Control plane benchmark specification

Heikki Lindholm, Taneli Vähäkangas

Helsinki November 27, 2006
Final
University of Helsinki
Department of Computer science

Abstract

There are not many comprehensive control plane and telecommunications benchmarks. This paper presents a component-based benchmark that is transport and hardware agnostic.

The benchmark specifies system under test (SUT) and driver components. The SUT consists of telephony server and load generator components. The telephony server is further composed of signalling protocol, user location register, and logging components. APIs for the components are provided for user implementation.

The presented benchmark specification assumes a transport capable of basic telephony signaling actions: call connecting and disconnecting. Furthermore, a mechanism for logging signalling layer events is required. The user location register is a database for looking up call participants. No connectivity to external databases is required.

The benchmark specifies a typical telephony usage profile and relevant metrics to measure. The central metric is the busy hour call attempts. The benchmark is constructed so that the performance of the components should linearly affect that metric. A skeleton for a reference implementation is also presented.

Contents

1 Introduction

1.1 Why a new benchmark?

There are a lot of existing benchmarks for telecom applications. Why specify a new one and not employ an existing, tried and commonly accepted benchmark? Some of the reasons are laid out in [1]. Microbenchmarks are good at pin-pointing performance problems. However, they can not reliably be used to compare performance of distinct systems. Application-level benchmarks exist, but only very few are applicable at all to the problem at hand: benchmarking control plane applications in telecom environments.

1.2 Related work

SIPStone[2] is a benchmark for SIP proxy servers. It resembles the work presented here in the context of that specific transport and configuration. However, our desire is to remain agnostic to the signaling layer. With SIPstone this is not possible, unless we heavily modify it.

The ETSI TISPAN working group has proposed a telecom benchmark called IMS. Their work is not yet available so details are not known, but the benchmark goals possibly overlap with the work presented here. Some work related to IMS is available at `http://www.open-ims.org/`.

1.3 Purpose

For the reasons stated above in chapters 1.1 and 1.2 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, 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 core components of the server, most importantly the signalling layer and the user location database. This allows us to change only one component at a time and see the difference it has on total system performance in typical control plane usage.

2 General system architecture and requirements

The control plane benchmark architecture is shown in Figure 1. The internal organization of the SUT is restricted in section 2.1.

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.

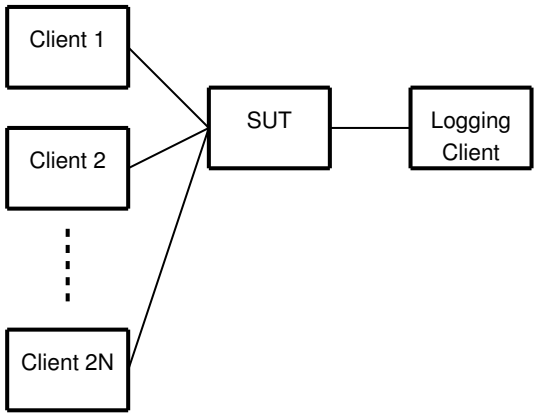

Figure 1: Control plane benchmark architecture.

2.1 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.

2.1.1 Signalling layer

2.1.1.1. The signalling layer is implemented in the SUT and in the driver. [1]

2.1.1.2. The signalling layer must be able to perform the echo test in chapter 3.2.

2.1.1.3. The SUT and driver must be able to perform the call scenario presented in chapter 3.1. All the functions must be implemented.

[1] 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.

2.1.1.4. 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).

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

2.1.3 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.

2.1.4 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.

2.1.5 Logging

2.1.5.1. The telephony server must log all actions that are specified in the corresponding test description in chapter 3.1.

3 Test profiles

3.1 Telephony service test

The telephone service test simulates the usual telephone call scenario.

3.1.1 Description

One possible call sequence is shown in Figure 2.

