SIMPLEstone - Benchmarking Presence Server Performance[*]

Vishal K. Singh and Henning Schulzrinne
Department of Computer Science, Columbia University
{*vs2140, hgs*}@cs.columbia.edu

Abstract: Presence is an important enabler for communication in Internet telephony systems. Presence-based services depend on accurate and timely delivery of presence information. Hence, presence systems need to be appropriately dimensioned to meet the growing number of users, varying number of devices as presence sources, the rate at which they update presence information to the network and the rate at which network distributes the user's presence information to the watchers. SIMPLEstone is a set of metrics for benchmarking the performance of presence systems based on SIMPLE. SIMPLEstone benchmarks a presence server by generating requests based on a work load specification. It measures server capacity in terms of request handling capacity as an aggregate of all types of requests as well as individual request types. The benchmark treats different configuration modes in which presence server interoperates with the Session Initiation protocol (SIP) server as one block.

1. Introduction

SIMPLEstone is an extension of SIPstone [1] for presence. It defines the benchmarking mechanism and metrics to be used for evaluating the performance of presence servers and deployments. The benchmark can be used to determine the impact of the presence traffic on the network and the request handling capacity of the presence server. The benchmark generates presence traffic for the server and measures the maximum request rate that the server can process in a timely manner without dropping new incoming requests. The presence traffic to be generated is specified in the workload specification which contains details like request rates for each message type and transport protocol to be used for tests. The test results are reported in terms of request handling capacity along with other details such as use of privacy filter and composition policy [14].

One way in which presence systems behave differently from the Session Initiation Protocol (SIP)-based call processing systems is that every incoming presence update (PUBLISH) generates multiple notification messages which are sent to all the watchers [5]. Hence the outgoing network traffic can be high for every presence update. Additionally, the benchmark metric for presence server cannot be only based on measuring request handling capacity as the processing for each type of request is different, e.g., a PUBLISH [11] message causes composition and filtering [11] operations and generates multiple NOTIFY messages, whereas a SUBSCRIBE [10] message triggers creation or renewal of subscription and filtering to generate a single NOTIFY message. Also, the processing can be different based on different event types. Hence, to benchmark a presence server it is not sufficient to consider only the user population and

[*] This work is supported by Verizon Labs.

the number of requests; rather, it is also important to consider details which determine the amount of processing done by the server, e.g., the number of watchers per presentity [5] and filter complexity for each of them.

The objective of SIMPLEstone is to define presence benchmarking specifications to perform network and server capacity planning and dimensioning. The impact of presence traffic on the network needs to be considered to appropriately provision the network resources. This traffic may also influence quality of service of other delay sensitive applications. Additionally, the capacity of the server to handle presence requests must be determined. A presence service provider needs to determine how many servers are required for a given user population. Similarly, a server software provider needs to specify the request handling capacity of his server. The benchmark metric can be used for comparing different presence servers. Another design objective is repeatability of tests and ease of specifying the workload. This is also useful for acceptance testing after an upgrade or change in the network, e.g., changes in configuration or changes in the network topology.

The remainder of this document is organized as follows: Section 2 presents an overview of presence server, Section 2.1 explains steps involved in presence processing, Section 3 explains factors affecting presence server performance; Section 4 explains the issues in choosing a benchmarking metric. Section 5 explains the SIMPLEstone benchmarking tool, our implementation architecture and components, workload specification and test types we have implemented. Section 6 discusses benchmarking methodology, measurement methodology and SIMPLEstone metrics to be reported. Finally, we present conclusions in Section 7 and references in Section 8.

2. Presence Server

A presence system allows for users to subscribe to each others presence [5] (availability and willingness for communication) information. The users (watchers) subscribe to presence information of other users (presentity) using SIP SUBSCRIBE requests and are notified about the changes in state of other users by SIP NOTIFY messages. Presence data for a user (presentity) is published from different presence sources using SIP PUBLISH. Figure 1 shows a basic flow of messages in a presence system. Diverse sources of presence information like wireline and wireless phones, applications like calendars and meeting makers, location sensors update presence information to the server using SIP PUBLISH message. This presence data is processed to give to the watchers a consistent view of the status of the presentities they are interested in.

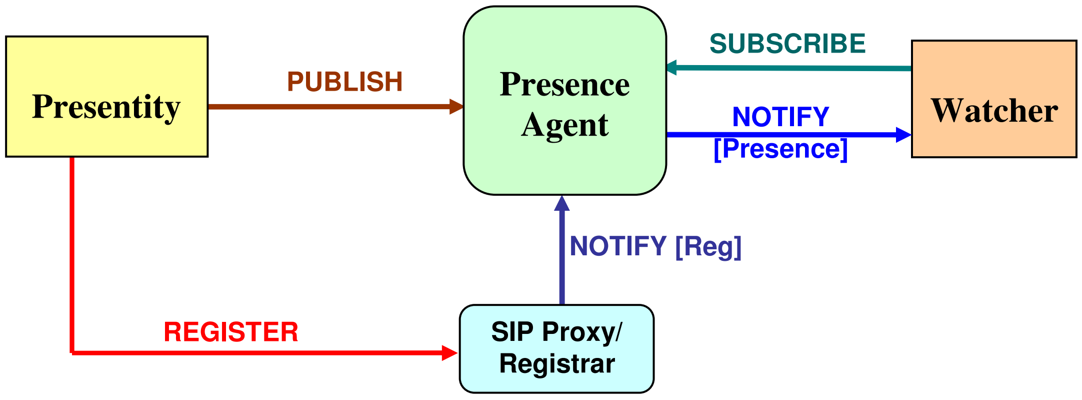


Figure 1 Basic block diagram of a presence system

2.1 Presence Processing Overview

A presence system supports three primary operations which internally may involve multiple operations. The three primary operations are (a) subscription. (b) notification and (c) publication. We briefly describe these and then explain the presence data processing on presence server.

