A New Approach of Performance Improvement for Server Selection in Reliable Server Pooling Systems*

Xing Zhou
Hainan University, College of Information Science and Technology
Renmin Road 58, 570228 Haikou, Hainan, China
University of Duisburg-Essen, Institute for Experimental Mathematics
Ellernstrasse 29, 45326 Essen, Germany
xing.zhou@uni-due.de

Thomas Dreibholz, Erwin P. Rathgeb
University of Duisburg-Essen, Institute for Experimental Mathematics
Ellernstrasse 29, 45326 Essen, Germany
{thomas.dreibholz,erwin.rathgeb}@uni-due.de

Abstract

Reliable Server Pooling (RSerPool) is a light-weight protocol framework for server redundancy and session failover, currently still under standardization by the IETF RSerPool WG. While the basic ideas of RSerPool are not completely new, their combination into a single, resource-efficient and unified architecture is. Server redundancy directly leads to the issues of load distribution and load balancing, which are both important for the performance of RSerPool systems.

While there has already been some research on the server selection policies of RSerPool, an interesting question still remains open: Is it possible to further improve the load balancing performance of certain policies by simply letting servers reject inappropriately scheduled requests? In this case, the failover handling mechanisms of RSerPool could choose a possibly better server instead.

The purpose of this paper is, after presenting an outline of the RSerPool framework, to analyse and evaluate the performance of our new approach. In particular, we will also analyse the impact of RSerPool protocol parameters on the performance of the server selection functionalities as well as on the overhead.

Keywords: *Reliable Server Pooling, Redundancy, Load Balancing, Performance Evaluation*

*Parts of this work have been funded by the German Research Foundation (Deutsche Forschungsgemeinschaft).

1 Introduction and Scope

The Reliable Server Pooling (RSerPool) architecture [11,18,28] currently under standardization by the IETF RSerPool WG is an overlay network framework to provide server replication [7] and session failover capabilities [3, 14] to its applications. Server redundancy leads to load distribution and load balancing [24], which are also covered by RSerPool [13, 15, 21]. But in strong contrast to already available solutions in the area of GRID and high-performance computing [22, 23], the fundamental property of RSerPool is to be "light-weight", i.e. it must be usable on devices providing only limited memory and CPU resources (e.g. embedded systems like telecommunications equipment or routers). This property restricts the RSerPool architecture to the management of pools and sessions only, but on the other hand makes a very efficient realization possible [4, 12, 16]. A generic classification of load distribution algorithms can be found in [24]; the two most important classes – also supported by RSerPool – are non-adaptive and adaptive algorithms. Adaptive strategies base their assignment decisions on the current status of the processing elements and therefore require up-to-date information. Non-adaptive algorithms – on the other hand – do not require such status data. More details on such algorithms can be found in [1, 27].

There has already been some research on the performance of RSerPool usage for applications like SCTP-based endpoint mobility [9, 10], VoIP with SIP [2], web server pools [29], IP Flow Information Export (IPFIX) [8], real-time distributed computing [6, 7, 14, 15, 19, 37] and battlefield networks [34]. A generic application model for RSerPool systems has been introduced by [7, 13], which includes performance metrics for the provider side (pool uti-

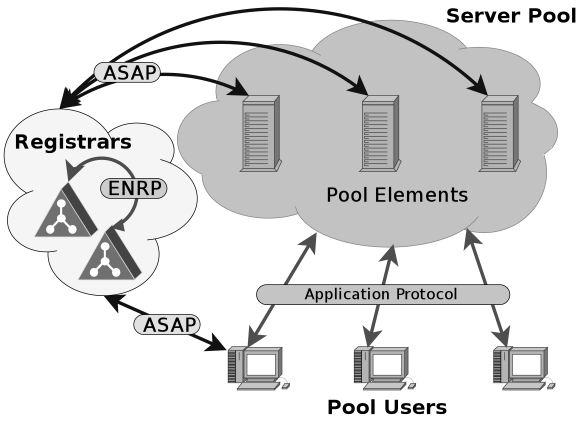

Figure 1. The RSerPool Architecture

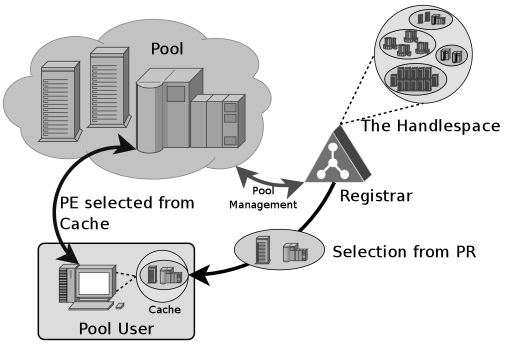

Figure 2. The Server Selection by PR and PU

lization) and user side (request handling speed). Based on this model, the load balancing quality of different pool policies has been evaluated [7, 13, 21]. A question arisen from these results is whether it is possible to improve the load balancing by allowing servers to reject requests. In this case the failover mechanisms of RSerPool could choose a possibly better one instead. The goal of this paper is therefore to evaluate the performance of this strategy, with respect to the accruing overhead. We also identify critical configuration parameter ranges in order to provide a guideline for designing and configuring efficient RSerPool systems.

