RC24336 (W0708-086) August 29, 2007
Computer Science

IBM Research Report

SERvartuka: Enhancing SIP Server Scalability with Dynamic State Management

Vijay Balasubramaniyan*, Arup Acharya,
Mustaque Ahamad*, Charles P. Wright
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

*College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

SERvartuka[*] : Enhancing SIP Server Scalability with Dynamic State Management

Vijay A. Balasubramaniyan[†], Arup Acharya[‡], Mustaque Ahamad[†], Charles P. Wright[‡]

[†]College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA
{vijayab,mustaq}@cc.gatech.edu
[‡]Network Server System Software, IBM Research T.J. Watson, Hawthorne, NY, USA
{arup,cpwright}@us.ibm.com

ABSTRACT

A growing class of applications, including VoIP, IMS 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. Our approach for solving this problem leads to the design and evaluation of SERvartuka, a more scalable SIP server that dynamically determines the optimal fraction of SIP requests to be handled statefully per server in order to maximize call throughput. This design is in contrast to existing SIP servers that are statically configured to be either stateful or stateless and therefore result in sub-optimal call throughput. We implemented SERvartuka on top of OpenSER and measured performance benefits of different server configurations on a BladeCenter. An example of our results is a 15% percent increase in call throughput when using our algorithm for a configuration of two servers in sequence.

1. INTRODUCTION

The Session Initiation Protocol (SIP)[1] is a control plane protocol that is used in connection setup and teardown for a variety of applications including VoIP, IMS[3], Presence[2] and now 3GPP[19]. In addition, there are proposals for using SIP as an off path signaling mechanism for any kind of data or media session[21]. In SIP, call requests traverse through an application overlay of proxies each of which performs some setup function. These functions include host discovery, routing, maintaining state (for retransmissions, accounting etc.) and authentication. As more applications adopt SIP for connection handshake, the functionality provided by SIP proxies will grow. The traditional approach to supporting this functionality is to assign each function to a particular proxy in the application overlay. If the number of proxies exceeds the functions that need to be provided, then core proxies provide the necessary functionality while the remaining proxies simply route the request. If the functions outnumber the proxies then certain proxies perform multiple functions. In either case network operators are left with the difficult task of making this static assignment, having to consider resources available at each proxy to get the best call throughput. Experience has shown that such assignments typically lead to resource bottlenecks and suboptimal throughput.

We challenge the traditional approach and claim that the most scalable mechanism would be to dynamically distribute call functions across the servers. Each server then provides a particular functionality for a fraction of the requests traversing through the server network. Scalability is particularly important for SIP server deployments due to the increasing loads that need to be handled by such servers. Estimates predict that by the year 2010, the global number of civilian VoIP users will reach 197.2 million, about 40 times that of the 4.8 million in 2004 [15]. Examples of current real world deployments of SIP include AT&T, CallVantage[16], Yahoo Messenger[18], and Vonage[17] to name a few. The large anticipated VoIP user base will require an infrastructure that is capable of handling large volumes of call traffic.

We develop SERvartuka, a SIP server that implements the algorithm that distributes state across the server configuration. In addition, the algorithm is designed such that each server can decide the amount of state that it maintains locally and still achieve close to optimal call throughput. Thus, each server dynamically reconfigures itself 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. Since realistic deployments will have many proxies, the benefits will be much higher.

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 define the problem of distributing state. The dynamic state distribution algorithm and the SERvartuka design are discussed in Section 3. We evaluate SERvartuka performance is Section 4. The applicability of our solution is discussed in Section 5. Related work is described in Section 6 and we conclude the paper in Section 7

[*]Sanskrit, adapted to all seasons.

2. DISTRIBUTING STATE

Most SIP deployments are organized like a DNS hierarchy. A call from a SIP URI *sip:burdell@cc.gatech.edu* will traverse through the proxies responsible for the *cc.gatech.edu* domain and the *gatech.edu* domain. In commercial deployments the proxy is also responsible for billing. It then needs to maintain state to record the start and the end of the call. In this case either the *cc.gatech.edu.* or the *gatech.edu.* proxy can maintain this information. However such a static decision will lead to that proxy becoming a bottleneck. If on the other hand we split the state maintenance load across the 2 proxies such that each proxy maintains state for some fraction of the requests then we can improve the call throughput.