a) **Subscription:** A watcher subscribes to the presence status of a user by sending SIP SUBSCRIBE request to the presence server. Upon receiving SUBSCRIBE request, the server performs authorization [12] of subscription and sends the status of presentity using a SIP NOTIFY request. The subscription can be rejected, approved or put in pending state depending upon the result of authorization. Once a subscription is approved, the presence document is delivered whenever the presentity's status changes. The watchers specify the rules for watcher filtering [13] in the SUBSCRIBE request.

b) **Notification:** The presence state of the presentity is conveyed to the watchers by delivering the presence document to them. The presence server delivers presence state information documents by sending SIP NOTIFY messages to the watchers. The presence document can either be in PIDF [6] or RPID [9] format. The presence document is filtered according to the presentity-specified and watcher-specified filters before being delivered to the watchers.

c) **Publication:** The sources of presence send information to the server for aggregation and distribution using SIP PUBLISH messages. The PUBLISH request triggers presence data processing which eventually generates a consistent view of the presentity on the server. This also triggers NOTIFY requests to be sent to all the watchers of the presentity, to update them with latest presence status.

Figure 2 shows the different stages of presence processing once the presence data is received. The published presence information for each presentity is composed [14] to a candidate presence document. Composition is done based on a composition policy [14] which in turn can be determined by presence authorization [12]. The composition policy can be same for all presentities or it can be different for different presentities. The presence server applies a privacy filter [13] on the candidate presence document to generate another candidate presence document. Different information is available to different subscribers after applying the privacy filter. The output of privacy filtering is the candidate presence document to which watcher filtering is applied. The filtered presence document is processed again to generate a difference from the previous NOTIFY body (for partial notifications [4]) or to ensure that final document after privacy and watcher filtering does not contain redundant information.

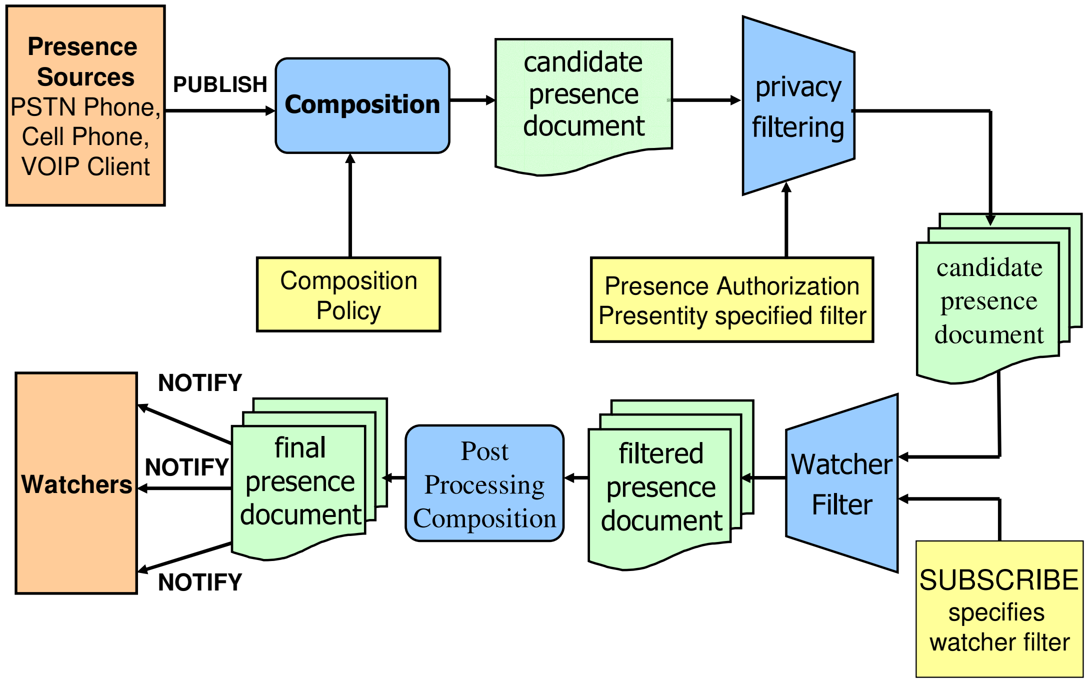

Presence Sources PSTN Phone, Cell Phone, VOIP Client
PUBLISH
Composition
candidate presence document
privacy filtering
Composition Policy
Presence Authorization Presentity specified filter
candidate presence document
Watchers
NOTIFY
NOTIFY
NOTIFY
final presence document
Post Processing Composition
filtered presence document
Watcher Filter
SUBSCRIBE specifies watcher filter

Figure 2 Presence processing overview

3. Factors Affecting Presence Server Performance

In this section, we explain some of the factors which affect the presence server performance.

