
- [21] LINDHOLM, H., AND VÄHÄKANGAS, T. Control plane benchmark technical documentation. Technical report, University of Helsinki, Department of Computer Science, May 2007.
- [22] LU, H., YU, J. X., WANG, G., ZHENG, S., JIANG, H., YU, G., AND ZHOU, A. What makes the differences: Benchmarking XML database implementations. *ACM Transactions on Internet Technology* 5, 1 (Feb. 2005), 154–194.
- [23] MCVOY, L., AND STAELIN, C. Imbench: Portable tools for performance analysis. In *Proceedings of the USENIX 1996 Annual Technical Conference* (San Diego, Calif., Jan. 1996), USENIX Association, pp. 279–284.
- [24] MEMIK, G., AND MANGIONE-SMITH, W. H. Evaluating network processors using NetBench. *ACM Transactions on Embedded Computing Systems* 5, 2 (May 2006), 453–471.
- [25] OPTICAL INTERNETWORK FORUM. NPF benchmarking implementation agreements. Available from http://www.oiforum.com/public/NPF_IA.html, 2007.
- [26] PERICAS-GEERTSEN, S. EXI measurement framework. Release Notes, W3C, 2007. Available from http://www.w3.org/XML/EXI/framework/RELEASE_NOTES.txt.
- [27] RUNAPONGSA, K., PATEL, J. M., AND AL-KHALIFA, S. The Michigan benchmark: A microbenchmark for XML query processing systems. In *Proceedings of VLDB 2002 Workshop EEXTT* (2002), Springer Verlag, pp. 160–161. Lecture Notes in Computer Science, Vol. 2590.
- [28] RUNAPONGSA, K., PATEL, J. M., JAGADISH, H. V., CHEN, Y., AND AL-KHALIFA, S. The Michigan benchmark: Towards XML query performance diagnostics. Tech. rep., University of Michigan, Feb. 2002. Available from <http://www.eecs.umich.edu/db/mbench/>.
- [29] SCHMIDT, A., WAAS, F., KERSTEN, M., CAREY, M. J., MANOLESCU, I., AND BUSSE, R. Xmark: A benchmark for XML data management. In *Proceedings of the 28th International Conference on Very Large Data Bases* (2002), pp. 974–985.
- [30] SCHMIDT, A., WAAS, F., KERSTEN, M., FLORESCU, D., CAREY, M. J., MANOLESCU, I., AND BUSSE, R. Why and how to benchmark XML databases. *ACM SIGMOD Record* 30, 3 (Sept. 2001), 27–32.
- [31] SCHULZRINNE, H., NARAYANAN, S., LENNOX, J., AND DOYLE, M. SIPstone – Benchmarking SIP server performance. Tech. rep., Columbia University, Apr. 2002. Available from <http://www.columbia.edu/>.
- [32] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queuing using deficit round robin. In *Proceedings of SIGCOMM'95* (Cambridge, Mass., Aug. 1995), ACM, pp. 231–242.
- [33] Standard Performance Evaluation Corporation. <http://www.spec.org/>, 2007.
- [34] STAELIN, C. Imbench: an extensible micro-benchmark suite. *Software: Practice and Experience* 35, 11 (Sept. 2005), 1079–1105.
- [35] STRANDELL, T. Open source database systems: Systems study, performance and scalability. Master's thesis, University of Helsinki, Department of Computer Science, Aug. 2003. Available from <http://ethesis.helsinki.fi/>.
- [36] STRANDELL, T. Open source database benchmark, 2007. Available from <http://hoslab.cs.helsinki.fi/savane/projects/ndbbenchmark/>.
- [37] SUN MICROSYSTEMS. XMLTest Project. <https://xmltest.dev.java.net/>, 2007.
- [38] SUN MICROSYSTEMS. XMLTest 1.0. White Paper, Sun Microsystems, 9999. Available from <http://java.sun.com/performance/reference/whitepapers/WS.Test-1.0.pdf>.
- [39] TPC. TPC Benchmark C. Standard Specification Revision 5.8.0, Transaction Processing Performance Council, Dec. 2006. Available from <http://www.tpc.org/pcc/spec/tpcc-current.pdf>.
- [40] Transaction processing performance council. <http://www.tpc.org/>, 2007.
- [41] W3C. Efficient XML Interchange Working Group. <http://www.w3.org/XML/EXI/>, 2007.
- [42] WEICKER, R. P. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM* 27, 10 (Oct. 1984), 1013–1030.
- [43] WEICKER, R. P. Dhrystone benchmark: Rationale for version 2 and measurement rules. *ACM SIGPLAN Notices* 23, 8 (Aug. 1988), 49–62.
- [44] WEISS, A. R. Dhrystone benchmark: History, analysis, "scores" and recommendations. White paper, EEMBC Certification Laboratories (ECL), Oct. 2002. Available from <http://www.eembc.org/techlit/Datasheets/dhrystone.wp.pdf>.
- [45] WOLF, T., AND FRANKLIN, M. CommBench — a telecommunications benchmark for network processors. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (Austin, Texas, Apr. 2000), pp. 154–162.
- [46] XML Benchmark. <http://xmlbench.sourceforge.net/>, 2007.
- [47] YAO, B. B., ÖZSU, M. T., AND KEENLEYSIDE, J. XBench—A family of benchmarks for XML DBMSs. In *Proceedings of VLDB 2002 Workshop EEXTT* (2002), Springer Verlag, pp. 142–163. Lecture Notes in Computer Science, Vol. 2590.

