Self-Adapting Load Balancing for DNS

Jörg Jung, Simon Kiertscher, Sebastian Menski, Bettina Schnor
Institute for Computer Science, University of Potsdam, Germany
Email: {jozinke, kiertsch, menski, schnor}@uni-potsdam.de

Abstract—The Domain Name System belongs to the core services of the Internet infrastructure. Hence, DNS availability and performance is essential for the operation of the Internet and replication as well as load balancing are used for the root and top level name servers.

This paper proposes an architecture for credit based server load balancing (SLB) for DNS. Compared to traditional load balancing algorithms like round robin or least connection, the benefit of credit based SLB is that the load balancer can adapt more easily to heterogeneous load requests and back end server capacities. The challenge of this approach is the definition of a suited credit metric. While this was done before for TCP based services like HTTP, the problem was not solved for UDP based services like DNS.

In the following an approach is presented to define credits also for UDP based services. This UDP/DNS approach is implemented within the credit based SLB implementation salbnet. The presented measurements confirm the benefit of the self-adapting credit based SLB approach. In our experiments, the mean (first) response time dropped significantly compared to weighted round robin (WRR) (from over 4 ms to about 0.6 ms for dynamic pressure relieve (DPR)).

Keywords—Load Balancing, Cluster Computing, Performance Evaluation

I. INTRODUCTION

Dispatcher based server load balancing (SLB) as shown in Figure 1 is an efficient way to provide scalable, flexible and, fault tolerant services.

SLB is a popular technique to build high-availability services. Current load balancer (LB) implementations like the Linux Virtual Server (LVS) [2] use SLB algorithms like round robin (RR) and least connection (LC) and the corresponding weighted variants weighted round robin (WRR) and weighted least connection (WLC). The latter can be configured to use different weights reflecting the different capacities of machines in the back end. On the other hand, credit based SLB strategies have been proposed in the literature, where the back end servers dynamically report a metric called credit to the LB. The credits reflect their current capacity and are pushed based server probes [3], [4], [5].

The credit calculation tries to apply a mapping from the current metric values to the number of possible additional requests. Ideally one credit represents a single request which can be handled by the back end server without being an overloaded server.

In the context of the credit based SLB approach, the self-adapting load balancing network (salbnet) implementation is presented in [6], which provides the associated components to setup an application independent credit based SLB environment. The salbnet implementation is written for Unix-like systems, and runs on Linux distributions using LVS. Further, the salbnet implementation takes care of collecting the implicit metrics, the credit calculation, as well as the reporting, and the scheduling based on the credits. The implementation consists of the following main components:

- salbd: The salbd daemon handles the credit metric collecting and reporting of the calculated credits, which runs on the LB as well as on the back end servers.
- LVS scheduler: The LVS scheduler is a kernel module which runs on the LB and implements the credit based scheduling algorithms.
- libnethook: The libnethook library is used for hooking into (socket) system calls as utilized on the back end servers.
- libnetmsg: The libnetmsg library is a network abstraction library for sending messages over Ethernet and InfiniBand.

The next Section discusses some related work. Thereafter, the credit based approach for UDP is presented in detail in Section III. Section IV presents the implementation of the concept within salbnet and the corresponding components. This includes the collection and calculation of credits. Section V presents the performance evaluation based on the measurements with the prototype. Next, some further considerations regarding the recent Linux kernel and Domain Name System Security Extensions (DNSSEC) are presented in Section VI followed by a conclusion.

II. RELATED WORK

This paper focuses on SLB for DNS traffic using clusters with back end name servers to answer DNS requests.

A. DNS Traffic

DNS traffic is analyzed in several research papers [7], [8], [9]. DNS is hierarchical with the so called root name servers and top level domains on top of the hierarchy.
There are 13 logical root name servers available named with the letters A till M [10].

For DNS workload, an analysis of the traffic of the F root name server within the period from January 7–24, 2001 is provided in [7]. The analysis revealed DNS request load peaks at about 5,000 requests per second, with the F root name server answering about 93 percent while the other unanswerable 7 percent are for example malformed queries. In addition, two denial of service (DoS) attacks are found within the time period, with one using the F root server as reflector flooding a target with answers.