Request Rate: Number of messages received and distributed by the presence server per second affects the amount of processing done by the server. This determines the throughput of the server. The server needs to maintain the dialog state for all the subscriptions to send the NOTIFY requests. High rate of updates of presence information causes higher load on the server and adversely affects server's throughput.

Presence Privacy Filtering: Every notification is generated after performing the filtering on composed presence document. Filtering involves rule matching and transformation on presence document based on matched filter rules. This implies matching the request attributes with conditions in the filter document and applying transformations specified in the presentity's filter document. The amount of load generated in filtering depends on the complexity of privacy filter document and the size of presence document. A transformation on a larger presence document may have higher complexity then a transformation on a smaller presence document.

Composition Policy: The type of composition policy that the server uses to compose the presence documents determines the processing done by the server in the composition step. Hence, it affects the server's performance. There can be simple composition policies like union or overriding composition policy or complex policies which do composition based on a rule language.

Watcher Filtering: The size and complexity of watcher filter sent by the watcher in SUBSCRIBE request affects both the processing on the server as well as the size of presence document sent to the watcher which in turn affects the presence traffic generated by the server.

Partial Notification: Partial notification is mechanism used to conserve bandwidth by sending only the changes in the presence document to the watchers. The server needs to create the change document either by comparing the composed and filtered presence document with the last sent document or using any other mechanisms.

Other Factors Affecting Server Performance: The server's request handling capacity also depends on factors like transport protocol used (TCP, UDP, or TLS), DNS look up time, authentication mechanism used, database optimizations and caching, use of in-memory vs. network based database, connection handling capacity of server, caching mechanisms, e.g., the filter rules which are frequently used can be stored in memory (working set of filters which are most frequently used) and need not be queried from database always. As in SIPstone, benchmarking scheme should generate sufficient load such that server performance is not overestimated because of caching, etc. A large user population may require higher look up time for a database based system because of more number of entries in the database or may require a higher memory for an in-memory database. Implementation specific factors like the type of XML parsing used for presence documents i.e., DOM vs. SAX parsing, presence document persistence mechanism i.e., storing presence document in database as a string vs. storing serialized DOM, also affects the server's performance. Features enabled on the presence server like SNMP support may also impact server's performance and must also be considered.

4. Issues in SIMPLEstone Benchmarking Metric

In this section, we explain the issues related to choosing a presence benchmarking metric and discuss metrics which can be used for benchmarking presence server performance. Some of the issues related to SIP benchmarking metric e.g., user population, are also applicable to presence and are explained in Section 3 of SIPstone [1].

User population: The number of users supported by the presence server can be a measure of presence server performance. However, the number of messages generated and processed by the server does not strictly depend on the user population. The number of messages generated per presentity depends on three factors. 1) Average number of users to whom the presentity has subscribed to. 2) Average number of watchers subscribed to the presentity. 3) The publication rate from each of the presentity sources. The presentity sends SUBSCRIBE requests to the user's he is watching. The presentity sends NOTIFY request to the user's watching him. Notification rate for each presentity varies with number presence sources it has. Each presence source behaves differently and may have different rates of PUBLISH messages. The PUBLISH rate depends on user behavior and device behavior. For example, cell phone or Wifi phone's rate of PUBLISH may depend on users mobility pattern. Thus, a server with smaller number of users configured might be processing more messages then a server with more number of users.

Here, we described expressing presence server benchmark in terms of user population that the presence server can support for a given number of watchers per presentity and a given rate of PUBLISH requests generated per presentity. However, a better benchmarking unit can be expressed in terms of number of messages processed by the server per unit time, which we explain in the next section.

Request rate or number of messages per second: As explained above, the number of messages processed per unit time for a given user population depends on many factors. Since, the number of messages processed per second represents the load on the server more accurately then the number of users, we can use the number of messages processed and generated per second as a presence server benchmarking metric. The maximum number of messages that can be processed by the server depends on the ratio of message types, i.e., ratio of SUBSCRIBE, PUBLISH and NOTIFY requests. To account for variation in server performance because of different ratios of message types, SIMPLEstone proposes to measure the message handling capacity for each of the message types independently as well as for different combinations of message types. Thus, the benchmarking can be done by loading the server by varying the rate of PUBLISH message for each presentity, average number of watchers per presentity, their subscription rates and the number of presentities. This can be expressed using the rate of PUBLISH requests processed and NOTIFY requests generated by the server.

However, it should be noted that different message types involve different amount of processing on the server. For example, the processing involved for PUBLISH message type is different from processing involved for SUBSCRIBE message. Thus, an aggregate number of messages processed by the server per unit time do not represent the server's actual capacity. Additionally, there can be clients requesting presence information on demand by polling mechanism and not by periodic updates, i.e., clients send SUSBCRIBE message with expire field set to zero which generates a NOTIFY response with the presence data without creating a subscription.

5. SIMPLEstone Benchmarking Architecture (and Implementation)

SIMPLEstone defines a load to be generated for the presence server whose performance is to be determined. The specified load is generated and response is received by different components of the benchmarking architecture described in following sections. The measurements are performed on the server under different load conditions. The servers request handling capacity is measured in terms of success rate vs. load (requests/sec). In the following sections we describe the workload specification, our implementation architecture and a description of tests we used to benchmark the presence server.

