RC24459 (W0801-004) January 2, 2008
Computer Science

IBM Research Report

SERvartuka: Dynamic Distribution of State to Improve SIP Server Scalability

Vijay A. Balasubramaniyan[1], Arup Acharya[2], Mustaque Ahamad[1], Mudhakar Srivatsa[2], Italo Dacosta[1], Charles P. Wright[2]

[1]College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

[2]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

SERvartuka*: Dynamic Distribution of State to Improve SIP Server Scalability

Vijay A. Balasubramaniyan† Arup Acharya‡ Mustaque Ahamad† Mudhakar Srivatsa‡
Italo Dacosta†
Charles P. Wright‡

†College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA
{vijayab,mustaq,idacosta}@cc.gatech.edu
‡IBM Research T.J. Watson, Hawthorne, NY 10532, USA
{arup,msrivats,cpwright}@us.ibm.com

Abstract

A growing class of applications, including VoIP, IM and Presence, are enabled by the Session Initiation Protocol (SIP). Requests in SIP typically traverse through multiple proxies. The availability of multiple proxies offers the flexibility to distribute proxy functionality across several nodes. In particular, after experimentally demonstrating that the resource consumption of maintaining state is significant, we define the problem of state distribution across multiple nodes when the goal is to increase overall call throughput. We first formulate this as an optimization problem and then derive a distributed algorithm from it. This distributed algorithm leads to the design and evaluation of SERvartuka, a more scalable SIP server that dynamically determines the number of SIP requests for which the server is stateful while delegating state maintenance for the remainder of the requests to a server further downstream. This design is in contrast to existing SIP servers that are statically configured to either be stateless or stateful and therefore result in sub-optimal call throughput. We implement SERvartuka on top of OpenSER, a commercial SIP proxy server and measure performance benefits of different server configurations. An example of our results is a 20% percent increase in call throughput when using our algorithm for a configuration of two servers in series.

1. Introduction

The Session Initiation Protocol (SIP)[12] is a control plane protocol that is used in connection setup and teardown for a variety of applications including VoIP, IMS[3], Presence[11] and now 3GPP[10]. In addition, there are proposals for using SIP as an off path signaling mechanism for any kind of data or media session[5]. In SIP, connection requests traverse through an application overlay of proxy servers each of which performs some setup function. These functions include host discovery, routing, maintaining state and authentication. As more applications adopt SIP for the connection handshake, the functionality provided by SIP servers will grow.

The traditional approach to supporting this functionality is to assign each function to a particular server in the application overlay. If the number of servers exceed the functions that need to be provided, then core servers provide the necessary functionality while the remaining simply route the request. If the functions outnumber the servers then certain servers perform multiple functions. In either case this assignment is statically decided and in this paper we show that this leads to suboptimal throughput. Instead we propose a mechanism to dynamically distribute functionality across the servers. Each server then provides a particular functionality only for a fraction of the requests traversing through the server network.

We profiled OpenSER[1], an open source SIP Server, at low call loads to experimentally measure the processing resources consumed when a VoIP call is serviced. Our measurements indicate that resource consumptions vary significantly based on the functionality being provided. In particular creation, maintenance and deletion of call related state is one of the most significant consumers of CPU resources. A server that maintains state is known as a stateful server and one that does not is known as a stateless server. Such state is used by a server to maintain context across a set of messages. Depending on the context the state maintained is used to provide a variety of functionality includ-

*Sanskrit, adapted to all seasons.

ing absorbing unnecessary retransmissions and providing accounting services. We further measure CPU utilization of the OpenSER server for state maintenance as the call load increases. Again our measurements indicate that the maximum call load that can be supported statefully is significantly lower than what can be supported statelessly.

We then model state distribution as an optimization problem and this leads to two results. First that statically configuring a set of servers to be stateful or stateless to all calls (as is done now) will lead to sub-optimal call throughput. Second that we can increase call throughput significantly by distributing state across the servers. Each server then maintains state only for a fraction of requests while remaining stateless for the remaining requests. If we can ensure that each call request has state maintained at least at one of the servers in the system, then the system as a whole is stateful for the set of requests passing through it. This motivates the creation of SERvartuka, a SIP server that implements a state distribution algorithm that tries to achieve optimal call throughput for any configuration of servers. Each SERvartuka server dynamically reconfigures the fraction of requests that it maintains state for such that the system as a whole provides higher call throughput. For a simple hierarchy that contains two servers in series, we show an increase in throughput of 15% when state distribution is determined dynamically using our algorithm compared to a configuration where a server statically decides if it operates in stateful or stateless mode.

The following are the major contributions of this paper.

- Detailed CPU resource consumption profiles of a SIP server,
- Identifying state maintenance as a major source of CPU consumption,
- Providing an optimization formulation for the state distribution problem
- Creation of a dynamic state distribution algorithm that allows one server to maintain state associated with a call while other servers handle the call statelessly,
- Design, implementation and evaluation of SERvartuka, a SIP server that distributes call state to enhance overall system throughput.

Although we specifically look at state distribution, we contend that new ways for distributing functionality for connection requests must be explored to achieve close to optimal throughput in SIP proxy overlays. The rest of the paper is as follows: In Section 2, we provide a quick overview of SIP. Our detailed evaluation of a SIP server is presented in Section 3. We create a formulation for the dynamic state distribution formulation in Section 4 and realize it in a decentralized fashion in Section 5. We evaluate SERvartuka performance is Section 6. Related work is described in Section 7 and we conclude the paper in Section 8

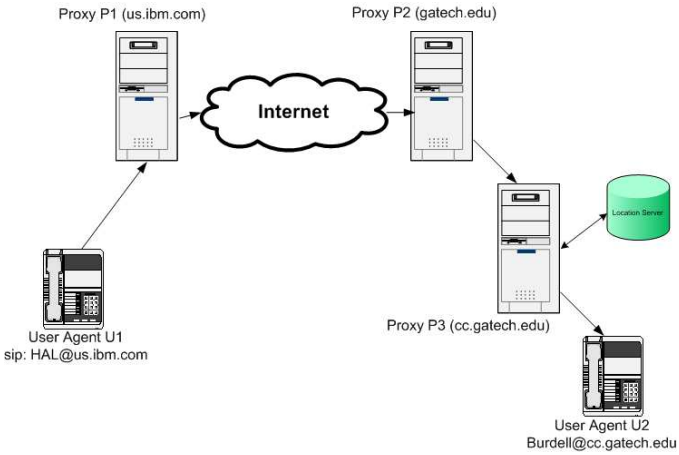

Figure 1. VoIP Call Setup

2 Background

This section starts with a basic background of VOIP before going to concepts relevant to this paper such as SIP application state and server functionality.

2.1 VoIP Fundamentals

In order to make VoIP calls with SIP, analogous to telephone numbers, a SIP URI (Uniform Resource Identifier) is used to communicate with a user. For example a user HAL might be associated with the SIP URI *sip:HAL@us.ibm.com*. Multiple devices can be registered to the same URI. SIP then uses an application overlay consisting of proxy servers and location services to locate the right device to send the call request to. A typical VoIP call traversal is shown in figure 1.

