Design, Implementation, and Performance of A Load Balancer for SIP Server Clusters

Hongbo Jiang*, Arun Iyengar†, Erich Nahum†, Wolfgang Segmuller†, Asser Tantawi†, and Charles P. Wright†
*Huazhong University of Science and Technology
†IBM T.J. Watson Research Center

***Abstract*—This paper introduces several novel load balancing algorithms for distributing Session Initiation Protocol (SIP) requests to a cluster of SIP servers. Our load balancer improves both throughput and response time versus a single node, while exposing a single interface to external clients. We present the design, implementation and evaluation of our system using a cluster of Intel x86 machines running Linux. We compare our algorithms with several well-known approaches and present scalability results for up to 10 nodes. Our best algorithm, Transaction Least-Work-Left (TLWL), achieves its performance by integrating several features: knowledge of the SIP protocol; dynamic estimates of back-end server load; distinguishing transactions from calls; recognizing variability in call length; and exploiting differences in processing costs for different SIP transactions. By combining these features, our algorithm provides finer-grained load balancing than standard approaches, resulting in throughput improvements of up to 24 percent and response time improvements of up to two orders of magnitude. We present a detailed analysis of occupancy to show how our algorithms significantly reduce response time.**

I. Introduction

The Session Initiation Protocol (SIP) is a general-purpose signaling protocol used to control various types of media sessions. SIP is a protocol of growing importance, with uses in Voice over IP, Instant Messaging, IPTV, Voice Conferencing, and Video Conferencing. Wireless providers are standardizing on SIP as the basis for the IP Multimedia System (IMS) standard for the Third Generation Partnership Project (3GPP). Third-party VoIP providers use SIP (e.g., Vonage, Gizmo), as do digital voice offerings from existing legacy Telcos (e.g., AT&T, Verizon) as well as their cable competitors (e.g., Comcast, Time-Warner).

While individual servers may be able to support hundreds or even thousands of users, large-scale ISPs need to support customers in the millions. A central component to providing any large-scale service is the ability to *scale* that service with increasing load and customer demands. A frequent mechanism to scale a service is to use some form of a load-balancing dispatcher that distributes requests across a cluster of servers. However, almost all research in this space has been in the context of either the Web (e.g., HTTP [27]) or file service (e.g., NFS [1]). This paper presents and evaluates several algorithms for balancing load across multiple SIP servers. We introduce new algorithms which outperform existing ones. Our work is relevant not just to SIP but also for other systems where it is advantageous for the load balancer to maintain sessions in which requests corresponding to the same session are sent by the load balancer to the same server.

SIP has a number of features which distinguish it from protocols such as HTTP. SIP is a transaction-based protocol designed to establish and tear down media sessions, frequently referred to as calls. Two types of state exist in SIP. The first, session state, is created by the `INVITE` transaction and is destroyed by the `BYE` transaction. Each SIP transaction also creates state that exists for the duration of that transaction. SIP thus has overheads that are associated both with sessions and with transactions, and taking advantage of this fact can result in more optimized SIP load balancing.

The session-oriented nature of SIP has important implications for load balancing. Transactions corresponding to the same call must be routed to the same server; otherwise, the server will not recognize the call. Session-aware request assignment (SARA) is the process where a system assigns requests to servers such that sessions are properly recognized by that server, and subsequent requests corresponding to that same session are assigned to the same server. In contrast, sessions are less significant in HTTP. While SARA can be done in HTTP for *performance* reasons (e.g., routing SSL sessions to the same back end to encourage session reuse and minimize key exchange [14]), it is not necessary for *correctness*. Many HTTP load balancers do not take sessions into account in making load balancing decisions.

Another key aspect of the SIP protocol is that different transaction types, most notably the `INVITE` and `BYE` transactions, can incur significantly different overheads: On our systems, `INVITE` transactions are about 75 percent more expensive than `BYE` transactions. A load balancer can make use of this information to make better load balancing decisions which improve both response time and throughput. Our work is the first to demonstrate how load balancing can be improved by combining SARA with estimates of relative overhead for different requests.

This paper introduces and evaluates several novel algorithms for balancing load across SIP servers. Each algorithm combines knowledge of the SIP protocol, dynamic estimates of server load, and Session-Aware Request Assignment (SARA). In addition, the best-performing algorithm takes into account the variability of call lengths, distinguishing transactions from calls, and the difference in relative processing costs for different SIP transactions.

1) Call-Join-Shortest-Queue (CJSQ) tracks the number of calls (in this paper, we use the terms *call* and *session*

interchangeably) allocated to each back-end server and routes new SIP calls to the node with the least number of active calls.

2) Transaction-Join-Shortest-Queue (TJSQ) routes a new *call* to the server that has the fewest active *transactions*, rather than the fewest calls. This algorithm improves on CJSQ by recognizing that calls in SIP are composed of the two transactions, `INVITE` and `BYE`, and that by tracking their completion separately, finer-grained estimates of server load can be maintained. This leads to better load balancing, particularly since calls have variable length and thus do not have a unit cost.
3) Transaction-Least-Work-Left (TLWL) routes a new call to the server that has the least *work*, where work (i.e., load) is based on relative estimates of transaction costs. TLWL takes advantage of the observation that `INVITE` transactions are more expensive than `BYE` transactions. On our platform, a 1.75:1 cost ratio between `INVITE` and `BYE` results in the best performance.

We implement these algorithms in software by adding them to the OpenSER open-source SIP server configured as a load balancer. Our evaluation is done using the SIPp open-source workload generator driving traffic through the load balancer to a cluster of servers running a commercially available SIP server. The experiments are conducted on a dedicated testbed of Intel x86-based servers connected via Gigabit Ethernet.

This paper makes the following contributions:

- We introduce the novel load balancing algorithms, CJSQ, TJSQ, and TLWL, described above, and implement them in a working load balancer for SIP server clusters. Our load balancer is implemented in software in user space by extending the OpenSER SIP proxy.
- We evaluate our algorithms in terms of throughput, response time, and scalability, comparing them to several standard "off the shelf" distribution policies such as round-robin or static hashing based on the SIP Call-ID. Our evaluation tests scalability up to 10 nodes.
- We show that two of our new algorithms, TLWL and TJSQ, scale better, provide higher throughputs and exhibit lower response times than any of the other approaches we tested. The differences in response times are particularly significant. For low to moderate workloads, TLWL and TJSQ provide response times for `INVITE` transactions that are an order of magnitude lower than that of any of the other approaches. Under high loads, the improvement increases to two orders of magnitude.
- We present a detailed analysis of why TLWL and TJSQ provide substantially better response times than the other algorithms. *Occupancy* has a significant effect on response times, where the occupancy for a transaction T assigned to a server S is the number of transactions already being handled by S when T is assigned to it. As described in detail in Section VI, by allocating load more evenly across nodes, the distributions of occupancy across the cluster are balanced, resulting in greatly improved response times. The naive approaches, in contrast, lead to imbalances in load. These imbalances result in the distributions of occupancy that exhibit large tails, which contribute significantly to response time as seen by that request. To our knowledge, we are the first to observe this phenomenon experimentally.
- We show how our load balancing algorithms perform using heterogeneous back ends. With no knowledge of the server capacities, our approaches adapt naturally to variations in back-end server processing power.
- We evaluate the capacity of our load balancer in isolation to determine at what point it may become a bottleneck. We demonstrate throughput of up to 5500 calls per second, which in our environment would saturate at about 20 back-end nodes. Measurements using oprofile show that the load balancer is a small component of the overhead, and suggest that moving it into the kernel can improve its capacity significantly if needed.

These results show that our load balancer can effectively scale SIP server throughput and provide significantly lower response times without becoming a bottleneck. The dramatic response time reductions that we achieve with TLWL and TJSQ suggest that these algorithms should be adapted for other applications, particularly when response time is crucial.

We believe these results are general for load balancers, which should keep track of the number of uncompleted requests assigned to each server in order to make better load balancing decisions. If the load balancer can reliably estimate the relative overhead for requests that it receives, this can improve performance even further.

The remainder of this paper is organized as follows: Section II provides a brief background on SIP. Section III presents the design of our load balancing algorithms, and Section IV describes their implementation. Section V overviews our experimental software and hardware, and Section VI shows our results in detail. Section VII discusses related work. Section VIII presents our summary and conclusions, and briefly mentions plans for future work.

II. Background

This section presents a brief overview of SIP. Readers familiar with SIP may prefer to continue to Section III.

A. Overview of the Protocol

SIP is a signalling (control-plane) protocol designed to establish, modify, and terminate media sessions between two or more parties. The core IETF SIP specification is given in RFC 3261 [31], although there are many additional RFCs that enhance and refine the protocol. Several kinds of sessions can be used, including voice, text, and video, which are transported over a separate data-plane protocol. SIP does not allocate and manage network bandwidth as does a network resource reservation protocol such as RSVP [38]; that is considered outside the scope of the protocol.