5.1 Architecture

The different components of SIMPLEstone architecture are shown in Figure 3. The architecture consists of the Server under Test (*SUT*), which is the presence server, one or more request load generators (*Loader*) that simulates the presentity sources, one or more response handlers (*Handler*) that simulates the watchers and a test coordinator (*Controller* – not shown in Figure 3) that coordinates the execution of the benchmark.

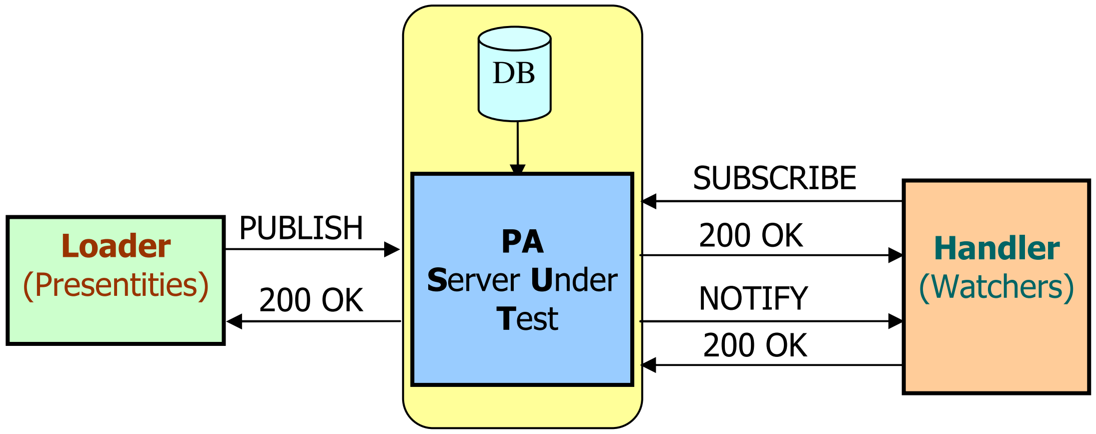
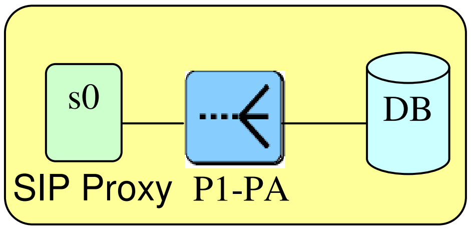


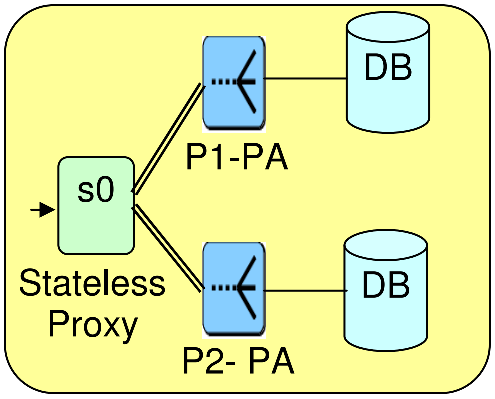
Figure 3 SIMPLEstone Architecture

5.1.1 Server Under Test (SUT)

The *SUT* consists of the host system(s), including hardware and software, required to support the SIP-based presence agent and any other components including database(s), if applicable. All network components between host machines which handle intra-SUT communications are part of the SUT. The software can consist of the SIP presence agent and the SIP proxy server either collocated or interoperating. In cases where the SIP proxy server does not co-reside with the presence server, the proxy server forwards the presence specific messages to the presence agent so that they can be appropriately processed. The *SUT* can have different configurations for failover and load sharing. The SIMPLEstone benchmark views the different configuration as one black box. To illustrate the idea, two examples are shown in Figure 4.



Configuration 1 – Single SIP proxy and presence server.



Configuration 2 - Two presence agents in load sharing mode.

Figure 4 SUT – Different Test Configurations

In Figure 4, there are two different sample configurations of SUT boxes. The first configuration consists of a single SIP proxy server forwarding presence request to a presence server. The second configuration consists of a single stateless SIP proxy server which distributes load between two different presence servers using a load distribution algorithm. Also, the different configurations can have different database redundancy models, e.g., *N+1 or 2N*. The SIMPLEstone benchmark is agnostic to internal configuration details of the SUT and determines the load handling capacity of the specified SUT block. Hence, it allows determining how the SUT capacity scales with increasing the number of presence servers and/or SIP proxy servers and for different load distribution algorithms.

5.1.2 Loader

The *Loader* emulates the presence sources for the presentities and generates SIP PUBLISH messages at the specified request rate. SIP requests for the benchmark are generated by user agent clients (UACs). Depending on the request rate to be generated, one or more *Loader* instances may be required to generate the load. The requests are sent to the specified server(s). The *Loader* should allow the following configuration parameters to be specified:

1 Rate of requests: This is the rate of generation of PUBLISH requests.
2 Number of presentity: This determines the user population or the number of users (URI's) used to generate the PUBLISH request at the specified rate.
3 Transport protocol (UDP or TCP, TLS)
4 Presence message body. This can be PIDF or RPID.

There are additional details like SIP addresses for presentities, the server's address, but they are static and do not affect the benchmark.