When user Hal, *sip:HAL@us.ibm.com*, calls user Burdell, *sip:Burdell@cc.gatech.edu*, the call request message is sent to the server responsible for the *us.ibm.com* domain, proxy server $P1$. $P1$ determines how to route the call to the server ($P2$) responsible for Burdells domain. The request is then sent across to $P2$ over the internet backbone. $P2$ is responsible for the top level domain *gatech.edu* and it decides where to route the call request message by determining the sublevel domain that houses the called party. In this case *sip:burdell@cc.gatech.edu* is located in the *cc.gatech.edu* domain and the call request is now routed to $P3$ responsible for this sub level domain. $P3$ then contacts a database generically called a location service to determine the current IP address of the phone associated with Burdell. $P3$ then routes the call setup request to Burdell's phone (user

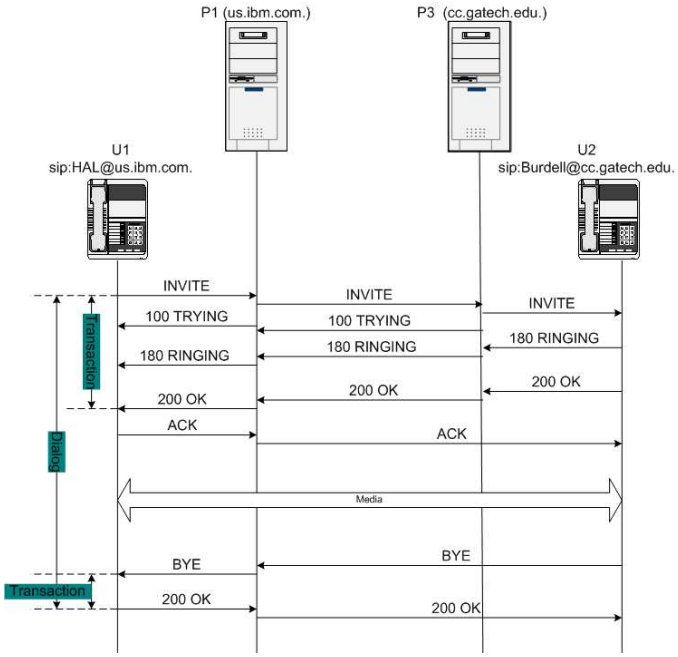

Figure 2. SIP Trapezoid

agent $U2$) which then accepts or rejects the call. Once the location (IP address) of the end points are determined in the above fashion, and the call is accepted, the media is routed directly between them and does not need to traverse through the servers.

As the above call setup indicates, SIP servers are typically organized into a hierarchy [14], which means that within a single domain a request will traverse through multiple servers. This is the basis for one of our design assumptions: the existence of multiple servers within a single domain over which state can be distributed.

2.2 SIP Application State

The messages that are exchanged when a call is setup and torn down are shown in figure 2 which is a simplified version of figure 1 obtained by abstracting away the intermediate hop by hop proxy, $P2$, and the location server. As shown in figure 2, the entire call is composed of a series of transactions. A transaction comprises all messages starting from a request till its final response. Two such transactions are shown in figure 2, the call setup transaction and the call tear-down transaction. The call setup transaction starts with the $INVITE$ message and includes all messages exchanged till the 200 OK final response. The intermediate 1xx messages are provisional responses and are used to indicate progress. A dialog comprises all transactions that are part of the entire call (or dialog). In figure 2, the dialog comprises both the call setup and call tear-down transactions.

A server that maintains state for the duration of a transaction is known as a transaction stateful server. $P1$ and $P3$ are both transaction stateful servers. On the other hand, if the server maintains state for the length of the entire dialog, it is dialog stateful. Only $P1$ is dialog stateful, which explains why the BYE and the final 200 OK message continues to pass through it. By virtue of maintaining state, transaction stateful servers absorb retransmissions, handle forking requests, redirect requests and registrations. Dialog stateful servers are used when state needs to tie down the $INVITE$ transaction to subsequent transactions within the dialog such as a subsequent $REINVITE$ or BYE transaction. This is useful for servers that maintain accounting information, or conference servers that need to maintain the conference parameters, for new users as and when they join the conference. Unless otherwise specified we use state to mean transaction state. Stateless servers maintain no state. Their chief advantage is the ability to process requests very quickly. Most widely used proxy servers including OpenSER can be both stateless and stateful and can be statically configured to behave in one of these modes.

This state is different from the state maintained by lower level protocols such as TCP or IP as this state is purely application related state.

3 SIP server evaluation

We extensively profiled OpenSER, a representative SIP server at low call loads to determine the CPU consumptions for the various functionality that it provides.

3.1 CPU Resource Consumption Profiles for SIP Server

The experimental setup consists of a set of SIPp[2] clients that send requests to a set of SIPp servers through a proxy running OpenSER[1]. We profile the functionality of the proxy using OProfile[8].

OpenSER was configured to profile five typical server modes of operation. Each mode represents a service that the server is providing for the call and successive modes provide additional service (and are thus resource wise costlier). OpenSER does a lookup in order to make a translation from the URI to the IP address of the endpoint. The various modes are as follows:

1. *Stateless with No Lookup*: No call related state is maintained as a result of handling the call message. Also, the message contains sufficient information such as the IP address within the SIP URI of the endpoint, so no lookup is necessary.

2. *Stateless with Lookup*: No call state is maintained as in the previous case but a database lookup is performed

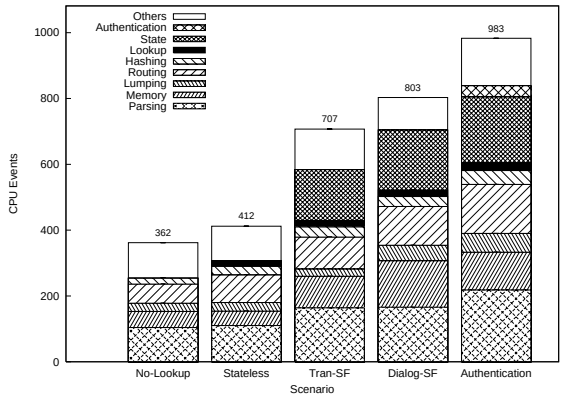

Figure 3. Server Functionality costs

to map the URI to an IP address.

3. *Transaction Stateful with Lookup*: In this case, a lookup is performed to map the URI in the message to an IP address. In addition, state is maintained only for individual transactions.

4. *Dialog Stateful with Lookup*: Here too the IP address is looked up. In addition the state is maintained for the entire call duration, spanning multiple transactions

5. *Dialog Stateful with Authentication*: In this mode, all the functionality of the previous modes are executed. In addition the proxy checks the credentials of the client.