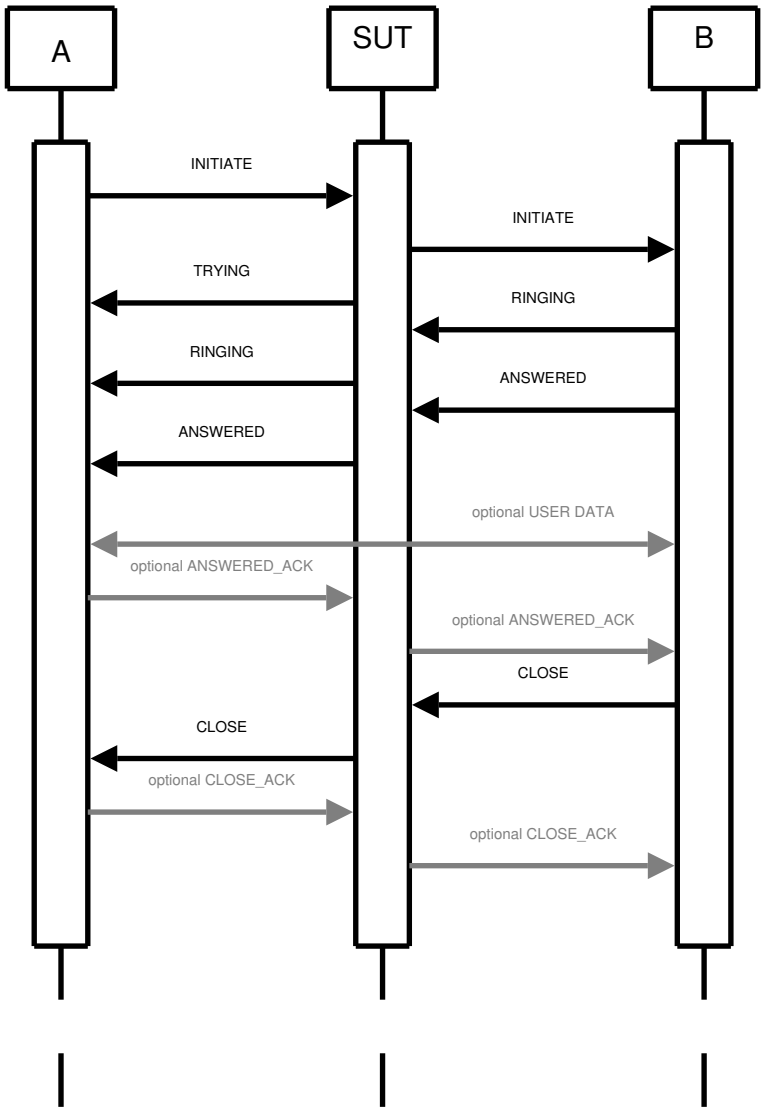

Figure 2: Basic call sequence diagram.

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 (i.e. 15 minutes $*R_i < N_i$).

- Initialize the SUT (including the load generator)
- Initialize the driver
- Optionally, the driver primes the SUT by performing N_i' calls, where N_i' equals $0.01 * N_i$.
- The driver waits until all priming calls are finished and there are no outstanding requests
- The driver performs calls at rate R_i for at least 15 minutes
- 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 evenly distributed random variable in the range 1 to 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 has exponential distribution with parameter $\lambda = \frac{1}{7s}$.

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 x seconds after the ANSWERED message, where x has exponential distribution with parameter $\lambda = \frac{1}{30s}$.

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.

3.2 Echo test

The echo test measures the performance of the signalling layer. It is meant as an auxiliary test case for isolating the effects of the signalling layer or layers below it.

3.2.1 Description

The echo test is performed between the driver and the SUT. Sequence diagram for the echo test is shown in Figure 3.

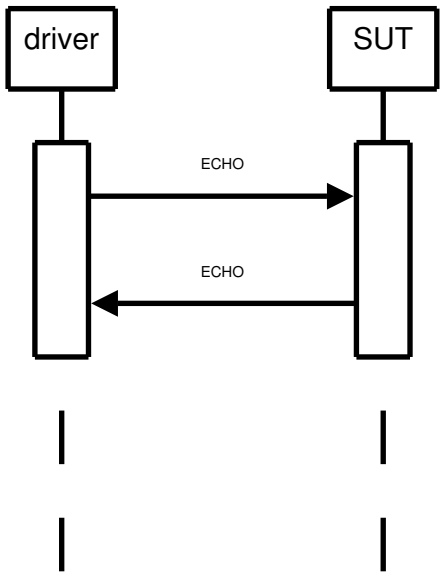

Figure 3: Echo test sequence diagram.

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

- Initialize the SUT
- Initialize the driver
- Transmit 1,000 ECHO messages from the driver to the SUT
- Measure response times of the received ECHO messages
- Report test results

3.2.1.2. The rate between ECHO messages is one second.

4 Performance metrics

4.1 Measuring

4.1.1 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.

4.1.2 The SUT

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.

4.1.3 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 3.1.

4.1.3.1. Every test run is separate and done according to the clauses in 3.1.

4.1.3.2. Every test run should run for 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* 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.