Several of the existing malware may result in significant amounts of additional DNS requests as analyzed for example for a virus named Antinny in [8].

B. Anycast Load Balancing

In [9], measurements are presented which collect latencies of several high traffic DNS name servers, like for example the K and F root name servers. The data is measured within the period from September 15 to October 8, 2004, and confirms that using anycast for high traffic DNS (root) name servers, in general increases the availability and decreases the query latencies. Nevertheless, the results depend on the interaction with Border Gateway Protocol (BGP) and the chosen routes. With anycast multiple hosts are connected with the same anycast IP address and a packet sent to an anycast address is delivered to one of the interfaces identified by that address, as described in RFC 1546 [11]. The servers can be distributed globally and with anycast usually the “nearest” (closest) server with the shortest (best) route is chosen by BGP.

Similar results are presented in the analysis of the measured traffic for the C, E, F, and K root name servers in [12]. The presented measurements from a two day period in January 2006 validates, that using anycast SLB for the root name servers improves the DNS service, due to clients using the instance closest to them (resulting in significantly localized DNS traffic). The conclusion in [12] is: “Overall, the transition to anycast[] by the DNS root name[ ]servers not only extended the original design limit of 13 DNS roots, but it also provides increased capacity and resilience, thereby improving DNS service worldwide.”

Nevertheless, below the layer of the global mechanisms like anycast, SLB within a local cluster can also be improved using more sophisticated distribution methods. This is where the self-adapting and credit based approach of salbnet comes in.

III. CREDIT BASED LOAD BALANCING FOR UDP

The salbnet application transparency can be extended to the underlying network as well. The underlying InfiniBand as well as the according one-sided communication using remote direct memory access (RDMA) capabilities are not required within salbnet. The credit reporting itself can utilize Ethernet and TCP or UDP as well.

For reporting credits, a metric is required, which is suited to reflect the current load on the back end servers. In case of TCP based services, the fill level of the backlog queue has shown to be a good measure [6]. The details of the extension of the basic backlog approach and the associated UDP based metrics are discussed within this Section. When it comes to UDP the counterpart to the TCP backlog is the UDP receive queue. Both, the TCP backlog and the UDP receive queue are implicit metrics. From the information provided by the latter, the following facts arise:

- A filled UDP receive queue indicates a busy application.
- If the UDP receive queue is full, the application on the server will not respond to requests anymore and is fully engaged.

In contrast to the TCP backlog, which holds connection information of fixed size, the UDP receive queue holds packets of different sizes. Hence, there is no information available how many packets still fit into the UDP receive queue. Instead, the following information is available:

- The maximum size of the receive queue $q_{\text{max}}$ and the current fill level $q_{\text{current}}$ which is the sum of the sizes of all packets which are queued and waiting to be processed by the application.
- The difference between the maximum possible size of the UDP receive queue and the current fill level of the receive queue indicates a hard limit: How much space is left for new packets which can be handled by the application on the server before new incoming packets are dropped (silently).

Thus, the UDP receive queue is providing implicit information about the current load of the back end server application. For example, a back end server application which did not call recvmsg() in a while, to process entries from the UDP receive queue, may be considered to be a saturated or possibly even an overloaded server.

The drawback of the UDP receive queue compared to the TCP backlog is that the packet size is not known for the back end server application. Hence, history values are used to predict the expected packet size. For the prediction, the median $p_{\text{median}}$ of the packet sizes seen in the UDP receive queue before is used. The size of DNS queries may vary over time, but not as much as for example the size of DNS answers. Hence, the prediction should work fine.

Further, the number of dropped packets $p_{\text{drop}}$ (since the last calculation) is used. In case of $p_{\text{drop}} > 0$, the server is overloaded and zero credits are reported.

The credit based SLB algorithms introduced in [13] use two different credit types, hard and soft credits. In case of UDP, the hard credits represent the maximum number of requests which a back end server is currently able to handle, while the soft credits represent the recommended number of requests which the back end server currently wants to receive from the LB. The simulations in [13] show that soft credits avoid an early overloading of a server.

