On the Performance of Reliable Server Pooling Systems

Thomas Dreibholz
University of Duisburg-Essen
Ellernstrasse 29
45326 Essen, Germany
dreibh@exp-math.uni-essen.de

Erwin P. Rathgeb
University of Duisburg-Essen
Ellernstrasse 29
45326 Essen, Germany
rathgeb@exp-math.uni-essen.de

Abstract

Reliable Server Pooling (RSerPool) is a protocol framework for server redundancy and session failover, currently under standardization by the IETF RSerPool WG. While the basic ideas of RSerPool are not new, their combination into one architecture is. Some research into the performance of RSerPool for certain specific applications has been made, but a detailed, application-independent sensitivity analysis of the system parameters is still missing.

The goal of this paper, after an application-independent, generic quantification of RSerPool systems and definition of performance metrics for both service provider and user, is to systematically investigate RSerPool's behavior on changes of workload and system parameters to give basic guidelines on designing efficient RSerPool systems.

Keywords: *RSerPool, server pooling, load distribution, performance analysis, parameter sensitivity*

1. Introduction

The Reliable Server Pooling (RSerPool) architecture currently under standardization by the IETF RSerPool WG is an overlay network framework to provide server replication and session failover capabilities to its applications. These functionalities themselves are not new, but their combination into one application-independent framework is.

While there has already been some research on the performance of RSerPool for applications like SCTP-based mobility [11], distributed computing [9, 12, 14–16, 37] and battlefield networks [34], a generic application-independent performance analysis is still missing. The goal of our work is therefore an application-independent quantitative characterization of RSerPool systems and a generic sensitivity analysis on changes of workload and system parameters. In particular, we want to identify critical parameter spaces to provide guidelines for designing efficient RSerPool systems. In this paper we concentrate our analysis on failure-free scenarios, since servers are usually available in 99.9x% of their lifetime and therefore best performance in this case is most crucial to a system's cost benefit ratio.

In section 2, we present the scope of RSerPool and related work, section 3 gives a short overview over the RSerPool architecture. A quantification of RSerPool systems including the definition of performance metrics is given in section 4. This is followed by the description of our OMNeT++-based simulation model in section 5 and finally our results in section 6.

2. Scope and Related Work

A basic method to improve the availability of a service is server replication. Instead of having one server representing a single point of failure, servers are simply duplicated. Most approaches like Linux Virtual Server [23], Cisco™ Distributed Director [5] or application-layer anycast [3] simply map a client's session to one of the servers and use the abort-and-restart principle in case of server failure (i.e. when the sessions of the failed server are lost and have to be restarted). While this approach is sufficient for its main application - web server farms - it causes unacceptable delays for long-lasting sessions and is useless for applications like video conferences or real-time transactions [34]. More sophisticated approaches like FT-TCP [1], M-TCP [31] or RSerPool [33] provide a session layer to allow a resumption of the interrupted session on a new server. A survey of methods for the necessary server state replication can be found in [35], a very handy technique is the client-based state sharing [8].

The existence of multiple servers for redundancy automatically leads to load distribution and load balancing. While load distribution [2] refers only to the assignment of work to a processing element, load balancing refines this definition by requiring the assignment to maintain a balance across the processing elements. The balance refers to an application-specific parameter like CPU load or memory usage.

A classification of load distribution algorithms can be found in [19]; the two classes important for this paper are non-adaptive and adaptive ones. Adaptive strategies base their assignment decisions on the processing elements' current status (e.g. CPU load) and therefore require up-to-date information. On the other hand, non-adaptive algorithms do not require such status data. An analysis of adaptive load distribution algorithms can be found in [22]; performance

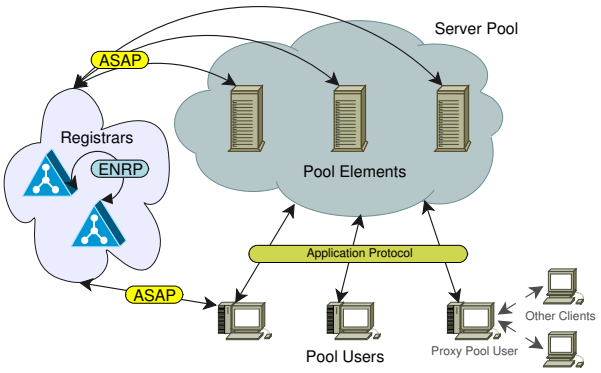

Figure 1. The RSerPool Architecture

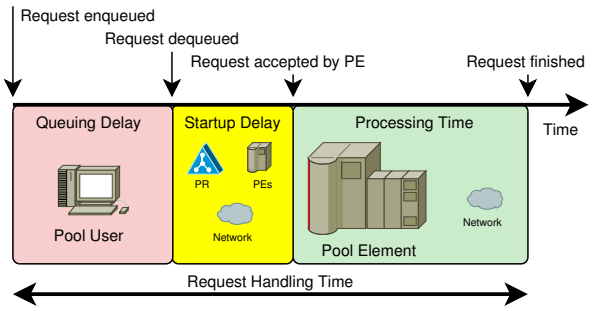

Figure 2. Request Handling Delays

evaluations for web server systems using different algorithms are presented in [4, 6, 18].

The scope of RSerPool [33] is to provide an open, application-independent and highly available framework for the management of server pools and the handling of a logical communication between a client and a pool. Essentially, RSerPool constitutes a communications-oriented overlay network, where its session layer allows for session migration comparable to [1, 31]. While server state replication is highly application-dependent and out of the scope of RSerPool, it provides mechanisms to support arbitrary schemes [7, 8, 14]. The pool management provides sophisticated server selection strategies [15, 16, 32] for load balancing, both adaptive and non-adaptive. Custom algorithms for new applications can be added easily [13].