5.1.3 Handler

The *Handler* emulates the watchers and sends SUBSCRIBE requests to the server and handles receipt of NOTIFY requests by sending back the 200 OK response. The main details of the configuration for the *Handler* are:

1 Number of subscription per presentity i.e., Number of SUBSCRIBE requests per presentity with a unique “FROM” address in SIP header. It determines the number of NOTIFY requests generated for each PUBLISH request for that presentity.
2 Rate of SUBSCRIBE request.
3 Transport protocol (UDP or TCP, TLS)
4 Optionally, SUBSCRIBE body i.e., watcher filter, if the impact of watcher filtering on request handling capacity of server is to be measured.

Other details include subscriber’s SIP addresses, presentities SIP addresses and server IP addresses. Additionally, the *Handler* needs to maintain a count of NOTIFY messages received, which is then correlated with PUBLISH and SUBSCRIBE rates, number of presentities and subscriptions per presentity to determine the success rate. The *Handler* must be able to respond, under load, to an incoming NOTIFY so that the server does not do retransmission and thus get overloaded because of test infrastructure failure.

5.1.4 Controller

The *Controller* is the test coordinator that starts the SUT (SIP proxy server(s) and presence server(s)) as well as *Loader* and *Handler* instances on the specified systems. It also starts the programs or scripts to perform measurements like CPU utilization, memory utilization and store the measurement results for analysis after the test runs are done. Depending upon implementation it may use mechanisms such as rsh/rcmd, ssh, .shosts to login to different hosts and start appropriate application instances and measurement infrastructure pieces. The measurement infrastructure can do performance measurement based on scripts or based on SNMP-based tools such as MRTG. However, this is also independent of the benchmarking mechanism. The benchmark only proposes what measurements should be performed when the tests are done.

5.2 SIMPLEstone Benchmarking Workload Specification

SIMPLEstone benchmarking workload specifies the following parameters on a per test basis:

1 Number of presentities and their SIP addresses which the *Loader* uses to generate PUBLISH request and *Handler* subscribes to.

2 Number of watchers and their SIP addresses which the *Handler* uses for sending SUBSCRIBE request and accepting NOTIFY requests.

3 Request rate

  a. Rate of PUBLISH request. This can be specified per *Loader* instance or per presentity. This controls the overall PUBLISH rate.

  b. Total initial unique SUBSCRIBE per presentity.

  c. Rate of SUBSCRIBE refresh rate. This is optional parameter.

4 Presence (PUBLISH) body: This is required to perform testing with different body sizes and content types e.g., PIDF, RPID, PIDF-LO.

5 Transport protocol to be used for the test (UDP,TCP,TLS)

6 Filter documents (for testing with different filter sizes).

7 Timeout interval for PUBLISH responses and receipt of NOTIFY for each PUBLISH message.

8 The host addresses on which *SUT*, *Loader* and *Handler* run and configuration details for each of them like port numbers, database if any, etc.

5.3 Type of Tests

We have implemented two types of tests to benchmark the presence server.

5.3.1 SUBSCRIBE-NOTIFY TEST(S-N)

SUBSCRIBE-NOTIFY test is performed by varying the SUBSCRIBE request rate to the server and is used to determine the server's capacity to handle SUBSCRIBE requests per second. The test is called SUBSCRIBE-NOTIFY test as every SUBSCRIBE request generates a NOTIFY request. This test can be used to determine the NOTIFY generation capacity of the server and subscription handling capacity of server. In particular, this test can be used to determine the upper limit on number of unique subscriptions that the server allows, the maximum rate at which the subscriptions can be refreshed and the maximum SUBSCRIBE request rate that the server can handle. The upper limit on the number of unique subscriptions is determined by loading the server by sending SUBSCRIBE requests to unique presentities. The maximum subscription refresh rate can be determined by sending multiple SUBSCRIBE's in the same dialog.

Additionally, this test internally measures the impact of presence authorization rules on the server's request handling capacity. The subscription may be declined or forbidden or may be put in pending state depending on the authorization rules for the subscriber. Each of the subscription states impacts the request processing capacity of server. Therefore, the result must report the conditions of the test along with the results. Figure 5 shows the message flow for SUBSCRIBE-NOTIFY test and the actual messages are shown in figure 6.