In conclusion, the calculation of the hard credits and the soft credits for UDP applications is described in Equation 1 and Equation 2.

$$c_h = \begin{cases} q_{\text{max}} - q_{\text{current}} & p_{\text{drop}} = 0 \\ 0 & p_{\text{drop}} > 0 \end{cases}$$ (1)

$$c_s = \begin{cases} q_{\text{max}} - q_{\text{current}} & p_{\text{drop}} = 0 \\ 0 & p_{\text{drop}} > 0 \end{cases}$$ (2)
As shown in Equation 1, the hard credits for UDP applications $c_h$ are calculated as the available receive queue capacity divided by the packet size median $p_{\text{median}}$. The former is the difference of the receive queue maximum $q_{\text{max}}$ and the current value $q_{\text{current}}$. So $c_h$ states the expected additional number of requests which the back end server currently can accept.

Instead of the current value, the median $p_{\text{median}}$ of the receive queue is used in the definition of the UDP soft credits $c_s$. Thus, $c_s$ is an estimation of the additional requests which the back end server probably can accept considering the request history.

IV. IMPLEMENTATION

On implementation related question is: When and how often should metrics be captured or calculated? On the one hand, the LB should not work with old, out-dated credit values, on the other hand overhead due to unnecessary updates should be avoided. A good occasion to update statistics is in case when the application makes a system call. How the salbnet libnethook library intercepts the `accept()` system call from the Apache HTTP server is described in [6]. In a similar manner it is possible to intercept system calls of a name server. For example, the Berkeley Internet Name Domain (BIND) name server uses the `recvmsg()` system call to gather a request from the receive queue. Hence, the libnethook library was extended to intercept this specific system call. This is completely transparent for the BIND name server. Actually, this method can be used for every UDP application, which uses the `recvmsg()` system call.

Every call of `recvmsg()` gathers a DNS request from the receive queue and also returns the received size. This received size is collected by the libnethook library and forwarded to the salbd client within the back end server. But the received size does not match the size of the packet in the receive queue exactly as there is additional overhead introduced by the managing socket structures used within the Linux kernel. The correlation between the DNS packet size and the required memory size in the receive queue depends on the Kernel implementation, in particular on the `struct sk_buff`. To avoid kernel modifications, this correlation was experimentally determined for the Linux kernel v2.6.18. In the experiments, DNS packets with increasing sizes are send to the name server and for each of them the occupied memory size was determined. In the results, a minimal DNS packet is 376 Bytes and (starting with a packet size of 64 Bytes) every 128 Bytes an additional memory of 128 Bytes is required to store a packet. Therefore, with the determined pattern, the memory consumption can be calculated as shown in Equation 3.

$$\text{size}_{\text{queue}} = \left\lfloor \frac{\text{size}_{\text{recvmsg}} + 63}{128} \right\rfloor \cdot 128 + 376 \quad (3)$$

The remaining implementation of the UDP credit metric collection within the salbd is realized through the `/proc` file system interface. The current capacity level of the receive queue is parsed from the `/proc/net/udp` file. This file contains several data points for every open UDP socket.

Further, the number of dropped UDP packets is received from the `/proc/net/snmp` file. It contains information for different protocols like IP, TCP, and ICMP as well as UDP. It should be mentioned that the received drop count is the sum of all dropped packets for all UDP sockets. However, in a real world scenario the name server may be the main UDP application on the back end server, which means that dropped UDP packets are critical for the name server anyway.

In particular, the collection of the dropped packets and the calculation of the median packet size of the receive queue are rather complex and implemented only as proof of concept within the salbd client. Further overhead is considered in the implementation of the UDP based credit calculation for the $p_{\text{median}}$ (see Equations 1 and 2).

In conclusion, the current (proof of concept) implementation is dependent on the specific environment, in particular dependent on the Linux kernel release and the utilized software versions.

