A Performance Evaluation of RSerPool Server Selection Policies in Varying Heterogeneous Capacity Scenarios*

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

Abstract

Reliable Server Pooling (RSerPool) is a protocol framework for server redundancy and session failover, currently still under standardization by the IETF RSerPool WG. Server redundancy influences load distribution and load balancing, which both are important for the performance of RSerPool systems. Especially, a good load balancing strategy is crucial if the servers of a pool are heterogeneous. Some research on this subject has already been performed, but a detailed analysis on the question of how to make best use of additional capacity in dynamic pools is still open.

Therefore, the aim of this paper is, after an outline of the RSerPool framework, to simulatively examine the performance of RSerPool server selection strategies in scenarios of pools with varying server heterogeneity. In particular, this paper examines and evaluates a simple but very effective new policy, achieving a significant performance improvement in such situations.

Keywords: *RSerPool, Redundancy, Server Selection, Heterogeneous Pools, Performance Analysis*

1 Introduction

A RSerPool system provides various mechanisms to detect and handle component failures, so that it supports applications in providing a reliable service. RSerPool provides a complete Session Layer which maintains a logical session of a client with a pool and also supports the failover between servers. More details on this subject – which is out of this paper's scope – can be found in [4, 7, 12, 13]. Server redundancy directly influences load distribution and load balancing. While load distribution according to [1] only refers to the assignment of work to a processing element, load balancing refines this definition by requiring the assignment to maintain a certain application-specific balance across the processing elements (e.g. CPU load or memory usage). A classification of load distribution algorithms can be found in [19]; 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 [2, 18, 23].

There has already been some research on the performance of RSerPool for applications like SCTP-based mobility [9,10], VoIP with SIP [3], IP Flow Information Export (IPFIX) [8], real-time distributed computing [6,7,12,13,16] and battlefield networks [28]; [14] has already analysed the provider-side performance (utilization) of heterogeneous pools for normalized capacity distributions. But, a generic application-independent performance analysis and evaluation of how to take advantage of an increased pool capacity in heterogeneous pools – in particular also for the user side – is still missing.

Our goal is therefore to further set up an application-independent quantitative characterization for RSerPool systems, providing more insights into the implications of different heterogeneous pool capacity scenarios under varying server selection policies – from the perspective of both, the service provider and the users. We also identify critical configuration parameter ranges to provide a guideline for designing and configuring efficient RSerPool systems.

2 The RSerPool Architecture

Figure 1 provides an illustration of the RSerPool architecture as defined by the Internet Draft [27]. 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, which is the set of all pools. The handlespace is managed by *pool registrars* (PR). PRs of an *operation scope* synchronize their view of the han-

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

†Hainan University, College of Information Science and Technology, Renmin Road 58, 570228 Haikou, Hainan, China

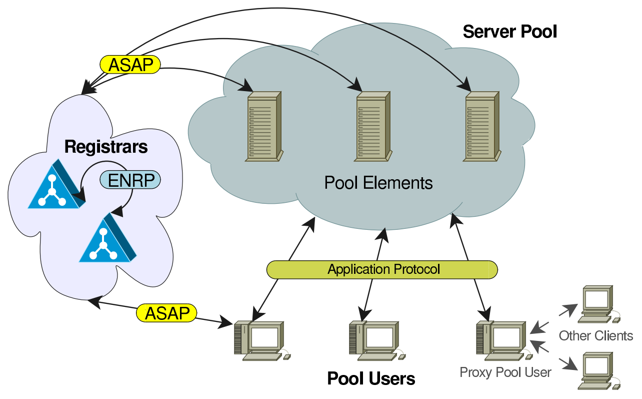

Figure 1. The RSerPool Architecture

dlespace using the Endpoint haNdlespace Redundancy Protocol (ENRP [30]), transported via SCTP [20–22, 24]. 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 globally, for their service to survive localized disasters [15].

PEs choose an arbitrary PR to register into a pool by using the Aggregate Server Access Protocol (ASAP [25]), 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 using ASAP Endpoint 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 Update messages.