Each run has the server configured in one of the above modes and two SIPp clients make and break calls through the server at the rate of 1 call per second for 10 minutes. During this time OProfile profiles the various functionality blocks of OpenSER. The results for these runs is as shown in figure 3.

The bar graph in figure 3 shows that as the server executes more functionality, it results in higher CPU consumption. This is expected but the size of increase is noteworthy. In the most basic configuration where the server is stateless and performs no lookup, the CPU cycles consumed are approximately one third compared to a server that performs lookups, maintains transaction state and performs authentication. This clearly shows that control plane costs for simple call establishment vary widely with the complexity of the service being provided by the SIP server. In our experience, we find most SIP vendors providing blanket throughput specifications of the number of calls per second. From our graphs, we however see that the throughput could differ by a factor of three depending on the functionality executed by the server.

Compared to the no lookup bar, all other cases (Stateless to Authentication) have lookup processing which involves either querying a DB or an internal cache. This is reflected as a thin lookup band in figure 3. Similarly we see increase in CPU cycles for state maintenance and authentication. Most of the granular functionality performed by the server also monotonically increases with scenario/service. In particular we see costs associated with parsing, memory and state increasing significantly with service provided. Parsing in most SIP servers is lazy which means they parse only as much of the message that is required to be able to either dispatch the request to the next hop or create an appropriate response or both. Richer services require more of the message to be parsed. Lookups do not change the parsing costs significantly as the Request-URI always needs to be parsed to decide whether a route lookup needs to be done. Hence parsing costs in the first two scenarios consume almost the same resources. To create/maintain state, however, more headers need to be parsed. This is because state maintenance requires the request to be uniquely identified. This is done by hashing together a set of fields including *From*, *To* and all these headers thus need to be parsed.

State maintenance requires allocation and deallocation of memory and this results in an increase in the memory processing across the last three scenarios. This extra state causes increased parsing and increased memory processing at the server. From the graph we can see that, at a low request rate of 1 call per second, being dialog stateful or transaction stateful is 2 times or 1.75 times, respectively, costlier that being stateless. State maintenance thus is a good candidate function to distribute and we further explore how stateful and stateless servers behave when the call rate increases.

3.2 Impact of Maintaining State with Increasing Call Rates

The goal of the experiment was to determine how stateful servers behave with increasing call load as opposed to stateless servers. However at high loads, a single SIPp UAC (client) and UAS (server) reaches 100% CPU utilization at about the same call rate as OpenSER and therefore skews the measurements. We therefore split this load among four machines, two running the SIPp server scenario and two running the SIPp client scenario. Associated with the SIPp servers are two URIs which the two SIPp clients make calls to respectively. The OpenSER database is populated with these two URIs and is configured to run in 2 modes (i) stateless with lookup and (ii) transaction stateful with lookup. In order to determine the complete behavior of these servers, the call load was increased till the servers reached 100% CPU utilization. To ensure that servers were saturated only due to CPU utilization we configured OpenSER with adequate amount of memory (1024 Mb) and used Gigabit ethernet interfaces on a private network.

The SIPp clients generated a load starting with 20 calls

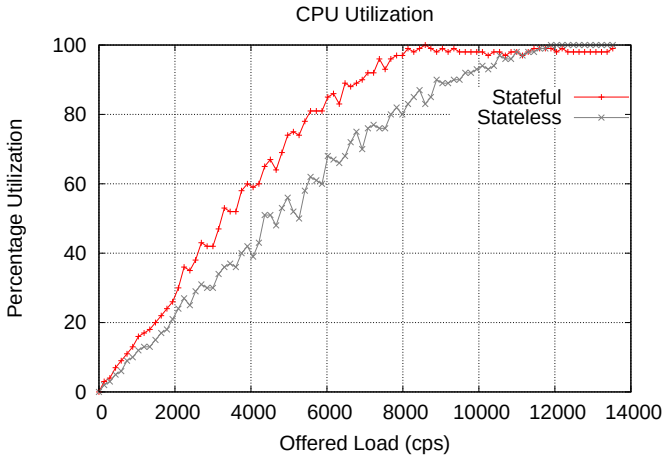

Figure 4. CPU Increasing Load Utilization

per second and increased this load steps of 20 calls per second. Saturation of the OpenSER server was determined by observing that CPU utilization of the server was at 100% (using top logs) and that the call throughput at the SIPp servers did not increase with increasing call load generated at the SIPp clients. At the onset of the saturation point there is also a large increase in SIP 500 Server Busy messages and increased retransmission of call requests from the SIPp client. Top logs were kept throughout the run to ensure that when OpenSER saturated, the SIPp clients were operating far below 100% CPU utilization. The results are shown in figure 4.

As seen in figure 4, as the call rate increases, the statefully configured server's CPU utilization increases at a faster rate when compared to the statelessly configured server. The stateful server saturates at $\approx$ 10360 calls per second while a stateless server saturates at $\approx$ 12300 calls per second. This difference between stateful and stateless servers forms the basis for obtaining higher throughput by distributing state maintenance functions. In the next section we create a state distribution algorithm starting with providing an intuition towards why state distribution will potentially increase call throughput.

4 Distribution of State

Consider a simple case where a request traverses through two servers in series ($S1$ and $S2$), within a domain. This is similar to the scenario depicted in figure 1 where the call traverses through proxy servers responsible for the *cc.gatech.edu* domain and the *gatech.edu* domain. The possible configurations that these servers can be arranged in are (i) both stateful, (ii) one stateful and the other stateless, and (iii) both stateless. In case (i), each server by virtue of being stateful will maintain state for each request that passes through it. As a result, the maximum number of such requests that it will be able to service will be equal to the saturation limit of a stateful server, say T^{SF} (from our experiments this is $\approx 10360cps$). Since both servers see the same request load the maximum system throughput will be around $10360cps$. Both servers will also be operating at 100% CPU utilization. Case (ii) is when one of these servers is configured to be stateless. Such a system will continue to have maximum throughput of $10360cps$ because the stateful server is the bottleneck and hence will dictate the overall throughput. The stateful server will operate at 100% utilization while the stateless server will be underutilized. Case(iii) is when both servers are stateless, in which case the maximum system throughput will be the saturation limit of a stateless server, T^{SL} (from our experiments $\approx 12300cps$). This throughput will be significantly higher than the first two cases but here state is not maintained in the system at all. Such a system is unusable for call requests that need to maintain application state.

Therefore, for the two servers in series, where each server is pre-configured statically as stateful or stateless, the maximal throughput will be T^{SF} for call requests that need to maintain application state. However as observed in case (ii), $S2$ is under utilized, with $S1$ being the bottleneck. If server $S1$ offloaded half of its requests to be handled statefully at $S2$, while being stateless for these requests then we could potentially ensure that both servers are equally utilized at each stage thereby being able to support a larger call load. A state distribution algorithm then has to satisfy the following requirements. First, the call related state must be maintained by some node on the call request path. Other nodes forward the call statelessly and thus do not incur the computational overhead associated with state maintenance. Second, we must dynamically determine which node in a path should maintain the call state such that the system is able to handle an incoming call as long as the there are resources available to handle that call in a stateful manner. In order to characterize the nature of such an algorithm we start by modeling the problem as an optimization problem.