3. The RSerPool Architecture

An illustration of the RSerPool architecture defined in [33] is shown in figure 1. It consists of three component classes: servers of a pool are called *pool elements* (PE). Each pool is identified by a unique *pool handle* (PH) in the handlespace, i.e. the set of all pools; the handlespace is managed by *registrars* (PR). PRs of an operation scope synchronize their view of the handlespace using the Endpoint haNdlespace Redundancy Protocol (ENRP [30, 36]), transported via SCTP [20, 21, 28]. An operation scope has a limited range, e.g. a company or organization; RSerPool does not intend to scale to the whole Internet. Nevertheless, it is assumed that PEs can be distributed worldwide, for their service to survive localized disasters (e.g. earthquakes or floodings).

A client is called *pool user* (PU) in RSerPool terminology. To use the service of a pool given by its PH, a PE has to be selected. The selection works in two stages: first, an arbitrary PR of the operation scope is asked for a *handle resolution* of the PH to a list of PE identities. This communication between PU and PR uses the Aggregate Server Access Protocol (ASAP [29, 30]). The PR selects the requested list of PE identities using a pool-specific selection rule, called *pool policy*. The PU writes this list into its local cache, denoted as PU-side cache, and selects again one PE for its communication. Subsequent handle resolutions may be directly satisfied from the cache, until its entries time out. This timeout is called *stale cache value*. Using a stale cache value of zero, every handle resolution must query a PR.

Adaptive and non-adaptive pool policies are defined in [32], relevant for this paper are the non-adaptive policies Round Robin (RR) and Random (RAND) and the adaptive policy Least Used (LU). LU selects the least-used PE, according to up-to-date load information. Round robin selection is applied among multiple least-loaded PEs [13]. The definition of *load* is application-specific and could e.g. be the current number of users, bandwidth or CPU load.

For more detailed information on RSerPool, see [10, 11, 13, 14, 16, 17].

4. Quantifying an RSerPool System

4.1. System Parameters

The service provider side of an RSerPool system consists of a pool of PEs, using a certain server selection policy. Each PE has a request handling *capacity*, which we define in the abstract unit of calculations per second. Depending on the application, an arbitrary view of capacity can be mapped to this definition, e.g. CPU cycles, bandwidth or memory usage. Each request consumes a certain amount of calculations, we call this amount *request size*. A PE can handle multiple requests simultaneously, in a processor sharing mode as commonly used in multitasking operating systems.

On the service user side, there is a set of PUs. The amount of PUs can be given by the ratio between PUs and PEs (PU:PE ratio), which defines the parallelism of the request handling: the higher the PU:PE ratio, the more requests have to be simultaneously handled by the PEs. Each PU generates a new request in an interval denoted as *request interval*. Requests are queued in the *request queue* and are sequentially assigned to PEs selected by the RSerPool mechanisms (see section 3).

The total delay for handling a request d_{handling} is defined as the sum of queuing delay (stay in request queue), startup delay (dequeuing until reception of acceptance acknowledgement from PE) and processing time (acceptance

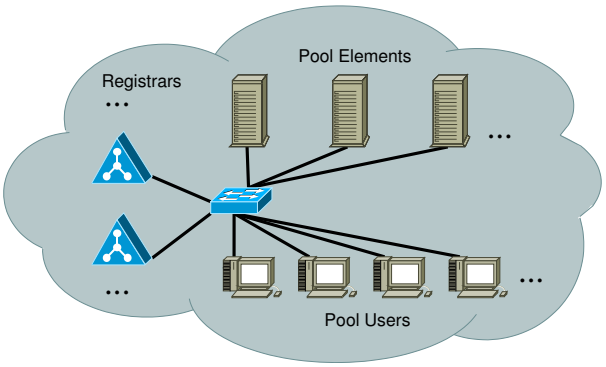

Figure 3. Our Simulation Setup

until finish) as illustrated in figure 2:

$$d_{\text{handling}} = d_{\text{queuing}} + d_{\text{startup}} + d_{\text{processing}} \quad (1)$$

Using d_{handling}, we define the *handling speed* in calculations/s as:

$$\text{handlingSpeed} = \frac{\text{reqSize}}{d_{\text{handling}}} \quad (2)$$

Using the definitions above, it is now possible to give a formula for the system's utilization:

$$\text{sysUtil} = \text{puToPERatio} * \frac{\frac{\text{reqSize}}{\text{reqInt}}}{\text{peCapacity}} \quad (3)$$

The load fraction generated by a single PU is given by the following formula:

$$\text{puLoad} = \frac{\text{reqSize}}{\text{reqInt} * \text{peCapacity}} \quad (4)$$

In summary, the workload of a RSerPool system is given by the three dimensions (1) PU:PE ratio, (2) request interval and (3) request size. In a well-designed client/server system, the amount and capacities of servers are provisioned for a certain *target system utilization*, e.g. 80%. That is, by setting any two of the parameters, the value of the third one can be calculated using equation 3.

4.2. Performance Metrics

To evaluate the performance impacts of parameter variation, we define two performance metrics. For the service provider, performance obviously refers to the system utilization as defined in equation 3: only utilized servers gain revenue. On the other hand, we denote service efficiency for service users as the achieved handling speed defined in equation 2. This definition not only includes the processing time itself but also the waiting time for the service.

5. Our Simulation Model

For our performance analysis, we have developed a simulation model using OMNeT++ [24], containing full implementations of the protocols ASAP [29] and ENRP [36], a PR module and PE and PU modules modelling the request handling scenario defined in section 4.

Our scenario setup is shown in figure 3: all components are connected by a switch; network delay is introduced by link delays only. We neglect component latencies here, because a RSerPool system is usually geographically distributed using WANs, to survive localized disasters (e.g. an earthquake); therefore, only the network delay is significant. The latency of the pool management by PRs is also negligible, as we show in our paper [13].