V. MEASUREMENTS AND EVALUATION

To evaluate the salbnet DNS extension, a dispatcher based SLB scenario is setup similar to the one shown in Figure 1. This setup is two armed, NAT based and uses route path as introduced in [14]. LVS is used as load balancer with salbnet reporting algorithm dynamic pressure relieve (DPR) and the salbnet scheduling, algorithm next credits as known from [13], [6].

For the measurements the servload benchmark is used and a DNS trace from the University of Potsdam as workload.

A. DNS Benchmark servload

Since DNS benchmarks like queryperf from the BIND name server distribution [15] or DNSPerf and ResPerf [16] are not able to replay queries, we have decided to add support for the DNS protocol to the benchmark servload [17]. Originally, servload is designed for web server benchmarking and supports only HTTP requests. Obviously, measurements with further application protocols like DNS require the extension of servload to support them.

In contrast to the previously mentioned benchmarks servload tries to simulate real user sessions. It therefore expects a real server log file which then is replayed with respect to different user sessions and think times between the requests in the log file.

To apply this for the DNS protocol, we concentrated on the two BIND log formats as described in [19]. From every entry the timestamp, client IP address as well as the query name, type, and class are extracted. The timestamp is used to calculate the think times between subsequent requests of a session. Where a session consists of all requests of a single user, which is identified by its IP address. The DNS request, which servload sends to the back end server, is generated from the extracted query name, type and class.

The DNS protocol implementation conforms to RFC 1035 [20], but supports only the UDP and not the TCP on truncated responses. It would be much more complex to support optional resending of a request over.

\footnote{The arbitrary chosen history value for the salbd implementation is 16.}
TCP and also could affect the benchmark results. Further, the servload benchmark currently supports only 13 query types, which however covers the major query types of the DNS protocol.

B. Outcomes and Metrics

One possible outcome is the successful processing of the request including the successful transfer of a complete response. Another possible outcome could be failures in processing of the request for example due to an overloaded server, resulting in aborted requests and in wrong, incomplete, or aborted responses. Finally, failures in the network connection result in aborted or incomplete requests and responses as well.

The metrics used as criteria to compare the performance are the normalized (first) response time $\bar{r}_a$, the normalized (request) errors $\bar{e}_a$, and the normalized duration $\bar{t}_a$ (see Equations 4, 5, and 6). These are all lower-is-better metrics. For every tested algorithm $a$ (out of the set of algorithms $A$) and metric ($r$, $e$, and $t$) the mean of $n$ measurements is calculated. The resulting means are normalized against the maximal mean per metric. Thus every normalized metric is in the interval $[0, 1]$.

$$\bar{r}_a = \frac{1}{n} \sum_{i=1}^{n} r_{a,i}$$  
$$\bar{e}_a = \frac{1}{n} \sum_{i=1}^{n} e_{a,i}$$  
$$\bar{t}_a = \frac{1}{n} \sum_{i=1}^{n} t_{a,i}$$

(4) (5) (6)

They are combined in the SLB Internet service provider (ISP) penalty and the SLB full penalty (see Equation 7 and 8) which were introduced in [6]). Both ISP penalty metrics are also within $[0, 1]$, where a penalty can only be 1, if the algorithm $a$ is the worst of all compared in every of the three metrics.

$$P_{ISP}(a) = \bar{r}_a \times \bar{e}_a$$  
$$P_{Full}(a) = \bar{r}_a \times P_{ISP}(a)$$

(7) (8)

Furthermore, the CPU usage and Load Average are collected through the SNMP. The values are collected once a minute through SNMPv1 from a shell script running on the LB.

C. Workload

Instead of generating and using synthetic workloads, representative workloads are generated and prepared from existing traces.

The DNS protocol measurements are based on captured traces collected through the DNS servers of the Institute of Computer Science at the University of Potsdam.

The input for the DNS measurements is prepared from an anonymized\(^3\) trace of the DNS servers running the haiti.cs.uni-potsdam.de zone of the Institute for Computer Science at the University of Potsdam. The trace is reduced to the requests within the five minutes from September 29, 2011, 05:55–06:00 a.m. CEST. The resulting final sequence consists of 22,594 requests in total.