Figure 1 illustrates a typical SIP VoIP scenario, known as the "SIP Trapezoid." Note the separation between control and

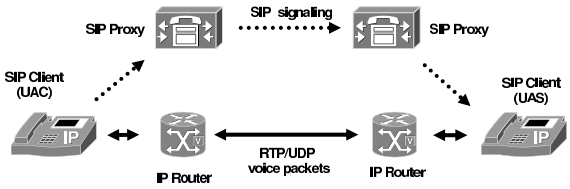

Fig. 1. SIP Trapezoid

data paths: SIP messages traverse the SIP overlay network, routed by proxies, to find the eventual destinations. Once endpoints are found, communication is typically performed directly in a peer-to-peer fashion. In this example, each endpoint is an IP phone; however, an endpoint can also be a server providing services such as voicemail, firewalling, voice conferencing, etc. This work focuses on scaling the server (in SIP terms, the UAS, described below), rather than the proxy.

The separation of the data plane from the control plane is one of the key features of SIP and contributes to its flexibility. SIP was designed with extensibility in mind; for example, the SIP protocol requires that proxies forward and preserve headers that they do not understand. As another example, SIP can run over many protocols such as UDP, TCP, TLS, SCTP, IPv4 and IPv6.

B. SIP Users, Agents, Transactions, and Messages

A SIP Uniform Resource Identifier (URI) uniquely identifies a SIP user, e.g., sip:hongbo@us.ibm.com. This layer of indirection enables features such as location-independence and mobility.

SIP users employ end points known as *user agents*. These entities initiate and receive sessions. They can be either hardware (e.g., cell phones, pages, hard VoIP phones) or software (e.g., media mixers, IM clients, soft phones). User agents are further decomposed into *User Agent Clients* (UAC) and *User Agent Servers* (UAS), depending on whether they act as a client in a transaction (UAC) or a server (UAS). Most call flows for SIP messages thus display how the UAC and UAS behave for that situation.

SIP uses HTTP-like request/response *transactions*. A transaction consists of a request to perform a particular method (e.g., `INVITE`, `BYE`, `CANCEL`, etc.) and at least one response to that request. Responses may be *provisional*, namely, that they provide some short term feedback to the user (e.g., `100 TRYING`, `180 RINGING`) to indicate progress, or they can be *final* (e.g., `200 OK`, `407 UNAUTHORIZED`). The transaction is only completed when a final response is received, not a provisional response.

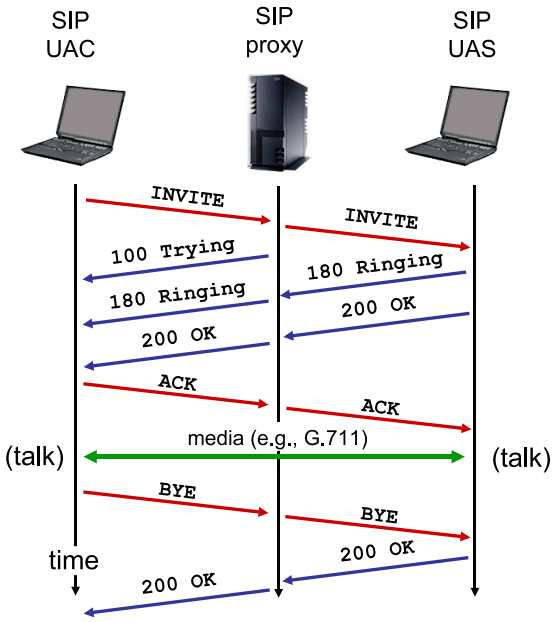

Fig. 2. SIP Message Flow

A SIP session is a relationship in SIP between two user agents that lasts for some time period; in VoIP, a session corresponds to a phone call. This is called a *dialog* in SIP and results in state being maintained on the server for the duration of the session. For example, an `INVITE` message not only creates a transaction (the sequence of messages for completing the `INVITE`), but also a session if the transactions completes successfully. A `BYE` message creates a new transaction and, when the transaction completes, ends the session. Figure 2 illustrates a typical SIP message flow, where SIP messages are routed through the proxy. In this example, a call is initiated with the `INVITE` message, and accepted with the `200 OK` message. Media is exchanged, and then the call is terminated using the `BYE` message.

C. The SIP Message Header

SIP is a text-based protocol that derives much of its syntax from HTTP [12]. Messages contain headers and additionally bodies, depending on the type of message.

In VoIP, SIP messages contain an additional protocol, the Session Description Protocol (SDP) [30], which negotiates session parameters (e.g., which voice codec to use) between end points using an offer/answer model. Once the end hosts agree to the session characteristics, the Real-time Transport Protocol (RTP) is typically used to carry voice data [33].

RFC 3261 [31] shows many examples of SIP headers. An important header to notice is the `Call-ID:` header, which is

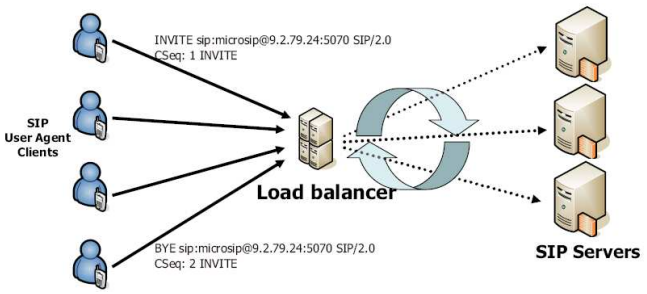

Fig. 3. System Architecture

a globally unique identifier for the session that is to be created. Subsequent SIP messages must refer to that Call-ID to look up the established session state. If a SIP server is provided by a cluster, the initial `INVITE` request will be routed to one back-end node, which will create the session state. Barring some form of distributed shared memory in the cluster, subsequent packets for that session must also be routed to the same back-end node, otherwise the packet will be erroneously rejected. Thus, many SIP load balancing approaches use the `Call-ID` as hashing value in order to route the message to the proper node. For example, Nortel's Layer 4-7 switch product [24] uses this approach.

III. Load Balancing Algorithms

This section presents the design of our load balancing algorithms. Figure 3 depicts our overall system. User Agent Clients send SIP requests (e.g., `INVITE`, `BYE`) to our load balancer which then selects a SIP server to handle each request. The distinction between the various load balancing algorithms presented in this paper are *how* they choose which SIP server to handle a request. Servers send SIP responses (e.g., `180 TRYING` or `200 OK`) to the load balancer which then forwards the response to the client.

Note that SIP is used to establish, alter, or terminate media sessions. Once a session has been established, the parties participating in the session would typically communicate directly with each other using a different protocol for the media transfer which would not go through our SIP load balancer.

A. Novel Algorithms

A key aspect of our load balancer is that requests corresponding to the same call are routed to the same server. The load balancer has the freedom to pick a server *only* on the *first* request of a call. All subsequent requests corresponding to the call must go to the same server. This allows all requests corresponding to the same session to efficiently access state corresponding to the session.

Our new load balancing algorithms are based on assigning calls to servers by picking the server with the (estimated) least amount of work assigned but not yet completed. While the concept of assigning work to servers with the least amount of work left to do has been applied in other contexts [16], [32], the specifics of how to do this efficiently for a real application are often not at all obvious. The system needs some method to reliably estimate the amount of work that a server has left to do at the time load balancing decisions are made.

In our system, the load balancer can estimate the work assigned to a server based on the requests it has assigned to the server and the responses it has received from the server. All responses from servers to clients first go through the load balancer which forwards the responses to the appropriate clients. By monitoring these responses, the load balancer can determine when a server has finished processing a request or call and update the estimates it is maintaining for the work assigned to the server.

1) Call-Join-Shortest-Queue: The *Call-Join-Shortest-Queue (CJSQ)* algorithm estimates the amount of work a server has left to do based on the number of *calls* (sessions) assigned to the server. Counters are maintained by the load balancer indicating the number of calls assigned to each server. When a new `INVITE` request is received (which corresponds to a new call), the request is assigned to the server with the lowest counter, and the counter for the server is incremented by one. When the load balancer receives a `200 OK` response to the `BYE` corresponding to the call, it knows that the server has finished processing the call and decrements the counter for the server.

A limitation of this approach is that the number of calls assigned to a server is not always an accurate measure of the load on a server. There may be long idle periods between the transactions in a call. In addition, different calls may consist of different numbers of transactions and may consume different amounts of server resources. An advantage of CJSQ is that it can be used in environments in which the load balancer is aware of the calls assigned to servers but does not have an accurate estimate of the transactions assigned to servers.

2) Transaction-Join-Shortest-Queue: An alternative method is to estimate server load based on the number of *transactions* (requests) assigned to the servers. The *Transaction-Join-Shortest-Queue (TJSQ)* algorithm estimates the amount of work a server has left to do based on the number of transactions (requests) assigned to the server. Counters are maintained by the load balancer indicating the number of transactions assigned to each server. New calls are assigned to servers with the lowest counter.

A limitation of this approach is that all transactions are weighted equally. In the SIP protocol, `INVITE` requests are more expensive than `BYE` requests, since the `INVITE` transaction state machine is more complex than the one for non-`INVITE` transactions (such as `BYE`). This difference in processing cost should ideally be taken into account in making load balancing decisions.