A state distribution algorithm has to satisfy the following requirements. First, the call related state must be maintained by one node on the path that is traversed by the call. Other nodes forward the call statelessly and thus do not incur the computational overhead associated with state maintenance. We must dynamically determine which node in a path should maintain the call state in such a way that the system is able to handle an incoming call as long as the resources available in the system are sufficient to handle the call in stateful manner at one node and stateless manner at all other nodes on some path in the overlay network.

In order to characterize the nature of such an algorithm we start by profiling OpenSER[12], an open source SIP Server, as a representative SIP server to experimentally measure the processing resources consumed by call handling functions at low call load and at increasing call load.

2.1 Low Load CPU Profiling

The run setup includes the OpenSER server setting up a call between two SIPp[14] clients. SIPp is a workload generator for SIP and uses XML documents to create call flows. OpenSER is 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 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. A transaction in SIP consists of all messages starting with the initial request till the first non provisional response.

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 previous functions are executed. In addition, the proxy checks the credentials of the client.

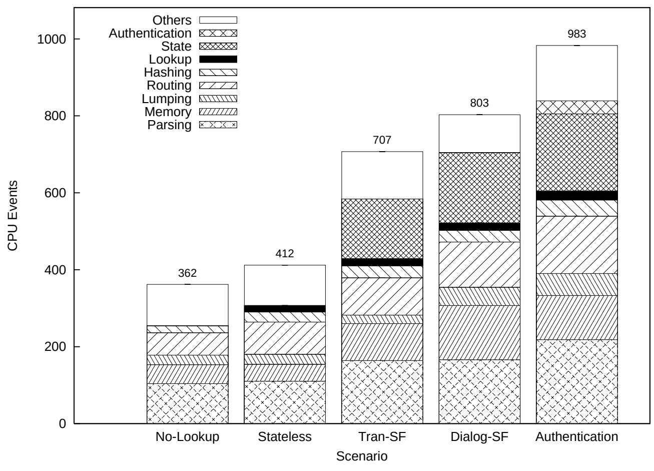

Figure 1: Server Functionality costs

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[13] profiles the various functionality blocks of OpenSER. The results for these runs is as shown in Figure 1.

The bar graph in Figure 1 shows that as the server executes more functionality, it results in higher CPU load. 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 such 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 1. 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 to maintain state, the request needs to be uniquely identified. This is done by hashing together a set of fields including *From*, *To*,

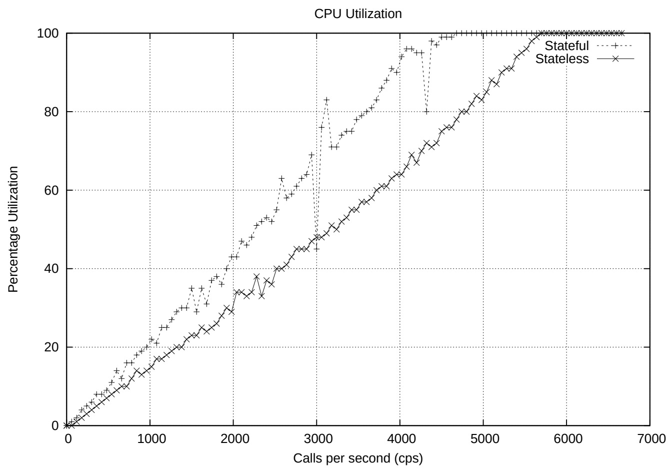

Figure 2: CPU Increasing Load Utilization

which requires these fields to be parsed.

The increase in CPU processing by being transaction stateful or dialog stateful is largely due to the extra state that needs to be maintained. 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. We next explore how stateful and stateless servers behave when the call rate increases.

2.2 Performance with Increasing Call Rates

At high loads, a single SIPp client saturates (reaches 100% CPU utilization) at about the same loads as OpenSER and therefore skews the measurements. We therefore split this load among 2 SIPp clients on equally powerful blade center machines. Associated with these servers were two URIs which the two SIPp clients made calls to respectively. The OpenSER database was populated with these two URIs and was configured to run in 2 modes (i) stateless with lookup and (ii) transaction stateful with lookup.