2 The RSerPool Protocol Framework

Figure 1 illustrates the RSerPool architecture as defined in [28]. It contains three classes of components: in RSerPool terminology, servers of a pool are called *pool elements* (PE), a client is denoted as *pool user* (PU). The *handlespace* – which is the set of all pools – is managed by redundant *pool registrars* (PR). Within the handlespace, each pool is identified by a unique *pool handle* (PH). PRs of an *operation scope* synchronize their view of the handlespace using the Endpoint haNdlespace Redundancy Protocol (ENRP [36]), transported via SCTP [25, 26, 30, 33]. An operation scope has a limited range, e.g. a company or organization. In particular, it is restricted to a single administrative domain – in contrast to GRID computing [22, 23] – in order to keep the management complexity [12, 16] at a minimum. Nevertheless, it is assumed that PEs can be distributed globally, for their service to survive localized disasters [17].

PEs choose an arbitrary PR of the operation scope to register into a pool by using the Aggregate Server Access Protocol (ASAP [31]), again transported via SCTP. Upon registration at a PR, the chosen PR becomes the Home-PR (PR-H) of the newly registered PE. A PR-H is responsible for monitoring its PEs' availability by keep-alive messages (to be acknowledged by the PE within a given timeout) and propagates the information about its PEs to the other PRs of the operation scope via ENRP updates.

In order to access the service of a pool given by its PH, a PU requests a PE selection from an arbitrary PR of the operation scope, using the Aggregate Server Access Protocol (ASAP [31]), transported via SCTP. As illustrated in figure 2, the PR selects the requested list of PE identities by applying a pool-specific selection rule, called *pool policy*. Adaptive and non-adaptive pool policies are defined in [32], relevant to this paper are the non-adaptive policies Round Robin (RR) and Random (RAND) as well as the adaptive policy Least Used (LU). LU selects the least-used PE, according to up-to-date load information; the actual definition of *load* is application-specific. Round robin selection is applied among multiple least-loaded PEs [12]. Detailed discussions of pool policies can be found in [7, 13, 15, 17, 21].

The PU writes the list of PE identities selected by the PR into its local cache (denoted as *PU-side cache*). From the cache, the PU selects – again using the pool's policy – one element to contact for the desired service. The PU-side cache constitutes a local, temporary and partial copy of the handlespace. Its contents expire after a certain timeout, denoted as *stale cache value*. In many cases, the stale cache value is simply 0s, i.e. the cache is used for a single handle resolution only [13].

3 Quantification and Performance Metrics

In order to evaluate the performance of an RSerPool system, it is necessary to quantify it. We therefore extend the model of [13], in which the service provider side of a RSerPool system consists of a pool of PEs. 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 or memory usage. Each request consumes a certain number of calculations, we call this number *request*

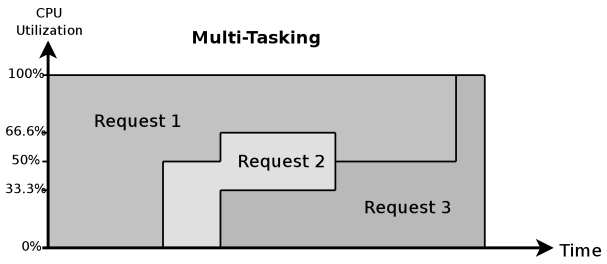

Figure 3. The Processing of Requests

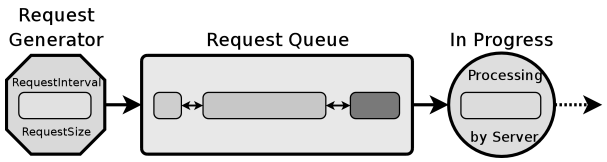

Figure 4. Request Handling by the Pool User

size. A PE can handle multiple requests simultaneously, in a processor sharing mode as commonly used in multi-tasking operating systems. An illustration can be found in figure 3. The maximum number of simultaneously handled requests is limited by the parameter $\mathrm{MaxRequests}$. If reached, any further request is rejected.

On the user side, there is a set of PUs. The number of PUs can be given by the ratio between PUs and PEs (*PU:PE ratio*), which defines the parallelism of the request handling. Each PU generates a new request in an interval denoted as *request interval*. The requests are queued and sequentially assigned to PEs, as illustrated in figure 4.