Since our goal is a generic parameter sensitivity analysis being independent of specific applications, we use negative exponential distribution for request intervals and request sizes. Unless otherwise specified, the used target system utilization (see section 4) is 80% and the PU-side handle resolution cache is turned off (*stale cache value* set to 0s). For the LU policy, we define *load* as the current amount of simultaneously handled requests. The capacity of a PE is 10^6 calculations/s, we use 10 PEs and the simulation runtime is 120 minutes; each simulation has been repeated 18 times with different seeds to achieve statistical accuracy.

The amount of PRs has been set to 1. We will show in section 6.2 that this parameter does not significantly affect the results; PR synchronization via ENRP only introduces the delay of the network. For this paper, we neglect congestion and failure scenarios of these connections, because we assume ENRP connections to be highly reliable, due to the usage of SCTP multi-homed associations [13]. Furthermore, they are established in controlled networks (e.g. of a company or an organization) where QoS mechanisms could be applied easily.

For the statistical post-processing of our results, we used R Project [26] for the computation of 95% confidence intervals and plotting.

6. Results

6.1. Workload Parameter Variation

In our first set of simulations, we examine the performance impact of varying the three workload parameters: PU:PE ratio, request size (normalized by $\frac{\text{Request Size}}{\text{PE Capacity}}$) and request interval on a system designed for a target utilization of 80%. In this simulation set, we do not want the network delay to affect the results, therefore we turn it off. The impact of network delay will be examined later in section 6.3. Figure 4 presents the resulting average utilization and figure 5 the average handling speed (normalized by $\frac{\text{Request Size}}{\text{PE Capacity}}$). In each figure, the left part shows the variation of the PU:PE ratio r, the middle part the variation of the request size:PE capacity ratio s and the right part the variation of the request interval i. In the following three subsections, we discuss these results.

6.1.1. Impact of the PU:PE Ratio For our first simulation set, we varied the PU:PE ratio r from 1 to 20 for request size:PE capacity ratios s from 1s to 100s. For each pair of both values, the request interval can be calculated

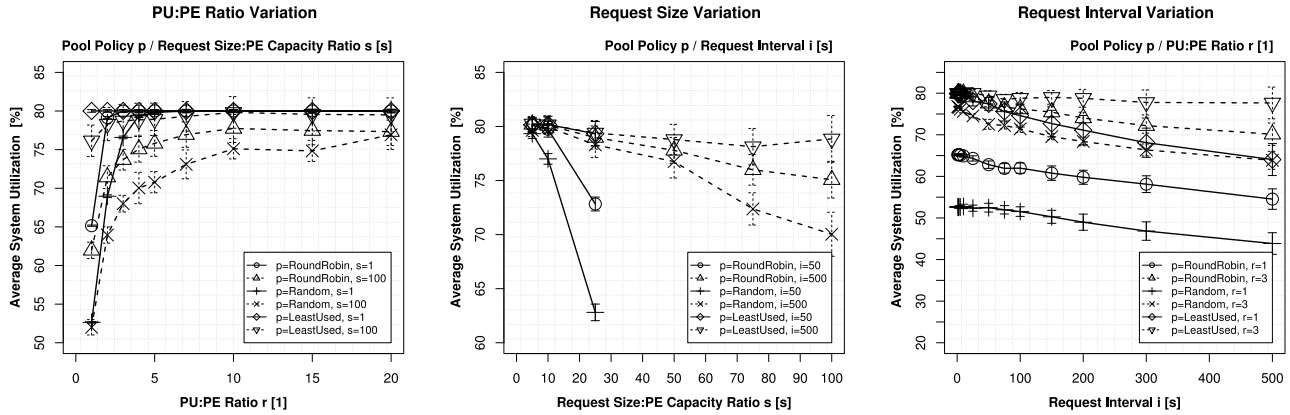

Figure 4. Impact of Workload Variation on the System Utilization

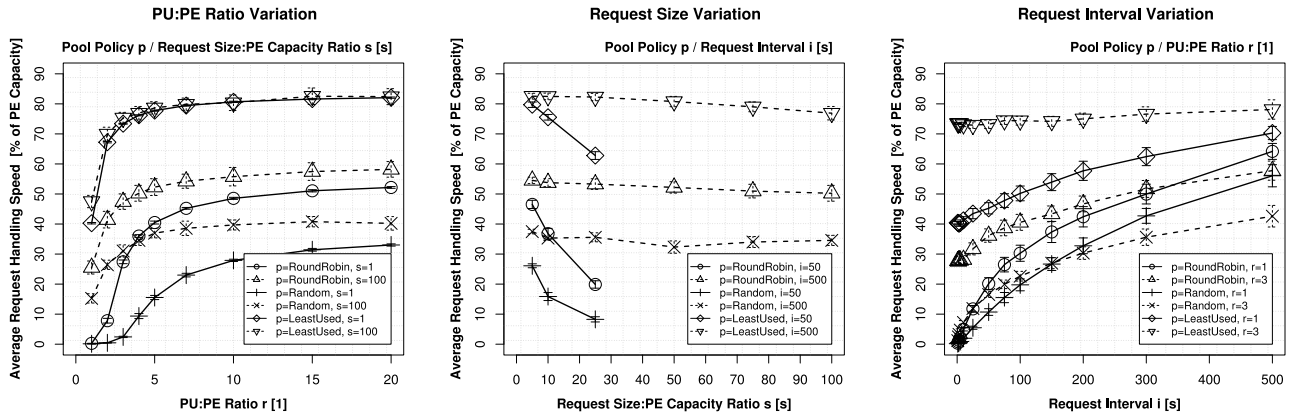

Figure 5. Impact of Workload Variation on the Request Processing Speed

based on equation 3 described in section 4:

$$\text{reqInt} = \frac{\text{puToPERatio} * \text{reqSize}}{\text{targetSysUtil} * \text{peCapacity}}$$

The left sides of figure 4 (utilization) and figure 5 (handling speed) show a carefully chosen subset of our results which contains the essential results.