The SIPp clients generated a load starting with 20 calls per second and increasing it in steps of 20 calls per second. Saturation of the OpenSER server was determined by observing that CPU utilization of the server was at 100% and the call throughput at the SIPp server did not increase any longer with increased call rate generated at the SIPp client. At 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.

As seen in Figure 2, 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 ≈ 4700 calls per second while a stateless server saturates at 5700 calls per second. This difference between stateful and stateless servers forms the basis for obtaining higher throughput by distributing state maintenance functions. The theoretical basis for this is described in the next section (Section 2.3).

2.3 Formulation of State Distribution as an Optimization Problem

We show that dynamically maintaining state can increase call throughput by formulating state distribution as an optimization problem similar to [10]. Intuitively, the proxy overlay network can be considered as a graph. Call set up requests enter the system at some node in this graph and they traverse a set of nodes along links in the graph. A request exits at a proxy node that forwards the request to the called user agent.

Let us consider a network of N servers arranged in any topography, where each server is represented as a number from 1 to N. Let the expected traffic at node i be t_i. This expected traffic will be routed to node i by a set of upstream servers denoted by L_i. Thus, for any upstream server, $l \epsilon L_i$, i is a downstream server. Node l will thus route requests to many such downstream servers. Let the fraction of requests that a server l routes to i be ϕ_{li}. This fraction can be either predetermined because of existing routing tables or can be calculated in case there is flexibility in determining routes dynamically. We note that if ϕ_{li} is pre-determined then the optimization problem is reduced to a linear programming problem. As the incoming flow at l will be routed to some downstream server

$$\sum_i \phi_{li} = 1$$

The incoming traffic at node i then is the sum total of the fraction of requests that the l upstream servers route to it.

$$t_i = \sum_l t_l * \phi_{li}$$

Each server then determines the amount of incoming traffic t_i to forward statefully, represented by t_i^{SF} and the rest statelessly as t_i^{SL}. Therefore

$$t_i = t_i^{SF} + t_i^{SL}$$

The entire premise of creating this optimization problem is the fact that stateful resource consumption is different from stateless resource consumption. Thus, if the CPU utilization due to stateful request forwarding at node i is represented by $U_i^{SF} = f^{SF}(t_i^{SF})$ and stateless by $U_i^{SL} = f^{SL}(t_i^{SL})$ we have the CPU utilization represented by

$$U_i = U_i^{SF} + U_i^{SL}$$

The utilization at each node should not exceed 100%. If the utilizations are normalized then this translates to

$$U_i <= 1$$

The first server node will receive the call traffic from a set of VoIP clients. Without any loss in generality, we assume that there is only one initial entry node. If there are multiple entry points we can always assume an imaginary source that sends requests to these multiple entry points. The theoretical results do not change in this case. The first node, by our initial convention, is then node 1 and the incoming traffic is t_1. This represents the total incoming load or the call throughput. In our case we assume that state for all of this incoming load has to be maintained at some node in the overlay network. Therefore

$$t_1 = \sum_{i=1}^{N} t_i^{SF}$$

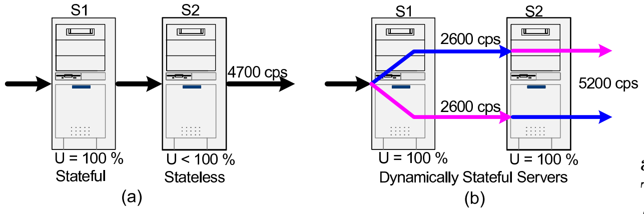

Figure 3: Static vs Dynamic state maintenance

With these constraints in place, the objective of the optimization problem is then to maximize the throughput or t_1. Thus the problem has the form

$$\begin{aligned} &Maximize\ t_1 \\ &Subject\ to \\ t_i &= \sum_l t_l * \phi_{li} &(1) \\ \sum_i \phi_{li} &= 1 &(2) \\ t_i &= t_i^{SF} + t_i^{SL} &(3) \\ f^{SF}(t_i^{SF}) + f^{SL}(t_i^{SL}) &<= 1 &(4) \\ t_1 &= \sum_{i=1}^{N} t_i^{SF} &(5) \end{aligned}$$