4.1 Formulation of State Distribution as an Optimization Problem

The SIP proxy servers (nodes) can be arranged in any network topology and we can represent the set of nodes as a graph. Call setup requests enter the system at some node in this graph, traverses a set of nodes along links in the graph and exits at a proxy node that forwards the request either to the internet backbone or to the call recipient's user agent. A topology can have many entry nodes. This is equivalent to having an imaginary source node 0 which generates calls to all the entry nodes. Similarly there might be multiple exit nodes in the system and this is equivalent to a single

sink node z to which all exit nodes route calls through. The advantage of this is that the network can be represented as a single source, single sink topology and provides a cleaner formulation without any loss in generality.

Consider a server in the system, i, and a particular call request that path through node i. There is an upstream node from which the call was routed to i, represented as l. This call is representative of all calls routed from l to i, t_{li}. For the rest of the discussion all call traffic terms represent call rate (load). Therefore t_{li} is the call load from node l to i. t_{li} has two components, calls for which state has already been maintained, t_{li}^{ASF} and calls for which state has yet to be maintained, $t_{li}^{\overline{ASF}}$. Therefore $t_{li} = t_{li}^{ASF} + t_{li}^{\overline{ASF}}$. Now the total aggregate flow incoming at node i, t_i is the sum of the individual flows t_{li}, $t_i = \sum_{l \in US_i} t_{li}$. Similarly we have expressions for the total call load that has state already maintained, $t_i^{ASF} = \sum_{l \in US_i} t_{li}^{ASF}$ and the total call load for which state is yet to be maintained, $t_i^{\overline{ASF}} = \sum_{l \in US_i} t_{li}^{\overline{ASF}}$. US_i is the set of all possible upstream servers for i.

At node i the incoming load t_i will be redistributed to all possible downstream paths. For a downstream server d if the call load is t_{id} and DS_i represents all downstream servers for i, then by conservation of flow, $t_i = \sum_{d \in DS_i} t_{id}$. t_{id} consists of three components, t_{id}^{FASF} - the call flow from i to d that has state already maintained at some node previous to i, t_{id}^{SF} - the call flow from i to d for which i maintains state, and $t_{id}^{\overline{ASF}}$ - the call flow from i to d for which state is yet to be maintained. The flow constraints mandate the following equations:

$$t_i^{ASF} = \sum_{d \in DS_i} t_{id}^{FASF}$$

$$t_i^{\overline{ASF}} = \sum_{d \in DS_i} t_{id}^{SF} + \sum_{d \in DS_i} t_{id}^{\overline{ASF}}$$

For downstream node d's, the amount of flow that is already stateful from i, $t_{id}^{ASF} = t_{id}^{FASF} + t_{id}^{SF}$. A similar split would have occurred for traffic from upstream node l to i. Therefore $t_{li}^{ASF} = t_{li}^{FASF} + t_{li}^{SF}$. This ensures that at each node the decision variables are t_{id}^{FASF}, t_i^{SF} and $t_i^{\overline{ASF}}$. The constraints discussed so far ensure that state is handled in a mutually exclusive fashion. However, we also need to ensure that by the time the requests exit the system, state is maintained in at least one of the nodes. Therefore the number of calls for which state is yet to be maintained for any flow from an exit node, k to the imaginary sink node should be 0, $t_{kz}^{\overline{ASF}} = 0$.

From section 3.2, we know that the CPU utilization for handling requests statefully is different from handling requests statelessly. The number of calls being handled statefully at node i is $\sum_{d \in DS_i} t_{id}^{SF}$ and the number being handled statelessly is $\sum_{d \in DS_i} (t_{id}^{\overline{ASF}} + t_{id}^{FASF})$. Thus, if we assume that the CPU utilization due to stateful request forwarding at node i, represented by U_i^{SF}, is dependent on the number of requests handled statefully, then $U_i^{SF} = f^{SF}(\sum_{d \in DS_i} t_{id}^{SF})$. Similarly the stateless utilization is $U_i^{SL} = f^{SL}(\sum_{d \in DS_i} (t_{id}^{\overline{ASF}} + t_{id}^{FASF})$. The total CPU utilization at node i is the sum of these two utilizations and this should not exceed 100%. If the utilizations are normalized then this translates to $U_i^{SF} + U_i^{SL} \leq 1$

The above equations form the constraints of the system. The goal is to determine a distribution of state such that the throughput is maximized. The throughput is basically the call flow from the imaginary source node 0 to all the entry nodes. At thispoint state has to be maintained for all calls, $t_{0a}^{FASF} = 0, t_{0a}^{SF} = 0, \forall\, a \in$ entry node. We are then trying to maximize the sum of all flows from the imaginary node to the entry nodes, $\sum_{a \in \text{entry node}} t_{0a}^{\overline{ASF}}$. The complete linear problem is as follows:

$$\text{Maximize} \sum_{a \in \text{entry node}} t_{0a}^{\overline{ASF}} \tag{1}$$

Subject to

$$t_{0a}^{FASF} = 0, t_{0a}^{SF} = 0, \forall a \in \text{entry node}$$

$$\sum_{l \in US_i} (t_{li}^{FASF} + t_{li}^{SF}) = \sum_{d \in DS_i} t_{id}^{FASF} \tag{2}$$

$$\sum_{l \in US_i} t_{li}^{\overline{ASF}} = \sum_{d \in DS_i} t_{id}^{SF} + \sum_{d \in DS_i} t_{id}^{\overline{ASF}} \tag{3}$$

$$t_{kz}^{\overline{ASF}} = 0, \forall k \in \text{exit node}$$

$$f^{SF}(\sum_{d \in DS_i} t_{id}^{SF}) + f^{SL}(\sum_{d \in DS_i} (t_{id}^{\overline{ASF}} + t_{id}^{FASF})) \leq 1 \tag{4}$$

Solving this optimization formulation would help in identifying how much state needs to be maintained at each server in order to maximize throughput. We have not considered including routing constraints in the formulation. The formulation assumes that a server can decide which direction to send a request among the many downstream paths possible and that all the call destinations are reachable by any of these paths. However, in real world scenarios, the call request will traverse a path determined by underlying network routing mechanisms. Adding routing constraints is easy. In addition to the constraints specified in equation (1) we need to add constraints that relate the incoming flow, at any node to each of the outgoing flows at that node. That is at a node i we need to introduce constraints of the form $t_{id} = \phi_{id} * t_i, 0 \leq \phi_{id} \leq 1$, where ϕ_{id} is the fractional split of the incoming load into the downstream path d. In addition we need to ensure flow conservation by requiring that all the fractions sum up to 1, $\sum_{d \in DS_i} \phi_{id} = 1$. The outgoing flow t_{id} is then some predetermined fraction of the total

incoming flow t_i. In the design of SERvartuka we also take into account the routing constraints.

