Failover, Load Sharing and Server Architecture in SIP Telephony

Kundan Singh and Henning Schulzrinne
Department of Computer Science, Columbia University
1214 Amsterdam Ave, Mail Code 0401, New York, NY 10027, USA
Email: {kns10,hgs}@cs.columbia.edu, tel:+1-212-9397000, fax:+1-212-6660140

Abstract

We apply some of the existing web server redundancy techniques for high service availability and scalability to the relatively new IP telephony context. The paper compares various failover and load sharing methods for registration and call routing servers based on the Session Initiation Protocol (SIP). In particular, we consider SIP server failover techniques based on the clients, DNS (Domain Name Service), database replication and IP address takeover, and load sharing techniques using DNS, SIP identifiers, network address translators and servers with same IP addresses. We describe our two-stage reliable and scalable SIP server architecture in which the first stage proxies the request to one of the second stage server group based on the destination user identifier. We quantitatively evaluate the performance improvement of the load sharing architecture using our SIP server. We quantitatively compare the effect of SIP server architecture such as event-based and thread pool. Additionally, we present an overview of the failover mechanism we implemented in our test-bed using the open source MySQL database.

Keywords: Availability; scalability; failover; load sharing; SIP

1 Introduction

The Session Initiation Protocol (SIP) [1] is a distributed signaling protocol for IP telephony. SIP-based telephony services have been proposed as an alternative to the classical PSTN (public switched telephone network) and offers a number of advantages over the PSTN [2]. Traditionally, telephony service is perceived as more reliable than the Internet-based services such as web and email. To ensure wide acceptance of SIP among carriers, SIP servers should demonstrate similar quantifiable guarantees on service availability and scalability. For example, PSTN switches have a "5 nines" reliability requirement, i.e., are available for 99.999% of the time, which implies at most 5 minutes of outage a year.

The SIP proxy servers are more light-weight compared to PSTN switches because they only route call signaling messages without maintaining any per-call state. The SIP proxy server of a *domain* is responsible for forwarding the incoming requests destined for the logical address of the form *user@domain* to the current transport address of the device used by this logical entity, and forwarding the responses back to the request sender. Consider the example shown in Fig. 1. When a user, Bob, starts his SIP phone, it registers his unique identifier *bob@home.com* to the SIP server in the *home.com* domain. The server maintains the mapping between his identifier and his phone's IP address. When another user, Alice, calls *sip:bob@home.com*, her phone does a DNS (Domain Name Service) lookup for the SIP service [3] of *home.com* and sends the SIP call initiation message to the resolved server IP address. The server "proxies" the call to Bob's currently registered phone. Once Bob picks up the handset, the audio packets can be sent directly between the two phones without going through the server. Further details [2, 1] of the call are skipped for brevity.

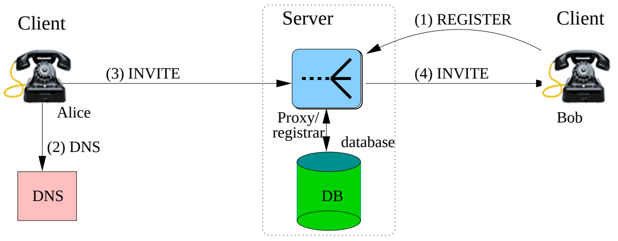


Figure 1: An example SIP call

If the server fails for some reason, the call initiation or termination messages cannot be proxied correctly. (Note that the call termination message need not traverse proxy servers unless the servers record-route.) We can improve the service availability by adding a second server that automatically takes over in case of the failure of the first server. Secondly, if there are thousands of registered users and a single server cannot handle the load, then a second server can work along with the first server such that the load is divided between the two. Our goal is to provide the carrier grade capacity of one to ten million BHCA (busy hour call attempts) for IP telephony using commodity hardware. We describe some of the failover and load sharing techniques for SIP servers in Sections 3 and 4, respectively. These techniques also apply beyond telephony, for example, for SIP-based instant messaging and presence that use the same SIP servers for registration and message routing. Section 5 quantitatively evaluates the performance improvement of our load sharing architecture. Section 6 further evaluates and improves the performance based on the software architecture such as event-based and thread pool.

2 Related Work

Failover and load sharing for web servers is a well-studied problem [4, 5, 6, 7]. TCP connection migration [8], IP address takeover [9] and MAC address takeover [10] have been proposed for high availability. Load sharing via connection dispatchers [11] and HTTP content or session-based request redirection [12, 13, 10] are available for web servers. Some of these techniques such as DNS-based load sharing [14, 15] also apply to other Internet services like email and SIP. Although SIP is an HTTP like request-response protocol, there are certain fundamental differences that make the problem slightly different. For example, SIP servers can use both TCP and UDP transport, the call requests and responses are usually not bandwidth intensive, caching of responses is not useful, and the volume of data update (REGISTER message) and lookup (INVITE message) is often similar, unlike common read-dominated database and web applications.

For SIP server failover, IP anycast does not work well with TCP and the backend database requires synchronization between the primary and backup servers [16]. Section 3.5 describes how to apply the IETF's Reliable Server Pooling (Rserpool [17, 18, 19]) architecture for SIP telephony. The primary disadvantage of Rserpool is that it requires new protocol support in the clients.

SIP-based telephony services exhibit three bottlenecks to scalability: signaling, real-time media data and gateway services. The signaling part requires high request processing capacity in the SIP servers. The data part requires enough network bandwidth and capacity (CPU and memory) in the end systems. The gateway part requires optimal placement of media gateways and switching components [20]. This paper focuses on the signaling part only. SIP allows redirecting a request to a less loaded server using the 302 response, or transferring an existing call dialog to a less loaded endpoint or gateway [1, 21].

Optimizations such as memory pool, counted strings and lazy parsing have been proposed for SIP servers [22, 23]. These optimizations can further improve our load sharing performance. Event and thread-based architectures, particularly for web servers, are well known in systems research. We study the effect of server architecture on SIP server performance using commodity hardware and standard POSIX threads.

3GPP's IP Multimedia Subsystem (IMS) uses SIP for call control to support millions of users. It defines different server roles such as outbound proxy in visited network, interrogating proxy as the first point of contact for incoming calls in the home network, and serving proxy providing services based on subscriber's profile.

Identifier-based load balancing has been used for emails. We combine this with DNS-based server redundancy for a two-stage reliable and scalable architecture. Novelty of our work lies in the application of existing techniques to relatively new Internet telephony, and quantitative performance evaluation of the architecture for SIP-based telephony.

We describe and compare some of these techniques in the context of SIP. We also present an overview of our implementation of failover and describe some practical issues.

3 Availability: Failover

High availability is achieved by adding a backup component such as the SIP server or user record database. Depending on where the failure is detected and who does the failover, there are various design choices: client-based, DNS-based, database failover and IP takeover.

3.1 Client-based failover

In the client-based failover (Fig. 2), Bob's phone knows the IP addresses of the primary and the backup servers, P_1 and P_2. It registers with both, so that either server can be used to reach Bob. Similarly, Alice's phone also knows about the two servers. It first tries P_1, and if that fails it switches to P_2.

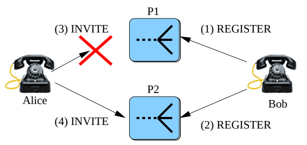
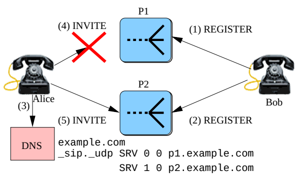


Figure 2: Client-based failover



```
example.com
_sip._udp SRV 0 0 p1.example.com
          SRV 1 0 p2.example.com
```

Figure 3: DNS-based failover

All failover logic is built into the client. The servers operate independently of each other. This method is used by the Cisco IP phones [24]. Configuring phones with the two server addresses works well within a domain. However, DNS is used to allow adding or replacing backup servers without changing the phone configurations as described next.

3.2 DNS-based failover

DNS-based failover using NAPTR and SRV records is the most clean and hence, preferred way, to failover [3]. For instance, Alice's phone can retrieve the DNS SRV [14] record for *_sip._udp.home.com* to get the two server addresses (Fig. 3). In the example, P_1 will be preferred over P_2 by assigning a lower numeric priority value to P_1.

Alternatively, dynamic DNS can be used to update the A-record for *home.com* from the IP address of P_1 to P_2, when P_1 fails. P_2 can periodically monitor P_1 and update the record when P_1 is dead. Setting a low time-to-live (TTL) for the A-record bindings can reduce the failover latency due to DNS caching [25].

3.3 Failover based on database replication

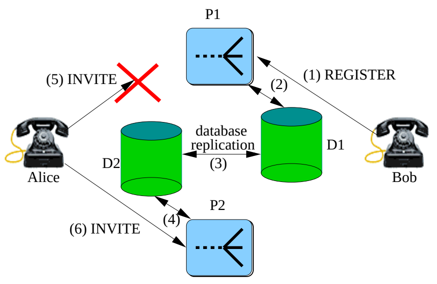


Figure 4: Failover based on database replication

Not all the SIP phones are capable of registering with multiple servers. Moreover, to keep the server failover architecture independent of the client configuration, the client can register with only P_1, which can then propagate the registration to P_2. If a database is used to store the user records, then replication can be used as shown in Fig. 4. Bob's phone registers with the primary server, P_1, which stores the mapping in the database D_1. The secondary server, P_2, uses the database D_2. Any change in D_1 is propagated to D_2. When P_1 fails, P_2 can take over and use D_2 to proxy the call to Bob. There could be small delay before D_2 gets the updated record from D_1.