The total number of requests from the DNS trace results in insufficient workload for the three back end name servers in the testbed. Therefore, the number of requests is increased using the multiply method of the servload benchmark. The numbers of requests for the resulting three increased traces are shown in Table I. All three number of requests entries in the Table correspond to overloaded servers in the back end.

Table I. Number of requests from the DNS protocol trace.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Requests</th>
<th>Sessions</th>
<th>Mean requests/s</th>
<th>Max requests/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22,594</td>
<td>33</td>
<td>75</td>
<td>204</td>
</tr>
<tr>
<td>400</td>
<td>9,037,600</td>
<td>12,200</td>
<td>30,125</td>
<td>81,600</td>
</tr>
<tr>
<td>800</td>
<td>18,075,200</td>
<td>26,400</td>
<td>60,250</td>
<td>163,200</td>
</tr>
<tr>
<td>1,600</td>
<td>36,150,400</td>
<td>52,600</td>
<td>125,501</td>
<td>326,600</td>
</tr>
</tbody>
</table>

In Figure 2 the workload request distribution is given. The requests per second over the time-frame for all three variations from Table I are shown. The Figure underpins that the distribution of the three variations is similar.

The packet payload sizes of the DNS queries from the trace are distributed between 24 and 78 Bytes, as shown on the left in Figure 3. Applying Equation 3 from Section IV, results in only two different kernel buffer sizes (376 and 504 Bytes) in the UDP receive queue, as shown on the right.

D. Measurement Environment

The corresponding measurement environment consists of five machines. Three BIND v9.3.6 name servers (IB4, IB6, and IB8) are used in the back end. Another machine is used as LB running Linux Virtual Server [2] with ipvsadm v1.24 (compiled with IPVS v1.2.0). The LB and the three name server nodes run CentOS Linux v5.7 with kernel v2.6.18-274.12.1.el5 and GCC v4.1.2 (see Table II).

Table II. Hardware specifications of the web server cluster.

<table>
<thead>
<tr>
<th>Hostname</th>
<th>CPU</th>
<th>Memory</th>
<th>GE NIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVS and IB4</td>
<td>AMD Opteron 244</td>
<td>4 GB</td>
<td>BCM95784</td>
</tr>
<tr>
<td>IB6</td>
<td>Intel Pentium 4 2.8 GHz</td>
<td>4 GB</td>
<td>BCM5721</td>
</tr>
<tr>
<td>IB8</td>
<td>Intel Xeon 3040 1.86 GHz Dual Core</td>
<td>4 GB</td>
<td>BCM95784</td>
</tr>
</tbody>
</table>

Further, on the client machine is servload v0.5.1 running. The client is an Intel Xeon E5520 2.27 GHz Quad Core with 12 GByte RAM, running Debian Linux v5.0.10 with kernel v2.6.26-2-amd64 and GCC v4.3.2 installed and connected with a Dell PowerConnect 6248 Gigabit Ethernet switch.

E. Algorithms and Scenarios

The measurements are done on a Gigabit Ethernet network using TCP for the credit reporting. The goal of this scenario is to compare the de facto standard LVS using weighted round robin (WRR) with the proposed self-adapting credit based approach. For WRR, the LB has to be configured with suited weights for the back

\(^3\)The client IP addresses are replaced by an incrementing identifier number.
end servers. We did several measurements and found that the weights triple (788, 623, 1181) gave a good overall performance for WRR [1].

Similar to root or top level domain name servers, the three back end name servers are configured as iterative name servers. An iterative name server replies to requests only with resource records entries known from its own zones. This avoids variations in the (first) response time due to recursive lookups through the Internet.

Each measurement run is repeated 51 times. Further, the servload benchmark on the client machine is configured to run 1,021 concurrent sessions at maximum, which is the default setting for the number of open file descriptors for Debian Linux distribution user, minus three reserved descriptors for standard input, standard output, and standard error. Increasing this value did not gain better performance.