- [48] YAO, B. B., ÖZSU, M. T., AND KHANDELWAL, N. XBench benchmark and performance testing of XML DBMSs. In *Proceedings of the 20th International Conference on Data Engineering (ICDE'04)* (2004), IEEE, pp. 621–632.

Notes

¹ETSI is European Telecommunications Standardisation Institute and TISPAN its technical committee on Telecommunications and Internet converged Services and Advanced Networking.

²Standard Performance Evaluation Corporation or SPEC [33] has seven up-to-date benchmarks categories. The benchmarks are available by purchase, but the results are freely available in documents at www.spec.org and in many scientific studies. The CPU benchmarks are stand-alone in the sense that they do not utilize network in any way. SPEC CPU2006 is a collection of 12 integer (CINT2006) and 17 floating point (CFP2006) benchmarks. The most significant difference to the SPEC CPU2000 [12] is the increased size of test programs: 3 334 000 lines of code and 5 621 modules in CPU2006 when the corresponding numbers in CPU2000 were 811 000 and 1 432. A detailed analysis of SPEC CPU2006 can be found in [13].

Appendix: Event Handler

```
%\begin{lstlisting}[frame=single,showstringspaces=false,basicstyle=\tiny]
void handle_event(enum EVENT e, struct connection *c)
{
    switch(enum EVENT e) {
        ECHO:
            send_message(ECHO, c);
            break;
        INITIATE:
            log_action(ts, "INITIATE received from %s", c->URI);
            lookup(c);
            log_action(ts, "User lookup started for URI %s", c->destURI);
            break;
        LOOKUP_DONE:
            /* create call structure to which connections to A and B are associated */
            struct call *cs;
            struct connection *b;
            cs = alloc(struct call);
            cs->first_connection = c;
            c->call = cs;
            send_message(TRYING, c);
            acquire_mutex(c->mutex);
            log_action(ts, "TRYING sent to %s", c->URI);
            b->URI = c->lookupresult;
            release_mutex(c->mutex);
            create_connection(b);
            send_message(INITIATE, b);
            acquire_mutex(b->mutex);
            acquire_mutex(c->mutex);
            /* make a circular list of connections */
            c->next_connection = b;
            b->next_connection = c;
            release_mutex(c->mutex);
            b->call = cs;
            log_action(ts, "INITIATE sent to %s", b->URI);
            release_mutex(b->mutex);
            break;
        LOOKUP_TIMEOUT:
        LOOKUP_FAILED:
            log_action(ts, "Lookup timed out for %s", c->destURI);
            send_message(ERROR, c);
            break;
        INITIATE_TIMEOUT:
            struct connection *a;
            acquire_mutex(c->mutex);
            log_action(ts, "INITIATE timed out for %s", c->URI);
            a = c->next_connection;
            release_mutex(c->mutex);
            send_message(ERROR, a);
            break;
        RINGING:
            struct connection *a;
            acquire_mutex(c->mutex);
            a = c->next_connection;
            release_mutex(c->mutex);
            send_message(RINGING, a);
            log_action(ts, "RINGING sent to %s", a->URI);
    }
}
%\end{lstlisting}
```

```

    break;
ANSWERED:
    struct connection *a;
    acquire_mutex(c->mutex);
    a = c->next_connection;
    release_mutex(c->mutex);
    send_message(ANSWERED, a);
    log_action(ts, "ANSWERED sent to %s", a->URI);
    break;
ANSWERED_ACK_TIMEOUT:
    struct connection *b;
    acquire_mutex(c->mutex);
    log_action(ts, "ANSWERED_ACK timed out for %s", c->URI);
    b = c->next_connection;
    release_mutex(c->mutex);
    send_message(ERROR, b);
    break;
ANSWERED_ACK:
    struct connection *b;
    acquire_mutex(c->mutex);
    b = c->next_connection;
    release_mutex(c->mutex);
    send_message(ANSWERED_ACK, b);
    log_action(ts, "ANSWERED_ACK sent to %s. Connection established.", a->URI);
    break;
CLOSE:
    struct connection *o;
    acquire_mutex(c->mutex);
    o = c->next_connection;
    release_mutex(c->mutex);
    send_message(CLOSE, o);
    break;
CLOSE_ACK_TIMEOUT:
    struct connection *o;
    acquire_mutex(c->mutex);
    o = c->next_connection;
    release_mutex(c->mutex);
    send_message(ERROR, o);
    break;
CLOSE_ACK:
    struct connection *o;
    acquire_mutex(c->mutex);
    o = c->next_connection;
    c->next_connection = NULL;
    release_mutex(c->mutex);
    send_message(CLOSE_ACK, o);
    acquire_mutex(o->mutex);
    log_action(ts, "CLOSE_ACK sent to %s. Connection closed.", o->URI);
    o->next_connection = NULL;
    acquire_mutex(o->call->mutex);
    o->call->first_connection = NULL;
    release_mutex(o->call->mutex);
    free(o->call);
    release_mutex(o->mutex);
    break;
}
}

```