3.4 Failover using IP address takeover

If DNS-based failover cannot be used due to some reason (e.g., not implemented in the client), then IP takeover [9] can also be used (Fig. 5). Both P_1 and P_2 have identical configuration but run on different hosts on the same Ethernet. Both servers are configured to use the external master database, D_1. The slave D_2 is replicated from D_1. The clients know the server IP address as P_1's 10.1.1.1 in this example.

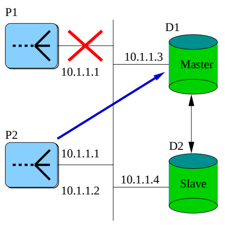


Figure 5: When the primary server fails

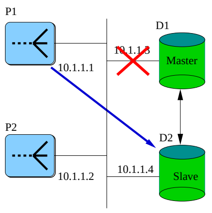


Figure 6: When the master database fails

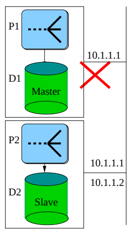


Figure 7: co-located database and proxy

P_2 periodically monitors the activity of P_1. When P_1 fails, P_2 takes over the IP address 10.1.1.1. Now, all requests sent to the server address will be received and processed by P_2. When D_1 fails, P_1 detects and switches to D_2 (Fig. 6). IP takeover is not used by D_2 since the SIP servers can be modified to switch over when D_1 fails. There can be a small failover latency due to the ARP cache.

The architecture is transparent to the rest of the network (clients and DNS) and can be implemented without external assumptions. However, if the replication is only from the master to the slave, it requires modification in the SIP server software to first try D_1, and if that fails use D_2 so that all the updates are done to the master server. To avoid replicating the database, P_1 can propagate the REGISTER message also to P_2.

Alternatively, to avoid the server modification, the server and the associated database can be co-located on the same host as shown in Fig. 7. If the primary host fails, both P_2 and D_2 take over. P_1 always uses

D_1, whereas P_2 always uses D_2.

3.5 Reliable server pooling

In the context of IETF's Reliable Server Pooling architecture [17], Fig. 8 shows the client phone as the pool user(PU), P_1 and P_2 as the pool elements (PE) in the "SIP server pool", and D_1 and D_2 as PEs in the "database pool". P_1 and P_2 register with their home name server, NS_2, which supervises them, and informs the other name servers (NS) about these PEs. Similarly, D_1 and D_2 also register with the NS. The SIP servers are the pool users of the "database pool". A pool element is removed from the pool if it is out of service.

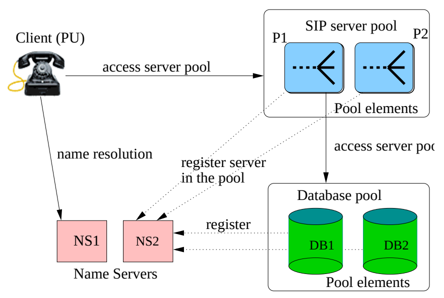


Figure 8: Reliable server pooling for SIP

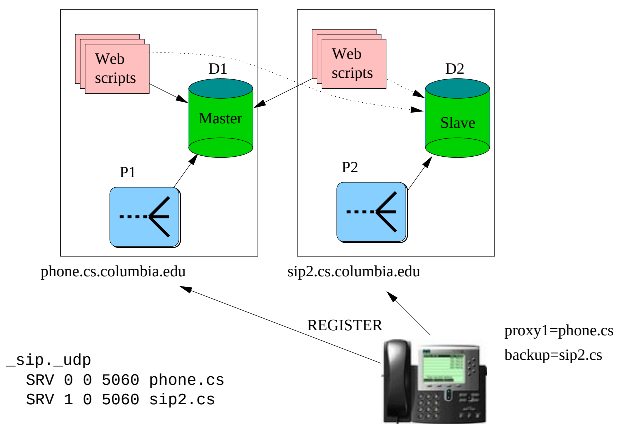
When the client wants to contact the "SIP server pool", it queries one of the name servers, NS_1, to get the list of P_1 and P_2 with relative priority for failover and load sharing. The client chooses to connect to P_1 and sends the call invitation. If P_1 fails, the client detects this and sends the message to P_2. For stateful services, P_1 can exchange state information with another server, P_2, and return the backup server, P_2, to the client in the initial message exchange. This way the client knows which backup server to use in the case of failure. P_1 can also give a signed cookie similar to HTTP cookie to the client, which sends it to the new failover server, P_2, in the initial message exchange. This is needed for call stateful services such as conferencing, but not for SIP proxy server failover.

The SIP server, P_1, queries the NS to get the list, D_1 and D_2, for the "database pool". D_1 and D_2 are backed up and replicated by each other, so they can return this backup server information in the initial message exchange.

The primary limitation is that this requires new protocol support for name resolution and aggregate server access in the clients. A translator can be used to interoperate with the clients that do not support reliable server pooling. However, this makes the translator as a single point of failure between the client and the server, hence limiting the reliability. Secondly, the name space is flat unlike DNS hierarchy, and is designed for a limited scale (e.g., within an enterprise), but may be combined with wide area DNS based name resolution, for example. More work is needed in that context.

3.6 Implementation

We have used some of the above techniques in our Columbia InterNet Extensible Multimedia Architecture (CINEMA). The architecture [26, 27] consists of our SIP server, sipd and a MySQL database for user profile and system configuration. Other components such as the PSTN gateway and media servers are outside the scope of this paper. The configuration and management are done via a web interface that accesses various CGI (Common Gateway Interface) scripts written in Tcl on the web server. All the servers may run on a single machine for an enterprise setup.



```
_sip._udp
   SRV 0 0 5060 phone.cs
   SRV 1 0 5060 sip2.cs
```

Figure 9: Failover in CINEMA

For failover, we use two sets of identical servers on two different machines as shown in Fig. 9. The database and SIP server share the same host. The databases are replicated using MySQL 4.0 replication [28] such that both D_1 and D_2 are master and slave of each other. MySQL propagates the binary log of the SQL commands of master to the slave, and the slave runs these commands again to do the replication. Our technical report [29] contains the details of the two-way replication.

MySQL 4.0 does not support any locking protocol between the master and the slave to guarantee the atomicity of the distributed updates. However, the updates from the SIP server are additive, i.e., each registration from each device is one database record, so having two devices for the same user register with two database replicas does not interfere with the other registration. For example, if *bob@home.com* registers *bob@location1.com* with D_1 and *bob@location2.com* with D_2, both, D_1 and D_2, will propagate the updates to each other such that both D_1 and D_2 will have both of Bob's locations. There is a slight window of vulnerability when one contact is added from D_1 and the same contact is removed in D_2, then after the propagation of updates the two databases will be inconsistent with different contacts for the user. It turns out that this does not occur for the simple failover as we describe next. We can safely use the two-way replication as long as updates are done by only the SIP server.

For a simple failover case, the primary server P_1 is preferred over the secondary server P_2. So all the REGISTER requests go to P_1 and are updated in D_1. The replication happens from D_1 to D_2, not the other way. Only in the case of failure of P_1, will the update happen to D_2 through P_2. But D_1 will not be updated by the server in this case. By making sure that database becomes consistent before the failed server is brought up, we can avoid the database inconsistency problem mentioned above.

Web scripts are used to manage user profiles and system configuration. To maintain database consistency, the web scripts should not be allowed to modify D_2 if D_1 is up. To facilitate this we modified the MySQL-Tcl client interface to accept a list of connection attributes. For example, if D_1 and D_2 are listed then the script tries to connect to D_1 first, and if that fails then tries D_2 as shown in Fig. 9. For our web scripts, the short-lived TCP connection to MySQL is active as long as the CGI script is running. So the failover at the connection setup is sufficient. In the future, for long-lived connection, it should be modified to provide failover even when the TCP connection breaks.

3.7 Analysis

The architecture provides high reliability due to redundancy. Assuming the reliability of primary and backup sets of servers as R, the overall reliability is $(1 - (1 - R)^2)$.

Server failure affects the *call setup latency* (since the client retries the call request to the secondary server after a timeout) and the *user availability* (the probability that the user is reachable via the server given that her SIP phone is up). If the primary server is down for a longer duration, the DNS records can be updated to change the secondary server into primary. If the individual server reliability is R (such that $0 \leq R \leq 1$), client retry timeout is T_R, and DNS TTL is T_D, then the average call setup latency increases by $T_R(1 - R)\mathrm{P}[t_M < T_D]$ (assuming no network delay and $R \approx 1$), where $\mathrm{P}[t_M < T_D]$ is the probability that the time, t_M (random variable), to repair the server is less than the DNS TTL. For example, if the repair time is exponentially distributed with mean T_M, then $\mathrm{P}[t_M < T_D] = 1 - e^{-\frac{T_D}{T_M}}$ assuming that the mean time to failure is much larger than the mean time to repair. (i.e., $(1 - R)T_M \approx 0$).