A client is called *pool user* (PU) in RSerPool terminology. To access the service of a pool given by its PH, a PE has to be selected. This selection procedure – called *handle resolution* – is performed by an arbitrary PR of the operation scope. A PU can request a handle resolution from a PR using the ASAP protocol. The PR selects PE identities by using a pool-specific server selection rule, denoted as *pool policy*. A set of adaptive and non-adaptive pool policies is defined in [26]; for a detailed discussion of these policies, see [7, 11, 12, 14]. For this paper, the following pool policies are relevant:

- The adaptive Least Used (LU) policy selects the least-used PE, according to up-to-date load information. The definition of *load* is application-specific and could e.g. be the current number of users, bandwidth or CPU load.

- The Priority Least Used (PLU) policy also takes the load increment into account, besides up-to-date load information.

- The non-adaptive Round Robin (RR) policy realizes a selection in turn.

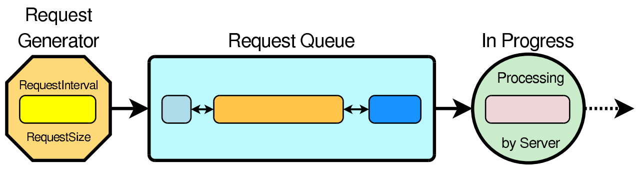

Figure 2. Request Handling by the Pool User

- The non-adaptive Weighted Random (WRAND) policy provides random selection, with a probability being proportional to a PE's weight constant. The Random (RAND) policy is a specialization of WRAND, having set all weights to 1.

For further information on RSerPool, see also [5, 7, 11–17].

3 Quantifying a RSerPool System

In order to evaluate the behaviour of an RSerPool system, it is necessary to quantify RSerPool systems first. The system parameters relevant to this paper can be divided into two groups: RSerPool system parameters and server capacity distributions. These two groups will be introduced in the following subsections.

3.1 System Parameters

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, bandwidth or memory usage. Each request consumes a certain number of calculations, we call this number *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 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 2.

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 $d_{\text{Processing}}$ (acceptance until finish) as illustrated in figure 3:

$$d_{\text{Handling}} = d_{\text{Queuing}} + d_{\text{Startup}} + d_{\text{Processing}}.$$

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.

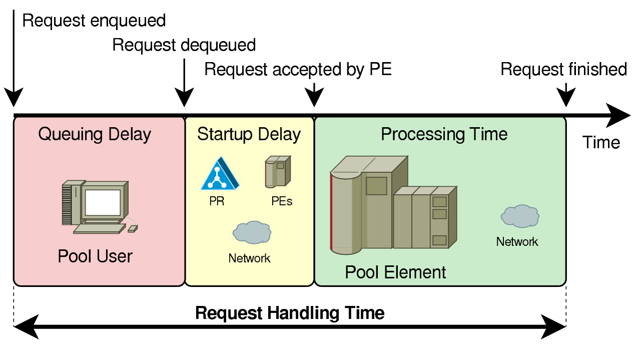

Figure 3. Request Handling Delays

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

$$\text{handlingSpeed} = \frac{\text{requestSize}}{d_{\text{handling}}}. \quad (1)$$

For convenience reasons, the handling speed can also be represented in % of the average PE capacity. Clearly, the user-side performance metric is the handling speed – which should be as high as possible.

Using the definitions above, it is now possible to delineate the system utilization in a formula:

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

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 interval and request size), the value of the third one can be calculated using equation 2. See also [7,12] for more details on this subject.

3.2 Server Capacity Distribution

In order to present the effects introduced by heterogeneous servers, we have considered three different and realistic capacity distributions: a single powerful server, multiple powerful servers and a linear capacity distribution.