Conservation of flow is implicit in equation 1 of the formulation as the sum of all outgoing flow at node i is the total incoming flow. The solution of this optimization problem then gives the optimal values at each node i for t_i^{SF}, t_i^{SL}. Reducing it to an optimization problem implies that an algorithm that solves it will yield an optimal throughput solution. Our formulation is applicable to any configuration of servers and can accommodate multiple flows and does not require a strictly ordered direction of requests. Any algorithm created can thus be measured against the optimal throughput determined by this formulation. In the next section we use this formulation and compute the optimal fraction of requests maintained by each server in a simple configuration to show how state distribution can in fact lead to higher throughput.

2.3.1 *Illustrative example*

Let us consider a simple case where a request traverses through two servers in series ($S1$ and $S2$), within a domain. The possible configurations that these servers can be arranged 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, T_{SF} (from our experiments this is $\approx$ 4700 cps). Since both servers see the same request load the maximum system throughput will be around 4700 cps.

Case (ii) is when one of these servers is configured to be stateless. Such a system will continue to have maximum throughput of 4700 cps because the stateful server is the bottleneck and hence will dictate the overall throughput. 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 $\approx$ 5700 cps). 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 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} and this configuration is shown in Figure 3(a).

If we now use the LP formulation to calculate the optimal throughput, we obtain the values shown in Figure 3(b). Here each server maintains 2600 cps statelessly and 2600 cps statefully, giving a total throughput of 5200 cps. This is higher than the throughput of the static configuration in 3(a). This is because in configuration (a), where one server is stateful and the other stateless, we notice that though the stateful server at 4700 cps will be operating at 100% utilization (and thus becoming the bottleneck), the stateless server is under-utilized. The formulation obtains the optimal throughput by ensuring that all servers are utilized to the maximal possible extent.

We can think of a centralized solver which is fed the topography of the servers and is then able to communicate to each server the exact fraction of stateful to stateless requests to maintain. In section 3 we create SERvartuka, a distributed solution to the problem where each server dynamically reconfigures itself to decide the amount of state that it maintains locally in an attempt to achieve close to optimal call throughput for the entire system.

3. SERVARTUKA DESIGN

The above sections show that by distributing state among a set of servers, we can increase system throughput. Each server then needs to decide the fraction of incoming requests it will forward statefully. The intuition behind an algorithm that achieves this stems from a parallel we draw from network flows. The max flow problem[4] tries to maximize a flow in a network given that the edges of the graph have bounded capacities. Analogous to this, we are trying to maximize the amount of state that the system can sink given that the amount of state sinkable by each server is bounded. Maximizing the amount of state that the system can sink is equivalent to maximizing the throughput as is seen in equation 5 of the LP. The Floyd Fulkerson method[4] determines the max flow in a network and follows the algorithm:

```
while there exists augmenting path $p$ do
    augment flow $f$ along $p$
end
```

In the state distribution case this translates to:

```
while there exists augmenting path p that can sink state do
    sink state $s$ along $p$
end
```

However, being able to sink state into an augmenting path also implies that the nodes connected to this path have enough resources to be able to send this extra flow. This means if a node j and nodes reachable from j are able to sink extra state s, then all nodes i preceding j should have available CPU resources to be able to forward the flow s statelessly. This forms the basis of SERvartuka where each server in the system tries to sink as much state while making sure that it can also sink state along augmenting paths by forwarding calls statelessly to downstream nodes. We follow a modified greedy strategy, where the server forwards requests statefully until a utilization threshold is reached. Thereafter, with the increasing call load it routes the extra call load statelessly, and reduces the amount of state it maintains by small amounts when required. The latter is necessary because the server needs to free up utilization to be able to route this extra call load statelessly.

Now if the servers downstream also follow the above mechanism, we will have a system of dynamically self configuring servers, each taking up the state that the upstream servers have not been able to maintain. Each server in this system is then constantly trying to find an operating point where the number of requests that it forwards statefully is maximized while making sure that the number of requests that the system as a whole forwards is maximized.