As shown, the PU:PE ratio r giving the degree of parallelism in request handling has a significant impact on the utilization: At $r = 1$, the utilization is at 53% for the RAND policy and at about 65% for RR. Using LU, it nearly reaches 80%. The utilization difference for the policies becomes significantly smaller when r increases: for $r = 5$, the difference is about 6% and for $r = 10$, it decreases to about 3%.

The reason for this behavior is the amount of simultaneous requests processed by the PEs: a PU:PE ratio $r = 1$, there should be exactly one PE for every PU. That is, each PU expects to get a PE exclusively, which processes its requests during 80% (target utilization) of its runtime (see equation 4). Each time a "bad" PE is selected for a request, one PE is idle while another one has to split up its capacity to handle two requests simultaneously. Obviously, this behavior is most frequent when PEs are randomly chosen. For RR selection, the PE just selected should be chosen again only after having used every other PE before. This method already achieves a significant improvement over RAND. Finally, LU has the knowledge of the PEs' current load states; therefore - except for the rare cases of simultaneous selection - the least-loaded PE can be used. This is the reason for the good performance of LU.

Observing the utilization for a variation of the request size:PE capacity ratio s, only minor differences are shown. Even for a change of two orders of magnitude as presented in figure 4, the utilization between $s = 1$ (solid lines) and $s = 100$ (dotted lines) only decreases by 1%-2% for LU, about up to 4% for RR and up to about 7% for RAND.

The reason for the small decrement is that longer requests increase the impact duration of the selection decision: for example, a heavily-loaded PE may become idle within the next few seconds while a 75%-loaded one - putatively the appropriate choice - may stay at this load level for quite some time. Clearly, the probability of a non-optimal assignment is highest for RAND and lowest for LU, explaining the performance differences between these policies.

Comparing the different policies, it can be observed that the higher the PU:PE ratio r, the smaller the utilization differences: trying to assume which PE is a good choice to select - based on load information for LU or by list position for RR - becomes more and more inappropriate.

The results for the handling speed normalized by $\frac{\text{Request Size}}{\text{PE Capacity}}$ shown in figure 5 (left) reflect the results for the system utilization: Since the per-PU load (see equation 4) becomes higher with a lower PU:PE ratio, a "bad" selection decision leads to queuing of requests. Clearly, this "request jam" contributes significantly to the handling speed loss.

While the utilization for higher request sizes s decreases, the handling speed increases: for example 100 requests of size $s = 1$ are affected 100 times by the queuing delay, while one large request of $s = 100$ is only affected once - for the same amount of calculations to be processed. Clearly, this results in a significant handling speed gain.

Unlike the utilization curves for the different policies, the handling speed does not converge to a certain value for all policies when the PU:PE ratio r becomes high enough. Instead, a significant gap between the handling speeds for RAND, RR and LU remains: RAND and RR frequently make "bad" assignments, leading to low handling speeds and therefore longer request queues of the PUs. While the per-PU load (see equation 4) decreases with r and therefore lowers the penalty of queuing delay (leading to an improved handling speed), the probability to make non-optimal selection decisions remains significantly higher for RAND and RR than for LU.

6.1.2. Impact of the Request Size In this simulation set, we vary the request size:PE capacity ratio s between 5s and 100s for values of the request interval i between 1s and 500s. Based on equation 3 described in section 4, we can calculate the PU:PE ratio:

$$\text{puToPERatio} = \frac{\text{reqInt} * \text{targetSysUtil} * \text{peCapacity}}{\text{reqSize}}$$

Note, that the smaller the request size, the higher the PU:PE ratio.

The middle blocks of figure 4 (utilization) and figure 5 (handling speed) show a carefully chosen subset of our results which contains the essential results.

Obviously, the utilization is highest for small values of the request size:PE capacity ratio s, since here the PU:PE ratio r is highest. As already observed in the analysis of the PU:PE ratio in section 6.1.1, the difference between the policies becomes small for high values of r due to the parallelism of request handling.

Larger values of i lead to a more shallow descent of the PU:PE ratio r, therefore, the utilization decrement rate for rising s becomes smaller. Compare the results for $i = 50$ (solid lines) with the curves for $i = 500$ (dotted lines) at $s = 25$: the PU:PE ratio at $s = 25$ has already reached 1 for $i = 50$ and therefore the gap between the three policies as explained in section 6.1.1 can be observed. On the other hand, the PU:PE ratio r is 16 for $i = 500$; therefore, the difference between the policies is only about 2%.

The handling speed results shown in figure 5 (middle) reflect the results of the utilization curves. r sinks with rising request size s, i.e. the possibility to compensate "bad" selections by parallelity becomes smaller and the handling speed decreases. Obviously, this effect becomes smaller for higher request intervals i.

Again, as explained in section 6.1.1, there is a huge difference between the three policies, due to their different selection decision qualities.

6.1.3. Impact of the Request Interval The request interval is the third and last workload parameter. We examined it in the range from 1 to 500 for PU:PE ratios from 1 to 10. Based on equation 3 described in section 4, we can calculate the request size:

$$\text{reqSize} = \frac{\text{reqInt} * \text{targetSysUtil} * \text{peCapacity}}{\text{puToPERatio}}$$

Note, that the request size increases with the request interval.

The right sides of figure 4 (utilization) and figure 5 (handling speed) show a carefully chosen subset of our results, containing the essential information.

With increasing request interval i, the request size:PE capacity ratio s also increases. As already explained in section 6.1.1 and section 6.1.2, this leads to a small decrement of the utilization due to the longer durability of selection decisions.