The total delay for handling a request d_{Handling} is defined as the sum of queuing delay d_{Queuing}, startup delay d_{Startup} (dequeuing until reception of acceptance acknowledgement) and processing time $d_{\mathrm{Processing}}$ (acceptance until finish):

$$d_{\mathrm{Handling}} = d_{\mathrm{Queuing}} + d_{\mathrm{Startup}} + d_{\mathrm{Processing}}. \quad (1)$$

That is, d_{Handling} not only incorporates the time required for processing the request, but also the latencies of queuing, server selection and protocol message transport.

The *handling speed* (in calculations/s) is defined as:

$$\mathrm{handlingSpeed} = \frac{\mathrm{requestSize}}{d_{\mathrm{handling}}}.$$

For convenience reasons, the handling speed can also be represented in % of the average PE capacity. Clearly, the main user-side performance metric is the handling speed and should of course be as high as possible. A secondary performance metric is the number of handle resolutions at a PR per request. This overhead – consuming time and network bandwidth – should be kept small.

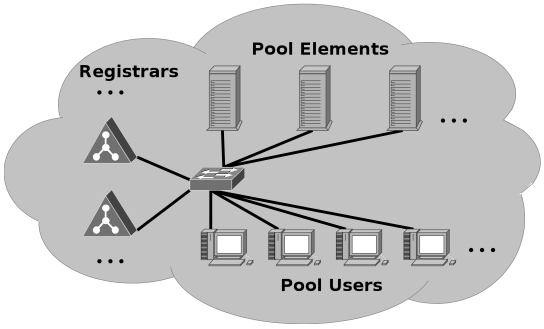

Figure 5. The Simulation Setup

Using the definitions above, the system utilization can be delineated as:

$$\mathrm{systemUtilization} = \mathrm{puToPERatio} * \frac{\frac{\mathrm{requestSize}}{\mathrm{requestInterval}}}{\mathrm{peCapacity}} \quad (2)$$

Obviously, the provider-side performance metric is the system utilization, since only utilized servers gain revenue. In practise, a well-designed client/server system is dimensioned for a certain *target system utilization*, e.g. 80%. That is, by setting any two of the parameters (PU:PE ratio, request size and request interval), the value of the third one can be calculated using equation 2. See also [7,13] for more details on this subject.

4 Setup Simulation Model

For the performance analysis, the RSerPool simulation model RSPSIM [7, 13] has been used. This model is based on the OMNeT++ [35] simulation environment and contains the protocols ASAP [31] and ENRP [36], a PR module and PE as well as PU modules for the request handling scenario defined in section 3. Network latency is introduced by link delays only. Therefore, only the network delay is significant. The latency of the pool management by PRs is negligible [12, 16].

Unless otherwise specified, the basic simulation setup – which is also presented in figure 5 – uses the following parameter settings:

- The target system utilization is 80%.
- Request size and request interval are randomized using a negative exponential distribution (in order to provide a generic and application-independent analysis).
- There are 10 PEs; each providing a capacity of 10^6 calculations/s (i.e. we use a homogeneous capacity distribution).
- No network latency is used (we will examine the impact of delay in subsection 5.4).

- We use a single PR only, since we do not examine failure scenarios here (see [13] for the impact of multiple PRs).
- The simulated real-time is more than 60 minutes; each simulation run is repeated at least 24 times with a different seed in order to achieve statistical accuracy.

GNU R has been used for the statistical post-processing of our results – including the computation of 95% confidence intervals – and plotting. Each resulting plot shows the average values and their corresponding confidence intervals.

5 Performance Analysis

In [13], we have already evaluated the load balancing capabilities of the LU, RR and RAND policies in situations where a PE accepts as many requests as offered. The interesting question arisen from the previous research results has been whether it is possible to improve the system performance by limiting the number of requests a PE is able to accept. The goal of the following simulations is to find out in which situation such a limit can be beneficial.

5.1 A Proof of Concept

In order to provide a first proof of concept for the usefulness of a requests limit, we have varied $\mathrm{MaxRequests}\ x$ for the PU:PE ratios r=1 to r=10 and the request size:PE capacity ratio s=5. The performance results are presented in figure 6; the left-hand part shows the system utilization, the middle one the handling speed and the right-hand one the number of handle resolutions per request (we will also use this three-plots structure for the following figures).

For the system utilization, there is a clear impact of the $\mathrm{MaxRequests}$ setting for the PU:PE ratio of r=1. As already shown in [13, 15], a small ratio r is critical: at r=1, each PU expects to get a PE exclusively. Therefore, the performance of RR and RAND is lower than for LU – since there is no knowledge about PE load states. This leads to a low performance if $\mathrm{MaxRequests}$ is too high: some PEs may remain idle while others have to perform multiple requests in parallel. That is, using a low setting of $\mathrm{MaxRequests}$ leads to a better distribution of the requests, which in turn also results in a significantly improved handling speed.