User availability is mostly unaffected by the primary server failure, because most registrations are REGISTER refreshes. However, if the primary server fails after the phone registers a new contact for the first time, but before the registration is propagated to the secondary server, then the phone contact location is unreachable until the next registration refresh. In this case, assuming that the server uptime is exponentially distributed, and given the memoryless property, the time-to-failure has the same distribution. Suppose the mean-time-to-failure is T_F and the database replication latency is T_d, then the probability that the server goes down before the replication is completed (given that it is up at $t = 0$) is $\mathrm{P}[\text{lifetime} < T_d] = 1 - e^{-\frac{T_d}{T_F}}$. If this happens, the user record is unavailable for at most $T_r + T_R$, where T_r is the registration refresh interval (typically one hour), and T_R is client retry timeout. After this time, the client refreshes the registration and updates the secondary server making the user record available.

We use an in-memory cache of user records inside the SIP server to improve its performance [26, 23]. This causes more latency in updating the user registration from P_1 to P_2. If the failure happens before the update is propagated to the P_2, then it may have an old and expired record. However, in practice the phones refresh registrations much before the expiry and the problem is not visible. For example, suppose the record expires every two hours and the refresh happens every 50 minutes. Suppose P_1 receives the registration update from a phone and fails before propagating the update to D_1. At this point, the record in D_2 has 70 minutes to expire so P_2 can still handle the calls to this phone. The next refresh happens in 50 minutes, before expiration of the record in D_2. If a new phone is setup (first time registration) just before failure of P_1, it will be unavailable until the next refresh. Suppose T_d and T_F are defined as before, and T_c is the database refresh interval, then the probability that the server goes down before the replication is completed is $1 - e^{-\frac{T_d + T_c}{T_F}}$.

With the Cisco phone [24] that has the primary and backup proxy address options (Section 3.1), the phone registers with both P_1 and P_2. Both D_1 and D_2 propagate the same contact location change to each other. However, since the contact record is keyed on the user identifier and contact location, the second write just overrides the first write without any other side effect. Alternatively, the server can be modified to perform the immediate synchronization between the in-memory cache and external database if the server is not loaded.

The two-way replication can be extended to more servers by using circular replication such as D_1-D_2-D_3-D_1 using the MySQL master/slave configuration [28]. To provide failover of individual servers (e.g., D_1 fails but not P_1), the SIP server P_1 should switch to D_2 if D_1 is not available.

4 Scalability: Load Sharing

In failover, the backup server takes over in the case of failure whereas in load sharing all the redundant servers are active and distribute the load among them. Some of the failover techniques can also be extended to load sharing.

4.1 DNS-based load sharing

The DNS SRV [14] and NAPTR [15] mechanisms can be used for load sharing using the priority and weight fields in these resource records [3], as shown below:

```
example.com
_sip._udp 0 40 a.example.com
          0 40 b.example.com
          0 20 c.example.com
          1  0 backup.somewhere.com
```

The above DNS SRV entry indicates that the servers a, b, c should be used if possible (priority 0), with backup.somewhere.com as the backup server (priority 1) for failover. Within the three primary servers, a and b are to receive a combined total of 80% of the requests, while c, presumably a slower server, should get the remaining 20%. Clients can use weighted randomization to achieve this distribution.

However, simple random distribution of requests is not sufficient since the servers need to access the same registration information. Thus, in the example above, each server would have to replicate incoming REGISTER requests to all other servers or update the common shared and replicated database(s). In either case, the updates triggered by REGISTER quickly become the bottleneck. The SIP phones typically do REGISTER refresh once an hour, thus, for a wireless operator with one million subscribers, it has to process about 280 updates per second.

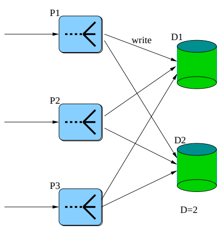


Figure 10: DNS-based

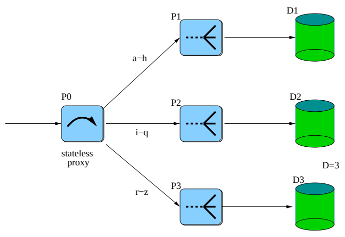


Figure 11: Identifier-based load sharing

Fig. 10 shows an example with three redundant servers and two redundant databases. For every REGISTER, it performs one read and one write in the database. For every INVITE-based call request, it performs one read from the database. Every write should be propagated to all the D databases, whereas a read can be done from any available database. Suppose there are N writes and $r * N$ reads, e.g., if the same number of INVITE and REGISTER are processed then $r = 2$. Suppose, the database write takes T units of time, and database read takes $t * T$ units. Total time per database will be $(\frac{tr}{D} + 1)TN$.

This architecture also provides high reliability due to redundancy. Assuming that the mean-time-to-repair is much less than mean-time-to-failure, and the reliability of individual proxy server as R_p and database server as R_d, and suppose there are P proxy servers and D database servers, the reliability of the system becomes $(1-(1-R_p)^P)(1-(1-R_d)^D)$. The reliability increases with increasing D and P.

4.2 Identifier-based load sharing

For identifier-based load sharing (Fig. 11), the user space is divided into multiple non-overlapping groups. A hash function maps the destination user identifier to the particular group that handles the user record, e.g., based on the first letter of the user identifier. For example, P_1 handles a-h, P_2 handles i-q and P_3 handles r-z. A high speed first stage server (P_0), proxies the call request to P_1, P_2 and P_3 based on the destination user identifier. If a call is received for destination *bob@home.com* it goes to P_1, whereas *sam@home.com* goes to P_3. Each server has its own database and does not need to interact with the others. To guarantee almost uniform distribution of call requests to different servers, a better hashing algorithm such as SHA1 can be used or the groups can be re-assigned dynamically based on the load.

Suppose N, D, T, t and r are as defined in the previous section. Since each read and write operation is limited to one database and assuming uniform distribution of requests to the different servers, total time per database will be $(\frac{tr+1}{D})TN$. With increasing D, this scales better than the previous method. Since the writes do not have to be propagated to all the databases and the database can be co-located on the same host with the proxy, it reduces the internal network traffic.

However, because of lack of redundancy this architecture does not improve system reliability. Assuming that the mean-time-to-repair is much less than mean-time-to-failure, and the reliability of the first stage proxy, second stage proxy and database server as R_0, R_p and R_d, and suppose there are D groups, then the system reliability becomes $R_0 \cdot (R_p)^D \cdot (R_d)^D$. The least reliable component affects the system reliability the most and the reliability decreases as D increases.

The only bottleneck may be the first stage proxy. We observed that the stateful performance is roughly similar to stateless performance (Section 5), hence a single stateless load balancing proxy may not work well in practice.

4.3 Network address translation

A network address translator (NAT) device can expose a unique public address as the server address and distribute the incoming traffic to one of the several internal private hosts running the SIP servers [30]. Eventually, the NAT itself becomes the bottleneck making the architecture inefficient. Moreover, the transaction-stateful nature of SIP servers require that subsequent re-transmissions should be handled by the same internal server. So the NAT needs to maintain the transaction state for the duration of the transaction, further limiting scalability.

4.4 Multiple servers with the same IP address

In this approach, all the redundant servers in the same broadcast network (e.g., Ethernet) use the same IP address. The router on the subnet is configured to forward the incoming packets to one of these servers' MAC address. The router can use various algorithms such as "round robin" or "response time from server" to choose the least loaded server.

To avoid storing SIP transaction states in the subnet router, this method is only recommended for stateless SIP proxies that use only UDP transport and treat each request as independent without maintaining any transaction state.

In the absence of DNS SRV and NAPTR, we can use this method for the first stage in Fig. 12. This is less efficient since the network bandwidth of this subnet may limit the number of servers in the cluster. Moreover, this method does not work if the network itself is unreachable.

4.5 Two-stage reliable and scalable architecture

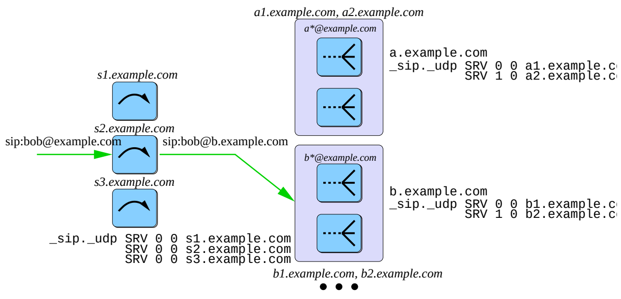


Figure 12: Two-stage reliable and scalable architecture

Since none of the mechanisms above are sufficiently general or infinitely scalable, we propose to combine the two methods (Fig. 10 and 11) in a two-stage scaling architecture (Fig. 12) to improve both reliability and scalability. The first set of proxy servers selected via DNS NAPTR and SRV performs request routing to the particular second-stage cluster based on the hash of the destination user identifier. The cluster member is again determined via DNS. The second-stage server performs the actual request processing. Adding an additional stage does not affect the audio delay, since the media path (usually directly between the SIP phones) is independent of the signaling path. Use of DNS does not require the servers to be co-located, thus allowing geographic diversity.