Obviously, the most influencing factor is the PU:PE ratio r: while the policies' utilizations for $r = 1$ differ in the range of about 20% to 25%, the variation already sinks to less than about 10% for $r = 3$ for the reason of parallelism as explained in section 6.1.1.

The handling speed curves presented in figure 5 (right) mainly reflect the results for the system utilization. But while the utilization slightly decreases, the handling speed slowly increases. The reason is that s increases with i. Therefore, as explained in detail in section 6.1.1, the amount of requests decreases and the fraction of processing delay $d_{processing}$ in equation 1 gains more importance over the queuing delay $d_{queuing}$. This implies a handling speed improvement (see equation 2).

Note, that for sufficiently high values of i, the handling speed curves for RR and RAND at $r = 1$ exceed the curves for $r = 3$. Here, the gain by larger (and therefore fewer) requests at $r = 1$ oversteps the gain by parallelity at $r = 3$.

6.2. Registrar Parameter Variation

An important parameter for redundancy is the amount of PRs. In section 5 we noted that this amount does not significantly influence the performance. To substantiate our asser-

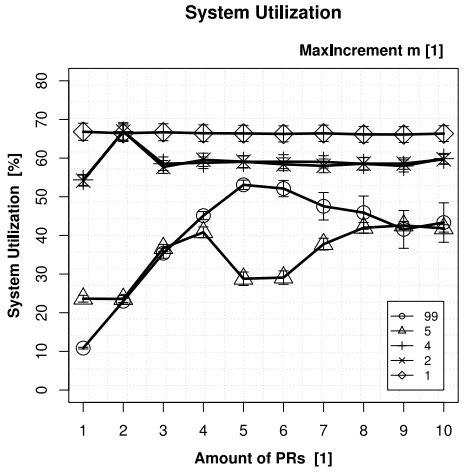

Figure 6. Registrar Parameters

tion, we made simulations with different workload sets and PR ranges from 1 to 10.

PRs synchronize their handlespace copies using the ENRP protocol. Neglecting the network delay (analyzed later in section 6.3), it is obvious that the amount of PRs does not affect the LU and RAND policies, since they are stateless. However, RR is stateful, i.e. the selection of the next PE depends on the previous selections. This introduces a problem, since multiple PRs select independently from each other.

Figure 6 shows the impact of varying PR amounts on the utilization for different settings of *MaxIncrement* m. We denote the number of steps the round robin pointer in the PE list is forwarded after selection as *MaxIncrement*; e.g. a setting of $m = 3$ means that after selecting n elements, the pointer is incremented by $\min(m, n)$. As we have already shown in our paper [16], the selection performance highly depends on the setting of m. In short, if the amount of PE identities selected by the PR is larger than the number of PEs actually used by the PU, this leads to systematical skipping of PEs while other PEs are highly loaded. The setting $m = 1$ has been found to be a useful value.

The example in figure 6 shows the utilization for a PU:PE ratio $r = 1$ and a request size:PE capacity ratio $s = 10$. Obviously, $m = 1$ is also useful to avoid RR problems when scaling the amount of PRs. Note, that the utilization slightly decreases with the amount of PRs: round robin selection on independent components differs from a global round robin selection. LU and RAND (not shown here) are not affected by the amount of PRs: the utilizations for both of them are constant at 80%.

6.3. Network Delay

The performance impact of network delay is most significant when (1) the PU:PE ratio r is small, so that the per-PU load (see equation 4) is high and/or (2) the ratio between delay and request size:PE capacity ratio s is high. The first case reduces the system's capability to absorb the impact of "bad" selection decisions (see section 6.1.1); in the second case the delay decreases the handling speed (see equation 2).

We made simulations for a wide range of workload parameters (r from 1 to 10, s from 1 to 10) and delays (given as ratio between component RTT and s, from 0.0 to 1.0); a carefully chosen subset ($s = 1$) of the results containing the essential information is shown in figure 7. The left side presents the utilization, the right side the handling speed (normalized by PE capacity).

Obviously, the PU:PE ratio r has the main impact on the utilization for a rising delay: while the curves only slightly decrease for $r = 3$, there is a steep descent for $r = 1$. As explained in section 6.1.1, the lower the PU:PE ratio r, the higher the amount of capacity a single PU expects from its PE. That is, at $r = 1$ a PE is expected to exclusively provide 80% of its runtime to request handling. Only 20% remain to absorb both, the delay of communications and the speed degradation due to simultaneously handled requests.

An important observation can be made for LU: while the utilization of LU converges to the curve of RR for $r = 1$, it converges to the curve of RAND for higher values of r. The reason is clear: while for $r = 1$ RR provides the best chance to get an unloaded PE, the probability of RR to make a good choice decreases with rising parallelism (see section 6.1.1).

The handling speed results presented on the right side of figure 7 reflect the observations for the utilization. Clearly, the network delay has a significant impact on the handling speed for small settings of s (here: $s = 1$), since it affects the handling speed (see equations 2 and equation 1) by the communication of PU and PR (handle resolution) and PU and PE (request handling).

6.4. Handle Resolution Cache

As explained in section 3, the PU provides a cache to provide local handle resolutions. Using this cache, the overhead of asking a PR for handle resolution may be skipped.

To be effective, the stale cache value has to be larger than the request interval i. We made simulations for a large parameter range using PU:PE ratios r from 1 to 10, request size:PE capacity ratios s from 1 to 100 (the corresponding request interval i has been calculated using equation 3) and stale cache value:request interval ratios sir from 0 to 10. The value sir is a measure for how many times the cache could be used for a handle resolution before querying a PR.

Figure 8 presents a subset ($i = 3$; 60% target utilization) of the results for utilization (left side) and handling speed (right side), carefully chosen to present the essential effects.