Note, that even the handling speed of LU is improved for $\mathrm{MaxRequests}\ x$=1 and $r > 1$: x=1 enforces requests to be handled exclusively – even if there are more PUs in startup state than PEs available. However, while the system performance is clearly improved by a low setting of $\mathrm{MaxRequests}$, the number of handle resolutions per request also increases: for each rejected request, a new trial is necessary. But before we will address this handle resolution overhead later in subsection 5.3, it is necessary to observe and explain the impact of the workload parameters in more detail.

5.2 Changing Workloads

5.2.1 PU:PE Ratio

The PU:PE ratio r is the most critical workload parameter; figure 7 shows the performance results for its variation (the request size:PE capacity ratio s has been 5 again). Clearly, the higher r, the better utilization and handling speed – as expected from the results in [13]. Furthermore, a smaller setting of $\mathrm{MaxRequests}\ x$ leads to an improved performance: inappropriate choices of PEs are avoided, even for the LU policy. However, x=1 and r>1 also leads to a significantly increased request rejection rate – and therefore much more overhead for querying the PR. The reason for the overhead increasing with r is that there are frequently more requests in processing/startup phase than the number of PEs. That is: for r>1, a setting of x>1 is recommended to keep the overhead small – which results in configuring a trade-off between overhead reduction and performance improvement.

5.2.2 Request Size

The results of varying the request size:PE capacity ratio s (for r=1, since this is the most critical setting) are presented in figure 8. As expected from [13], the utilization for all policies slightly decreases with s, while the handling speed slightly increases (due to reduced queuing of longer requests). Comparing with $\mathrm{MaxRequests}\ x$=3, it can clearly be observed that the performance is significantly improved by x=1 – as already expected from the previous results. The rejection rate (and therefore the handle resolution overhead) keeps quite constant for different settings of s: for load distribution, it does not matter how long the request is. In general, only the degree of parallelism – i.e. the number of PUs – is relevant, as shown in subsubsection 5.2.1.

The most interesting part of the request size variation analysis is the handling speed result for a small setting of s. Here (i.e. from s=1 to s=5), a significant difference among the three policies can be found: while LU still achieves a high speed, the performance of RR and especially for RAND significantly decays at x=3. The reason for this behaviour is the load distribution quality of the policies: at x=1, RAND has the highest chance to get its request rejected, while RR is somewhat better and the best result is achieved by LU (due to knowledge of the current PE load states, see also [13]). For each rejection, there is the average delay of 100ms before retrying. In combination with a small request size (and therefore a short processing time), the proportion of the startup delay gains an increasing importance in the overall request handling time (see equation 1). However, for larger requests (which do not influence the rejection rate) the delay penalty fraction of the request handling time becomes negligible.