From figure 4 we see that both the stateful utilization and the stateless utilization are linear and pass through the origin. We can therefore approximate these functions to be of the form $f^{SF}(x) = \frac{x}{T^{SF}}$ and $f^{SF}(x) = \frac{x}{T^{SL}}$, where T^{SF} and T^{SL} are the stateful and stateless threshold, respectively. From our experiments $T^{SF} \approx 10360$ and $T^{SF} \approx 12300$. In such a case the optimization formulation is a linear programming (LP) problem which can be solved efficiently. For the rest of the paper we assume that the system can be represented as an LP. For the two server in series scenario we can now use the formulation to calculate the optimal throughput with the threshold values specified. The optimal solution is one where each server maintains $5620cps$ statefully and the remaining $5620cps$ statefully, giving a total throughput of $11240cps$ which is higher than the throughput of the static configuration ($\approx 10360cps$). This increase in throughput is because both servers are now being equally utilized, thus leading to higher throughput. Reducing it to an optimization problem implies that an algorithm that solves the optimization problem will yield an optimal throughput solution. An algorithm thus needs to try and emulate the behavior specified in the formulation and we outline how we determine such an algorithm in the next section.

4.2 Towards a State Distribution Algorithm

The solution to the LP will provide the values of t_{id}^{FASF}, t_{id}^{SF} and $t_{id}^{\overline{ASF}}$ for each node i. However, these are the values when the incoming call flow is maximum. We need to determine an operating point at any incoming load such that it provides a feasible distribution of state at that load. In addition this operating point should be feasible for all incoming loads till the maximum possible incoming load. Consider the equations of the LP from Section 4.1. The utilization constraint (4) primarily governs the amount of state maintained at each server. Equations (2) and (3) provide basic flow constraints and can be taken care of fairly easily in practice. The routing constraints can also be taken care of by assuming that we do not have the liberty of deciding routing paths in the final algorithm. Assuming that the utilization functions are linear, the utilization constraint can be rewritten as:

$$\frac{\sum_{d \epsilon DS_i} t_{id}^{SF}}{T^{SF}} + \frac{\sum_{d \epsilon DS_i} (t_{id}^{\overline{ASF}} + t_{id}^{FASF})}{T^{SL}} \leq 1 \quad (5)$$

At node i the incoming load is $\sum_{l \in US_i} t_{li}$ which by flow conservation is equal to the outgoing load $\sum_{d \in DS_i} t_{id}$. We can therefore think of the incoming flow as composed of many individual flows t_{id}. t_{id} as mentioned earlier consists of three components t_{id}^{FASF}, t_{id}^{SF} and $t_{id}^{\overline{ASF}}$, where $t_{id} = t_{id}^{FASF} + t_{id}^{SF} + t_{id}^{\overline{ASF}}$. Substituting for $t_i d^{FASF} + t_{id}^{\overline{ASF}}$ in (5) and rearranging the terms we get:

$$\sum_{d \in DS_i} t_{id}^{SF} \leq \frac{1 - \frac{\sum_{d \in DS_i} t_{id}}{T^{SL}}}{\frac{1}{T^{SF}} - \frac{1}{T^{SL}}} \quad (6)$$

(6) shows that the total state that should be maintained is some function of the incoming load. In addition the amount of requests that each server maintains statefully cannot be more than the total incoming request flow at that node.

$$\sum_{d \in DS_i} t_{id}^{SF} \leq \sum_{d \in DS_i} t_{id} \quad (7)$$

The above flows are in terms of summation of individual flows. For clarity we can use $t_i = \sum_{d \in DS_i} t_{id}$, where t_i represents the total incoming flow, and $t_i^{SF} = \sum_{d \in DS_i} t_{id}^{SF}$, where t_i^{SF} represents the flow for which node i maintains state. If $\alpha = \frac{1}{T^{SF}}$ and $\beta = \frac{1}{T^{SL}}$, then the above two equations can be rewritten as $t_i^{SF} = \min\{t_i, \frac{1-\beta t_i}{\alpha - \beta}\}$. The first term is lesser than the second when $t_i \leq T^{SF}$. This yields:

$$t_i^{SF} = \begin{cases} t_i & \text{if } t_i \leq T^{SF}, \\ \frac{1-\beta t_i}{\alpha - \beta} & \text{if } t_i > T^{SF}. \end{cases} \quad (8)$$

Equation (8) suggests that as long as the incoming flow is lesser than the stateful saturation limit T^{SF} a server can maintain all the requests that are not yet stateful as stateful. Once the incoming flow crosses T^{SF} the server begins to relinquish state as specified by the second case. The formulation provides an operating point if we were considering state maintained as a whole without the individual flows. The behavior of the individual flows should be such that the overall state maintained satisfies the equation (8). Once the incoming flow crosses T^{SF} then each of these individual flows need to relinquish state to servers downstream such that the total amount of state maintained at the server does not exceed the second case specified in equation (8). Thus upstream servers can relinquish state further downstream, until the exit nodes. Since these nodes have no downstream path to relinquish state, they keep maintaining state for increasing number of calls until they are close to maximum utilization at which point they communicate back an overload message to the upstream servers. When all downstream paths are saturated and the server is itself saturated it will communicate the overload message to servers further upstream and finally the system as a whole gets saturated. The next section details the actual implementation of SERvartuka on an opensource proxy OpenSER.

5 SERvartuka

The SERvartuka algorithm realizes the state distribution algorithm outlined in the previous section by calculating the number of call setup requests for which a given server maintains state for each downstream path. Equation (6) shows there exists a relation between the overall state maintained by a server and its input load. Measurements in any system cannot be instantaneous. Hence the incoming load and the state maintained by the server needs to be monitored periodically. Let the monitored incoming load at a server i be $obsv(t_i)$ and the load that is stateful be $obsv(t_i^{SF})$. From equation (6), based on the monitored incoming load we can calculate the amount of state that can be maintained by the system in order to satisfy the feasibility constraint. Let this be $calc(t_i^{SF})$. If $calc(t_i^{SF}) \geq obsv(t_i^{SF})$ then the system is maintaining lesser state than it feasibly can and therefore does not need to change anything. If $calc(t_i^{SF}) \leq obsv(t_i^{SF})$ then the system needs to relinquish state. In SERvartuka we relinquish state by choosing downstream paths on which state can be delegated to a server further downstream. This allows the possibility of accommodating extra flow on a particular path for which state has to be maintained, at this node, by reducing the state maintained for some other flow (as long as downstream servers on that flow can continue to take up the delegated state).