As expected, the cache does not affect the RAND policy's results, neither utilization nor handling speed. Clearly, the utilization of LU quickly decays for rising sir at $r = 1$ due to high per-PU load (see equation 4) and therefore high penalty of "bad" selection decisions. Caching results in using out-of-date load information, therefore inappropriate decisions become more likely for larger values of sir.

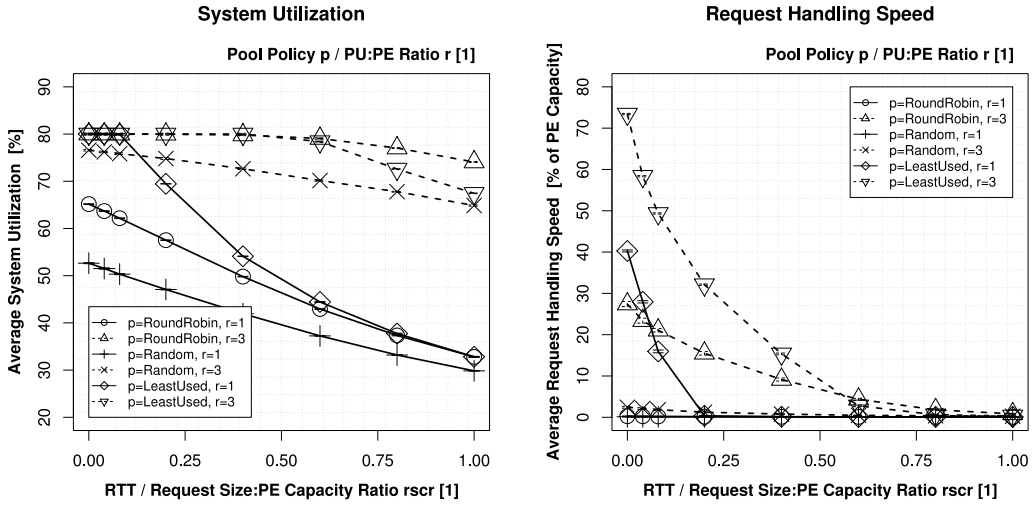

Figure 7. Impact of Network Delay

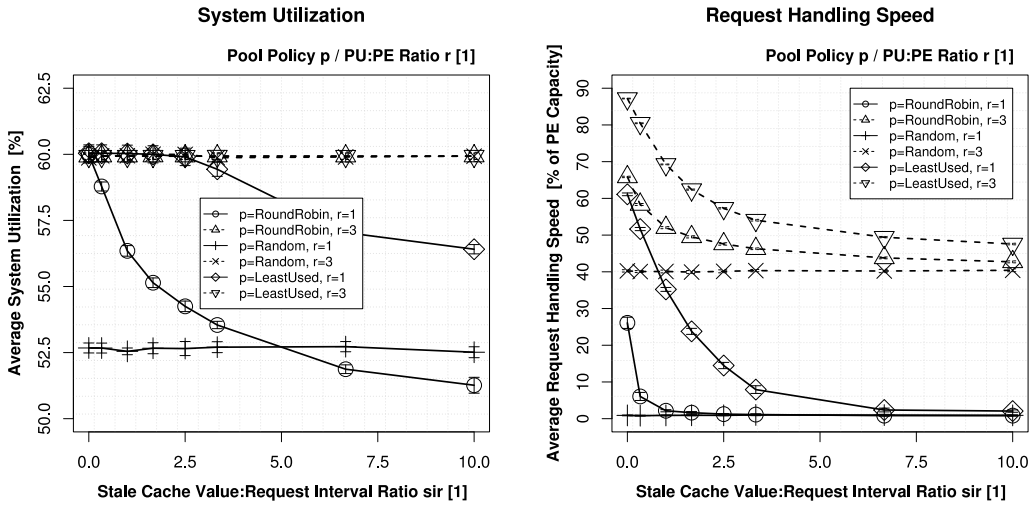

Figure 8. Impact of PU-side Handle Resolution Caching

For higher values of r, the situation becomes much better, since the per-PU penalty of a "bad" decision becomes smaller due to the parallelism (see section 6.1.1).

The reason for RR's utilization decay is the same as for the amount of PRs (see section 6.2): each component selects independently using round robin; but viewed globally, the selections differ from round robin behavior. For $r = 1$, it can be observed that RR's utilization even slightly falls short of RAND's utilization for sir greater than 5.0 - the higher the sir, the more independent are the RR selections, the more the global view of the round robin selections adapts to random.

Although the utilization penalty for using the cache becomes small when the PU:PE ratio r is high enough, no significant impact in form of a constant request handling speed can be observed for policies other than RAND: using LU or RR, there remains a significant loss in handling speed. This is caused by long request queues, due to inappropriate PE selection and therefore reduced processing speed.

As a conclusion, using the cache for a policy other than RAND does not make much sense. The gain of saving some overhead messages on handle resolution comes at a high price on performance. So, in which cases can the cache become valuable? We provide the answer in the next section.