3) Transaction-Least-Work-Left: The *Transaction-Least-Work-Left (TLWL)* algorithm addresses this issue by assigning different *weights* to different transactions depending on their relative costs. It is similar to TJSQ with the enhancement that transactions are weighted by relative overhead; in the special case that all transactions have the same expected overhead, TLWL and TJSQ are the same. Counters are maintained by the load balancer indicating the *weighted* number of transactions

assigned to each server. New calls are assigned to the server with the lowest counter. A ratio is defined in terms of relative cost of `INVITE` to `BYE` transactions. We experimented with several values for this ratio of relative cost. TLWL-2 assumes `INVITE` transactions are twice as expensive as `BYE` transactions and are indicated in our graphs as *TLWL-2*. We found the best performing estimate of relative costs was 1.75; these are indicated in our graphs as *TLWL-1.75*. Note that if it is not feasible to determine the relative overheads of different transaction types, TJSQ can be used which results in almost as good performance as TLWL-1.75 as will be shown in the results section.

TLWL estimates server load based on the weighted number of transactions a server is currently handling. For example, if a server is processing an INVITE (relative cost of 1.75) and a BYE transaction (relative cost of 1.0), the server has a load of 2.75.

TLWL can be adapted to workloads with other transaction types by using different weights based on the overheads of the transaction types. In addition, the relative costs used for TLWL could be adaptively varied to improve performance. We did not need to adaptively vary the relative costs because the value of 1.75 was relatively constant.

CJSQ, TJSQ, and TLWL are all novel load balancing algorithms. In addition, we are not aware of any previous work which has successfully adapted least work left algorithms for load balancing with SARA.

B. Comparison Algorithms

We also implemented several standard load balancing algorithms for comparison. These algorithms are not novel but are described for completeness.

1) Hash and FNVHash: The *Hash* algorithm is a static approach for assigning calls to servers based on the SIP Call-ID, which is contained in the header of a SIP message identifying the call to which the message belongs. A new `INVITE` transaction with Call-ID x is assigned to server $(Hash(x) mod N)$, where $Hash(x)$ is a hash function and N is the number of servers. This is a common approach to SIP load balancing; both OpenSER and the Nortel Networks Layer 2-7 Gigabit Ethernet Switch module [24] use this approach. We have used both the original hash function provided by OpenSER and FNV hash [25].

2) Round Robin: The hash algorithm is not guaranteed to assign the same number of calls to each server. The *Round Robin (RR)* algorithm guarantees a more equal distribution of calls to servers. If the previous call was assigned to server M, the next call is assigned to server $(M + 1) mod N$, where N is again the number of servers in the cluster.

3) Response-time Weighted Moving Average: Another method is to make load balancing decisions based on server response times. The *Response-time Weighted Moving Average (RWMA)* algorithm [29] assigns calls to the server with the lowest weighted moving average response time of the last n (20 in our implementation) response time samples. The formula for computing the RWMA linearly weights the measurements so that the load balancer is responsive to dynamically changing loads, but does not overreact if the most recent response time measurement is highly anomalous. The most recent sample has a weight of n, the second most recent a weight of $n - 1$, and the oldest a weight of one. The load balancer determines the response time for a request based on the time when the request is forwarded to the server and the time the load balancer receives a 200 `OK` reply from the server for the request.

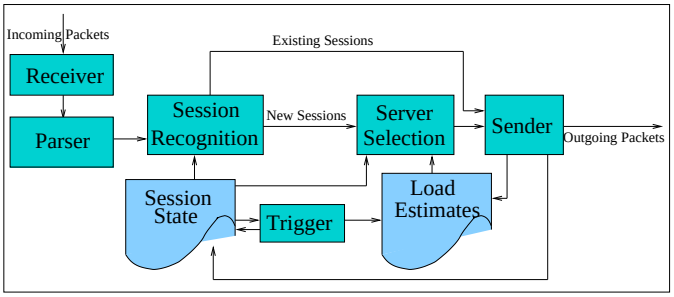

Fig. 4. Load Balancer Architecture

IV. Load Balancer Implementation

This section describes our implementation. Figure 4 illustrates the structure of the load balancer. The rectangles represent key functional modules of the load balancer, while the irregular shaped boxes represent state information that is maintained. The arrows represent communication flows.

The *Receiver* receives requests which are then parsed by the *Parser.* The *Session Recognition* module determines if the request corresponds to an already existing session by querying the *Session State* which is implemented as a hash table as described below. If so, the request is forwarded to the server to which the session was previously assigned. If not, the *Server Selection* module assigns the new session to a server using one of the algorithms described earlier. For several of the load balancing algorithms we have implemented, these assignments may be based on *Load Estimates* maintained for each of the servers. The *Sender* forwards requests to servers and updates *Load Estimates* and *Session State* as needed.

The *Receiver* also receives responses sent by servers. The client to receive the response is identified by the *Session Recognition* module which obtains this information by querying the *Session State*. The *Sender* then sends the response to the client and updates *Load Estimates* and *Session State* as needed. The *Trigger* module updates *Session State* and *Load Estimates* after a session has expired.

Figure 5 shows the pseudocode for the main loop of the load balancer. The pseudocode is intended to convey the general approach of the load balancer; it omits certain corner cases and error handling (for example, for duplicate packets). The essential approach is to identify SIP packets by their Call-ID and use that as a key for identifying calls. Our load balancer selects the appropriate server to handle the first request of a call. It also maintains mappings between calls and servers

```
01:  h = hash call-id
02:  look up session in active table
03:  if not found
04:    /* don't know this session */
05:    if INVITE
06:      /* new session */
07:      select one node d using algorithm
08:      (TLWL, TJSQ, RR, Hash, etc)
09:      add entry (s,d,ts) to active table
10:      s = STATUS_INV
11:      node_counter[d] += w_inv
12:    /* non-invites omitted for clarity */
13:  else /* this is an existing session */
14:    if 200 response for INVITE
15:      s = STATUS_INV_200
16:      record response time for INVITE
17:      node_counter[d] -= w_inv
18:    else if ACK request
19:      s = STATUS_ACK
20:    else if BYE request
21:      s = STATUS_BYE
22:      node_counter[d] += w_bye
23:    else if 200 response for BYE
24:      s = STATUS_BYE_200
25:      record response time for BYE
26:      node_counter[d] -= w_bye
27:      move entry to expired table
28:  /* end session lookup check */
29:  if request (INVITE , BYE etc.)
30:    forward to d
31:  else if response (200/100/180/481)
32:    forward to client
```

Fig. 5. Load Balancing Pseudocode

using two hash tables which are indexed by call ID. That way, when a new transaction corresponding to the call is received, it will be routed to the correct server.

The *active* hash table maintains state information on calls and transactions that the system is currently handling, and an *expired* hash table is used for routing duplicate packets for requests that have already completed. This is analogous to the handling of old duplicate packets in TCP when the protocol state machine is in the TIME-WAIT state [2]. When the load balancer receives a 200 status message from a server in response to a BYE message from a client, the session is completed. The load balancer moves the call information from the active hash table to the expired hash table in order to recognize retransmissions which may arrive later. If a packet corresponding to a session arrives that cannot be found in the active table, the expired table is consulted to determine how to forward the packet, but the systems' internal state machine is not changed (as it would be for a non-duplicate packet). Information in the expired hash table is reclaimed by garbage collection after an appropriate time-out period. Both tables are chained-bucket hash tables where multiple entities can hash to the same bucket in a linked list.

For the Hash and FNVHash algorithms, the process of maintaining an active hash table could be avoided. Instead, the server could be selected by the hash algorithm directly. This means that lines 2-28 in Figure 5 would be removed. However, the overhead for accesses to the active hash table is not a significant component of the overall CPU cycles consumed by the load balancer, as will be shown in Section VI E.

We found that the choice of hash function affects the efficiency of the load balancer. The hash function used by OpenSER did not do a very good job of distributing call IDs across hash buckets. Given a sample test with 300,000 calls, OpenSER's hash function distributed the calls to about 88,000 distinct buckets. This resulted in a high percentage of buckets containing several call ID records; searching these buckets adds overhead. We experimented with several different hash functions and found FNV hash [25] to be the best one. For that same test of 300,000 calls, FNV Hash mapped these calls to about 228,000 distinct buckets. The average length of searches was thus reduced by a factor of almost three.

When an INVITE request arrives corresponding to a new call, the call is assigned to a server using one of the algorithms described earlier. Subsequent requests corresponding to the call are always sent to the same machine where the original INVITE was assigned to. For algorithms that use response time, the response time of the individual INVITE and BYE requests are recorded when they are completed. An array of node counter values are kept that track the number of INVITE and BYE requests.

If a server fails, the load balancer stops sending requests to the server. If the failed server is later revived, the load balancer can be notified to start sending requests to the server again. A primary load balancer could be configured with a secondary load balancer which would take over in the event that the primary fails. In order to preserve state information in the event of a failure, the primary load balancer would periodically checkpoint its state, either to the secondary load balancer over the network or to a shared disk. We have not implemented this failover scheme for this paper, and a future area of research is to implement this failover scheme in a manner which both optimizes performance and minimizes lost information in the event that the primary load balancer fails.

V. Experimental Environment

We describe here the hardware and software that we use, our experimental methodology, and the metrics we measure.