Suppose there are S first stage proxy servers, P clusters in the second stage, and B proxy and database servers in each cluster. The second stage cluster has one primary server and $B-1$ backups. All the databases in a cluster are replicated using circular replication. Suppose the REGISTER message arrivals are uniformly distributed (because of the uniform registration refresh rate by most user agents) with mean λ_R and INVITE (or other requests that need to be proxied such as MESSAGE) arrivals are Poisson distributed with mean λ_P, such that the total request rate is $\lambda = \lambda_R + \lambda_P$. Suppose the constant service rates of first stage server be μ_s, and the second stage server be μ_r and μ_p for registration and proxying, respectively. We assume a hash function so that each cluster's arrival rate is approximately $\frac{\lambda}{B}$. Suppose the reliability (probability that the system is available for processing an incoming message) and maintainability (repair rate of the system after a failure) distributions for first stage proxy are represented by probability distribution functions (pdf) R_s and M_s, respectively, and that for second stage proxy be R_p and M_p respectively. Note that Fig. 9 is a special case where S=0, P=1 and B=2. Similarly, Fig. 11 is a special case where S=B=1.

The goal is to quantitatively derive the relationship between different service parameters (μ), system load (λ), reliability parameters (R, M) and redundancy parameters (S, B, P). We want to answer the questions such as (1) when is first stage proxy needed, and (2) what are the optimal values for redundancy parameters to achieve a given scalability and reliability. Our goal is to achieve carrier grade reliability (99.999% available) and scalability (10 million BHCA) using commodity hardware. We provide our performance measurement results for scalability parameters (S and P) and system load (λ) in the next section.

Suppose each server is 99% reliable, and $S = P = B = 3$, then overall system reliability is $(1-(1-R_s)^S)\cdot(1-(1-R_p)^B)^P = 99.9996\%$, i.e., "five nines".

We do not consider the case of load sharing by different proxies in the same cluster, because load sharing is better achieved by creating more clusters. For handling sudden load spikes within one cluster, the DotSlash on-demand rescue system [31] is more appropriate where a backup server in the same or another cluster temporarily shares the load with the primary server of the overloaded cluster.

5 Performance Evaluation

In this section, we quantitatively evaluate the performance of our two-stage architecture for scalability using our SIP registration and proxy server, sipd, and SIPstone test suite [32].

5.1 Test setup

We performed the SIPstone Proxy 200 tests, over UDP. The SIPstone test suite has *loaders* and *call handlers*, to generate SIP requests and to respond to incoming requests, respectively. The server under test (SUT) is a two-stage cluster of our SIP servers, sipd, implementing the reactive system model [23]. An example test setup is shown in Fig. 13. Each instance of sipd was run on a dedicated host with Pentium 4, 3 GHz CPU, on a 800 MHz motherboard, with 1 GB of memory, running Redhat Linux (Fedora). The hosts communicated over a lightly loaded 100base-T Ethernet connection. A single external MySQL database, running version 3.23.52 of the MySQL server was shared by all the sipd instances. But this is not an issue because the Proxy 200 test does not modify the database, but uses in-memory cache of sipd [26].

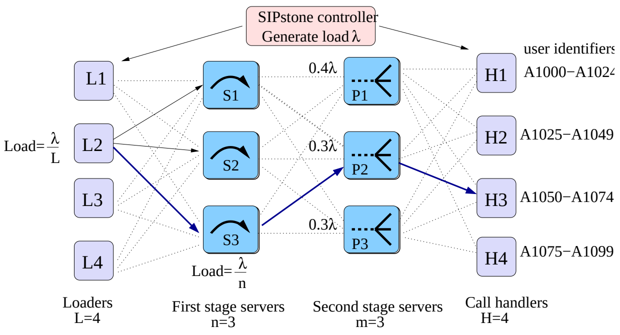


Figure 13: Example test setup for S3P3

To focus on only the scalability aspects we used one server in each group of the second stage (Fig. 12, B=1). We use the convention S_nP_m to represent n first stage servers, and m second stage groups with one server per group. S_0P_1 is same as a single SIP proxy server without any first stage load balancer.

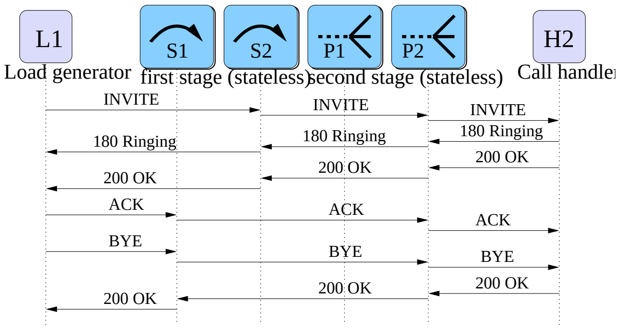


Figure 14: Example message flow

On startup, a number of call handlers (in our tests, four) register a number of destination locations (from non-overlapping user identifier sets as shown in Fig. 13) with the proxy server. Then for the Proxy 200 test, a number of loaders (in our tests, four) send SIP INVITE requests using Poisson distribution for call generation to the SUT, randomly selecting from among the registered addresses as shown in Fig. 14. If there is more than one first stage server ($n > 1$), then the loader randomly selects one of the first stage servers. The first stage server proxies the request on one of the second stage servers based on the destination user identifier. The second stage server forwards each request to the appropriate call handler responsible for this user identifier. The call handler immediately responds with 180 Ringing and 200 OK messages. These are forwarded back to the load generators in the reverse path. Upon receiving the 200 OK response, the load generator sends an ACK message for the initial transaction and a BYE request for a new transaction. The BYE is similarly forwarded to the call handler via the two-stage servers to reflect the record-route behavior in real operational conditions [32]. The call handler again responds with 200 OK. If the 200 OK response is not received by the loader within two seconds, or if any other behavior occurs, then the test is considered a failure. The loader generates the request for one minute for a given request rate. The server is then restarted, and the test is repeated for a higher request rate. We used an increment of 100 calls per second (CPS).

This process is repeated until 50% or more of the tests fail. Although [32] requires 95% success, we measure until 50% to show that the throughput is stable at higher loads. There is no retransmission on failure [32]. The complete process is repeated for different values of n and m in the cluster configuration, S_nP_m.

5.2 Analysis

Fig. 15 compares the performance of the different S_nP_m configurations. It shows the average of three experiments for each configuration at various call rates. A single sipd server handles about 900 calls/second (CPS) (see S_0P_1 in Fig. 15), which corresponds to about three million BHCA. When the load is more than the server capacity, the throughput remains almost constant at about 900 CPS. When throughput is same as load, i.e., 100% success rate, the graph is a straight line. Once the throughput reaches the capacity (900 CPS), the graph for S_0P_1 flattens indicating lower success rate for higher load. At a load of 1800 CPS, the system gives only 50% success rate (i.e., throughput is half of load), and the experiment stops. Note that for all practical purposes, success rate of close to 100% is desired.

When the server is overloaded, the CPU utilization is close to 100%. Introducing an extra server in the second stage and having a first stage load balancing proxy puts the bottleneck on the first stage server which has a capacity of about 1050 CPS (S_1P_2 in Fig. 15). An additional server in the first stage (S_2P_2) gives the throughput of approximately double the single second stage server capacity. Similarly, S_3P_3 has capacity of approximately 2800 CPS which is about three times the capacity of the single second stage server, and S_2P_3 has capacity of 2100 CPS which is double the capacity of the single first-stage server.

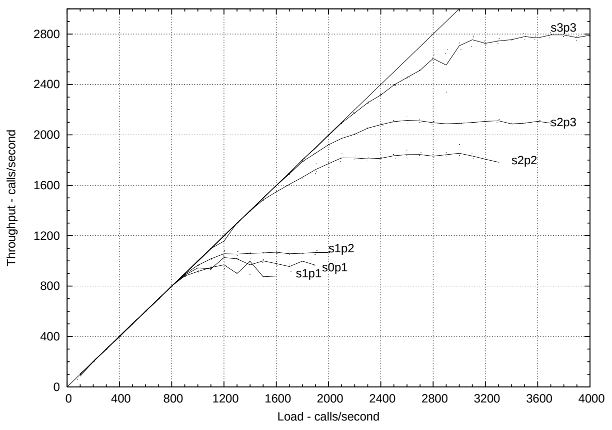


Figure 15: Server throughput in S_nP_m configuration (n first stage and m second stage servers). The results show that the performance increases linearly with the number of servers, i.e., s2p2 is twice and s3p3 is thrice that of s1p1 and s0p1 performance.

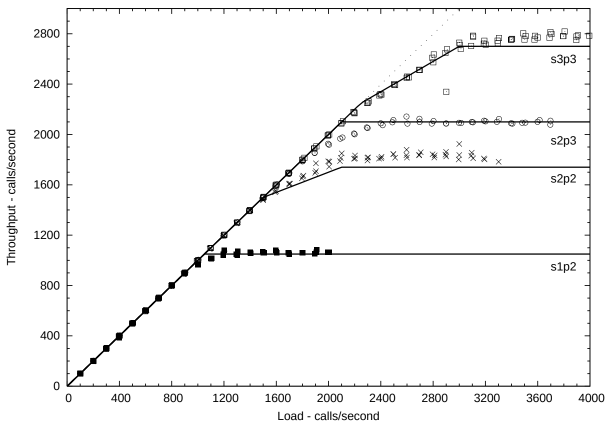


Figure 16: Theoretical and experimental capacity for configuration S_nP_m

The results show that we can achieve linear scaling by putting more servers in the first and second stages in our architecture. Below, we present the theoretical analysis for the two-stage architecture.