4.2 Calculated metrics

4.2.1 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.

4.2.2 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.

5 Benchmark environment

5.1 Requirements

5.1.1 Hardware specifications

5.1.1.1. It is required that the SUT has enough memory; the benchmark result must not be affected by the slowdown caused by paging memory from or to 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 today's systems this requirement translates to a network connection of at least 100 Mbps.

5.2 Software components

This section describes the software components that comprise the benchmark.

5.2.1 The signalling layer

The requests, which the clients (user agents) make to the server, and the corresponding answers are carried by the signalling layer. For example, the signalling 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.

5.2.2 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.

5.2.3 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 layer to accept requests and has a local or external database

connectivity for performing lookups from the user location register (and possibly for recording state information.)

5.2.4 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.

5.3 Network topology

The suggested network topology is shown in Figure 4. There may be several users. The database may reside within the SUT. It is assumed that the network topology resembles that of a real system.

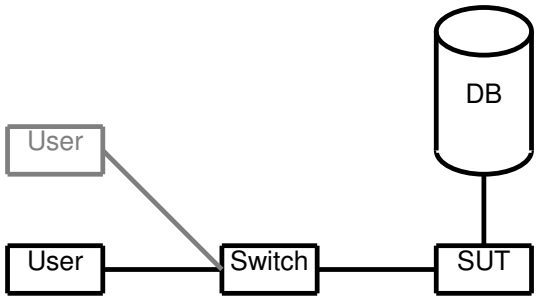

Figure 4: Suggested network topology.

6 Benchmark implementation

6.1 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 3.1 and 3.2. 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 measuring must be implemented at the points specified by clauses 4.1.1.1 and 4.1.1.2.

6.2 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.

6.2.1 Component overview

The components that make up the telephony server are the signalling 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.

6.2.2 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 messages can be handled as null operations by the signalling 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 acknowledgement to an ANSWERED message timed out. This event is optional.

CLOSE_ACK The user-agent acknowledgement to a CLOSE message 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 succesfully.

LOOKUP_FAILED Lookup failed for some reason other than timeout.

LOOKUP_TIMEOUT Lookup timed out.

6.2.3 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 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.

6.2.4 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`.

6.2.5 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 *) Send 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 *) Create 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.

6.2.6 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 *) Lookup the receiver of caller based on infomation 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.

6.2.7 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.

6.2.8 Event processor

State-machine for the server event processor is shown in Figure 5.

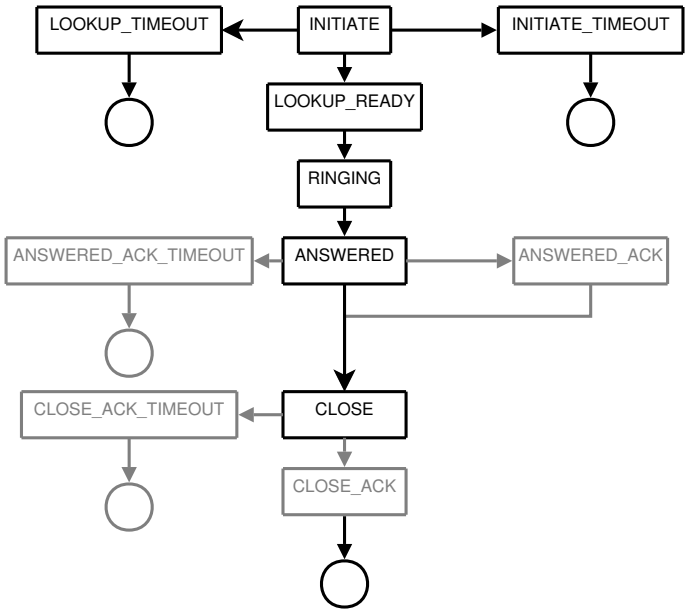

Figure 5: Telephony server state machine.

6.2.8.1. The event processing will be done as defined by the `handle_event` function below. 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.

```
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);
      aqcuire_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);
      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;
   }
}
```

7 Reporting

7.1 Report Format

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

7.1.1 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.

7.1.2 Report details

7.1.2.1. The network and node organization must be reported. For each node: number and type of processor, 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 4 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.

References

1 K. Raatikainen, H. Lindholm, and T. Vähäkangas. State-of-the-art in benchmarking. *Department of Computer Science Series of Publications B*, B-2006-N, 2006.

2 H. Schulzrinne, S. Narayanan, J. Lennox, and M. Doyle. SIPstone - benchmarking SIP server performance, Apr 2002.