F. Measurement Results

The absolute values for the mean (first) response time $\bar{r}_a$ and mean (request) errors $\bar{e}_a$ for both algorithms $a$ WRR and DPR are given in Table III. For a better comparison, first the SLB ISP Penalty is discussed.

Table III. Absolute Values for the Mean (First) Response Time and Mean (Request) Errors for 51 Measurement Runs with the WRR Triple and the DPR Configuration.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Algorithm</th>
<th>Mean response [ms]</th>
<th>Mean errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>WRR</td>
<td>4.05</td>
<td>1,273.39</td>
</tr>
<tr>
<td></td>
<td>DPR</td>
<td>0.64</td>
<td>743.16</td>
</tr>
<tr>
<td>800</td>
<td>WRR</td>
<td>4.33</td>
<td>2,038.53</td>
</tr>
<tr>
<td></td>
<td>DPR</td>
<td>0.62</td>
<td>759.86</td>
</tr>
<tr>
<td>1,600</td>
<td>WRR</td>
<td>4.24</td>
<td>2,910.16</td>
</tr>
<tr>
<td></td>
<td>DPR</td>
<td>0.60</td>
<td>1,612.55</td>
</tr>
</tbody>
</table>

1) Results: SLB ISP Penalty: The normalized (first) response time $\tilde{r}_a$, the normalized (request) errors $\tilde{e}_a$, and the resulting SLB ISP penalty $P_{ISP}(\alpha)$ are shown in Figure 4.

The DPR configuration clearly outperforms the WRR algorithm and has a much lower number of (request) errors and better (first) response time values.

All (request) errors are timeouts. The 51 measurement passes revealed no protocol errors due to overloaded
servers worth mentioning.

As result from the lower number of (request) errors and the better (first) response time values, the SLB ISP penalty $P_{ISP}(DPR)$ is also much lower and better compared to $P_{ISP}(WRR)$.

The credit based salbnet algorithm DPR outperforms the LVS algorithm. This also demonstrates the application and protocol independence of salbnet. The DPR algorithm works for TCP based applications for example based on protocols like HTTP as well as for UDP based applications for example based on protocols like DNS.

While the salbnet implementation presented in [6] uses RDMA for credit reporting, the measurements with DNS show that credits can be reported through TCP as well, without introducing a major impact on the performance.

2) Load Averages and CPU Usage: While metrics like the (first) response time or (request) errors are important for service level agreements, they do not completely characterize the behavior of the different SLB algorithms.

Figure 5 shows the load averages on the three back end servers for the highest workload (factor 1,600) during the 51 measurements. All three back end servers have a higher load average under DPR. Further, the duration of all 51 measurement runs is much longer compared to WRR.

Figure 6 shows the CPU usage of the three back end servers during the second measurement run under WRR with the weights triple (788, 623, 1181) and the DPR configuration for the factor 1,600 workload. The second measurement run was chosen arbitrarily as an example and the results are similar for the other measurement runs. The graphs show that the CPU usage is differing on the back end servers during the second pass of the measurement. Especially, IB8 receives less workload under DPR. Furthermore, the CPU usage is more constant for the DPR algorithm.

In conclusion from both Figure 5 and Figure 6, using the WRR algorithm results in less load on the back end servers.

3) Results: SLB Full Penalty: Since the SLB ISP penalty $P_{ISP}(a)$ is created with the requirements of an ISP in mind, it does not take the total duration of the measurement runs into account. Therefore, also the SLB full penalty $P_{full}(a)$ is compared.
The Figure 7 shows that the DPR configuration still performs better than the WRR algorithm, but the included total duration adjusted the SLB full penalty $P_{full}(a)$ values in favor of WRR. Therefore, for the SLB full penalty $P_{full}(a)$ on the right the maximum penalty value 1 is not reached anymore, because with WRR having a better total duration than DPR none of the two algorithms is the worst in all three metrics. Nevertheless, the results do not overturn.

VI. FURTHER CONSIDERATIONS

In the following some further considerations towards the usage of recent Linux kernel versions and DNSSEC are presented.