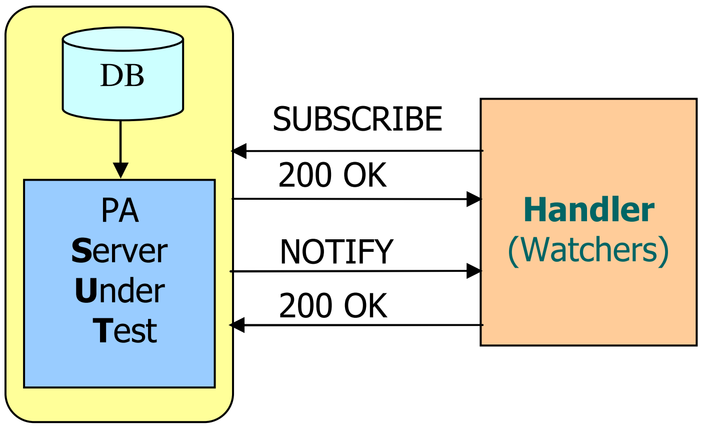


Figure 5 SUBSRIBE-NOTIFY Test

```
SUBSCRIBE sip:presentity@example.com SIP/2.0
```

```
Via: SIP/2.0/UDP host.example.com;branch=z9hG4bKnashds7
To: <sip:presentity@example.com>
From: <sip:watcher@example.com>;tag=12341234
Call-ID: 12345678@host.example.com
CSeq: 1 SUBSCRIBE
Max-Forwards: 70
Expires: 3600
Event: presence
Contact: sip:user@host.example.com
Content-Length: 0

NOTIFY sip:user@host.example.com SIP/2.0
Via: SIP/2.0/UDP pa.example.com;branch=z9hG4bK8sdf2
To: <sip:watcher@example.com>;tag=12341234
From: <sip:presentity@example.com>;tag=abcd1234
Call-ID: 12345678@host.example.com
CSeq: 1 NOTIFY
Max-Forwards: 70
Event: presence
Subscription-State: active; expires=3599
Contact: sip:pa.example.com
Content-Type: application/pidf+xml
Content-Length: ...

[PIDF document]
```

Figure 6 SUBSRIBE-NOTIFY Test.

5.3.2 SUBSCRIBE: PUBLISH-NOTIFY TEST(S: P-N)

PUBLISH-NOTIFY test is performed by varying the rate of PUBLISH request and is used to determine the server's capacity to handle PUBLISH request. PUBLISH request rate can be varied per presentity for a given user population to achieve the specified rate of PUBLISH request. In this test, the *Handler(s)* initially subscribe to the presentities by sending SUBSCRIBE messages to the *SUT*. After all the subscriptions are successful, the *Loader* starts sending PUBLISH requests at the specified request rate. The *Handler* receives NOTIFY messages and send response to NOTIFY messages.

The test can be done with different number of subscriptions per presentity. When the test is done with single subscription per presentity, it measures the PUBLISH handling capacity of the server, as in this case the output NOTIFY rate is equal to input PUBLISH rate. In general, *the number of NOTIFY requests = Number of presentity * rate of PUBLISH * number of subscription per presentity*. Each PUBLISH operation results in one composition and twice the number of watcher times filtering operations (privacy filtering and watcher filtering) for every presentity. Each one of the subscriber's may have different filters and varying in complexity and filter sizes. For the purpose of benchmark, an average filter size should be considered. The message flow is shown in the Figure 7 and actual messages are shown in Figure 8.

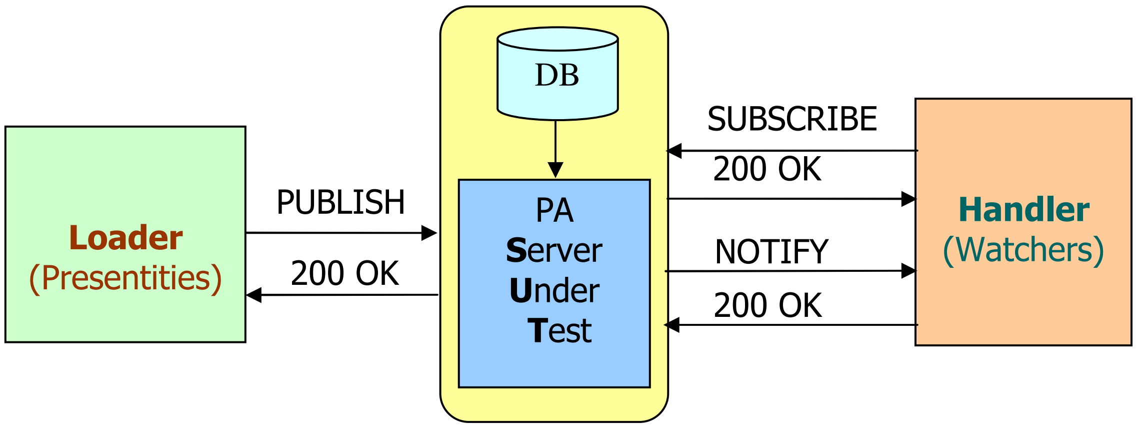


Figure 7 PUBLISH-NOTIFY Test

```
SUBSCRIBE sip:presentity@example.com SIP/2.0
Via: SIP/2.0/UDP host.example.com;branch=z9hG4bKnashds7
To: <sip:presentity@example.com>
From: <sip:watcher@example.com>;tag=12341234
Call-ID: 12345678@host.example.com
CSeq: 1 SUBSCRIBE
Max-Forwards: 70
Expires: 3600
Event: presence
Contact: sip:user@host.example.com
Content-Length: 0

PUBLISH sip:presentity@example.com SIP/2.0
Via: SIP/2.0/UDP pua.example.com;branch=z9hG4bK652hsge
To: <sip:presentity@example.com>
From: <sip:presentity@example.com>;tag=1234wxyz
Call-ID: 81818181@pua.example.com
CSeq: 1 PUBLISH
Max-Forwards: 70
Expires: 3600
Event: presence
Content-Type: application/pidf+xml
Content-Length: ...

[Published PIDF document]

NOTIFY sip:user@host.example.com SIP/2.0
Via: SIP/2.0/UDP pa.example.com;branch=z9hG4bK4cd42a
To: <sip:watcher@example.com>;tag=12341234
From: <sip:presentity@example.com>;tag=abcd1234
Call-ID: 12345678@host.example.com
CSeq: 2 NOTIFY
Max-Forwards: 70
Event: presence
Subscription-State: active; expires=3400
Contact: sip:pa.example.com
Content-Type: application/pidf+xml
Content-Length: ...
```