There could be multiple reasons that state has to be maintained at a particular node for a particular path flow of call requests: It could be the exit node for those call requests or all the servers downstream in that path are fully utilized. The second case is required because we assume routing constraints are also in place. In the real world, call request paths are typically determined by some underlying protocol and therefore we assume we cannot reroute the calls. In this case the relationship between the call load that is maintained statefully and the incoming call load (higher than T^{SF}) can be specified in terms of its individual call flows as:

$$\sum_{d \in DS_i} t_{id}^{SF} \leq \frac{1 - \beta \sum_{d \in DS_i} t_{id}}{\alpha - \beta}, \text{if } t_i > T^{SF} \qquad (9)$$

On a per flow basis this can be written as $t_{id}^{SF} = \frac{1}{n*(\alpha-\beta)} - \frac{\beta * t_{id}}{\alpha - \beta}, \forall\, d \epsilon DS_i$, if there are n downstream paths. Summing both sides over all d we get back equation (6). This implies that each flow can reduce a certain amount of state so that in the aggregate the effect is the same as Equation (6). However as discussed before it may not be possible to relinquish state along all paths. Assume that i is an exit node and it also has n downstream paths. In addition let k downstream paths be unsaturated while the remaining $n - k$ paths have been saturated (they have sent overload messages to i). Without any loss in generality we can order the paths such that $(i, 1)$ to (i, k) are unsaturated and $(i, k+1)$ to (i, n) are saturated. For all saturated nodes $k+1 \leq p \leq n, \overline{c_{ip}^{ASF}}$ is the load that was maintained statefully by the downstream path at the time of overload. As the path is overloaded it will not be able to maintain state for a higher call load than $\overline{c_{ip}^{ASF}}$. Therefore the amount of state that this server will need to accommodate for this path will be $t_{ip} - \overline{c_{ip}^{ASF}} - t_{ip}^{FASF}$. For the flow path for which i is the exit node, all state that is yet to be maintained must be maintained at i and this value is $t_{iz} - t_{iz}^{FASF}$. Expanding equation (9):

$$t_{i1}^{SF} + \ldots + t_{ik}^{SF} + t_{ik+1}^{SF} + \ldots + tin^{SF} + t_{iz}^{SF} \leq \frac{1}{\alpha - \beta} - \frac{\beta * (t_{i1} + \ldots + t_{ik} + t_{ik+1} + \ldots + t_{in} + t_{iz})}{\alpha - \beta}$$

Substituting for t_{ik+1}^{SF} to t_{in}^{SF} and t_{iz}^{SF} we get

$$t_{i1}^{SF} + \ldots + t_{ik}^{SF} \leq \frac{1}{\alpha - \beta} + (c_{ik+1}^{ASF} + \ldots + c_{in}^{ASF}) + (t_{ik+1}^{FASF} + \ldots + t_{in}^{FASF} + t_{iz}^{FASF}) - \frac{\alpha * (t_{ik+1} + \ldots + tin + t_{iz})}{\alpha - \beta} - \frac{\beta * (t_{i1} + \ldots + t_{ik})}{\alpha - \beta}$$

Except for the last term all the others are fixed and therefore can be equated to a constant value c. Thus each flow that can relinquish state will now relinquish:

$$t_{iq}^{SF} = \frac{c}{k} - \frac{\beta * t_{iq}}{\alpha - \beta}, 1 \leq q \leq k$$

This forms the basis of the SERvartuka algorithm. The dynamic SERvartuka server algorithm has two parts to it. The first part which is executed on receipt of each message is specified in algorithm 1 and the second part which is carried out periodically is specified in the algorithm 2. The algorithms are created from what has been discussed so far and are detailed and self explanatory.

6 Results and Analysis

We evaluated SERvartuka on a number of server configurations. These configurations were chosen for several reasons. First, they provide building blocks for more complex topologies and their evaluation can provide insights for larger network topologies. Second they represent some of the real world server topologies that we have seen.

6.1 Two Server Configuration

The most basic configuration where we can observe benefits from the SERvartuka algorithm is the two server configuration where the servers are arranged in series. This

```
   // myshare and msg_count obtained
      from algorithm 2
1  Determine index i of DS path for msg
   // rcv_msg_count[i] used in updating
      msg_count[i]
2  Increment rcv_msg_count[i], tot_msg_count
3  if state is not already maintained for msg
4  AND ( sf_count[i] ≤ myshare[i]
5  OR msg is part of existing transaction ) then
6  |  Increment sf_count[i], tot_sf_count
7  |  Forward msg statefully
8  else
9  |  Forward msg statelessly
10 end
```

Algorithm 1: Handle Message

```
   // α = 1/T^SF and β = 1/T^SL are known
1  Determine time elapsed, t
2  c = 0, not_ovld_count = 0
3  foreach DS path j do
4  |  Update msg_count[j], nasf_count[j], fasf_count[j]
5  |  if path j is overloaded then
6  |  |  Calculate myshare[j]
7  |  |  Update c appropriately
8  |  else
9  |  |  Increment not_ovld_count
10 |  end
11 end
12 if msg_per_sec > T^SF then
13 |  if not_ovld_count then
14 |  |  foreach path q that is not overloaded do
15 |  |  |  lt = t*c/not_ovld_count − β*msg_count[j]/(α−β)
16 |  |  |  if sf_count[q] > lt then
17 |  |  |  |  myshare[q] = lt
18 |  |  |  end
19 |  |  end
20 |  else
21 |  |  if tot_sf_count > (1−β*tot_msg_count)/(α−β) then
22 |  |  |  send overload message
23 |  |  end
24 |  end
25 end
26 foreach DS path j do
27 |  reset sf_count[j], rcv_msg_count[i]
28 end
```

Algorithm 2: Calculating myshare

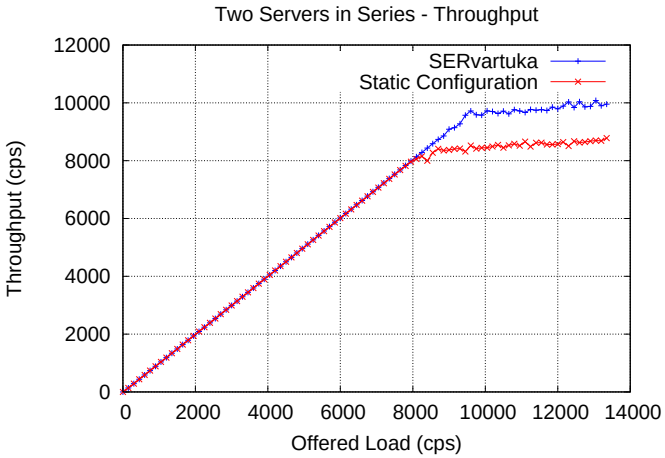

Figure 5. Two Server - Throughput

is similar to our example where requests from users in the $cc.gatech.edu$ domain traverse proxies responsible for $cc.gatech.edu$ and $gatech.edu$. The experimental setup is similar to the SIP Server Evaluation Section (Section 3). In this experiment, a set of SIPp clients send requests to the first proxy which then forwards these to the second proxy. The second proxy delivers the call requests to the SIPp servers, and the rest of the call path follows this pattern. Two runs are conducted, in the first the proxies are configured statically, and in the second the proxies are running the SERvartuka algorithm. Throughput is measured at the SIPp server. We ensure that SERvartuka is maintaining state for all requests by checking if the number of calls sent by the SIPp client is equal to the number of 100 Trying messages that it receives (see Section 2.2). The results are shown in figure 5. The static configuration saturates at $8540cps$ and SERvartuka saturates at $9790cps$ a performance improvement of 15%. Though we initially predicted that for a static two server in series configuration the maximum throughput will be equal to the maximum throughput of a single stateful server, we find that in practice it does worse. We find a similar trend for three servers arranged in series, where the static configuration throughput is $8780cps$ and SERvartuka's throughput is $10180cps$,a performance improvement is 16%. In essence, we observe that for most of these basic configurations we can expect a performance improvement of 15% - 20% when running SERvartuka in comparison with a static configuration.