SIP Software. For client-side workload generation, we use the the open source SIPp [13] tool, which is the de facto standard for generating SIP load. SIPp is a configurable packet generator, extensible via a simple XML configuration language. It uses an efficient event-driven architecture but is not fully RFC compliant (e.g., it does not do full packet parsing). It can thus emulate either a client (UAC) or server

Feature	Machine Type A	Machine Type B
Quantity	11	3
CPU	3.06 GHz	2.8 GHz
RAM	4 GB	2 GB
Kernel	2.6.9-55.0.6	2.6.9-11
Distro	RedHat AS 4.5	RedHat AS 4.5
Roles	Back-End Server, Load Balancer	Workload Generation

TABLE I
HARDWARE TESTBED CHARACTERISTICS

(UAS), but at many times the capacity of a standard SIP end-host. We use the Subversion revision 311 version of SIPp. For the back-end server, we use a commercially available SIP server.

Hardware and System Software. We conduct experiments using two different types of machines, both of which are IBM x-Series rack-mounted servers. Table I summarizes the hardware and software configuration for our testbed. Eight of the servers have two processors; however, for our experiments, we use only one processor. All machines are interconnected using a gigabit Ethernet switch.

To obtain CPU utilization and network I/O rates, we use nmon [15], a free performance monitoring tool from IBM for AIX and Linux environments. For application and kernel profiling, we use the open-source OProfile [26] tool. OProfile is configured to report the default GLOBAL_POWER_EVENT, which reports time in which the processor is not stopped (i.e. non-idle profile events).

Workload. The workload we use is SIPp's simple SIP UAC call model consisting of an `INVITE`, which the server responds to with `100 TRYING`, `180 RINGING`, and `200 OK` responses. The client then sends an ACK request which creates the session. After a variable pause to model call hold times, the client closes the session using a `BYE` which the server responds to with a `200 OK` response. This is the same call flow as depicted in Figure 2. Calls may or may not have *pause times* associated with them, intended to capture the variable call duration of SIP sessions. In our experiments, pause times are normally distributed with a mean of one minute and a variance of 30 seconds. While simple, this is a common configuration used in SIP performance testing. Currently no standard SIP workload model exists, although SPEC is attempting to define one [36].

Methodology. Each run lasts for 3 minutes after a warm-up period of 10 minutes. There is also a ramp-up phase until the experimental rate is reached. The request rate starts at 1 cps and increases by x cps every second, where x is the number of back-end nodes. Thus, if there are 8 servers, after 5 seconds, the request rate will be 41 cps. If load is evenly distributed, each node will see an increase in the rate of received calls of one additional cps until the experimental rate is reached. After the experimental rate is reached, it is sustained. SIPp is used in open-loop mode; calls are generated at the configured rate regardless of whether the other end responds to them.

Metrics. We measure both throughput and response time. We define throughput as the number of completed requests per second. The peak throughput is defined as the maximum throughput which can be sustained while successfully handling more than 99.99% of all requests. Response time is defined as the length of time between when a request (`INVITE` or `BYE`) is sent and the successful `200 OK` is received.

Component Performance. We have measured the throughput of a single SIPp node in our system to be 2925 calls per second (cps) without pause times and 2098 cps with pause times. The peak throughput for the back-end SIP server is about 300 cps in our system; this figure varies slightly depending on the workload. Surprisingly, the peak throughput is not affected much by pause times. While we have observed that some servers can be adversely affected by pause times, we believe other overheads dominate and obscure this effect in the server we use.

VI. RESULTS

In this section, we present in detail the experimental results of the load balancing algorithms defined in Section III.

A. Response Time

We observe significant differences in the response times of the different load balancing algorithms. Performance is limited by the CPU processing power of the servers and not by memory. Figure 6 shows the average response time for each algorithm versus offered load measured for the `INVITE` transaction. Note especially that the Y axis is in logarithmic scale. In this experiment, the load balancer distributes requests across 8 back-end SIP server nodes. Two versions of Transaction-Least-Work-Left are used. For the curve labeled *TLWL-1.75*, `INVITE` transactions are 1.75 times the weight of `BYE` transactions. In the curve labeled *TLWL-2*, the weight is 2:1. The curve labeled *Hash* uses the standard OpenSER hash function, whereas the curve labeled *FNVHash* uses FNVHash. Round-robin is denoted RR on the graph.

The algorithms cluster into three groups: TLWL-1.75, TLWL-2, and TJSQ, which offer the best performance; CJSQ, Hash, FNVHash, and Round Robin in the middle; and RWMA which results in the worst performance. The differences in response times are significant even when the system is not heavily loaded. For example, at 200 cps, which is less than 10% of peak throughput, the average response time is about 2 ms for the algorithms in the first group, about 15 ms for algorithms in the middle group, and about 65 ms for RWMA. These trends continue as the load increases, with TLWL-1.75, TLWL-2, and TJSQ resulting in response times 5-10 times smaller than those for algorithms in the middle group. As the system approaches peak throughput, the performance advantage of the first group of algorithms increases to two orders of magnitude.

Similar trends are seen in Figure 7, which shows average response time for each algorithm vs. offered load for `BYE` transactions, again using 8 back-end SIP server nodes. `BYE` transactions consume fewer resources than `INVITE` transactions resulting in lower average response times. TLWL-

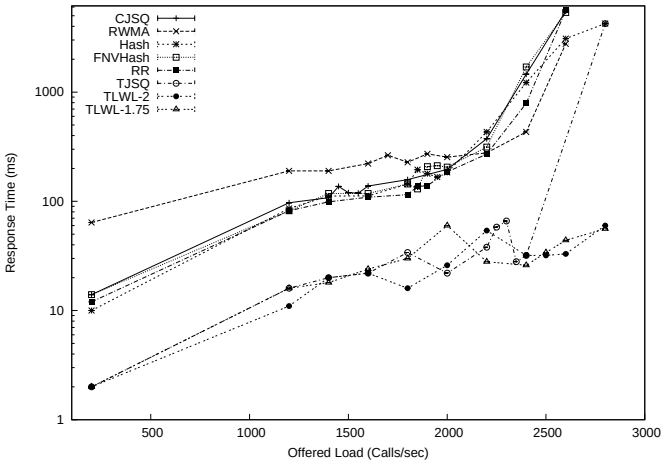

Fig. 6. Average Response Time for INVITE

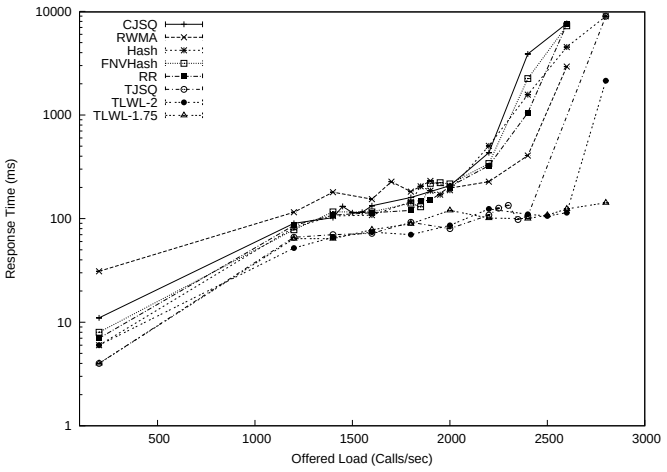

Fig. 7. Average Response Time for BYE

1.75, TLWL-2, and TJSQ provide the lowest average response times. However, the differences in response times for the various algorithms are smaller than is the case with `INVITE` transactions. This is largely because of SARA. The load balancer has freedom to pick the least loaded server for the first `INVITE` transaction of a call. However, a `BYE` transaction must be sent to the server which is already handling the call.

The sharp increases that are seen in response times for the final data points in some of the curves in Figures 6 and 7 are due to the system approaching overload. The fact that the curves do not always monotonically increase with increasing load is due to experimental error.

The significant improvements in response time that TLWL and TJSQ provide present a compelling reason for systems such as these to use our algorithms. Section VI-C provides a detailed analysis of the reasons for the large differences in response times that we observe.