Servers cannot sink unlimited state through its downstream path as there will be a limit after which the downstream servers will get saturated and will not be able to take up anymore state. Downstream servers can indicate this by the use of a throttle message mechanism to the immediate upstream server. Once the upstream server receives this message, it knows it cannot delegate additional state through that path of servers anymore. The throttle message is issued by a server only when the following two conditions are satisfied: (i) the server is saturated, and (ii) all paths downstream to it are saturated.

The upstream servers need to somehow indicate to the downstream servers that they have already maintained state for a set of requests. This way the downstream servers can forward these statelessly. This guarantees that state is distributed and only a single server maintains state for a particular transaction or a dialog.

Combining most of what has been discussed so far, we develop the *dynamically stateful algorithm (DSA)* for a server that is outlined here (Algorithm 1)

```
   // First time Csf_reduced = $\infty$
1  Csf_req = min( C - Csf_up, Csf_reduced )
2  U = ( Csf_req, C - Csf_req )
   // Check CPU utilization threshold
3  if $U <$ U_high then
4      Forward Csf_req statefully
5      Forward C - Csf_req statelessly
6  else if there exists downstream servers AND
7  downstream servers are not saturated then
8      Csf_reduced = min( Csf_req, Csf_reduced ) - deltaC
9      Forward Csf_reduced statefully
10     Forward C - Csf_reduced statelessly
11 else
12     Send throttle messages upstream
13 end
```

Algorithm 1: Dynamically stateful algorithm

The algorithm basically allows each server to decide the right operating point. Each server is subjected to an incoming load C cps. However, the upstream servers already maintain state for Csf_up calls, so this server will need to maintain state only for $Csf\_req = C - Csf\_up$ (line 1). It then calculates the utilization to route Csf_req calls statefully and $C - Csf\_req$ calls statelessly and checks to see if it is below the utilization threshold (lines 2 and 3). If so, then this is the right operating point. However if not, and there exists downstream servers that the server can delegate state to, then the server reduces the amount of state maintained by an amount *deltaC* and then forwards the resulting $Csf\_reduced$ statefully and $C - Csf\_reduced$ statelessly (lines 6 through 10). Thereafter, each time the threshold is crossed, it will reduce $Csf\_reduced$ by *deltaC*. We use $min(Csf\_req, Csf\_reduced)$(line 8) to achieve this. Essentially $Csf\_reduced$ keeps undergoing an additive decrease each time the server crosses the utilization threshold. This additive decrease is good enough because delegating *deltaC* amounts of state frees up significant amounts of CPU utilization which it can then use to route the extra load statelessly.

In case there are no downstream servers or all downstream servers are saturated then the server itself will then send throttle messages (line 12) to indicate saturation of itself and all paths (if they exist) downstream to it. By using this algorithm, the server constantly reconfigures itself to find the right amount of stateful and stateless calls to forward. The advantage of this algorithm is that other than for minimal communication, the server can decide this information in a local fashion. This algorithm however only shows the net effect at each server, a more refined algorithm is actually used and this includes other mechanisms such as unthrottle messages (to undo the throttle messages when utilization goes below a lower threshold) and providing a state delegation order (to reduce call setup times).

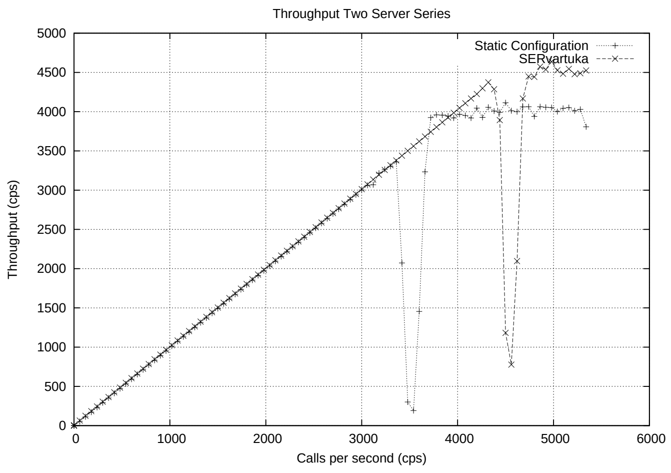

Figure 4: Comparison of Goodput

4. PRELIMINARY RESULTS AND ANALYSIS

In our preliminary evaluation, we arranged two servers in a series configuration and measured maximum throughput when (a) the servers are pre-configured statically as stateful, and (b) the servers run the load balanced DSA algorithm of SERvartuka. Our results are as shown in Figure 4. The static pre-configuration results in a maximal throughput of 4000 cps, actually lesser than the single pre-configured server call rate. SERvartuka on the other hand gives a maximal throughput of 4600 cps a 15% increase. There is a significant throughput drop when the system load is 3500 cps for the unmodified OpenSER static configuration and a similar drop at 4500 cps for the SERvartuka algorithm. We are right now unsure of the exact reason for the drop and are investigating it.