A. Recent Linux Kernel

As mentioned in Section IV the equation (3) was deduced from measurements on a Linux kernel v2.6.18. Since the internal memory consumption of a DNS package in the UDP receive queue depends on the struct sk_buff, it can be assumed that this estimation is not accurate for a newer kernel. To verify this assumption the measurements were repeated on a recent Linux kernel v3.17.3. Table IV shows that the results heavily differ from the previous measurements. For example, the minimal UDP packet size increased from 376 Bytes to 768 Bytes. This can be explained by numerous changes to the struct sk_buff between these kernel versions. In kernel v2.6.18 the struct sk_buff contained 41 members. This number increased to 63 with kernel v3.17.3.

If we apply the package size mappings from Table IV to the package sizes of our test data visualized in Figure 3, which were distributed between 24 and 78 Bytes, all packages take up 768 Bytes in the UDP receive queue. This information can be used to simplify the credit calculation. In general a DNS request will not exceed a package size of 645 Bytes which corresponds to a memory allocation of 1280 Bytes in the UDP receive queue in the Linux kernel v3.17.3. This size can be used as a conservative approximation for the $P_{median}$ value in the equations (1) and (2). Hence, the calculation of equation (3) becomes superfluous in our model. The credits can now calculated solely based on the current and the median available receive queue size as shown in the equations (1) and (2).

Table IV. SIZE OF A DNS REQUEST PACKAGE IN THE UDP RECEIVE QUEUE FOR THE LINUX KERNEL V3.17.3.

<table>
<thead>
<tr>
<th>Packet payload size (Bytes)</th>
<th>Receive queue size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12-133</td>
<td>768</td>
</tr>
<tr>
<td>134-645</td>
<td>1280</td>
</tr>
<tr>
<td>646-1669</td>
<td>2304</td>
</tr>
</tbody>
</table>

But again, this solution would be kernel dependent and may change in newer Linux kernel versions. This leads to the question if there is a way to dynamically determine the size of the next UDP package in the receive queue. The so-called netlink sockets [21] can be used for communication between the kernel and the user-space. The netlink interface supports multiple protocols to query for different information from the kernel. With the Linux kernel v3.3, the netlink protocol NETLINK_SOCKDIAG was extended to support UDP sockets. Using netlink sockets could be a promising alternative compared to the /proc/net/udp polling. The netlink socket can be reused by the salbd client and allows an asynchronous query using a standardized socket API for a single UDP socket. Whereas the /proc filesystem solution involves recurring blocking file read operations and string parsing of a file which contains all UDP sockets and which format may change in the future. However, the response of the netlink query does not contain the size of the next UDP packet.
package in the receive queue nor the number of dropped packages. So, additional implementation effort would be necessary to adapt the netlink API for our use-case.

Another interesting change regarding salbd was introduced in the Linux kernel v2.6.27, where the /proc/net/udp file also contains the dropped package count per UDP socket. Thus salbd does not have to additionally parse the /proc/net/snmp file to gather the dropped package count. Furthermore, the information from the /proc/net/udp file is more precise as it is listed for every socket separately.

### B. DNSSEC

DNS was designed with the focus on scalability. Security aspects were ignored in the early phase of the Internet. Domain Name System Security Extensions (DNSSEC) as defined in RFC 4033 [22] adds security support, while maintaining backward compatibility. Thus, DNSSEC provides origin authentication and integrity protection for DNS data, as well as a means of public key distribution.

The DNSSEC implementation may add significant load to some DNS servers. For example, common DNSSEC responses are larger than the default UDP size of 512 Bytes. This may lead to the use of TCP instead (see RFC 1035 [20, Section 4.2]). Actually, a DNSSEC measurement scenario may be a mix case using both, UDP and TCP.

While TCP is handled fine by the salbnet implementation as shown for HTTP in [6], the TCP overhead results in more resources required on the back end servers. In addition, the load on the back end resolvers may increase due to the DNSSEC validation itself. Thus due to the nature of DNSSEC the overall packet throughput of validating resolvers may be reduced compared to plain DNS resolvers.