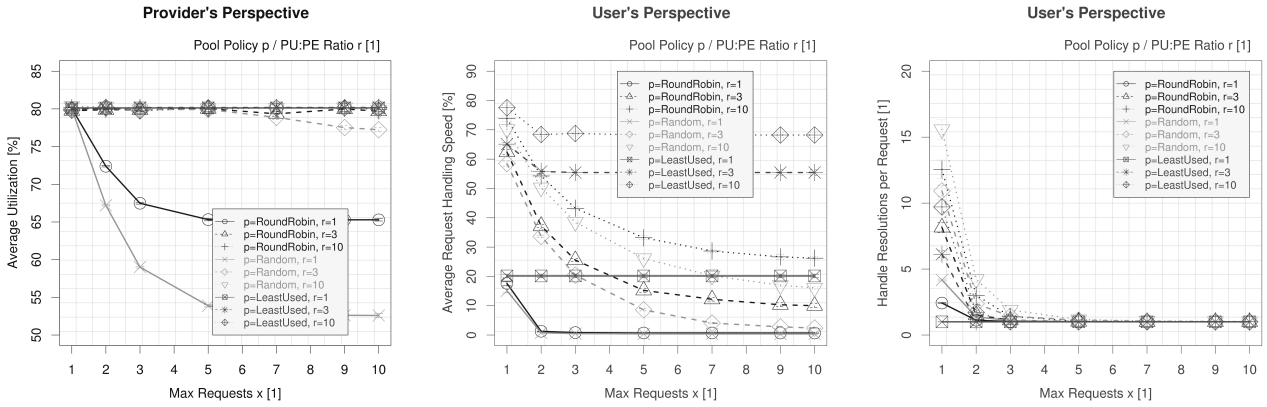

Figure 6. Limiting the Number of Requests

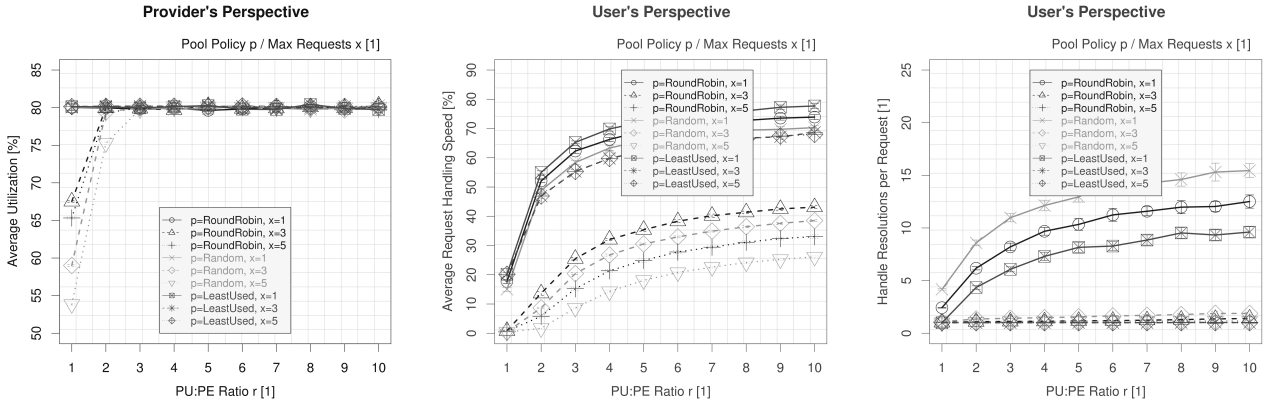

Figure 7. Changing the PU:PE Ratio

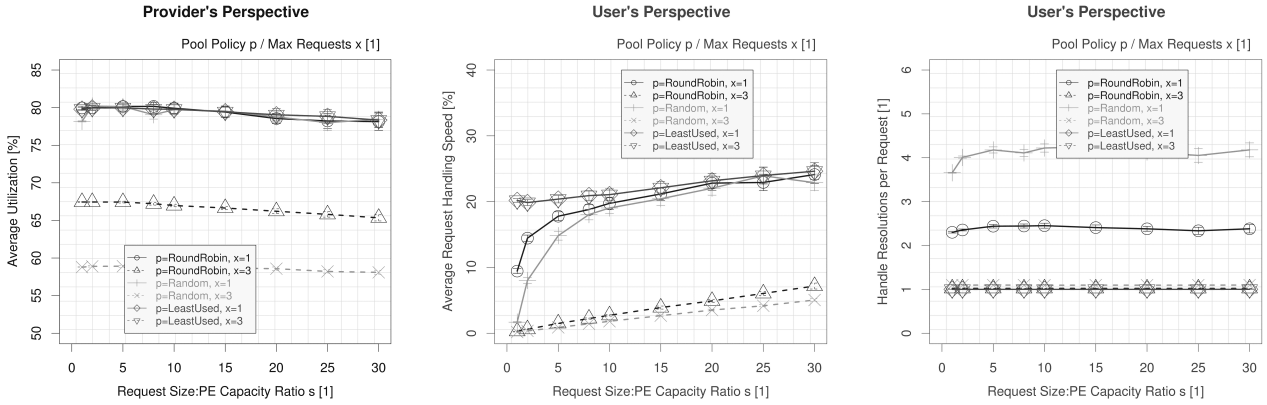

Figure 8. Changing the Request Size

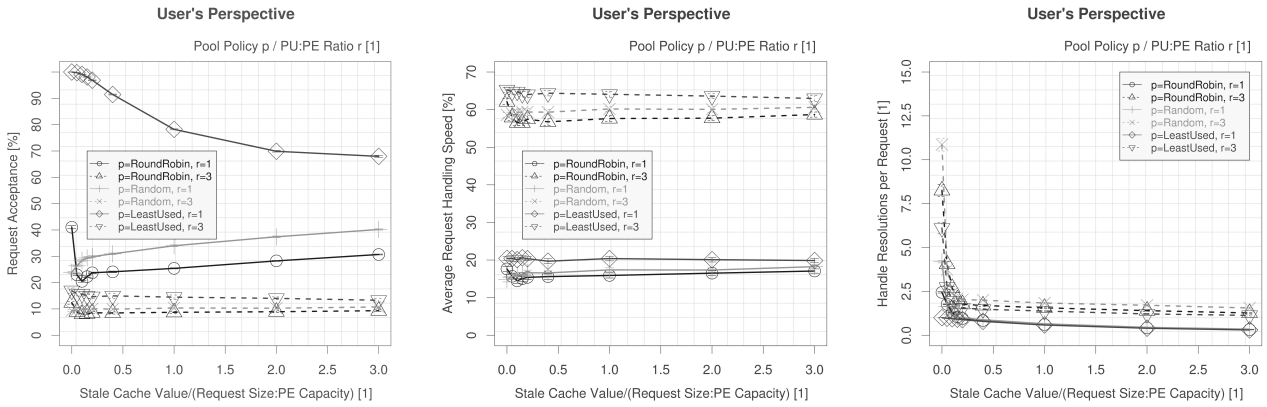

Figure 9. Utilizing the PU-Side Cache

5.3 Overhead Reduction by Caching

In order to reduce the handle resolution overhead, it is possible to utilize the PU-side cache. A new PE may directly be selected from the cache instead of querying a PR first, according to the setting of the stale cache value c. The results of varying c (given as ratio between the actual stale cache value and the request size:PE capacity ratio) for s=5 (i.e. not to critical for RR and RAND; see subsubsection 5.2.2) and $\mathrm{MaxRequests}$ x=1 is presented in figure 9. Since the utilization is always at 80% for all policies and settings of r, we have omitted a corresponding plot and instead show the request acceptance rate on the left-hand side.