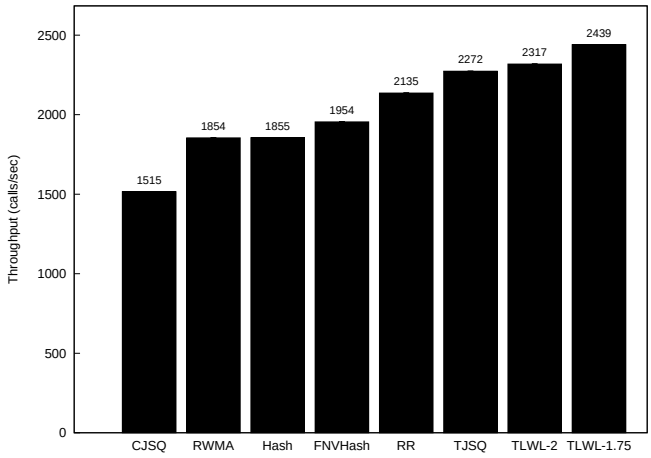

Fig. 8. Peak Throughput of Various Algorithms with 8 SIP Servers

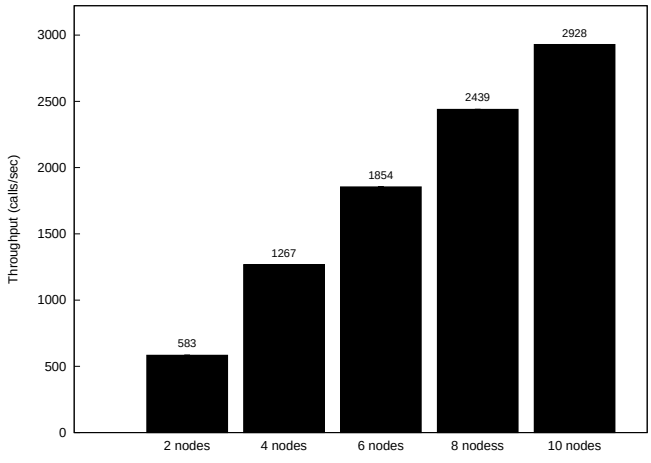

Fig. 9. Peak throughput vs. # of nodes (TLWL-1.75)

B. Throughput

We now examine how our load balancing algorithms perform in terms of how well throughput scales with increasing numbers of back-end servers. In the ideal case, we would hope to see 8 nodes provide 8 times the single-node performance. Recall that the peak throughput is the maximum throughput which can be sustained while successfully handling more than 99.99% of all requests and is approximately 300 cps for a back-end SIP server node. Therefore, linear scalability suggests a maximum possible throughput of about 2400 cps for 8 nodes. Figure 8 shows the peak throughputs for the various algorithms using 8 back-end nodes. Several interesting results are illustrated in this graph.

TLWL-1.75 achieves linear scalability and results in the highest peak throughput of 2439 cps. TLWL-2 comes close to TLWL-1.75, but TLWL-1.75 does better due to its better estimate of the cost ratio between `INVITE` and `BYE` transactions. The same three algorithms resulted in the best response times and peak throughput. However, the differences

in throughput between these algorithms and the other ones are not as high as the differences in response time. For a system in which the ratio of overheads between different transaction times is higher than 1.75, the advantage obtained by TLWL over the other algorithms would be higher.

The standard algorithm used in OpenSER, Hash, achieves 1954 cps. Despite being a static approach with no dynamic allocation at all, one could consider hashing doing relatively well at about 80% of TLWL-1.75. Round-robin does somewhat better at 2135 cps, or 88% percent of TLWL-1.75, illustrating that even very simple approaches to balancing load across a cluster are better than none at all.

We did not obtain good performance from Response-time Weighted Moving Average (RWMA), which resulted in the second lowest peak throughput and the highest response times. Response times may not be the most reliable measure of load on the servers. If the load balancer weights the most recent response time(s) too heavily, this might not provide enough information to determine the least loaded server. On the other hand, if the load balancer gives significant weight to response times in the past, this makes the algorithm too slow to respond to changing load conditions. A server having the lowest weighted average response time might have several new calls assigned to it resulting in too much load on the server before the load balancer determines that it is no longer the least loaded server. In contrast, when a call is assigned to a server using TLWL-1.75 or TJSQ, the load balancer takes this information immediately into account when making future load balancing decisions. Therefore, TLWL-1.75 and TJSQ would not encounter this problem. While we do not claim that *any* RWMA approach does not work well, we were unable to find one that performed as well as our algorithms.

Calls-Join-Shortest-Queue (CJSQ) is significantly worse than the others, since it does not distinguish call hold times in the way that the transaction-based algorithms do. Experiments we ran that did not include pause times (not shown due to space limitations) showed CJSQ providing very good performance, comparable to TJSQ. This is perhaps not surprising since, when there are no pause times, the algorithms are effectively equivalent. However, the presence of pause times can lead CJSQ to misjudgments about allocation that end up being worse than a static allocation such as Hash. TJSQ does better than most of the other algorithms. This shows that knowledge of SIP transactions and paying attention to the call hold time can make a significant difference, particularly in contrast to CJSQ.

C. Occupancy and Response Time

Given the substantial improvements in response time shown in Section VI-A, we believe it is worth explaining in depth how certain load balancing algorithms can reduce response time versus others. We show this in two steps: First, we demonstrate how the different algorithms behave in terms of *occupancy*, namely, the number of requests allocated to the system. The occupancy for a transaction T assigned to a server S is the number of transactions already being handled by S when T is assigned to it. Then, we show how occupancy has a direct influence on response time. In the experiments described in this section, requests were distributed among four servers at a rate of 600 cps. Experiments were run for one minute; thus, each experiment results in 36,000 calls.

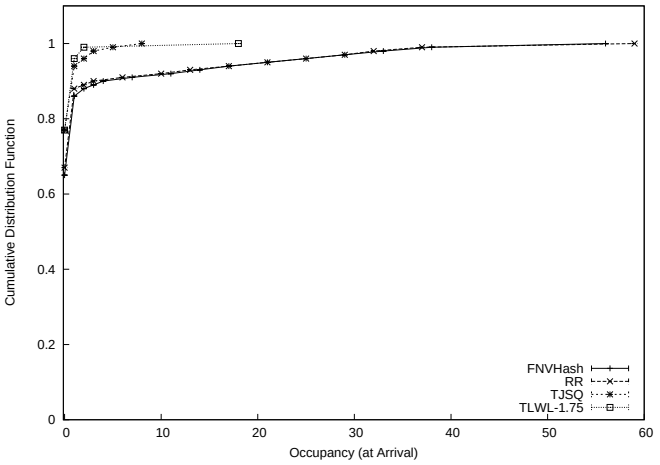

Fig. 10. CDF: Occupancy at one node xd017

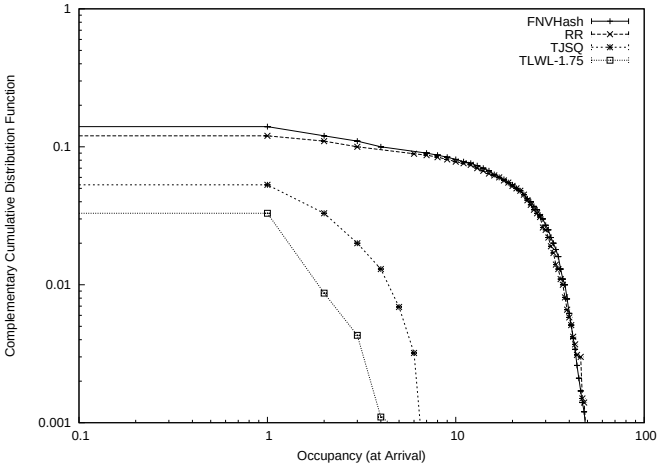

Fig. 11. CCDF: Occupancy at one node xd017

Figure 10 shows the cumulative distribution frequency (CDF) of the occupancy as seen by a request at arrival time for one back-end node for four algorithms: FNVHash, Round-Robin, TJSQ, and TLWL-1.75. This shows how many requests are effectively "ahead in line" of the arriving request. A point $(5, y)$ would indicate that y is the proportion of requests with occupancy no more than 5. Intuitively, it is clear that the more requests there are in service when a new request arrives, the longer that new request will have to wait for service. One can observe that the two Transaction-based algorithms see lower occupancies for the full range of the distribution, where 90 percent see fewer than two requests, and in the worst case never see more than 20 requests. Round-Robin and Hash, however, have a much more significant proportion of

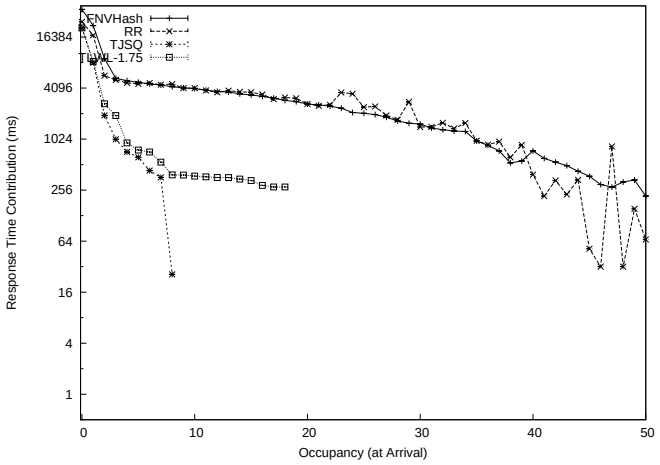

Fig. 12. Response Time Contribution

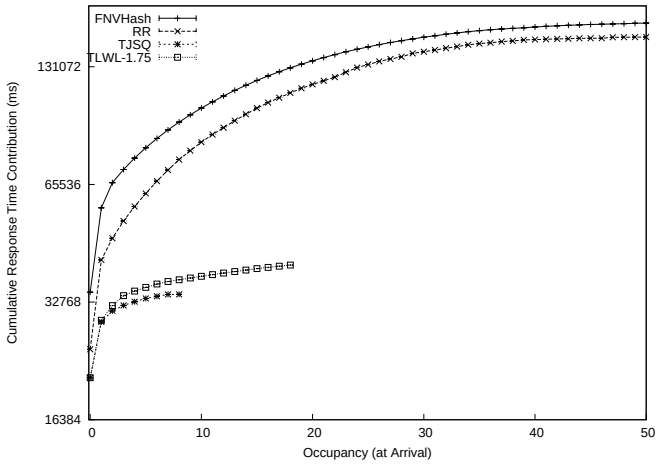

Fig. 13. Response Time Cumulative Contribution

their distributions with higher occupancy values; 10 percent of requests see 5 or more requests upon arrival. This is particularly visible when looking at the complement of the CDF, as shown in Figure 11: Round-robin and Hash have much more significant tails than do TJSQ or TLWL-1.75. While the medians of the occupancy values for the different algorithms are the same (note that over 60% of the transactions for all of the algorithms in Figure 10 have an occupancy of 0), the tails are not, which influences the average response time.