Early experiments confirm the increased load for DNSSEC [23]. The results show an (expected) higher response time of up to factor three in a client-server scenario without load balancer. While this additional overhead applies also in SLB scenarios, the relative behavior of the algorithms should not change. Nevertheless, the
performance of the proposed load balancing approach should be re-evaluated for DNSSEC.

VII. CONCLUSION AND FUTURE WORK

While the credit based SLB algorithm seemed to be promising in simulations, an implementation for UDP based services, like for example DNS was still missing. The proposed credit metric is based on the UDP receive queue and maps the current back end server load to credits. The presented credit calculation for UDP is based on an approximation of the expected UDP packet size of the client requests.

The presented approach was implemented as new module called salbnet within the load balancer LVS. In the presented measurements for DNS, the self-adapting approach based on the presented credit metrics performs better than the state of the art balancing method WRR. For example, the mean (first) response time dropped significantly compared to WRR (from over 4 ms to about 0.6 ms for DPR).

A sensitivity analysis to evaluate the influence of the heterogeneity of the back end servers is future work. For example, the positive effect of DPR in homogeneous clusters was not yet proven by experiments, but only by simulations [13]. Further, the influence of the quality of the UDP packet size prediction is not investigated and also future work.

For the evaluation, a DNS benchmark with a replay capability was required. Therefore, the HTTP benchmark servload was extended for DNS. Within the measurements of the salbnet and DNS scenario servload processed more than 36 million requests and up to 45 thousand requests per second in one pass, while being standard conform. This shows the scalability of both, the servload benchmark which generated the requests on single machine, and the salbnet module which implemented the credit based SLB. Further, the results show that there is no need for RDMA, since credit reporting over TCP works fine as well.

Our prototype is still considered a proof of concept implementation and depends heavily on specific software versions, like for example the used Linux kernel. An analysis of the recent kernel v3.17.3 has shown that the presented approach could be simplified. An updated salbnet version on the recent kernel v3.17.3 would only poll the /proc/net/udp file for the current and maximal available receive queue size and the package drop count. To calculate the credits, it would use a conservative approximation for the pmedian value of 1280 Bytes or alternatively more optimistic 768 Bytes. Again, this approach is kernel dependent and may have to be adjusted for future versions. Therefore, there is still a need for a portable suited interface for communication between user and kernel space. The netlink interface seems to be a promising candidate, but currently does not support the necessary information for the proposed credit calculation.

With DNSSEC the load on the back end servers may increase due to larger packet sizes and the additional validation. Hence, the performance of the proposed load balancing approach should be re-evaluated for DNSSEC.

ACKNOWLEDGMENT

The authors are grateful to the anonymous referees for their valuable comments and suggestions to improve the presentation of this paper.

REFERENCES


Jörg Jung received a graduate degree in computer science with focus on the area of intelligent systems from Brandenburg University of Applied Sciences, followed by an MSc degree in computer science from the University of Potsdam. Currently he is a PhD student at the chair of Operating Systems and Distributed Systems of the Potsdam University. His main research interests are server load balancing, providing scalable and fault-tolerant services through efficient and self adapting load balancing algorithms especially in Linux and Unix environments. He is also an OpenBSD developer.

Simon Kiertscher studied Computer Science at Potsdam University. Currently, he is a research associates and PhD student in the Operating Systems and Distributed Systems group at Potsdam University. His research interest is “Green IT” applied to cluster computing.

Sebastian Menski is currently a student working towards his MSc degree at the University of Potsdam. His current research interest includes network protocols and scalability of network applications.

Bettina Schnor In the 1990s, Bettina Schnor established the research field “Distributed Systems” at the Technical University Braunschweig, Germany. From 1996 to 2000 she was researcher and lecturer at the Institute of Telematics at the University of Luebeck, Germany. Since April 2000 she is head of the Department of Operating Systems and Distributed Systems at Potsdam University, Germany. Her research interests are in the field of distributed systems, especially cluster and grid computing, network security, and assisted living technologies.