3.2.1 A Single Powerful Server

A dedicated powerful server is realistic if there is only one powerful server to perform the main work and some other older (and slower) ones to provide redundancy. To quantify such a scenario, the variable φ (denoted as *capacity scale factor*) is defined as the capacity ratio between the new capacity ($\text{PoolCapacity}_{\text{New}}$) and the original capacity ($\text{PoolCapacity}_{\text{Original}}$) of the pool:

$$\varphi = \frac{\text{PoolCapacity}_{\text{New}}}{\text{PoolCapacity}_{\text{Original}}}. \quad (3)$$

A value of φ=1 denotes no capacity change, while φ=3 stands for a tripled capacity. In case of a single powerful server, the variation of φ results in changing the capacity of the designated PE only. That is, the capacity increment $\Delta_{\text{Pool}}(\varphi)$ of the whole pool can be calculated as follows:

$$\Delta_{\text{Pool}}(\varphi) = \underbrace{(\varphi * \text{PoolCapacity}_{\text{Original}})}_{\text{PoolCapacity}_{\text{New}}} - \text{PoolCapacity}_{\text{Original}}. \quad (4)$$

Then, the capacity of the i-th PE can be deduced using equation 4 by the following formula (where NumPEs denotes the number of PEs):

$$\text{Capacity}_i(\varphi) = \begin{cases} \frac{\text{PoolCapacity}_{\text{Original}}}{\text{NumPEs}} + \Delta_{\text{Pool}}(\varphi) & (i = 1) \\ \frac{\text{PoolCapacity}_{\text{Original}}}{\text{NumPEs}} & (i > 1) \end{cases}.$$

That is, $\text{Capacity}_1(\varphi)$ stands for the capacity of the powerful server.

3.2.2 Multiple Powerful Servers

If using $\text{NumPEs}_{\text{Fast}}$ more than one powerful servers instead of only one at one time, the capacity of the i-th PE can be calculated as follows (according to equation 4):

$$\Delta_{\text{FastPE}}(\varphi) = \frac{\Delta_{\text{Pool}}(\varphi)}{\text{NumPEs}_{\text{Fast}}},$$

$$\text{Capacity}_i(\varphi) = \begin{cases} \frac{\text{PoolCapacity}_{\text{Orig}}}{\text{NumPEs}} + \Delta_{\text{FastPE}}(\varphi) & (i \leq \text{NumPEs}_{\text{Fast}}) \\ \frac{\text{PoolCapacity}_{\text{Orig}}}{\text{NumPEs}} & (i > \text{NumPEs}_{\text{Fast}}) \end{cases}$$

3.2.3 A Linear Capacity Distribution

In real life, a linear capacity distribution is likely if there are different generations of servers. For example, a company could buy a state-of-the-art server every half year and add it to the existing pool. In this case, the PE capacities are distributed linearly. That is, the capacity of the first PE remains constant, the capacities of the following PEs are increased with a linear gradient, so that the pool reaches its desired capacity $\text{PoolCapacity}_{\text{New}}$. Therefore, the capacity of the i-th PE can be obtained using the following equations (again, using $\Delta_{\text{Pool}}(\varphi)$ as defined in equation 4):

$$\begin{aligned} \Delta_{\text{FastestPE}}(\varphi) &= \frac{2 * \Delta_{\text{Pool}}(\varphi)}{\text{NumPEs}}, \\ \text{Capacity}_i(\varphi) &= \underbrace{\underbrace{\frac{\Delta_{\text{FastestPE}}(\varphi)}{\text{NumPEs} - 1}}_{\text{Capacity Gradient}} * (i-1)}_{\text{Additional Capacity for PE } i} + \frac{\text{PoolCapacity}_{\text{Original}}}{\text{NumPEs}}. \end{aligned}$$

4 Setup Simulation Model

For the performance analysis, our RSerPool simulation model RSPSIM [7, 12] has been used. It is based on the

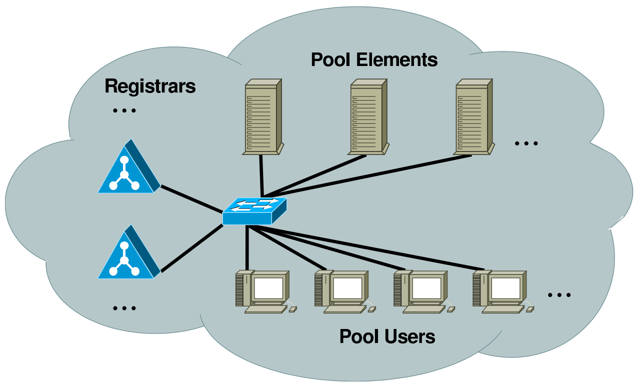

Figure 4. The Simulation Setup

OMNeT++ [29] simulation environment and contains the protocols ASAP [25] and ENRP [30], a PR module and PE and PU modules modelling the request handling scenario defined in subsection 3.1. Unless otherwise specified, the basic simulation setup, as illustrated in figure 4, uses the following configuration parameters:

- The target system utilization is 90%.
- Request size and request interval are randomized using a negative exponential distribution (in order to provide a generic, application-independent analysis).
- There are 10 PEs; the capacity of a PE for φ=1 is set to 10^6 calculations/s.
- The average request size:PE capacity ratio is 10, i.e. processing an average-sized request exclusively on an average PE takes 10s.
- We use a single PR only, since we do not examine failure scenarios here (see [12] for the impact of multiple PRs).
- The simulated real-time is 120m; each simulation run is repeated 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. All results plots show the average values and their confidence intervals.

5 Performance Analysis

An important feature of RSerPool is its support for dynamic pools: it is possible to adapt a pool's capacity to a changing demand by adding or removing PEs. An important question here is what happens with additional capacity if the pool is (temporarily) slightly loaded and removing some servers is not useful? Clearly, it is usually desirable that the additional capacity would result in an improvement of the request handling speed! But are the predefined policies capable to handle such a scenario? Some basic ideas have already been proposed by us in [16], but a more general evaluation has still been missing. Therefore, the goal of this paper is to provide an appropriate analysis.

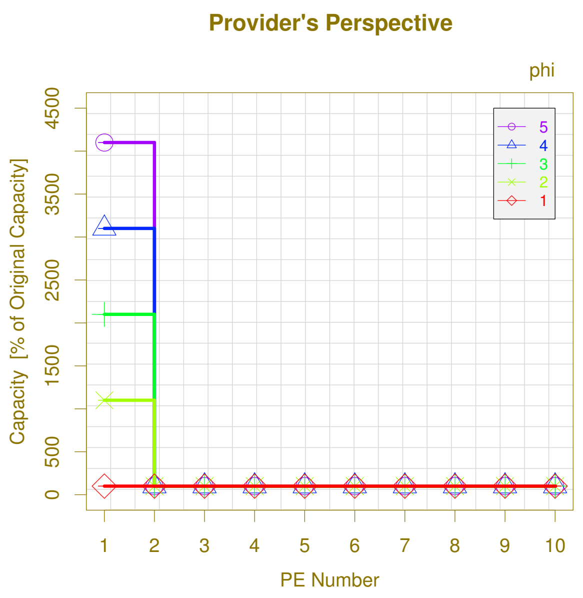

Figure 5. Server Capacities of the Single Powerful Server Scenario

In order to conveniently evaluate the performance in scenarios of varying heterogeneity, we have taken the three capacity distribution scenarios described in subsection 3.2 into account: a single powerful server, multiple powerful servers and a linear capacity distribution.

5.1 Results for a Single Powerful Server

In case of a single powerful server, changing φ results in varying the capacity of the designated PE only. Figure 5 illustrates the resulting PE capacities of each PE for varying φ, according to the equations of subsubsection 3.2.1.

Figure 6 presents the performance results (system utilization on the left-hand side, handling speed on the right-hand one) for varying φ and PU:PE ratios r=3 and r=10. Form the provider's performance perspective, the system utilization results are not surprising: the higher the capacity of the pool, the lower the utilization. Since the request workload remains constant and the policies are not used in critical parameter ranges (i.e. in particular, the PU:PE ratio r for the non-adaptive policies is sufficiently high, see also [12, 14]), no significant utilization differences among the used policies can be observed.