6.5. Efficient Cache Usage

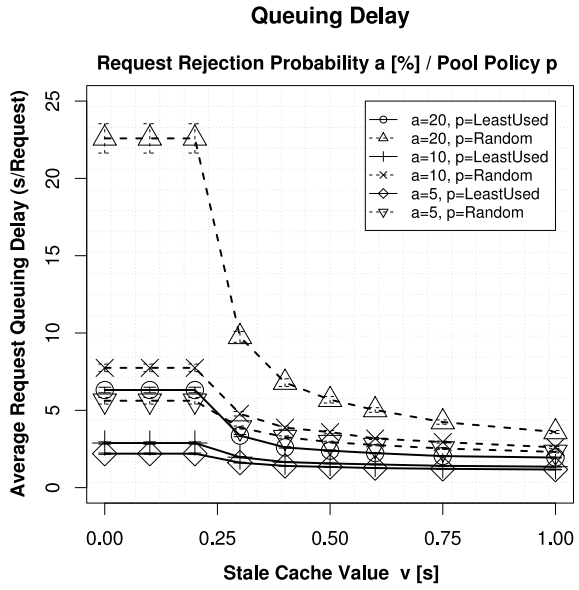

Figure 9. Efficient Cache Usage

Using the cache to reduce overhead traffic is usually inefficient, as shown in section 6.4. But there are situations where the cache becomes quite useful: (1) The request size:PE capacity ratio s is small, therefore network delay for the handle resolution at the PR significantly contributes to a reduction of the handling speed. (2) A PE may reject a request with a certain probability a_{PE}, implying an additional handle resolution to find a new PE. This is e.g. the case in telephone signaling, when the PE's request queue is full and it currently cannot accept any more request.

For a given a_{PE}, the total rejection probability after n trials is $a_{\text{total}} = (a_{\text{PE}})^n$. To reach a given value of a_{total} (e.g. 0.05), the number of trials n can be computed by:

$$n(a_{\text{total}}, a_{\text{PE}}) = \max(1, \lceil log_{a_{\text{PE}}}(a_{\text{total}}) \rceil).$$

Now, we want to cover the given number of trials by the cache instead of having to query a PR. That is, we compute the maximum delay for this amount of trials as follows:

$$d(a_{\text{total}}, a_{\text{PE}}) = \text{RTT}_{\text{PU}\leftrightarrow\text{PR}} + (n(a_{\text{total}}) - 1) * \text{RTT}_{\text{PU}\leftrightarrow\text{PE}}.$$

Using the computed delay as stale cache value, the trials are covered by the cache.

To show the effectiveness of the cache, we made an example simulation using a PU:PE ratio of $r = 3$, a request interval of $i = 1$ and a target utilization of 60%. The resulting request size:PE capacity ratio is $s = 0.2$ (such frequent and short transactions are typical for telecommunications signaling) for a component RTT of 200ms. Figure 9 shows the resulting queuing delay (see section 4 for the definition) for varying stale cache values from 0s (no cache) to 1s (equal to the request interval) and request rejection rates a_{PE} from 0.0 (0%) to 0.2 (20%). The system utilization is stable at about 60% for all settings, therefore we omit a figure. We also do not show plots for RR, since the results are quite similar.

Obviously, the cache has a huge impact on the reduction of the queuing delay (and therefore on an improvement of the handling speed). In the extreme case of RAND policy and $a_{PE} = 0.2$, a stale cache value of only 400ms reduces the queuing delay from 23s to 4s ($d(0.05, 0.20) = 400ms$); for $a_{PE} = 0.1$, the queuing delay of RAND drops from 8s to 4s at the same stale cache value ($d(0.05, 0.10) = 400ms$). Clearly, the reductions for LU are smaller, since LU's initial values without cache are much smaller than for RAND and the load values become inaccurate due to the cache. But nevertheless, there is still a significant queuing delay reduction by several seconds.

7. Conclusions and Future Work

In this paper, we first quantified the basic workload parameters of RSerPool systems (PU:PE ratio, request size and interval) and defined performance metrics for both service provider (system utilization) and service user (handling speed). We then provided a sensitivity analysis of the system performance for a broad range of the workload and system parameter space, providing application-independent guidelines for the behavior of RSerPool systems on workload changes.

The PU:PE ratio has been found to be the most critical parameter. A high parallelism in the request handling can compensate for deficiencies of the pool policies' load distribution quality, due to smaller per-PU load. It has been shown that the adaptive LU policy provides the highest stability on workload parameter changes. However, for increasing network delays the load information required by LU becomes inaccurate, resulting in convergence of LU's performance to the results of the non-adaptive RR and RAND policies.

Using the PU-side cache to avoid handle resolutions at a PR usually comes at a high price on system performance. But as we have shown, there are certain situations where this cache is able to achieve a significant performance gain.

The future goals of our ongoing RSerPool performance analysis include the examination of heterogeneous scenarios, where server capacities differ. It is also necessary to investigate the performance impacts of server failures and RSerPool's failover capabilities.

Furthermore, another goal of our future work is to transfer the results of our simulative research to our RSerPool prototype implementation [10, 27] and verify our simulation results in real life using the PLANETLAB [25].

References

[1] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov. Wrapping Server-Side TCP to Mask Connection Failures. In *Proceedings of the IEEE Infocom 2001*, 2001.

[2] E. Berger and J. C. Browne. Scalable Load Distribution and Load Balancing for Dynamic Parallel Programs. In *Proceedings of the WCBC 99*, 1999.

[3] S. Bhattacharjee, M. H. Ammar, E. W. Zegura, V. Shah, and Z. Fei. Application-Layer Anycasting. In *Proceedings of the IEEE Infocom 1997*, pages 1388–1396, 1997.

[4] V. Cardellini, M. Colajanni, and P. S. Yu. Geographic load balancing for scalable distributed Web systems. In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 20–27, 2000.

[5] Cisco™ Distributed Director. http://www.cisco.com.

[6] M. Colajanni and P. S. Yu. A Performance Study of Robust Load Sharing Strategies for Distributed Heterogeneous Web Server Systems. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):398–414, 2002.

[7] P. Conrad and P. Lei. Services Provided By Reliable Server Pooling. Internet-Draft Version 01, IETF, RSerPool WG, Jun 2004. draft-ietf-rserpool-service-01.txt, work in progress.

[8] T. Dreibholz. An efficient approach for state sharing in server pools. In *Proceedings of the 27th IEEE Local Computer Networks Conference*, Tampa, Florida/U.S.A., Oct 2002.