As already expected from the results in subsubsection 5.2.1, the handling speed for a PU:PE ratio r=3 is clearly better than for r=1. However, due to $\mathrm{MaxRequests}$ x=1, the acceptance rate of requests is significantly lower, leading to a much higher number of handle resolutions per request if the cache is turned off (i.e. c=0).

Obviously, using the PU-side cache (i.e. $c > 0$) leads to a significant reduction of the number of handle resolutions. However, the cache utilization slightly affects the system performance: For LU, the acceptance rate decreases: the higher c, the older the load state information in the cache. That is, the selection decision may be based on already out-of-date PE information. In this case, an additional trial may become necessary, leading to the 100ms rejection penalty and therefore implying a decreased handling speed (although small, since the penalty compared with the request size is small for s=5; see also subsubsection 5.2.2).

While RR achieves a better performance than RAND for c=0 (i.e. without cache), the situation changes when the cache is utilized. Without cache, only the PR performs RR selections and the overall view of the selections is Round Robin. However, each PU-side cache constitutes an own selection instance which independently performs its own Round Robin selections. In this case, the global selection view will not be "in turn" any more. Even worse, it will not just degrade to random selection, but perform a systematic selection of possibly already loaded PEs. For example at r=1, there will be 11 selection instances: the PR and 10 PEs. This leads to a systematic selection of loaded PEs and therefore to a reduced handling speed: if the PR selects $\{$PE #1, PE #2, PE #3$\}$ for PU φ, a subsequent selection for PU ψ will contain $\{$PE #2, PE #3, PE #4$\}$ (the step size for RR should always be 1, for the reasons described in [19]). That is, after the request for PU φ has been completed on PE #1, it will use PE #2 (selected from its cache) – which may still be in use by PU φ!

As an upshot of this section, the cache is beneficial for reducing the handle resolution in case of request rejections; in particular when using the RAND policy. But while it only has a small impact on LU, the performance of RR degrades below the level of RAND. Therefore, the cache acts counterproductively to RR.

5.4 The Impact of Network Delay

Although the network latency for RSerPool systems is negligible in many cases (e.g. if all components are situated on the same premises), there are some scenarios where components are distributed globally [17]. It is therefore also necessary to consider the impact of network delay on the system performance. Clearly, network latency only becomes significant for small request size:PE capacity ratios s. Therefore, figure 10 presents the performance results of varying the delay for s=1 and a PU:PE ratio r=3 (since a combination of r=1, s=1 is already too critical for RR and RAND; see subsubsection 5.2.2 and also [13]).

While there is no impact on the utilization as long as the delay is not too high, the latency clearly reduces the handling speed. Furthermore, the higher the network latency d, the more costly are request rejections and distribution retries: the time for querying the PR and contacting the chosen PE is increased. This time is added to the already existing average retry delay of 100ms. As a result, the handling speed for $\mathrm{MaxRequests}$ x=1 falls short of $x \geq 3$ at a certain point – already at $d \geq 37.5$ for LU, at $d \geq 50$ for RAND