However, the user's performance perspective becomes interesting: as shown in figure 6, the policies RR and

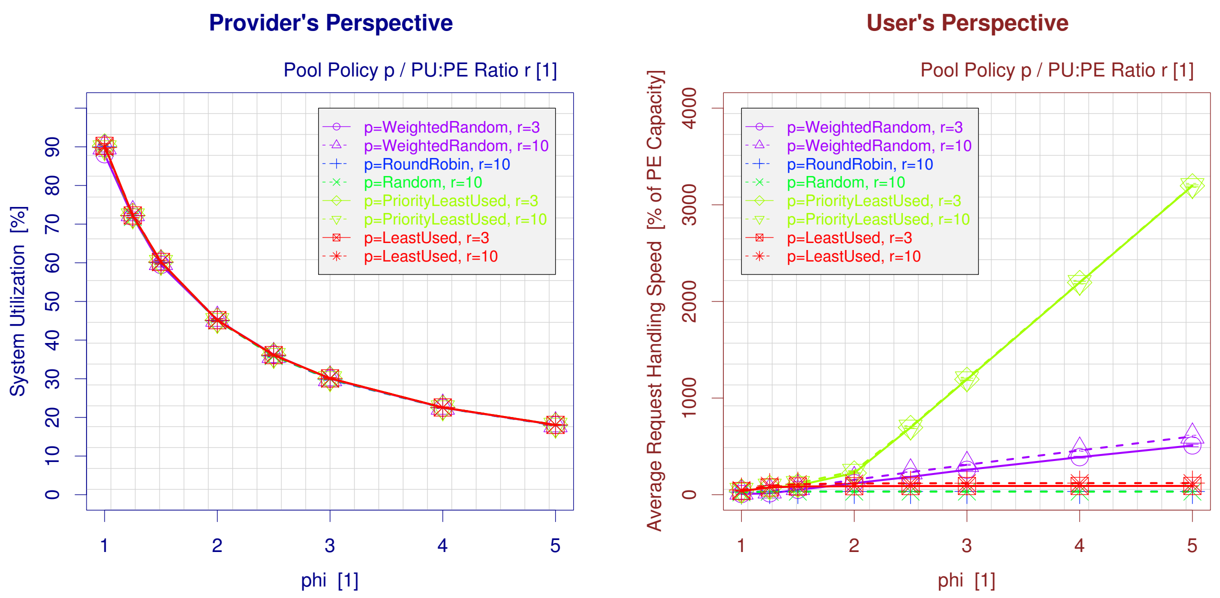

Figure 6. Increasing the Capacity of a Single Server

RAND only provide a minimal handling speed gain if the capacity of the designated server is increased. This is a result of these policies' lack of PE capacity knowledge. Therefore, these policies are only useful in homogeneous scenarios. The surprising result of this simulation is LU's bad performance: at φ=5 and for r=3 nad r=10, the handling speed is only 90% for LU, while it is already about 500% for WRAND policy. So, what is the problem of the LU policy in this scenario?

The reason for the bad performance of LU is the fact that the selection decision is based on the current PE load state only. Consider a powerful PE #1 loaded by 11% and a slow PE #2 loaded by 10%. Clearly, the LU policy would select PE #2, because it has the lowest load. But the new request may increase the slow PE #2's load by another 10%, while using PE #1 may only have increased its load by 2%. That is, PE #1 would have been able to handle the request more quickly. The lack of LU to incorporate the aspect of different load increments for different servers has led to the definition of the PLU policy: in [16], we propose that each PE can specify its load increment $\hat{l}$, i.e. the number of load units the server's load is increased by an additional request. Upon selection, the PE having the lowest sum of load and load increment is chosen[1]. In particular, while LU takes the PE *currently* having the lowest load, our PLU policy selects the PE having the lowest load *after* accepting a new request.

For the parametrization of PLU, it is only important that the settings of $\hat{l}$ reflect the relative PE capabilities. For this simulation, the load increment $\hat{l}_i$ for each PE i has been defined as:

$$\hat{l}_i = \frac{2.5 * 10^5}{\text{Capacity}_i(\varphi)}. \tag{5}$$

That is, we have defined that the load of a PE j of the average capacity (i.e. 10^6 calculations/s, see section 4) is rised by $\hat{l}_j$=25%. On the other hand, using a PE k with a capacity of $41{*}10^6$ calculations/s (i.e. the powerful PE's capacity for φ=5), the load increment would only be $\hat{l}_\text{k}$=0.61%.