6.1.1 Response Time

In addition to the SIPp server measuring throughput, it also maintains statistics of the response times for the messages it sends. For the two server in series test the results of measuring response time are shown in figure 6. The round trip

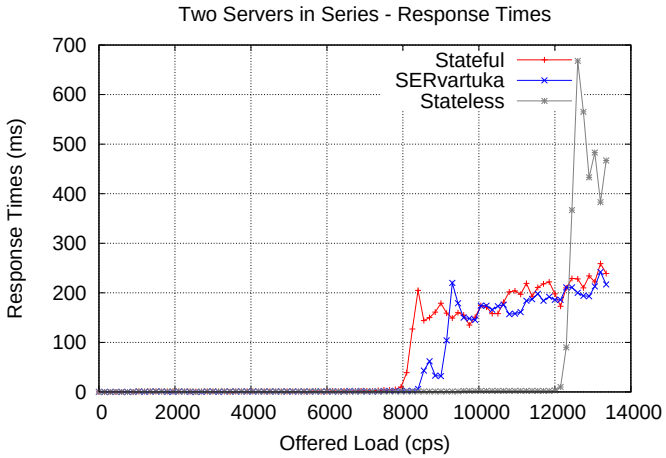

Figure 6. Two Server - Response Time

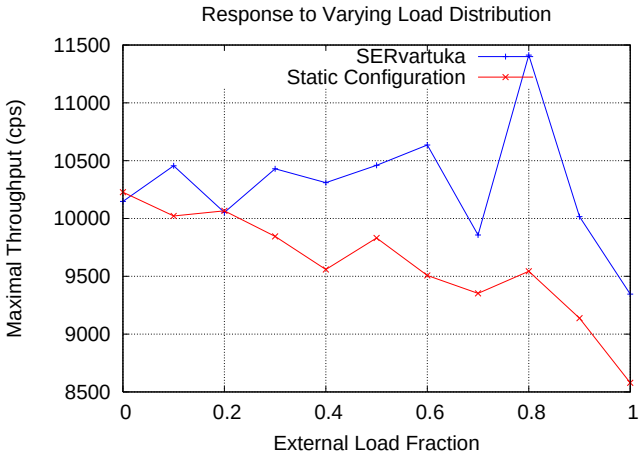

Figure 7. Changing Call Load Distribution

time from SIPp server to SIPp client is roughly $1.5ms$. The static stateful configuration by virtue of maintaining state is able to handle retransmission in a better fashion and thus bound the response time for server requests to under $200ms$ as seen in figure 6. However, it saturates at a low call rate of $8500cps$. In fact, the effects of saturation in increasing response times can be observed at $8000cps$. The static stateless configuration however is able to keep low response times till its saturation limit. Once it crosses this limit the response times becomes significantly higher. This is because any request that is lost must be reissued all the way from the SIPp server or client, thus increasing the average response time. As seen in figure 6, SERvartuka tries to bridge the gap by increasing the throughput as well as keeping the response times low. We see that the response times of SERvartuka are comparable to a stateful static configuration (under $200ms$). This is possible because one of the proxy servers in the path maintains state through the state distribution algorithm, thus allowing the system to absorb extraneous retransmissions.

6.1.2 Changing loads

For this experiment we considered a new flow path in the basic two server in series setup. In addition to the flows that can go through both servers we consider flows that terminate at the first server. Relating this to our real world analogy, users belonging to the $cc.gatech.edu$ domain can make calls to external users such as those in the $us.ibm.com$ domain, in which case call requests will traverse through both the $cc.gatech.edu$ proxy and the $gatech.edu$ proxy. In addition they can make calls to other users within the $cc.gatech.edu$ domain. In this case the call request will only traverse through $cc.gatech.edu$ proxy and will not touch the $gatech.edu$ proxy. Thus, there are two distinct call flows, the first set which we call the external call flow, and the second set which we call the internal call flow. We varied the fraction of external call load from 0 to 1 in steps of .1. When the external call load is 0, the two server in series configuration degenerates to a single serve configuration. When the external call load is 1, then it is the same as the above test cases. The results are shown in figure 7. We see that SERvartuka performs better than the static configuration for all possible loads. What this implies is that with SERvartuka, network operators do not need to worry about the flow distributions in the network. The algorithm based on the flow distribution will be able to determine the best state distribution to maximize throughput.

We see that SERvartuka and the static configuration are similar for the degenerated single server case and thereafter SERvartuka starts having better throughputs as the external load fraction increases until it hits a peak when the external load is 80% and the internal load is 20%. It is at this distribution that we get the maximum performance benefit over the static configuration. From our measurements, for the $80-20$ distribution, we see that the static configuration can only handle $9540cps$ while SERvartuka can handle $11410cps$, giving a performance improvement of close to 20% and the ability to handle 1500 extra cps. This behavior is predicted precisely by the LP which says that the throughput is maximal for such a distribution (the LP predicts a value of $11960cps$). This test case also shows that SERvartuka can handle multiple asymmetrical flows.

6.2 Three Server Configurations

As mentioned earlier we are able to perform 16% better in the three server in series configuration. The other possible configuration for three servers is the load balancing case where one proxy server forks requests along two par-

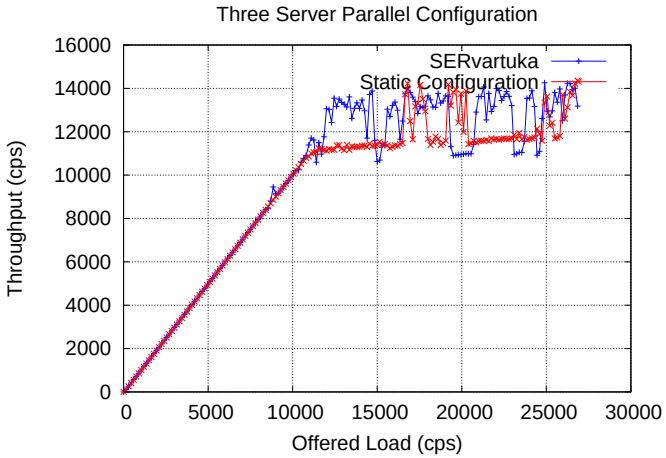

Figure 8. Three Server Parallel Configuration

allel paths. In this case the first server is typically maintained statelessly and the other two servers are maintained statefully. From the LP (and also intutively) in this configuration we cannot do better than servers that have been statically preconfigured. In our algorithm the first server should relinquish all of its state to the two servers that it forks to. In order to test this we setup an experiment with SIPp clients sending call requests to SIPp servers through a load balancing arrangement of servers. The SIPp clients are partitioned to send requests equally along the upper and lower forks, that is the first server sends half its incoming load to the upper fork proxy and the remaining half to its lower fork proxy. Two runs are conducted: (i) where the servers are configured statically and (ii) where the servers are running SERvartuka. The results are as shown in figure 8. Though the behavior near saturation is more erratic than previous scenarios the throughput of the static configuration is $11990cps$ and that of SERvartuka is $12830cps$. The behavior of SERvartuka is again significantly higher than the static case. However in this case we are unaware of why SERvartuka does so much better than the static case. We need to profile the internal workings of the system at these loads to determine answers and this is part of future work.