[New PIDF document]
Figure 8 SUBSRIBE: PUBLISH-NOTIFY Test

5.4 Consideration for SIP MESSAGE Request

The SIP MESSAGE [15] is an extension to SIP and allows transfer of instant messages. Since, instant messaging (IM) is closely related with presence. Hence, it is important to explain the impact of this on presence server performance. SIP MESSAGE is forwarded like any other SIP request by the proxy server and does not require any processing by the presence server. Hence, there is no direct impact on the performance of presence server. However, if the proxy server routing the messages is overloaded by instant message sessions, the presence traffic may get impacted. Also, the presence statuses of clients can potentially depend on if they are involved in IM sessions or in other words transmitting or receiving MESSAGE requests. The performance measurement of SIP proxy server for MESSAGE request can be done using the SIPstone specification. A new test can be added to the SIPstone specification with *Loader* sending MESSAGE and *Handler* receiving the message at different request rates. The server's capacity in successfully proxying the request determines the throughput for the MESSAGE request.

6. Benchmarking methodology

In our setup, SIMPLEstone benchmarking consists of a series of test runs, with increasing load levels generated by the load generators, and targeted at the server being tested (*SUT*).

Measurement Interval (MI): The measurement interval is defined as the steady state period during the execution of the test.

Publication rate (PR): Average number of PUBLISH messages per second from each *Loader* (source) to the *SUT* (presentity).

Subscription rate (SR): Average number of SUBSCRIBE sent to *SUT* per second. **Successful Subscriptions (SS):** The number of successful subscriptions per presentity.

Notification rate (NR): Average number of NOTIFY messages generated by server per second. The average number of subscription and average publication rate per presentity determines the average notification rate.

Transaction failure probability (TFP): The transaction failure probability is the fraction of transactions that fail, i.e., where the server does not return a provisional or final response within the time limit for the PUBLISH or SUBSCRIBE requests. It can also be NR going down the expected value. NR will go down for following reasons: The PUBLISH request is dropped or not processed, or, server cannot generate notification. The SUBSCRIBE request dropping is not considered as cause of NR going down because in that case the watcher knows that subscription is not created and expected NR is lowered by that value.

Success rate: The success rate is expressed in terms of successful PUBLISH, successful SUBSCRIBE and expected NOTIFY requests. Success rate for PUBLISH and SUBSCRIBE is based on receiving a 2XX response from the server. For, NOTIFY request this is ratio of obtained NOTIFY to the expected number of NOTIFY requests. *Expected notification rate = Number of PUBLISH per presentity * Number of SUBSCRIBE for the presentity*. If the expected notification rate is equal to the obtained notification rate as measured on handler, the success rate is 100%. The publication rate can be varied by changing the rate of PUBLISH requests per presentity. The notification rate can be varied by increasing the successful subscription per presentity or increasing the publication rate, for a given user population. In this way, performance, both in terms of number of messages as well as scalability in terms of number of users can be determined.

6.1 Measurement Methodology

The request rate is increased until the TFP increases to 5% (success rate comes down to 95%). The tests must run for sufficient duration so that the system reaches steady state as all the source of presence would PUBLISH the presence document by then. The highest sustained throughput is reported as the benchmark number. The test operates in an "open loop" mode, where the arrival of the *N+1st* request does not depend on the completion of the *Nth* request. After running the test at a given request rate, the request rate is increased to next level and the test is run for specified time duration.

6.2 SIMPLEstone Metrics

The SIMPLEstone result contains the following details.

- Description of the server farm or cluster configuration and configuration of servers within it (SIP and presence server interoperation topology). This includes:-.
  - The number of servers used (SIP server as well as presence server count),
  - The type of proxy server and its details (e.g., a local database, in-memory, or a network server) and
  - Whether presence server was located on the same host as the proxy server
  - Logging mechanism if used
  - SNMP and other features if used
  - Load balancing scheme used (if any)
- All aspects of the server hardware, in particular
  - the CPU count, type and speed,
  - the memory configuration,
  - the network interface type and speed,
  - the disk and disk controller configuration;
- The server operating system and version and any non-standard tunings or settings, e.g., for network parameters, number of file descriptors
- The type of network segments connecting the *Loader(s)*, the SUT and the *Handler* and the network bandwidth on the connecting links.
- The number of connections requested by the clients and accepted by the SUT per second. The intent is to count only the number of new connections made successfully by the clients in generating the load for the benchmark.
- CPU and memory utilization of server at various loads;
- Type of test (S-N, S:P-N)

The results of the tests can be reported in the scorecard format shown in the tables (1, 2 and 3) below. All columns except the throughput are configuration and test environment details. The tabular format takes care of the different functionalities, e.g., type of composition, support for filtering and their configurations, e.g., size of filters. However, SIMPLEstone does not mandate that these variables be a part of presence benchmark as it would introduce too many variations and make the benchmark less useful for comparing presence servers. Therefore, as described in Section 4 **the benchmark can be expressed only in terms of supported user population or the throughput of the server**. Table 1 show that the transport protocol is TCP and shows different combination of composition policies and filter configurations used for the test runs. Measured throughput is added in the last column after the test is completed. Depending upon the type of test the throughput is sum of number of PUBLISH messages per second and number of NOTIFY messages per second if the subscription is fixed i.e., if the test is of type (S: P-N) or the throughput can be sum of number of SUBSCRIBE messages per second and number of NOTIFY messages per second for a varying subscription rate i.e., if the test is S-N type. Similarly, we can get the results for tests with UDP and TLS protocols. The results should indicate the total i.e., aggregate success rate, PR, NR, SR and SS for the cluster of server as well as per-server.