Suppose the first and second stage servers in S_nP_m have capacity of C_s and C_p, respectively (usually, $C_s \geq C_p$). The servers are denoted as S_i and P_j, $1 \leq i \leq n$, $1 \leq j \leq m$, for the first and second stage, respectively. Suppose the incoming calls arrive at an average rate λ, with exponential inter-arrival time. Suppose the load is uniformly distributed among all the n first stage servers, so the each first stage server gets a request rate of $\frac{\lambda}{n}$. Suppose the hash function distributes the requests to the second stage server such that the i^{th} server, P_i, gets a fraction, f_i, of the calls (Note that $\sum f_i = 1$). Assuming that all the users are equally likely to get called, and the hash function uniformly distributes the user identifiers among the second stage servers, then all f_i will be same (i.e., $f_i = \frac{1}{n}$). However, differences in the number of incoming calls for different users will cause non-uniform distribution in reality.

The throughput, τ, at a given load, λ, is the combined throughput of the two stages. The throughput of the first stage is $\lambda' = \min(\lambda, nC_s)$, which is load (input) to the second stage. The server, P_j, in the second stage has throughput of $\min(\lambda' f_j, C_p)$. Thus,

$$\tau(\lambda) = \sum_{j=1}^{m} \min(f_j \min(\lambda, nC_s), C_p)$$

Without loss of generality, we assume that $f_i \geq f_j$ for $i > j$. The resulting throughput vs load graph is given by $m + 1$ line segments, L_i: $(\lambda_i, \tau_i) \rightarrow (\lambda_{i+1}, \tau_{i+1})$, for i=0 to m, where $(\lambda_k,\ \tau_k)$ is given as follows:

$$\begin{array}{rl} (0,\ 0) & for\ k = 0 \\ (\frac{C_p}{f_k},\ \tau_{k-1} + (\lambda_k - \lambda_{k-1})F_k) & for\ 1 \leq k \leq m; f_k \geq \frac{C_p}{nC_s} \\ (nC_s,\ \tau_{k-1} + (\lambda_k - \lambda_{k-1})F_k) & for\ 1 \leq k \leq m; f_k < \frac{C_p}{nC_s} \\ (\infty,\ \tau_m) & for\ k = m+1 \\ where\ F_k = (1 - \sum_{i=k}^{m} f_i) & \end{array}$$

The initial line segment represents 100% success rate with slope 1. At the request load of $\frac{C_p}{f_1}$, server P_1 reaches its capacity and drops any additional request load. So the capacity increases at rate equal to the remaining fraction of requests that go to the other non-overloaded servers, P_k, $k = 2, 3, ..., m$. This gives the slope $F_1 = (1 - (f_2 + f_3 + ... + f_m))$ for the second line segment. Similarly, P_2 reaches its capacity at load $\frac{C_p}{f_2}$, and so on. When all the second stage servers are overloaded the throughput remains constant, giving the last line segment. At the request load of nC_s, all the first stage servers, S_i, reach their capacity limit. If the second stage server P_j's capacity, C_p is more than the load it receives at that time, $f_j(nC_s)$, then the system throughput is not limited by P_j.

We used a set of hundred user identifiers for test. The hash function we used distributed these identifiers as follows: for $m = 2$, f is roughly $\{0.6, 0.4\}$, and for $m = 3$, f is roughly $\{0.4, 0.3, 0.3\}$. Note that with 1000 or 10 000 user identifiers the same hash function distributed the set more uniformly as expected, but our skewed distribution of hundred identifiers helps us verify the results assuming non-uniform call distribution for different users. The capacity C_s and C_p are 900 CPS and 1050 CPS, respectively. The resulting theoretical performance is shown in Fig. 16 for s1p2, s2p2, s2p3 and s3p3 with system capacity of 1050, 1740, 2100 and 2700 CPS, respectively. Although S_2P_2's second stage can handle $900 \times 2 = 1800$ CPS, the throughput of the first stage is only $1050 \times 2 = 2100$, out of which 60% (i.e., 1260 CPS) goes to $P1$ which drops $1260 - 900 = 360$ CPS. So the system throughput is $2100 - 360 = 1740$ CPS. Our experimental results are plotted as data points (not average, but individual throughput values) in the same graph for comparison.

5.3 Non-uniform call distribution

If the call requests to the user population among the different second stage servers is non-uniformly distributed, then the system starts dropping the call requests at a load lower than the combined capacity of the second stage servers. To prevent this, the user data should be redistributed among the second stage servers to provide an uniform distribution on an average, e.g., by changing the hash function. Fig. 17 compares the two experiments for the S_2P_2 configuration: one with the earlier skewed hash function that distributed the user identifiers in ratio 60:40 and another hash function (Bernstein's hash [33]), that distributed the user identifiers in ratio 50:50.

If the number of second-stage groups changes frequently, then a consistent hashing function [34] is desirable that avoid large redistribution of the user identifiers among the servers.

5.4 Stateful proxy

So far we have shown the test results using the stateless proxy mode. A SIP request over UDP that needs to be proxied to only one destination (i.e., no request forking), can be proxied statelessly. Our SIP server, sipd, can be configured to try the stateless mode, if possible, for every request that needs to be proxied. If a request can *not* be proxied statelessly, sipd falls back to the transaction stateful mode for that request. Stateful mode requires more processing and state in the server, e.g., for matching the responses against the request.

We ran one experiment by disabling the stateless proxy mode in the second stage. Fig. 18 shows the experimental results along with the theoretical through-

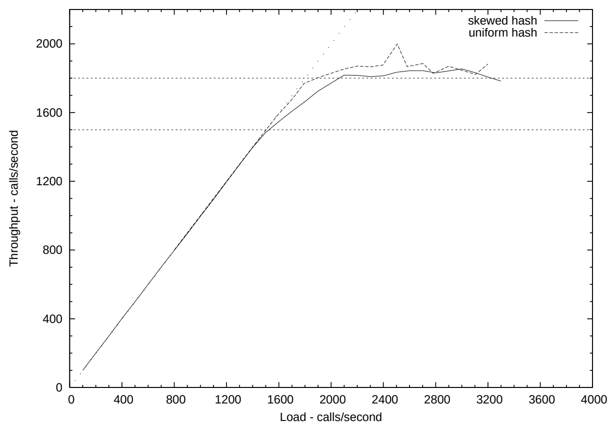


Figure 17: Effect of user identifier distribution among second stage servers for S2P2. Uniform distribution gives the best performance, i.e., success rate is close to 100% until the peak performance (1800 CPS), whereas for non-uniform distribution the success rate reduces as soon as one of the server is overloaded (at 1500 CPS).

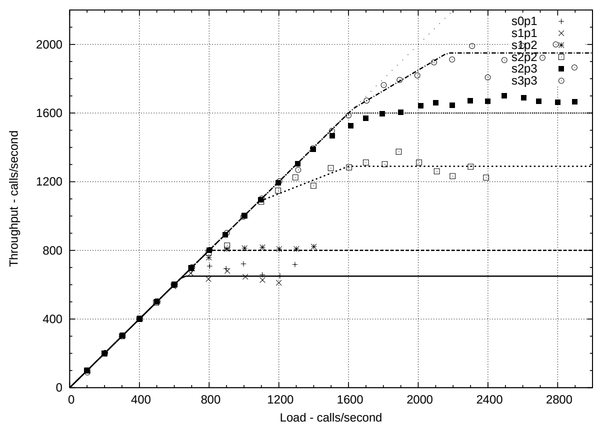


Figure 18: Performance of S_nP_m with stateful proxy in second stage. The results show that the performance increases linearly with the number of servers, i.e., s2p2 is twice and s3p3 is thrice that of s1p1 and s0p1 performance.

put using the earlier hash function. The first and second stage server capacities are C=800 and C'=650 CPS, respectively. The first stage server capacity is less if the second stage is stateful (800 CPS) compared to the case when the second stage is stateless (1050 CPS), because the stateful second stage server generates two additional 100 Trying SIP responses for INVITE and BYE in a call that increases the number of messages handled by the first stage server (See Fig. 19 and 14). If a fraction, f_s, of the input load needs to be handled using stateful mode (e.g., due to request forking to multiple callee devices), then the effective server capacity becomes $(1 - f_s)C + f_sC'$.

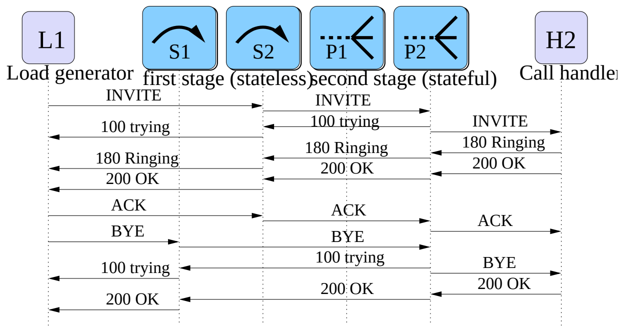


Figure 19: Stateful proxy message flow

Our more recent optimizations enhance the single second stage server throughput to 1200 CPS and 1600 CPS for stateful and stateless proxy, respectively.