Recall that average response time is the sum of all the response times seen by individual requests divided by the number of requests. Given a test run over a period at a fixed load rate, all the algorithms have the same total number of requests over the run. Thus by looking at contribution to *total* response time we can see how occupancy affects *average* response time.

Figure 12 shows the contribution of each request to the total response time for the four algorithms in Figure 10, where requests are grouped by the occupancy they observe when they arrive in the system. In this graph, a point $(5, y)$ would indicate that y is the sum of response times for all requests arriving at a system with 5 requests assigned to it. One can see that Round-Robin and Hash have many more requests in the tail beyond an observed occupancy of 20. However, this graph does not give us a sense of how much these observations contribute to the sum of all the response times (and thus the average response time). This sum is shown in Figure 13, which is the accumulation of the contributions based on occupancy.

In this graph, a point $(5, y)$ would indicate that y is the sum of response times for all requests with an occupancy up to 5. Each curve accumulates the components of response time (the corresponding points in Figure 12) until the total sum of response times is given at the top right of the curve. For example, in the Hash algorithm, approximately 12,000 requests see an occupancy of zero, and contribute about 25,000 milliseconds towards the total response time. 4,000 requests see an occupancy of one and contribute about 17,000 milliseconds of response time to the total. Since the graph is cumulative, the Y value for $x = 1$ is the sum of the two occupancy values, about 42,000 milliseconds. By accumulating all the sums, one sees how large numbers of instances where requests arrive at a system with high occupancy can add to the average response time.

Figure 13 shows that TLWL-1.75 has a higher sum of response times (40,761 milliseconds) than does TJSQ (34304 ms), a difference of about 18 percent. This is because TJSQ is exclusively focused on minimizing occupancy, whereas TLWL-1.75 minimizes work. Thus TJSQ has a smaller response time at this low load (600 cps), but at higher loads, TLWL-1.75's better load balancing allows it to provide higher throughput.

To summarize, by balancing load more evenly across a cluster, the transaction-based algorithms improve response time by minimizing the number of requests a new arrival must wait behind before receiving service. This clearly depends on the scheduling algorithm used by the server in the back end; however, Linux systems like ours effectively have a scheduling policy which is a hybrid between first-in-first-out (FIFO) and processor sharing (PS) [11]. Thus the response time seen by an arriving request has a strong correlation with the number of requests in the system.

D. Heterogeneous Back Ends

In many deployments, it is not realistic to expect that all nodes of a cluster have the same server capacity. Some servers may be more powerful than others, or may be running background tasks which limit the CPU resources which can be devoted to SIP. In this section, we look at how our load balancing algorithms perform when the back end servers have different capabilities. In these experiments, the load balancer is routing requests to two different nodes. One of the nodes is running another task which is consuming about 50% of its CPU capacity. The other node is purely dedicated to handling SIP requests. Recall that the maximum capacity of a single

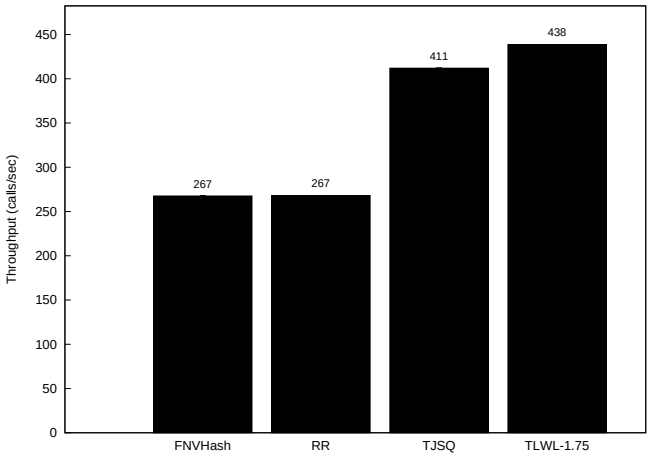

Fig. 14. Peak Throughput (Heterogeneous Backends)

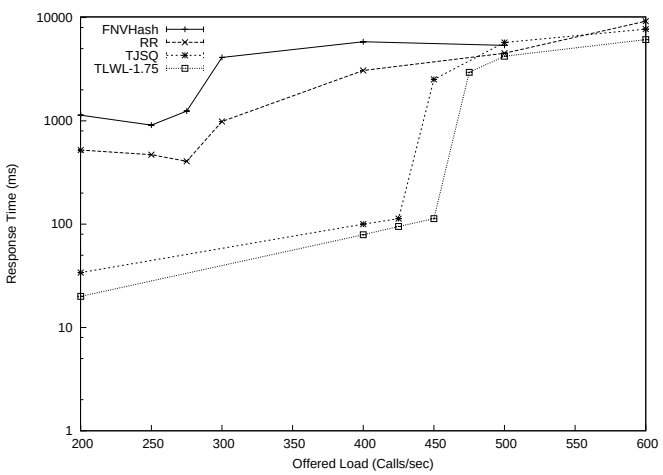

Fig. 15. Avg. Response Time (Heterogeneous Backends)

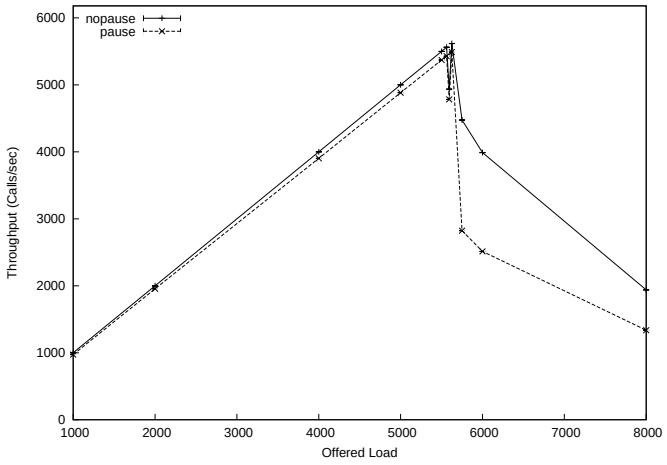

Fig. 16. Load Balancer Throughput vs. Offered Load

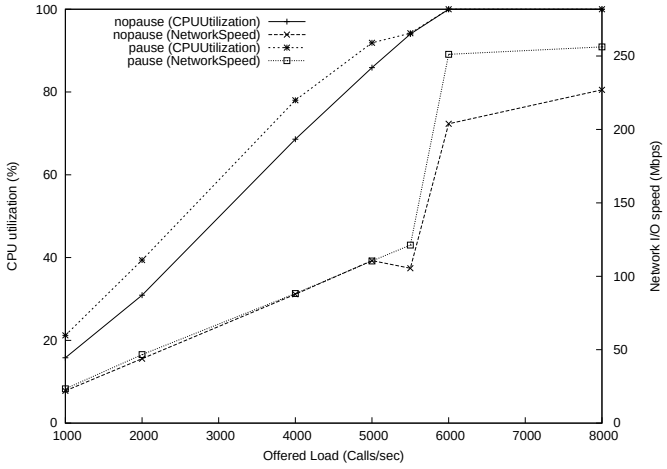

Fig. 17. CPU Utilization and Network Bandwidth vs. Load

server node is 300 cps. Ideally, the load balancing algorithm in this heterogeneous system should result in a throughput of about one and a half times this rate, or 450 cps.

Figure 14 shows the peak throughputs of four of the load balancing algorithms. TLWL-1.75 achieves the highest throughput of 438 cps, which is very close to optimal. TJSQ is next at 411 CPS. Hash and RR provide significantly lower peak throughputs.

Response times are shown in Figure 15. TLWL-1.75 offers the lowest response times followed by TJSQ. The response times for RR and Hash are considerably worse, with Hash resulting in the longest response times. These results clearly demonstrate that TLWL-1.75 and TJSQ are much better at adapting to heterogeneous environments than RR and Hash.

Unlike those two, the dynamic algorithms track the number of calls or transactions assigned to the back ends and attempt to keep them balanced. Since the faster machine satisfies requests twice as quickly, twice as many calls are allocated to it. Note that that is done *automatically* without the dispatcher having any notion of the disparity in processing power of the back-end machines.