Table 1 Throughput (request rate/second) for TCP

Protocol	Composition	Privacy Filter	Watcher Filter	Throughput
TCP	Default	None	None	
			Size = W1	
		Size = P1 (complexity)	None	
			Size = W2	
	Merge	None	None	
			Size = W1	
		Size = P1	None	
			Size = W2	
	Rule Based	None	None	
			Size = W1	
		Size = P1	None	
			Size = W2	

Table 2 Throughput (request rate/second) for UDP

Protocol	Composition	Privacy Filter	Watcher Filter	Throughput
UDP	Default	None	None	
			Size = W1	
		Size = P1	None	
			Size = W2	
	Merge	None	None	
			Size = W1	
		Size = P1	None	
			Size = W2	
	Rule Based	None	None	
			Size = W1	
		Size = P1	None	
			Size = W2	

Table 3 Throughput (request rate/second) for TLS

Protocol	Composition	Privacy Filter	Watcher Filter	Throughput
TLS	Default	None	None	
			Size = W1	
		Size = P1	None	
			Size = W2	
	Merge	None	None	
			Size = W1	
		Size = P1	None	
			Size = W2	
	Rule Based	None	None	
			Size = W1	
		Size = P1	None	
			Size = W2	

P1 is average number of rules in privacy filter. W1, W2 are average values of watcher filter sizes (complexity).

Thus, SIMPLEstone benchmark allows reporting the throughput in terms of

- Number of users supported (with assumptions about average number of sources and watchers per user and configuration details like average PIDF size, average filter size, composition policy)
- Number of messages per second

Additionally, SIMPLEstone can be used to estimate the bandwidth required on a per presentity basis that can be used for network capacity planning. However, given a user population (Number of presentity, number of watchers per presentity, the subscription and notification rate and average size of presence message bodies) we can approximate the average bandwidth required as a sum of incoming and outgoing traffic bandwidth.

The following relationship will apply.

$$BW \approx Number\_of\_presentity \times \begin{bmatrix} (SR \times Average\_SUBSCRIBE\_size) + \\ (NR \times Average\_NOTIFY\_size) + \\ (PR \times Average\_PUBLISH\_size \times Num\_of\_Publishers) \end{bmatrix}$$

Where *Num_of_Publishers* is average number of presence sources per presentity. Additionally, the affect of 200 OK for each message in above relationship must be considered. For a given number of messages, BW required for 200 OK can be calculated. Also, we need to account for retransmissions.

7. Conclusion

In this report, we described a benchmarking scheme for SIMPLE based presence servers. We discussed issues in designing benchmarks for presence servers mainly that the server performance depends on large number of factors. We presented architecture and the benchmarking methodology to measure presence server performance. Our proposed benchmark ensures coverage of all factors on which presence server performance may depend and reports the server capacity in terms of request rate for each request type individually as well as for tests which involve combination of request types in different proportions. We also consider performance under different configurations as well as with different features enabled.

8. References

1. Schulzrinne, H., Sankaran Narayanan, Jonathan Lennox and Michael Doyle, “SIPstone - Benchmarking SIP Server Performance”.
2. Rosenberg, J., "A Data Model for Presence"draft-ietf-simple-presence-data-model-04, August 23, 2005.
3. Rosenberg, J., "A Processing Model for Presence" draft-rosenberg-simple-presence-processing-model-01, July 17, 2005.
4. Lonnfors, M., "Session Initiation Protocol (SIP) extension for Partial Notification of Presence Information" draft-ietf-simple-partial-notify-05, May 24, 2005.
5. Day, M., Rosenberg, J. and H. Sugano, "A Model for Presence and Instant Messaging", RFC 2778, February 2000.
6. Sugano, H., Fujimoto, S., Klyne, G., Bateman, A., Carr, W. and J. Peterson, "Presence Information Data Format (PIDF)", RFC 3863, August 2004.
7. Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M. and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.
8. Rosenberg, J., "A Presence Event Package for the Session Initiation Protocol (SIP)", RFC 3856, August 2004.
9. Schulzrinne, H., Gurbani, V., Kyzivat, P. and J. Rosenberg, "RPID: Rich Presence: Extensions to the Presence Information Data Format (PIDF)", draft-ietf-simple-rpid-08, July 16, 2005.
10. Roach, A., "Session Initiation Protocol (SIP)-Specific Event Notification", RFC 3265, June 2002.
11. Niemi, A., “Session Initiation Protocol (SIP) Extension for Event State Publication ", RFC 3903, October 2004.
12. Rosenberg, J., "Presence Authorization Rules", draft-ietf-simple-presence-rules-03, July 18, 2005.
13. Khartabil, H., "An Extensible Markup Language (XML) Based Format for Event Notification Filtering" draft-ietf-simple-filter-format-05.txt, March 15, 2005.
14. Schulzrinne, H., “Composing Presence Information”, draft-schulzrinne-simple-composition-00, July 10, 2005
15. Campbell, B. and J. Rosenberg, "Session Initiation Protocol Extension for Instant Messaging", RFC 3428, September 2002.