5.5 Effect of DNS

In our earlier experiments, the call handler registered the DNS host name with the proxy server so that the server does DNS lookup for locating the call handler host. We observed comparatively poor performance, e.g., a single proxy server capacity with DNS was 110 CPS on the same hardware, compared to 900 CPS without DNS. There were two problems in our implementation: (1) it used a blocking DNS resolver that waits for the query to complete so the internal request queue builds up if the DNS latency is more than the average interarrival duration; and (2) it did not implement any host-cache for DNS, so the second stage server did DNS lookup for every call request. We also observed some fluctuations in throughput even before the server reached its capacity. This was due to the fact that the DNS server was not in the same network, and the DNS procedure took between 10 to 25 ms for each call. In our tests, sipd sent about 29 DNS queries for each call due to multiple resolver search domains (six in our tests) and DNS records (e.g., sipd tries NAPTR, SRV and A records, falling back in that order) used in the implementation.

Then, we implemented a simple DNS host-cache in sipd and observed same performance as that without DNS (i.e., 900 CPS for single second stage server). In practice, the first-stage servers access records for the second-stage servers within the same domain, thus, doing localized DNS queries in the domain. It will be interesting to measure the host-cache performance for the real callee host names by the second-stage servers, instead of a few call handler host names that were cached after the first lookups until the end of the test run in our tests. We plan to use an event-based DNS resolver to improve the performance and eliminate the potential bottleneck due to DNS access.

5.6 Other SIPstone tests

We also performed one experiment with Registration test without authentication. The performance is shown in Fig. 20 along with the expected throughput values. We used capacity values as C_s=2500 registrations/second (RPS) and C_p=2400 RPS for first and second stage servers respectively. Authentication requires two transactions, thus reducing the capacity to half. Thus, the S_3P_3 configuration will be able to support more than 10 million subscribers assuming one hour registration refresh interval.

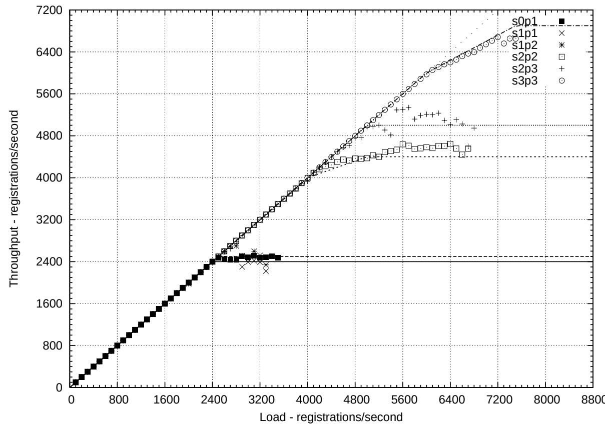


Figure 20: Performance for S_nP_m with registration server in second stage. The results show that the performance increases linearly with the number of servers, i.e., s2p2 is twice and s3p3 is thrice that of s1p1 and s0p1 performance.

Note that the second stage registrar is always stateful. Moreover, we used the database refresh rate to be more than the test duration, thus, removing the database synchronization variable from the results. The effect of database synchronization on performance is for further study. The first stage proxy server capacity for the registration test is more because the number of messages per transaction that it handles is two in the registration test compared to six in the Proxy 200 test (see Fig. 21 and 14).

The Proxy 200 test determines the BHCA (busy hour call attempts) metric, whereas the registration test determines the number of registered subscribers for the system.

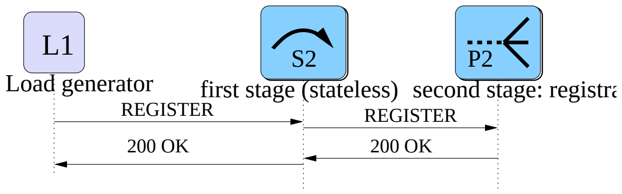


Figure 21: REGISTER message flow

6 Server Architecture

There are two components in providing high capacity IP telephony services: network components such as bandwidth, server location and load sharing, and server components such as server hardware (CPU, memory), features vs. performance tradeoff, non-blocking I/O and software architecture. In general, scaling any Internet service involves individual server performance enhancements and load distribution among multiple servers. We described and evaluated SIP load sharing in Sections 4 and 5. This section deals with performance enhancements on an individual SIP server on commodity server hardware. In particular, we evaluate the effect of software architecture - events, threads or processes - for the SIP proxy server. We try to answer the following questions: (1) For a SIP-style server, which of the basic architectures is likely to perform better in a given situation? (2) Does performance scale with CPU speed or is it memory dominated? (3) What can be done to improve the performance on a multiprocessor machine?

We built a very basic SIP server in different software architectures using the same set of libraries for SIP processing. This helps us in understanding the effect of the server architecture on performance. The server includes a parser module and has many simplifications such as memory-only lookups without any database write-through, no SIP Route header handling, minimal configuration, only UDP transport (i.e., no TCP or TLS), no programmable scripts, and no user authentication. We used standard POSIX threads, which map to kernel-level threads on Solaris and Linux. On a multi-processor hardware, concurrency is utilized via multi-threading and multi-processing software architecture. Our goal is to use commodity hardware without any custom tweaks, hence optimized user-level threads and CPU scheduler reconfiguration are not investigated.

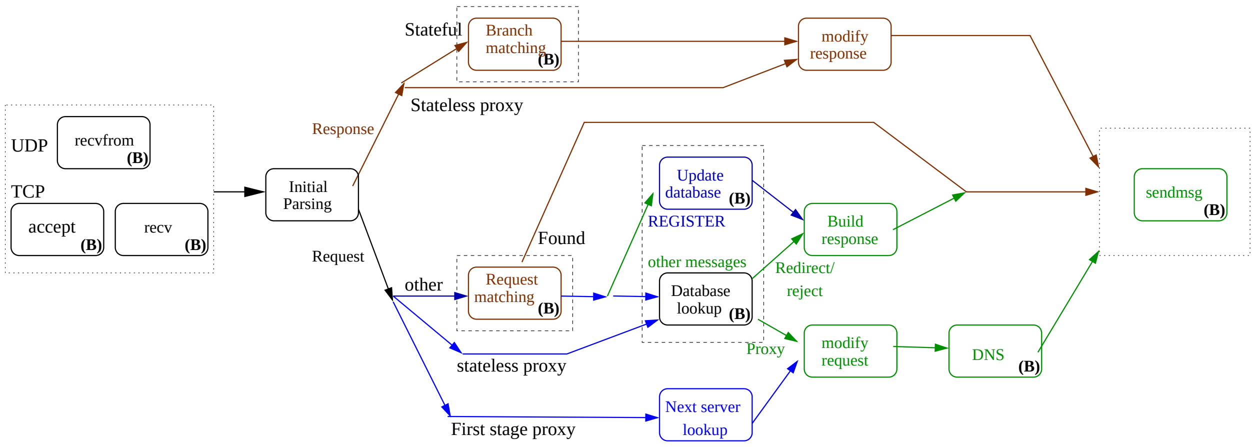


Figure 22: Processing steps in a SIP server

6.1 Processing steps

Figure 22 describes the steps involved in processing a SIP request in any SIP server. It includes both transaction stateless and stateful processing. The server receives a message on UDP, TCP, SCTP, or TLS transport. We use only UDP in our tests. The message is parsed using our unoptimized SIP parser. If the message is a SIP request, it is matched against the existing transactions. If a matching transaction is found, the request is a retransmission and the last response, if any, in the transaction is returned for the request. If a match is not found, and the request is a SIP REGISTER request, then the user contact records are updated for this registration and a response is sent. For any other request, the user record is looked up. Depending on the policy chosen, the call is then proxied, redirected or rejected. In the proxy mode, the server looks up the callee's current contact locations, forwards the request and waits for the response. During this process, the server may need to perform additional retransmissions for reliability. When receiving a response, the server looks up the appropriate matching transaction for this response and forwards the response. If the policy decides to redirect the request instead of proxying it, the server sends the response to the caller listing the new contact location(s). The first stage load balancing server selects the next stage server based on the destination user identifier, without doing any database query. The steps are based on our SIP server implementation, but are likely to be similar for other implementations.

These processing steps can be implemented in various software architectures for both stateless and stateful proxy modes.

6.2 Stateless proxy

A stateless proxy does not maintain any transaction state, and has a single control flow per message. That means, once a message is received, it can be processed to the end without interfering with other messages. We used only UDP transport for our tests and did not perform any DNS lookups. As shown in Figure 14, a single Proxy 200 test involves six messages. In Figure 22, for an incoming call request, the steps performed are recvfrom, initial parsing, database lookup, modify request and sendmsg. Similarly, for an incoming call response the steps performed are recvfrom, initial parsing, modify response and sendmsg. A first stage load balancer proxy is also a stateless proxy, but it does not include the database lookup stage. The processing steps can be implemented in different software architectures as follows:

Event-based: A single thread listens for incoming message and processes it to the end. There is no locking. This does not take advantage of the underlying multiprocessor architecture. If DNS is used, then the same thread also listens for events such as the DNS response and timeout.

Thread per message: A main thread listens for incoming messages. A new parsing thread is created to do the initial parsing. Then another processing thread is created to perform the remaining steps depending on whether the message is a request or a response. The thread terminates after the steps are completed. DNS lookups, if any, are performed synchronously in the processing thread. Locks (i.e., mutexes) are used for accessing shared data such as database. Potentially blocking operations include DNS, sendmsg, and database lookup.

Pool-thread per message: This is similar to the previous method, except that instead of creating a new thread, it reuses a thread from a thread pool. A set of threads are created in the thread pool on server initialization and persist throughout the server lifetime. This reduces the thread creation overhead and is the original architecture of our SIP server, sipd [23]. To further reduce lock contention, the user data can be divided into multiple sets (say, 100), each with its own transaction tables or user records. Thus, access to user records in different sets do not contend for the same lock.