When servers are arranged in such a fashion it is not always necessary that the first server being stateless would lead to optimal throughputs. A scenario where it might be necessary for the first server to be stateful, is if the incoming load is unevenly split along the two paths. The other possibility is when the three servers are non homogenous. If the first server has much larger capacity than the two downstream paths then it might be beneficial for it to maintain some state or even all state. In all such cases SERvartuka does better than the static configuration.

In essence these configurations (series and parallel blocks) are the basic building blocks of any network topology. We have seen that SERvartuka performs better in most configurations, and in the worst case does as well as the static configurations that exist today. In addition when multiple flows are involved or the load distribution varies the algorithm will try and optimize the distribution of state to maximize throughput. Though state has been considered here we can potentially distribute any other functionality and we have seen significantly larger improvements when we tried distributing authentication.

7 Related Work

In [4], the authors have recognized the scalability benefits of a transaction-stateless processing and have defined an algorithm that determines whether a request must be handled statefully if (a) the network link is lossy (BER $> 10^{-5}$), and the CPU utilization is (i) either low ($< 60\%$), or (ii) medium ($< 75\%$) and the transaction is an INVITE or a BYE, or (b) the transaction requires forking. Our algorithm is broader in multiple aspects: our algorithm seeks to determine an optimal ratio of stateful to stateless transaction processing in the *aggregate* when the input load is greater than what can be handled statefully at 100% CPU utilization. It is a dynamic algorithm which recomputes the ratio as the total input load changes. We leverage the per-server algorithm to establish a distributed algorithm which ensures that a request is handled statefully at some downstream server ("distributing state") when upstream servers prior to that handle calls statelessly. As mentioned before, our assumption in this paper that a request needs a set of functions to be executed in it call path or before exiting a domain, some of which may require stateful processing, and thus these functions can be performed over a sequence of proxies.

In general, the performance of SIP proxies has been investigated in [9],[6], [13],[16],[7]. In [7] authors study sipd, a SIP proxy server developed in Columbia University[17], and identify bottlenecks such as parsing, string operations and database access, compare performance of thread-based vs. process-based models for request processing in sipd and compare scalability of different proxy and database access combinations. In [18], the authors point out that ability to handle transactions statelessly could be used to thwart denial-of-service attacks.

Although the notion of trading off state for performance has been studied in other contexts to some degree, e.g for coupling link-state routing information only with long-lived flows for load-sensitivizing routing, thereby reducing route flapping [15], we believe our work is one of the first to design and implement a concrete detailed algorithm for SIP server systems.

8 Conclusion and Future Work

We have experimentally evaluated the performance of a SIP server under various call scenarios. Based on this performance study, we defined the state management problem and developed a mathematical model for deriving an optimal solution. This provides insights for developing a more scalable server design by dynamically distributing state across a set of servers. We evaluate our algorithm against existing pre-configured static algorithms and show a 15% - 20% increase in the maximum call throughput that can be achieved. Our work can be extended in several ways.

- Explore implications of state distribution on security issues such as privacy and confidentiality.
- Apply these ideas to distribute other functionality in SIP such as authentication as well on other overlay network protocols

Acknowledgements

We would like to thank Erich M. Nahum and John M. Tracey from IBM Research, T.J. Watson for valuable discussions.

References

[1] Openser - the open source sip server. http://www.openser.org/, 2000.

[2] Sipp - open source traffic generator for sip. http://sipp.sourceforge.net/, 2004.

[3] G. Camarillo and Miguel. *The 3G IP Multimedia Subsystem (IMS): Merging the Internet and the Cellular Worlds, Second Edition*. John Wiley & Sons, 2006.

[4] M. Cortes, J. O. Esteban, and H. Jun. Towards stateless core: Improving sip proxy scalability. San Francisco, CA, United States, 2007. Third Generation Partnership Project (3GPP);IP Multimedia Subsystem (MS);Retransmitted messages;.

[5] P. Francis and S. Guha. Path-decoupled signaling for data, 2006.

[6] J. Janak. Sip proxy server effectiveness. *Master's Thesis, Department of Computer Science, Czech Technical University, Prague, Czech*, 2003.

[7] H. S. K. Singh and J. Lennox. Sip server scalability, 2005.

[8] J. Levon and P. Elie. Oprofile: A system profiler for linux. *Web site: http://oprofile.sourceforge.net*, 2005.

[9] E. M. Nahum, J. Tracey, and C. P. Wright. Evaluating sip server performance. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 349–350, New York, NY, USA, 2007. ACM.

[10] G. Patel and S. Dennett. Gpp and 3gpp2 movements toward an all-ip mobile network, 2000.

[11] J. Rosenberg. A presence event package for the session initiation protocol (sip), Aug 2004. RFC 3856.

[12] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. Sip: Session initiation protocol, Jun 2002. RFC 3261.

[13] H. Schulzrinne, S. Narayanan, J. Lennox, and M. Doyle. SIPstone-benchmarking SIP server performance. *Columbia University*, 2002.

[14] H. Schulzrinne and E. Wedlund. Application-layer mobility using sip. *SIGMOBILE Mob. Comput. Commun. Rev.*, 4(3):47–57, 2000.

[15] A. Shaikh, J. Rexford, and K. Shin. Load-sensitive routing of long-lived ip flows. volume 29, pages 215 – 26, Cambridge, MA, USA, 1999/10/.

[16] K. Singh. *Reliable, Scalable and Interoperable Internet Telephony*. PhD thesis, COLUMBIA UNIVERSITY, 2006.

[17] K. Singh, X. Wu, J. Lennox, and H. Schulzrinne. Comprehensive multi-platform collaboration. volume 5305, pages 199 – 210, San Jose, CA, United States, 2004.

[18] D. Sisalem, J. Kuthan, and S. Ehlert. Denial of service attacks targeting a sip voip infrastructure: Attack scenarios and prevention mechanisms. *IEEE Network*, 20(5):26 – 31, 2006.