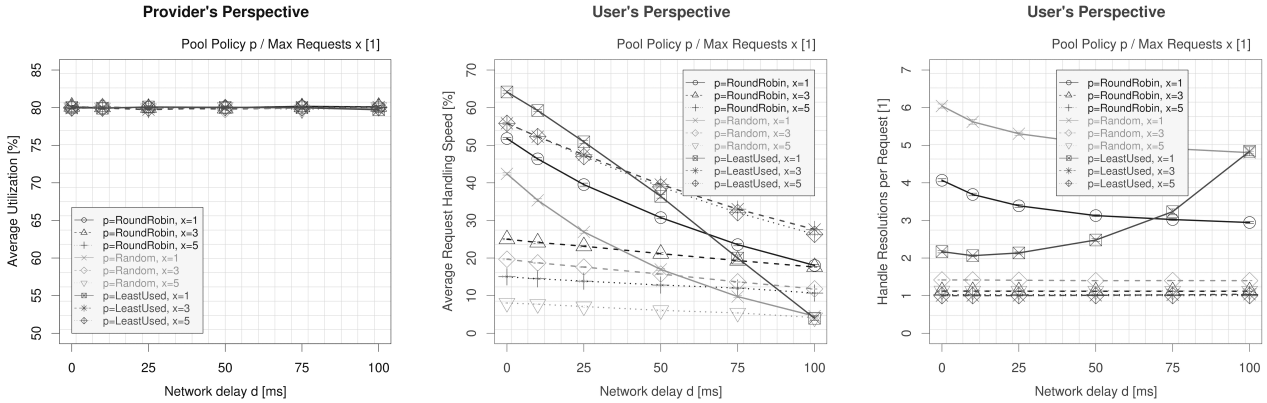

Figure 10. The Impact of Network Delay

and at $d \geq 100$ for RR. From these points, the penalty on rejected requests exceeds the performance benefit of the improved load balancing.

Since the load balancing quality of LU is superior in comparison with RR and RAND, the handling speed benefit of a requests limitation is only small: e.g. 59% at x=1 vs. 53% at x=3 for d=10ms. On the other hand, the speed for RR and RAND is significantly improved (due to their less optimal load balancing) – for a broad range of network delays.

As a result, using a small setting of $\mathrm{MaxRequests}\ x$ for RR and RAND is also useful if there is a significant network delay. But further improving the performance of LU is only possible if the latency is small. Otherwise, the request limitation may act counterproductively.

6 Conclusions

In summary, this paper has indicated that it is possible to improve the request handling performance of RSerPool systems by setting the per-PE limit $\mathrm{MaxRequests}$ for the maximum number of simultaneously handled requests – in particular for a low (i.e. critical) setting of the PU:PE ratio. The configuration of $\mathrm{MaxRequests}$ is a trade-off between performance improvement and handle resolution overhead. In order to reduce the overhead, the PU-side cache can be utilized. However, the usage of the cache also decreases the request acceptance rate for LU and leads to a performance worse than RAND for the RR policy. That is, the cache is mainly useful for the RAND policy only. We have also shown that our approach is useful if there is a significant network delay – in particular for RR and RAND.

As part of our ongoing research, we have also realized and evaluated our approach in scenarios of heterogeneous server capacities [38]. Furthermore, we are currently also validating our simulative performance results in real-life scenarios by using our RSerPool prototype implementation RSPLIB [5, 7, 17, 20] in the PLANETLAB; first results can be found in [7, 17].

References

[1] 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.

[2] P. Conrad, A. Jungmaier, C. Ross, W.-C. Sim, and M. Tüxen. Reliable IP Telephony Applications with SIP using RSerPool. In *Proceedings of the State Coverage Initiatives, Mobile/Wireless Computing and Communication Systems II*, volume X, Orlando, Florida/U.S.A., July 2002. ISBN 980-07-8150-1.

[3] T. Dreibholz. An Efficient Approach for State Sharing in Server Pools. In *Proceedings of the 27th IEEE Local Computer Networks Conference (LCN)*, pages 348–352, Tampa, Florida/U.S.A., Oct. 2002. ISBN 0-7695-1591-6.

[4] T. Dreibholz. Policy Management in the Reliable Server Pooling Architecture. In *Proceedings of the Multi-Service Networks Conference (MSN, Cosener's)*, Abingdon, Oxfordshire/United Kingdom, July 2004.

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

[6] T. Dreibholz. Applicability of Reliable Server Pooling for Real-Time Distributed Computing. Internet-Draft Version 03, IETF, Individual Submission, June 2007. draft-dreibholz-rserpool-applic-distcomp-03.txt, work in progress.

[7] T. Dreibholz. *Reliable Server Pooling – Evaluation, Optimization and Extension of a Novel IETF Architecture*. PhD thesis, University of Duisburg-Essen, Faculty of Economics, Institute for Computer Science and Business Information Systems, Mar. 2007.

[8] T. Dreibholz, L. Coene, and P. Conrad. Reliable Server Pooling Applicability for IP Flow Information Exchange. Internet-Draft Version 04, IETF, Individual Submission, June 2007. draft-coene-rserpool-applic-ipfix-04.txt, work in progress.

[9] 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 (LCN)*, pages 99–108, Königswinter/Germany, Nov. 2003. ISBN 0-7695-2037-5.

[10] T. Dreibholz and J. Pulinthanath. Applicability of Reliable Server Pooling for SCTP-Based Endpoint Mobility. Internet-Draft Version 02, IETF, Individual Submission, June 2007. draft-dreibholz-rserpool-applic-mobility-02.txt, work in progress.

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

[12] T. Dreibholz and E. P. Rathgeb. Implementing the Reliable Server Pooling Framework. In *Proceedings of the 8th IEEE International Conference on Telecommunications (ConTEL)*, volume 1, pages 21–28, Zagreb/Croatia, June 2005. ISBN 953-184-081-4.