Process pool: On server initialization, a pool of identical processes is created, all listening on the same socket. When a message is received, one of the processes gets the socket message and performs all the processing steps for that message. Shared memory is used for sharing the database among multiple processes. This is the architecture of the SIP express router [35].

Thread pool: This is similar to the previous method, but it uses threads instead of processes. Only one thread can call recvfrom on the listening socket. If a thread has called recvfrom, then another thread is blocked from calling this function until the first thread finishes the function call.

Software architecture /Hardware	1xP	4xP	1xS	2xS
Event-based	1550	400	150	600
Thread per message	1300	500	100	500
Pool-thread per message (sipd)	1400	850	110	600
Thread pool	1500	1300	152	750
Process pool	1600	1350	160	1000

Table 1: Performance (CPS) of stateless proxy for Proxy 200 test

We ran our tests on four different platforms as follows: (1xP) Pentium 4, 3 GHz, 1 GB running Linux 2.4.20, (4xP) four-processor Pentium 450 MHz, 512 MB running Linux 2.4.20, (1xS) ultraSparc-IIi, 300 MHz, 64 MB running Solaris 5.8, and (2xS) two-processor ultraSparc-III+, 900 MHz, 2 GB running Solaris 5.8. The results of our tests are shown in Table 1. The numbers presented in this section are different from earlier load sharing experiments of sipd, because these tests were done after some optimizations such as per-transaction memory pool to reduce memory deallocation and copy [23]. The performance of different architectures relative to the event-based model on different platforms is shown in Figure 23.

For a single processor system (1xP and 1xS), the performances of event-based, thread pool and process pool are roughly similar. We found that the thread pool model had higher context switches compared to process pool. In the process pool model the same process keeps getting scheduled for handling subsequent requests. This resulted in the slight difference in the performance. The process pool model performs the best. The thread-per-message and pool-thread-per-message models have many fold higher context switches resulting in much poorer performance. This is because every message processing must involve at least two context switches. One interesting observation is that both the single processor systems (1xP and 1xS) took approximately 2 MHz CPU cycle per CPS (calls per second) load.

For a multiprocessor system, the performance of the process pool implementation scales linearly with the number of processors. The performance of the pool-thread-per-message model is much worse than process pool because the former does not fully utilize the available concurrency of multiprocessor hardware. The processor running the main listening thread becomes the bottleneck.

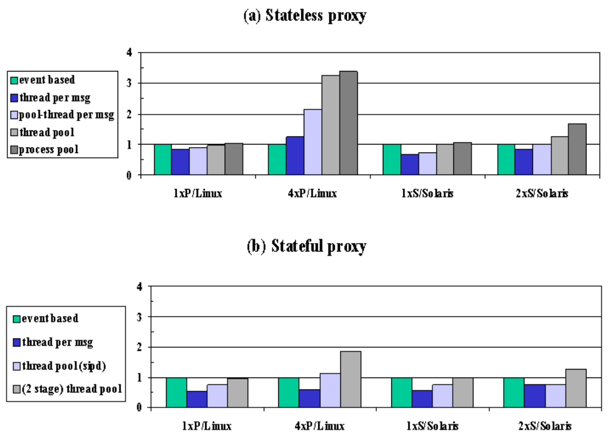


Figure 23: Performance of software architectures relative to event-based on different hardware. For example, the performance of stateless proxy on 4xP hardware in the thread pool architecture is approximately three times that in the event-based architecture on the same hardware.

6.3 Stateful proxy

Unlike the stateless proxy, a transaction stateful proxy needs to maintain the SIP transaction state for the duration of the transaction. We used only UDP transport for our tests and did not perform any DNS lookup. As shown in Figure 19, a single Proxy 200 test involves six incoming and eight outgoing messages. In Figure 22, compared to the stateless proxy, the stateful proxy performs additional steps such as transaction (or client branch) matching. The transactions data structures are locked for exclusive access in a multi-threaded system. The processing steps can be implemented in different software architectures as follows:

Event-based: Most of the blocking operations are made non-blocking using events. A single thread handles events from a queue (e.g., timer events) as well as messages from the listening socket. There is no locking or mutexes. There are only two operations that remain blocking: listening for incoming message on the socket, and listening for events on the event queue. A single threaded event-based system does not take advantage of the underlying multiprocessor architecture.

Thread per message (or transaction): A main thread listens for incoming messages. If the message is a request not matching any previous transaction, then a new thread is created to handle the new transaction associated with this message. The thread persists as long as the transaction exists. Similarly, a process-per-message model can be defined that creates a new process for each incoming connection and message.

Thread pool: This is similar to the previous method, except that instead of creating a new thread, it reuses a thread from the thread pool. This reduces the thread creation overhead. Locks are used for accessing shared data. Potentially blocking operations include DNS lookup, sendmsg, request matching and database access. This is the original architecture of our SIP server, sipd [23].

(Two-stage) thread pool: A pool of identical threads is created. Each thread handles a specific subset of the user population based on the hash value of the user identifier, similar to the second stage of our load sharing architecture. A request is processed in two stages. The first stage thread listens for incoming messages, does minimum parsing, and chooses the second stage thread based on the destination user identifier. The message is then handed over to the particular second stage thread. The second stage is purely event-based with no other locking. Since a single thread handles the requests for the same set of users, we do not need to lock the database or transaction data structures. Number of threads in the thread pool is determined by the number of processors.

The models can be further extended to processes as follows. We have not evaluated these extensions yet:

Process pool: A pool of identical processes is created, each listening on the same socket. When a message is received, the server performs all the processing steps for that message. Shared memory is used for sharing the transaction and user contacts among multiple processes. This is the architecture of the SIP express router [35].

Two-stage event and process-based: This is similar to the two-stage thread pool model, but using processes instead of threads. The inter-process communication is done using pipes or shared memory. Multiple first stage processes can be used to allow more concurrency.

A generic design of thread-per-message is easy to understand and implement. However this model suffers from poor performance at higher load [36]. As the load increases the number of threads in the system also increases. If the thread blocks waiting for a network response, the maximum number of simultaneous requests active in the system is small. Transaction lifetime further reduces the system capacity. For example, if the OS supports 10000 threads, and the SIP transaction lifetime is about 30 seconds, then there can be at most 10000/30 = 333 transactions/second processed in the system. Unlike a web server, this is further exacerbated in a SIP server by the fact that about 70% of calls are answered within roughly 8.5 seconds [37] while unanswered calls ring for 38 seconds. Thus, a bad design results in insufficient number of threads. This leads to higher call blocking or call setup delays at high call volume. Thus, we need to use a true event-driven architecture which requires the threads to be returned to the free-threads pool whenever they make a blocking call.

Table 2 and Figure 23 compare the performance of stateful proxy in different architectures on the same set of hardware, except that 1xS is replaced by a single-processor ultraSparc-IIi, 360 MHz, 256 MB, running Solaris5.9. Event-based system performs best for single processor machine. For an N-processor machine, the thread pool performance is much worse than N times the single-processor performance due to memory access contentions.

Software architecture /Hardware	1xP	4xP	1xS	2xS
Event-based	1150	300	160	400
Thread per message	600	175	90	300
Thread pool (sipd)	850	340	120	300
2-stage thread pool	1100	550	155	500

Table 2: Performance (CPS) for stateful proxy for Proxy 200 test

6.4 The best architecture

The two-stage thread pool model for stateful proxy and the thread pool model for stateless proxy combine the event and thread pool architectures. They provide an event-loop in each thread, and has a pool of threads for concurrency on multiprocessor machines. The lock contention is reduced by allowing the same thread to process all the steps of a message or transaction after initial parsing. For a multi-threaded software architecture this seems to give the best performance as per our tests. We have not yet evaluated the stateful proxy in process pool model.

The stateless proxy performance is usually limited by the CPU speed, whereas the memory utilization remains constant. On the other hand, the stateful proxy may be limited by either CPU or memory depending of various transaction timers. By default a SIP transaction state is maintained for about 30 seconds. Thus, a load of 1000 CPS creating 2000 transactions per second will require memory for about 60 thousand transactions. Assuming 10 kB for storing each transaction state, this requires 600 MB. In our tests, we have reduced the timer values significantly so that memory is not the bottleneck.

6.5 Effect on Load Sharing Performance

The software architecture choice of the SIP server further enhances the load sharing results since the best single stateless proxy capacity is about 1600 CPS on a 3 GHz Pentium 4 with 1 GB memory running Linux 2.4.20. In addition, we have achieved about 4000 CPS throughput for the first stage proxy in a simplified implementation. This means even S1P2 in stateless proxy mode can achieve close to 3200 CPS, i.e., 11 million BHCA on this hardware configuration. Similarly, S3P3 in stateful proxy mode can achieve close to 13 million BHCA.

7 Conclusions

We have shown how to apply some of the existing failover and load sharing techniques to SIP servers, and propose an identifier-based two-stage load sharing method. Using DNS is the preferred way to offer redundancy since it does not require network co-location of the servers. For example, one can place SIP servers on different networks. With IP address takeover and NATs, that is rather difficult. This is less important for enterprise environments, but interesting for voice service providers such as Vonage. DNS itself is replicated, so a single name server outage does not affect operation. We combine DNS, server redundancy and the identifier-based load sharing in our two-stage reliable and scalable server architecture that can theoretically scale to any capacity. A large user population is divided among independent second stage servers such that each server load remains below its capacity.