E. Load Balancer Capacity

In this section we evaluate the performance of the load balancer itself, to see how much load it can support before it becomes a bottleneck for the cluster. We use 5 SIPp nodes as clients and 5 SIPp nodes as servers which allow us to generate around 10,000 cps without SIPp becoming a bottleneck. Recall from Section V that SIPp can be used in this fashion to emulate both a client and a server, with a load balancer in between.

Figure 16 shows observed throughput vs. offered load for the dispatcher using TLWL-1.75. The load balancer can support up to about 5500 cps before succumbing to overload when no pause times are used, and about 5400 cps when pauses are introduced. Given that the peak throughput of the SIP server is about 300 cps, the prototype should be able to support about 18 SIP servers.

Figure 17 shows CPU utilization (on the left Y axis) and network bandwidth consumed (on the right Y axis) versus

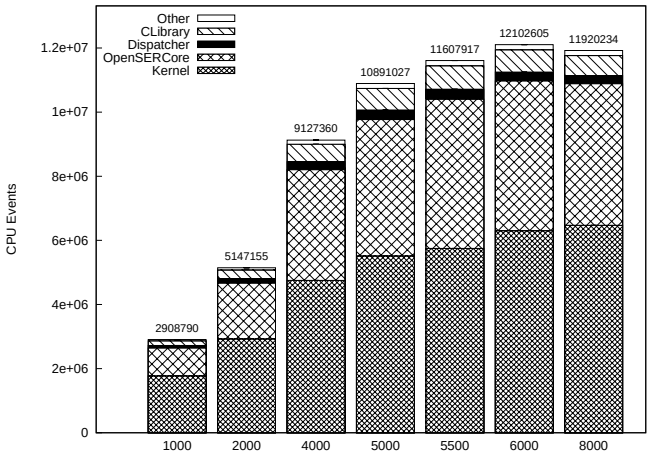

Fig. 18. Load Balancer CPU Profile

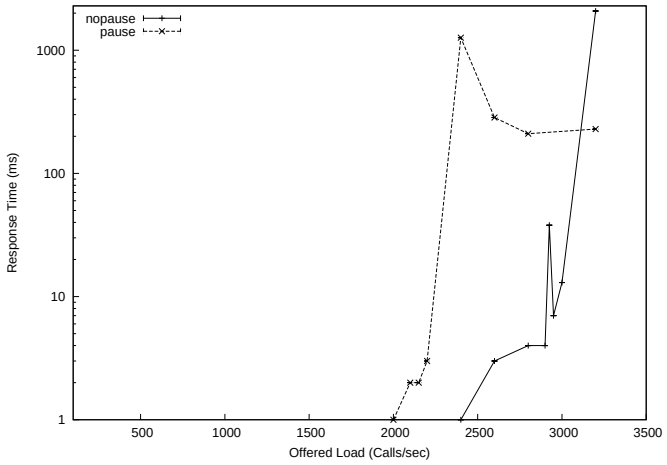

Fig. 19. SIPp Response Time

offered load for the load balancer. The graph confirms that the CPU is fully utilized at around 5500 cps. We see that the bandwidth consumed never exceeds 300 megabits per second (Mbps) in our gigabit testbed; thus, network bandwidth is not a bottleneck.

Figure 18 shows the CPU profiling results for the load balancer obtained via oprofile for various load levels. As can be seen, roughly half the time is spent in the Linux kernel, and half the time in the core OpenSER functions. The load balancing module, marked "dispatcher" in the graph, is a very small component consuming fewer than 10 percent of cycles. This suggests that if even higher performance is required from the load balancer, several opportunities for improvement are available. For example, further OpenSER optimizations could be pursued, or the load balancer could be moved into the kernel in a fashion similar to the IP Virtual Services (IPVS) [28] subsystem. Since we are currently unable to fully saturate the load balancer on our testbed, we leave this as future work. In addition, given that a user averages one call an hour (the "busy-hour call attempt"), 5500 calls per second can support over 19 million users.

F. Baseline SIPp and SIP Server Performance

This section presents the performance of our individual components. These results also demonstrate that the systems we are using are not, by themselves, bottlenecks that interfere with our evaluation.

These experiments show the load that an individual SIPp client instance is capable of generating in isolation. Here, SIPp is used in a back-to-back fashion, as both the client and the server, with no load balancing intermediary in between them. We measured the peak throughput that we obtained for SIPp on our testbed for two configurations: with and without pause times. Pause time is intended to capture the call duration that a SIP session can last. Here, pause time is normally distributed with a mean of 1 minute and a variance of 30 seconds.

Figure 19 shows the average response time vs. load of a call generated by SIPp. Note the log scale of the Y-axis in the graph. SIPp uses millisecond granularity for timing; thus calls completing in under 1 millisecond effectively appear as zero. We observe that response times appear and increase significantly at 2000 cps when pauses are used and 2400 cps when pauses are excluded. At these load values, SIPp itself starts becoming a bottleneck and a potential factor in performance measurements. To ensure this does not happen, we limit the load requested from a single SIPp box to 2000 cps with pauses and 2400 without pauses. Thus, our three SIPp client workload generators can produce an aggregate request rate of 6000 or 7200 cps with and without pauses, respectively.

We also measured peak throughput observed for the commercially available SIP server running on one of our back-end nodesr. Here, one SIPp client generates load to the SIP server, again with no load balancer between them. Again two configurations are shown: with and without pause times. Our measurements (not included due to space limitations) showed that the SIP server can support about 286 cps with pause times, and 290 cps without pauses. Measurements of CPU utilization vs. offered load confirm that the SIP server supports about 290 cps at 100 percent CPU utilization, and that memory and I/O are not bottlenecks.

VII. Related Work

A load balancer for SIP is presented in [35]. In this paper, requests are routed to servers based on the receiver of the call. A hash function is used to assign receivers of calls to servers. A key problem with this approach is that it is difficult to come up with an assignment of receivers to servers which results in even load balancing. This approach also does not adapt itself well to changing distributions of calls to receivers. Our study considers a wider variety of load balancing algorithms and shows scalability to a larger number of nodes. The paper [35] also addresses high availability and how to handle failures.

A number of products are advertising support for SIP load balancing including Nortel Networks' Layer 2-7 Gigabit Ethernet Switch Module for IBM BladeCenter [18], Foundry

Networks' ServerIron [23], and F5's BIG-IP [9]. Publicly available information on these products does not reveal the specific load balancing algorithms that they employ.

A considerable amount of work has been done in the area of load balancing for HTTP requests [5]. One of the earliest papers in this area describes how NCSA's Web site was scaled using round-robin DNS [20]. Advantages of using an explicit load balancer over round-robin DNS were demonstrated in [8]. Their load balancer is content unaware because it does not examine the contents of a request. Content-aware load balancing, in which the load balancer examines the request itself to make routing decisions, is described in [3], [4], [27]. Routing multiple requests from the same client to the same server for improving the performance of SSL in clusters is described in [14]. Load balancing at highly accessed real Web sites is described in [6], [19]. Client-side techniques for load balancing and assigning requests to servers are presented in [10], [21]. A method for load balancing in clustered Web servers in which request size is taken into account in assigning requests to servers is presented in [7].

Least-work-left (LWL) and join-shortest-queue (JSQ) have been applied to assigning tasks to servers in other domains [16], [32]. While conceptually TLWL, TJSQ, and CJSQ use similar principles for assigning sessions to servers, there are considerable differences in our work. Previous work in this area has not considered SARA, where only the first request in a session can be assigned to a server. Subsequent requests from the session must be assigned to the same server handling the first request; load balancing using LWL and JSQ as defined in these papers is thus not possible. In addition, these papers do not reveal how a load balancer can reliably estimate the least work left for a SIP server which is an essential feature of our load balancer.

VIII. Summary and Conclusions

This paper introduces three novel approaches to load balancing in SIP server clusters. We present the design, implementation, and evaluation of a load balancer for cluster-based SIP servers. Our load balancer performs session-aware request assignment (SARA) to ensure that SIP transactions are routed to the proper back-end node that contains the appropriate session state. We presented three novel algorithms: Call Join Shortest Queue (CJSQ), Transaction Join Shortest Queue (TJSQ), and Transaction Least-Work-Left (TLWL).

The TLWL algorithms result in the best performance, both in terms of response time and throughput, followed by TJSQ. TJSQ has the advantage that no knowledge is needed of relative overheads of different transaction types. The most significant performance differences were in response time. Under light to moderate loads, TLWL-1.75, TLWL-2, and TJSQ achieved response times for `INVITE` transactions that were at least 5 times smaller than the other algorithms we tested. Under heavy loads, TLWL-1.75, TLWL-2, and TJSQ have response times two orders of magnitude smaller than the other approaches. For SIP applications that require good quality of service, these dramatically lower response times are significant. We showed that these algorithms provide significantly better response time by distributing requests across the cluster more evenly, thus minimizing occupancy and the corresponding amount of time a particular request waits behind others for service. TLWL-1.75 provides 25% better throughput than a standard hash-based algorithm and 14% better throughput than a dynamic round-robin algorithm. TJSQ provides nearly the same level of performance. CJSQ performs poorly since it does not distinguish transactions from calls and does not consider variable call hold times.