[13] T. Dreibholz and E. P. Rathgeb. On the Performance of Reliable Server Pooling Systems. In *Proceedings of the IEEE Conference on Local Computer Networks (LCN) 30th Anniversary*, pages 200–208, Sydney/Australia, Nov. 2005. ISBN 0-7695-2421-4.

[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*, pages 396–403, Porto/Portugal, Aug. 2005. ISBN 0-7695-2431-1.

[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. ISBN 0-7803-9312-0.

[16] T. Dreibholz and E. P. Rathgeb. An Evalulation of the Pool Maintenance Overhead in Reliable Server Pooling Systems. In *Proceedings of the IEEE International Conference on Future Generation Communication and Networking (FGCN)*, Jeju Island/South Korea, Dec. 2007.

[17] T. Dreibholz and E. P. Rathgeb. On Improving the Performance of Reliable Server Pooling Systems for Distance-Sensitive Distributed Applications. In *Proceedings of the 15. ITG/GI Fachtagung Kommunikation in Verteilten Systemen (KiVS)*, Bern/Switzerland, Feb. 2007.

[18] T. Dreibholz and E. P. Rathgeb. Towards the Future Internet – A Survey of Challenges and Solutions in Research and Standardization. In *Proceedings of the Joint EuroFGI and ITG Workshop on Visions of Future Network Generations (EuroView)*, Würzburg/Germany, July 2007. Poster presentation.

[19] 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 (ICN)*, volume 2, pages 564–574, Saint Gilles Les Bains/Reunion Island, Apr. 2005. ISBN 3-540-25338-6.

[20] T. Dreibholz and M. Tüxen. High Availability using Reliable Server Pooling. In *Proceedings of the Linux Conference Australia (LCA)*, Perth/Australia, Jan. 2003.

[21] T. Dreibholz, X. Zhou, and E. P. Rathgeb. A Performance Evaluation of RSerPool Server Selection Policies in Varying Heterogeneous Capacity Scenarios. In *Proceedings of the 33rd IEEE EuroMirco Conference on Software Engineering and Advanced Applications*, pages 157–164, Lübeck/Germany, Aug. 2007. ISBN 0-7695-2977-1.

[22] I. Foster. What is the Grid? A Three Point Checklist. *GRID Today*, July 2002.

[23] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. *Grid Service Infrastructure WG, Global Grid Forum*, June 2002.

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

[25] A. Jungmaier. *Das Transportprotokoll SCTP*. PhD thesis, Universität Duisburg-Essen, Institut für Experimentelle Mathematik, Aug. 2005.

[26] A. Jungmaier, E. P. Rathgeb, and M. Tüxen. On the Use of SCTP in Failover-Scenarios. In *Proceedings of the State Coverage Initiatives, Mobile/Wireless Computing and Communication Systems II*, volume X, Orlando, Florida/U.S.A., July 2002. ISBN 980-07-8150-1.

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

[28] P. Lei, L. Ong, M. Tüxen, and T. Dreibholz. An Overview of Reliable Server Pooling Protocols. Internet-Draft Version 02, IETF, RSerPool Working Group, July 2007. draft-ietf-rserpool-overview-02.txt, work in progress.

[29] S. A. Siddiqui. Development, Implementation and Evaluation of Web-Server and Web-Proxy for RSerPool based Web-Server-Pool. Master's thesis, University of Duisburg-Essen, Institute for Experimental Mathematics, Nov. 2006.

[30] 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.

[31] R. Stewart, Q. Xie, M. Stillman, and M. Tüxen. Aggregate Server Access Protcol (ASAP). Internet-Draft Version 16, IETF, RSerPool Working Group, July 2007. draft-ietf-rserpool-asap-16.txt, work in progress.

[32] M. Tüxen and T. Dreibholz. Reliable Server Pooling Policies. Internet-Draft Version 05, IETF, RSerPool Working Group, July 2007. draft-ietf-rserpool-policies-05.txt, work in progress.

[33] E. Unurkhaan. *Secure End-to-End Transport - A new security extension for SCTP*. PhD thesis, University of Duisburg-Essen, Institute for Experimental Mathematics, July 2005.

[34] Ü. 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] A. Varga. *OMNeT++ Discrete Event Simulation System User Manual - Version 3.2*. Technical University of Budapest/Hungary, Mar. 2005.

[36] Q. Xie, R. Stewart, M. Stillman, M. Tüxen, and A. Silverton. Endpoint Handlespace Redundancy Protocol (ENRP). Internet-Draft Version 16, IETF, RSerPool Working Group, July 2007. draft-ietf-rserpool-enrp-16.txt, work in progress.

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

[38] X. Zhou, T. Dreibholz, and E. P. Rathgeb. Evaluation of a Simple Load Balancing Improvement for Reliable Server Pooling with Heterogeneous Server Pool. In *Proceedings of the IEEE International Conference on Future Generation Communication and Networking (FGCN)*, Jeju Island/South Korea, Dec. 2007.