We have also described the failover implementation and performance evaluation of our two-stage architecture for scalability using the SIPstone test suite in our test bed. Our results verify the theoretical improvement of load sharing for call handling and registration capacity. We achieve carrier grade scalability using commodity hardware, e.g., 2800 calls/second supported by our S_3P_3 load sharing configuration roughly translates to 10 million call arrivals per hour, using six servers (Lucent's 5E-XC^{TM} switch, a high-end 5ESS, can support four million BHCA for PSTN). This is further increased to 16 million BHCA in our memory pool and event-based architecture. We also achieved the 5-nines reliability goal even if each server has only uptime of 99% (3 days/year downtime) using the two-stage architecture. Other call stateful services such as voicemail, conferencing and PSTN interworking need more work to do failover and load sharing in the middle of the call without breaking the session.

Detection and recovery of wide area path outages [38] is complementary to the individual server failover. Instead of statically configuring the redundant servers, it will be useful if the servers can automatically discover and configure other available servers on the Internet, e.g., to handle temporary overload [31]. This gives rise to the service model where the provider can sell its SIP services dynamically by becoming part of another customer SIP network. Dynamic or adaptive load sharing based on the workload of each server is for further study. However, it is not clear how useful this will be for Internet telephony because the call distribution is more uniform unlike Zipf distribution of web page popularity. Therefore, a good static hash function can uniformly distribute the call requests among the servers. A peer-to-peer approach for the SIP service also seems promising for scalability and robustness [39].

Acknowledgment

Huitao Sheng helped with the initial performance measurement. Jonathan Lennox is the primary architect of our SIP server, sipd, and helped with SIPstone tools. Sankaran Narayanan implemented the efficient database interaction in sipd. The work is supported by a grant from SIPquest, Inc.

References

[1] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. R. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: session initiation protocol," RFC 3261, Internet Engineering Task Force, June 2002.

[2] H. Schulzrinne and J. Rosenberg, "Internet telephony: Architecture and protocols – an IETF perspective," *Computer Networks and ISDN Systems*, vol. 31, pp. 237–255, Feb. 1999.

[3] J. Rosenberg and H. Schulzrinne, "Session initiation protocol (SIP): locating SIP servers," RFC 3263, Internet Engineering Task Force, June 2002.

[4] H. BRYHNI, E. Klovning, and Øivind Kure, "A comparison of load balancing techniques for scalable web servers," *IEEE Network*, vol. 14, July 2000.

[5] K. Suryanarayanan and K. J. Christensen, "Performance evaluation of new methods of automatic redirection for load balancing of apache servers distributed in the Internet," in *IEEE Conference on Local Computer Networks*, (Tampa, Florida, USA), Nov. 2000.

[6] O. Damani, P. Chung, Y. Huang, C. Kintala, and Y. Wang, "ONE-IP: techniques for hosting a service on a cluster of machines," *Computer Networks*, vol. 29, pp. 1019–1027, Sept. 1997.

[7] D. Oppenheimer, A. Ganapathi, and D. Patterson, "Why do internet services fail, and what can be done about it?," in *4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, (Seattle, WA), Mar. 2003.

[8] A. C. Snoeren, D. Andersen, and H. Balakrishnan, "Fine-grained failover using connection migration," in *USENIX Symposium on Internet Technologies and Systems*, (San Francisco), Mar. 2001.

[9] High-Availability Linux Project, http://www.linux-ha.org/.

[10] Cisco Systems, Failover configuration for LocalDirector, http://www.cisco.com/warp/public/cc/pd/cxsr/400/tech/locdf_wp.htm.

[11] G. Hunt, G. Goldszmidt, R. P. King, and R. Mukherjee, "Network dispatcher: a connection router for scalable Internet services," *Computer Networks*, vol. 30, pp. 347–357, Apr. 1998.

[12] C.-L. Yang and M.-Y. Luo, "Efficient support for content-based routing in web server clusters," in *2nd USENIX Symposium on Internet Technologies and Systems*, (Boulder, Colorado, USA), Oct 1999.

[13] Akamai Technologies, Inc. http://www.akamai.com.

[14] A. Gulbrandsen, P. Vixie, and L. Esibov, "A DNS RR for specifying the location of services (DNS SRV)," RFC 2782, Internet Engineering Task Force, Feb. 2000.

[15] M. Mealling and R. W. Daniel, "The naming authority pointer (NAPTR) DNS resource record," RFC 2915, Internet Engineering Task Force, Sept. 2000.

[16] N. Ohlmeier, "Design and implementation of a high availability sip server architecture," Thesis, Computer Science Department, Technical University of Berlin, Berlin, Germany, July 2003.

[17] M. Tuexen, Q. Xie, R. Stewart, M. Shore, J. Loughney, and A. Silverton, "Architecture for reliable server pooling," Internet Draft draft-ietf-rserpool-arch-09, Internet Engineering Task Force, Feb 2005. work in progress.

[18] M. Tuexen, Q. Xie, R. J. Stewart, M. Shore, L. Ong, J. Loughney, and M. Stillman, "Requirements for reliable server pooling," RFC 3237, Internet Engineering Task Force, Jan. 2002.

[19] P. Conrad, A. Jungmaier, C. Ross, W.-C. Sim, and M. Tüxen, "Reliable ip telephony applications with sip using RSerPool," in *World Multiconference on Systemics, Cybernetics and Informatics (SCI)*, (Orlando, USA), July 2002.

[20] A. Srinivasan, K. G. Ramakrishnan, K. Kumaran, M. Aravamudan, and S. Naqvi, "Optimal design of signaling networks for Internet telephony," in *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, (Tel Aviv, Israel), Mar. 2000.

[21] R. Sparks, "The session initiation protocol (SIP) refer method," RFC 3515, Internet Engineering Task Force, Apr. 2003.

[22] J. Janak, "SIP proxy server effectiveness," Master's Thesis, Department of Computer Science, Czech Technical University, Prague, Czech, May 2003.

[23] J. Lennox, "Services for internet telephony," PhD. thesis, Department of Computer Science, Columbia University, New York, New York, Jan. 2004. http://www.cs.columbia.edu/~lennox/thesis.pdf.

[24] Cisco IP phone 7960, Release 2.1, http://www.cisco.com.

[25] J. Jung, E. Sit, H. Balakrishnan, and R. Morris, "DNS performance and the effectiveness of caching," in *ACM SIGCOMM Internet Measurement Workshop*, (San Francisco, California), Nov. 2001.

[26] K. Singh, W. Jiang, J. Lennox, S. Narayanan, and H. Schulzrinne, "CINEMA: columbia internet extensible multimedia architecture," technical report CUCS-011-02, Department of Computer Science, Columbia University, New York, New York, May 2002.

[27] W. Jiang, J. Lennox, S. Narayanan, H. Schulzrinne, K. Singh, and X. Wu, "Integrating Internet telephony services," *IEEE Internet Computing*, vol. 6, pp. 64–72, May 2002.

[28] MySQL, Open Source SQL server, http://www.mysql.com.

[29] K. Singh and H. Schulzrinne, "Failover and load sharing in SIP telephony," Tech. Rep. CUCS-011-04, Columbia University, Computer Science Department, New York, NY, USA, Mar. 2004.

[30] P. Srisuresh and D. Gan, "Load sharing using IP network address translation (LSNAT)," RFC 2391, Internet Engineering Task Force, Aug. 1998.

[31] W. Zhao and H. Schulzrinne, “Dotslash: A self-configuring and scalable rescue system for handling web hotspots effectively,” in *International Workshop on Web Caching and Content Distribution (WCW)*, (Beijing, China), Oct. 2004.

[32] H. Schulzrinne, S. Narayanan, J. Lennox, and M. Doyle, “SIPstone - benchmarking SIP server performance,” Technical Report CUCS-005-02, Department of Computer Science, Columbia University, New York, New York, Mar. 2002.

[33] B. Jenkins, “Algorithm alley,” *Dr. Dobb’s Journal*, Sept. 1997. http://burtleburtle.net/bob/hash/doobs.html.

[34] D. R. Karger, A. H. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B.-J. J. Kim, and L. Matkins, “Web caching with consistent hashing,” *Computer Networks*, vol. 31, pp. 1203–1213, May 1999.

[35] “SIP express router (ser): a high performance free sip server.” http://www.iptel.org/ser.

[36] M. Welsh, D. Culler, and E. Brewer, “SEDA: an architecture for well-conditioned, scalable Internet services,” in *Symposium on Operating Systems Principles (SOSP)*, (Chateau Lake Louise, Canada), ACM, Oct. 2001.

[37] F. P. Duffy and R. A. Mercer, “A study of network performance and customer behavior during-direct-distance-dialing call attempts in the USA,” *Bell System Technical Journal*, vol. 57, no. 1, pp. 1–33, 1978.

[38] D. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris, “Resilient overlay networks,” in *18th ACM SOSP*, (Banff, Canada), Oct. 2001.

[39] K. Singh and H. Schulzrinne, “Peer-to-peer Internet telephony using SIP,” Tech. Rep. CUCS-044-04, Department of Computer Science, Columbia University, New York, NY, Oct. 2004.