[9] T. Dreibholz. Applicability of Reliable Server Pooling for Real-Time Distributed Computing. Internet-Draft Version 00, University of Duisburg-Essen, Jul 2005. draft-dreibholz-rserpool-applic-distcomp-00.txt, work in progress.

[10] T. Dreibholz. Das rsplib–Projekt – Hochverfügbarkeit mit Reliable Server Pooling. In *Proceedings of the LinuxTag 2005*, Karlsruhe/Germany, Jun 2005.

[11] T. Dreibholz, A. Jungmaier, and M. Tüxen. A new Scheme for IP-based Internet Mobility. In *Proceedings of the 28th IEEE Local Computer Networks Conference*, Königswinter/Germany, Nov 2003.

[12] T. Dreibholz and E. P. Rathgeb. An Application Demonstration of the Reliable Server Pooling Framework. In *Proceedings of the 24th IEEE Infocom 2005*, Miami, Florida/U.S.A., Mar 2005.

[13] T. Dreibholz and E. P. Rathgeb. Implementing of the Reliable Server Pooling Framework. In *Proceedings of the 8th IEEE International Conference on Telecommunications 2005*, Zagreb/Croatia, Jun 2005.

[14] T. Dreibholz and E. P. Rathgeb. RSerPool - Providing Highly Available Services using Unreliable Servers. In *Proceedings of the 31st IEEE EuroMirco Conference on Software Engineering and Advanced Applications 2005*, Porto/Portugal, Aug 2005.

[15] T. Dreibholz and E. P. Rathgeb. The Performance of Reliable Server Pooling Systems in Different Server Capacity Scenarios. In *Proceedings of the IEEE TENCON '05*, Melbourne/Australia, Nov 2005.

[16] T. Dreibholz, E. P. Rathgeb, and M. Tüxen. Load Distribution Performance of the Reliable Server Pooling Framework. In *Proceedings of the 4th IEEE International Conference on Networking 2005*, Saint Gilles Les Bains/Reunion Island, Apr 2005.

[17] T. Dreibolz and M. Tüxen. High availability using reliable server pooling. In *Proceedings of the Linux Conference Australia 2003*, Perth/Australia, Jan 2003.

[18] S. G. Dykes, K. A. Robbins, and C. L. Jeffery. An empirical evaluation of client-side server selection algorithms. In *Proceesing of the IEEE Infocom 2000*, pages 1361–1370, 2000.

[19] D. Gupta and P. Bepari. Load Sharing in Distributed Systems. In *Proceedings of the National Workshop on Distributed Computing*, 1999.

[20] A. Jungmaier, E. Rathgeb, and M. Tüxen. On the Use of SCTP in Failover-Scenarios. In *Proceedings of the SCI 2002, Volume X, Mobile/Wireless Computing and Communication Systems II*, volume X, Orlando/U.S.A., Jul 2002.

[21] A. Jungmaier, M. Schopp, and M. Tüxen. Performance Evaluation of the Stream Control Transmission Protocol. In *Proceedings of the IEEE Conference on High Performance Switching and Routing*, Heidelberg/Germany, June 2000.

[22] O. Kremien and J. Kramer. Methodical Analysis of Adaptive Load Sharing Algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 3(6), 1992.

[23] Linux Virtual Server. http://www.linuxvirtualserver.org.

[24] OMNeT++ Discrete Event Simulation System. http://www.omnetpp.org.

[25] PlanetLab: Home. http://www.planet-lab.org.

[26] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2005. ISBN 3-900051-07-0.

[27] Thomas Dreibholz's RSerPool Page. http://tdrwww.exp-math.uni-essen.de/dreibholz/rserpool.

[28] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. Standards Track RFC 2960, IETF, Oct 2000.

[29] R. Stewart, Q. Xie, M. Stillman, and M. Tüxen. Aggregate Server Access Protcol (ASAP). Internet-Draft Version 12, IETF, RSerPool WG, Jul 2005. draft-ietf-rserpool-asap-12.txt, work in progress.

[30] R. Stewart, Q. Xie, M. Stillman, and M. Tüxen. Aggregate Server Access Protocol (ASAP) and Endpoint Handlespace Resolution Protocol (ENRP) Parameters. Internet-Draft Version 09, IETF, RSerPool WG, Jul 2005. draft-ietf-rserpool-common-param-09.txt, work in progress.

[31] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Highly available Internet services using connection migration. In *Proceedings of the ICDCS 2002*, 2002.

[32] M. Tüxen and T. Dreibholz. Reliable Server Pooling Policies. Internet-Draft Version 01, IETF, RSerPool WG, Jun 2005. draft-ietf-rserpool-policies-01.txt, work in progress.

[33] M. Tüxen, Q. Xie, R. Stewart, M. Shore, J. Loughney, and A. Silverton. Architecture for Reliable Server Pooling. Internet-Draft Version 10, IETF, RSerPool WG, Jul 2005. draft-ietf-rserpool-arch-10.txt, work in progress.

[34] U. Uyar, J. Zheng, M. A. Fecko, S. Samtani, and P. Conrad. Evaluation of Architectures for Reliable Server Pooling in Wired and Wireless Environments. *IEEE JSAC Special Issue on Recent Advances in Service Overlay Networks*, 22(1):164–175, 2004.

[35] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding Replication in Databases and Distributed Systems. In *Proceedings of the The 20th International Conference on Distributed Computing Systems*, page 464, 2000.

[36] Q. Xie, R. Stewart, M. Stillman, M. Tüxen, and A. Silverton. Endpoint Name Resolution Protcol (ENRP). Internet-Draft Version 12, IETF, RSerPool WG, Jul 2005. draft-ietf-rserpool-enrp-12.txt, work in progress.

[37] Y. Zhang. Distributed Computing mit Reliable Server Pooling. Masters thesis, Universität Essen, Institut für Experimentelle Mathematik, Apr 2004.