5. APPLICABILITY OF THE SOLUTION

The current algorithm will work when the configuration of servers are strictly ordered in the direction of requests. This is because each server needs to clearly know its upstream servers in order to be able to communicate saturation and unsaturation and provide a loading order for the algorithm. In typical SIP deployments where servers are arranged as a hierarchy, there exists such an order. However, this algorithm will not work for deployments that partition networks into core and edge proxy servers as these networks do not guarantee a strict ordering. We plan to determine such a distributed algorithm as part of future work.

6. RELATED WORK

In [9], 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 in that it 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 [5][6][7]. In [7], authors study sipd, a SIP proxy server developed in [20], 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 [8], 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 [11], we believe our work is one of the first to design and implement a concrete detailed algorithm for SIP server systems.

7. 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% increase in the maximum call throughput that can be achieved. Our work can be extended in several ways.

- Explore the algorithm's performance on more complex configurations
- Create a distributed algorithm for configurations where the direction of the flows are not strictly ordered.
- 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.

8. REFERENCES

[1] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, *SIP: Session initiation protocol.* RFC 3261, June 2002.

[2] J. Rosenberg, *A Presence Event Package for the Session Initiation Protocol (SIP).* RFC 3856, August 2004.

[3] G. Camarillo, and M. Garca-Martn, *The 3G IP Multimedia Subsystem (IMS): Merging the Internet and the Cellular Worlds.* John Wiley & Sons, 2006, ISBN 0-470-01818-6.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest. and C. Stein, *Introduction to Algorithms, Second Edition.* The MIT Press, 2001, ISBN-10: 0-262-03293-7.

[5] J. Janak, *SIP Proxy Server Effectiveness.* Master's thesis, Czech Technical University, May 2003.

[6] K. Singh, *Reliable, Scalable and Interoperable Internet Telephony.* PhD Thesis, Columbia University, 2006.

[7] K. Singh, H. Schulzrinne, and J. Lennox *SIP Server Scalability.* http://www1.cs.columbia.edu/ kns10/talks/, May 2005.

[8] D. Sisalem, and J. Kuthan, *Denial of Service attacks and SIP Infrastructure.* http://www.snocer.org/Paper/sisalem_dos.pdf.

[9] M. Cortes, J. Esteban, and H. Jun, *Towards Stateless Core : Improving SIP Proxy Scalability.* IEEE Globecom Conference, Nov 2006.

[10] R. Gallagher,*A Minimum Delay Routing Algorithm Using Distributed Computation.* IEEE Transactions on Communications, Pgs 73-85, Volume 25, Issue 1, Jan 1977.

[11] A. Shaikh, J. Rexford, and K. Shin, *Load-sensitive routing of long-lived IP flows.* ACM Sigcomm 1999.

[12] *OpenSER.* http://www.openser.org/

[13] *OProfile.* http://oprofile.sourceforge.net/

[14] *SIPp.* http://sipp.sourceforge.net/

[15] *4.8 million VoIP users in 2004. How many in 2010 ?.* http://www.voip-security-blog.com/comments.php?post=8

[16] *AT&T CallVantage.* https://www.callvantage.att.com/

[17] *Vonage broadband phone service.* http://www.vonage.com

[18] *Yahoo! Messenger with voice.* http://www.messenger.yahoo.com

[19] *The 3rd Generation Partnership Project (3GPP).* http://www.3gpp.org/

[20] *CINEMA.* http://www.cs.columbia.edu/IRT/cinema/

[21] A. Falk, P. Francis, *Path-decoupled Signaling for Data (offpath).* http://www.cs.cornell.edu/people/francis/offpath/