Our results show that by combining knowledge of the SIP protocol, recognizing variability in call lengths, distinguishing transactions from calls, and accounting for the difference in processing costs for different SIP transaction types, load balancing for SIP servers can be significantly improved.

The dramatic reduction in response times achieved by both TLWL and TJSQ, compared to other approaches, suggest that they should be applied to other domains besides SIP, particularly if response time is crucial. Our results are influenced by the fact that SIP requires SARA. However, even where SARA is not needed, variants of TLWL and TJSQ could be deployed and may offer significant benefits over commonly deployed load balancing algorithms based on round robin, hashing, or response times. A key aspect of TJSQ and TLWL is that they track the number of uncompleted requests assigned to each server, in order to make better assignments. This can be applied to load balancing systems in general. In addition, if the load balancer can reliably estimate the relative overhead for requests that it receives, this can further improve performance.

Several opportunities exist for potential future work. These include: evaluating our algorithms on larger clusters to further test their scalability; adding a fail-over mechanism to ensure that the load balancer is not a single point of failure; and looking at other SIP workloads such as instant messaging or presence.

Acknowledgments

Thanks to Mike Frissora and Jim Norris for their help with the hardware cluster.

References

[1] Darrell C. Anderson, Jeffrey S. Chase, and Amin Vahdat. Interposed request routing for scalable network storage. In *USENIX Operating Systems Design and Implementation (OSDI)*, San Diego, California, USA, October 2000.

[2] Mohit Aron and Peter Druschel. TCP implementation enhancements for improving Webserver performance. Technical Report TR99-335, Rice University Computer Science Dept., July 1999.

[3] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Efficient support for P-HTTP in cluster-based Web servers. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[4] Mohit Aron, Darren Sanders, Peter Druschel, and Willy Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. In *Proceedings of the USENIX 2000 Annual Technical Conference*, San Diego, CA, June 2000.

[5] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S. Yu. The state of the art in locally distributed Web-server systems. *ACM Computing Surveys*, 34(2):263–311, June 2002.

[6] Jim Challenger, Paul Dantzig, and Arun Iyengar. A scalable and highly available system for serving dynamic data at frequently accessed Web sites. In *Proceedings of ACM/IEEE SC98*, November 1998.

[7] Gianfranco Ciardo, Alma Riska, and Evgenia Smirni. EQUILOAD: A load balancing policy for clustered Web servers. *Performance Evaluation*, 46(2-3):101–124, 2001.

[8] Dan Dias, William Kish, Rajat Mukherjee, and Renu Tewari. A scalable and highly available Web server. In *Proceedings of the 1996 IEEE Computer Conference (COMPCON)*, February 1996.

[9] F5. F5 introduces intelligent traffic management solution to power service providers' rollout of multimedia services. http://www.f5.com/news-press-events/press/2007/20070924.html.

[10] Zongming Fei, Samrat Bhattacharjee, Ellen Zegura, and Mustapha Ammar. A novel server selection technique for improving the response time of a replicated service. In *Proceedings of IEEE INFOCOM*, 1998.

[11] Hanhua Feng, Vishal Misra, and Dan Rubenstein. PBS: A unified priority-based scheduler. In *Proceedings of ACM Sigmetrics*, San Diego, CA, June 2007.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2068, Internet Engineering Task Force, January 1997.

[13] Richard Gayraud and Olivier Jacques. SIPp. http://sipp.sourceforge.net.

[14] G. Goldszmidt, G. Hunt, R. King, and R. Mukherjee. Network dispatcher: A connection router for scalable Internet services. In *Proceedings of the 7th International World Wide Web Conference*, Brisbane, Australia, April 1998.

[15] Nigel Griffiths. nmon: A free tool to analyze AIX and Linux performance. http://www.ibm.com/developerworks/aix/library/au-analyze_aix/index.html.

[16] Mor Harchol-Balter, Mark Crovella, and Cristina D. Murta. On choosing a task assignment policy for a distributed server system. *Journal of Parallel and Distributed Computing*, 59(2):204–228, 1999.

[17] Volker Hilt and Indra Widjaja. Controlling overload in networks of SIP servers. In *International Conference on Network Protocols (ICNP)*, Orlando, Florida, USA, October 2008.

[18] IBM. Application switching with Nortel Networks layer 2-7 gigabit ethernet switch module for IBM BladeCenter. http://www.redbooks.ibm.com/abstracts/redp3589.html?Open.

[19] Arun Iyengar, Jim Challenger, Daniel Dias, and Paul Dantzig. High-performance Web site design techniques. *IEEE Internet Computing*, 4(2), March/April 2000.

[20] Thomas T. Kwan, Robert E. McGrath, and Daniel A. Reed. NCSA's World Wide Web server: Design and performance. *IEEE Computer*, 28(11):68–74, November 1995.

[21] Dan Mosedale, William Foss, and Rob McCool. Lessons learned administering Netscape's Internet site. *IEEE Internet Computing*, 1(2):28–35, March/April 1997.

[22] Erich Nahum, John Tracey, and Charles P. Wright. Evaluating SIP proxy server performance. In *17th International Workshop on Networking and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, Urbana-Champaign, Illinois, USA, June 2007.

[23] Foundry Networks. ServerIron switches support SIP load balancing VoIP/SIP traffic management solutions. http://www.foundrynet.com/solutions/sol-app-switch/sol-voip-sip/.

[24] Nortel Networks. Layer 2-7 GbE switch module for IBM BladeCenter. http://www-132.ibm.com/webapp/wcs/stores/servlet/ProductDisplay?productId=4611686018425170446&storeId=1&langId=-1&catalogId=-840.

[25] Landon Curt Noll. Fowler/Noll/Vo (FNV) hash. http://isthe.com/chongo/tech/comp/fnv/.

[26] OProfile. A system profiler for Linux. http://oprofile.sourceforge.net/.

[27] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich M. Nahum. Locality-aware request distribution in cluster-based network servers. In *Architectural Support for Programming Languages and Operating Systems*, pages 205–216, 1998.

[28] Linux Virtual Server Project. IP virtual server (IPVS). http://www.linuxvirtualserver.org/software/ipvs.html.

[29] Wikipedia Project. Moving average. http://en.wikipedia.org/wiki/Weighted_moving_average/.

[30] J. Rosenberg and Henning Schulzrinne. An offer/answer model with session description protocol (SDP). RFC 3264, Internet Engineering Task Force, June 2002.

[31] Jonathan Rosenberg, Henning Schulzrinne, Gonzalo Camarillo, Alan Johnston, Jon Peterson, Robert Sparks, Mark Handley, and Eve Schooler. SIP: Session initiation protocol. RFC 3261, Internet Engineering Task Force, June 2002.

[32] Bianca Schroeder and Mor Harchol-Balter. Evaluation of task assignment policies for supercomputing servers: The case for load unbalancing and fairness. *Cluster Computing*, 7(2):151–161, 2004.

[33] Henning Schulzrinne, Stephen Casner, Ron Frederick, and Van Jacobson. RTP: a transport protocol for real-time applications. RFC 3550, Internet Engineering Task Force, July 2003.

[34] Charles Shen, Henning Schulzrinne, and Erich M. Nahum. Session initiation protocol (SIP) server overload control: Design and evaluation. In *Principles, Systems and Applications of IP Telecommunications (IPTComm)*, pages 149–173, Heidelberg, Germany, July 2008.

[35] Kundan Singh and Henning Schulzrinne. Failover and load sharing in SIP telephony. In *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'05)*, July 2005.

[36] Systems Performance Evaluation Corporation (SPEC). SPEC SIP subcommittee. http://www.spec.org/specsip/.

[37] www.openser.org. The open SIP express router (OpenSER). http://www.openser.org.

[38] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, and David Zappola. RSVP: a new resource reservation protocol. *IEEE Communications Magazine*, 40(5):116–127, May 2002.

AUTHOR BIOGRAPHIES

Hongbo Jiang is an associate professor on the faculty of Huazhong University of Science and Technology. He received his Ph.D. from Case Western Reserve University in 2008. His research concerns computer networking, especially algorithms and architectures for wireless and high-performance networks.

Arun Iyengar received the Ph.D. degree in computer science from MIT. He does research on Web performance, distributed computing, and high availability at IBM's T.J. Watson Research Center. He is Co-Editor-in-Chief of the ACM Transactions on the Web, Chair of IFIP WG 6.4 on Internet Applications Engineering, and an IBM Master Inventor.

Erich Nahum is a research staff member in the IBM T.J. Watson Research Center. He received his Ph.D. in Computer Science from the University of Massachusetts, Amherst in 1996. He is interested in all aspects of performance in experimental networked systems.

Wolfgang Segmuller is a Senior Software Engineer at the IBM T.J. Watson Research Center. He has researched systems management, network management and distributed systems for 29 years at IBM.

Dr. Asser N. Tantawi is a research staff member at the IBM T.J. Watson Research Center. He received his Ph.D. degree in computer science from Rutgers University. His interests include performance modeling and analysis, multimedia systems, mobile computing and communications, telecommunication services, and high-speed networking.

Charles P. Wright received his Ph.D. in Computer Science from SUNY Stony Brook. He joined IBM's T.J. Watson Research Center, and has performed research on systems software for network servers and high-performance computers.