Using the PLU policy with the described setting of the load increment, a significant handling speed gain can be observed (see figure 6): about 3,250% for φ=5 (r=3 and r=10) vs. 600% for WRAND (r=10) and about 160% for LU (r=10). That is, the PLU policy provides the desired functionality of increasing the request handling speed in the scenario of a single designated server. But what about the other scenarios of increased PE capacity?

5.2 Results for Multiple Powerful Servers

Another realistic capacity distribution scenario is to have multiple powerful servers instead of a single one. In order to provide a simulation for this kind of distribution, we have chosen a scenario of using 3 fast servers out of 9. All other configuration parameters have remained the same as described in section 4.

Figure 7 shows the simulation results for the scenario having used 9 PEs, where 3 PEs are fast servers. The capacity $\text{Capacity}_i(\varphi)$ of PE i is calculated according to subsubsection 3.2.2. Since the utilization results are quite similar

[1]It is important to note that the PLU policy, similar to plain LU, is also very efficiently implementable using the approach presented in [11].

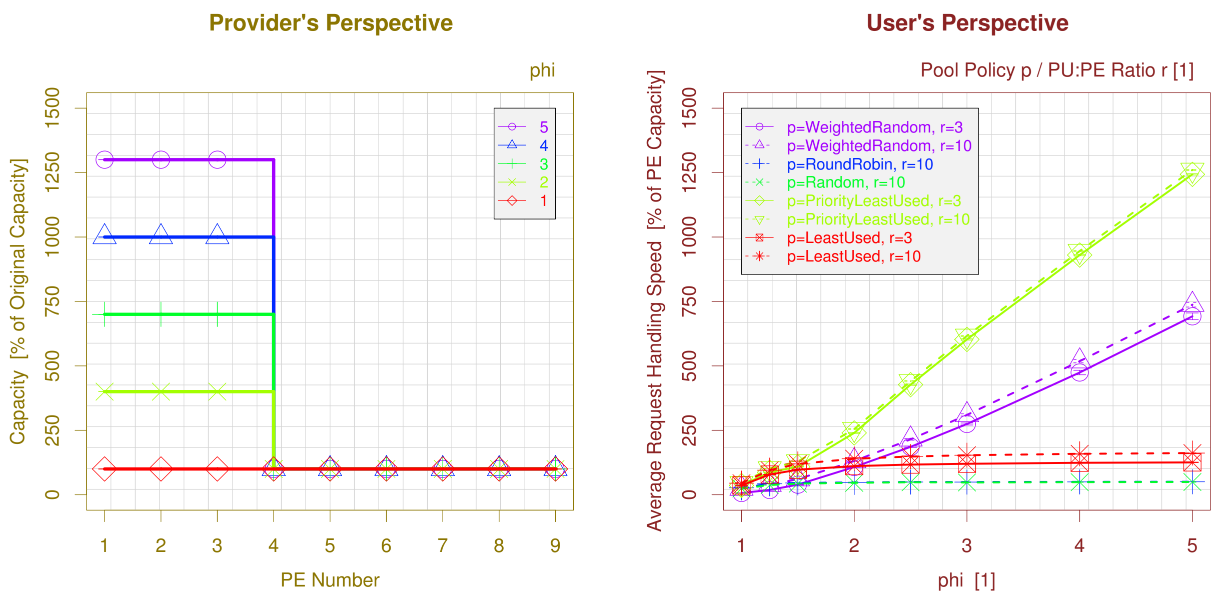

Figure 7. Increasing the Capacity of One Third of the Servers

to the results of subsection 5.1, we have omitted the corresponding plot here. Instead, the left-hand side of figure 7 presents the capacity of each server for varying settings of φ; the right-hand side shows the request handling speed.

As a general result of the curves, it can be observed that the behaviour of the policies is quite similar to the previous scenario presented in figure 6. That is, while RR and RAND are in fact useless, the performance of LU is significantly outperformed by WRAND (e.g. a handling speed of more than 700% vs. about 125% for LU for r=10). Again, a significant performance improvement is achievable by using PLU with the load increment $\hat{l}_i$ setting defined by equation 5.

Comparing the results with the observations for the "single powerful server" scenario of section 5.1, the effects of the changed capacity distribution can be observed: while the designated powerful PE incorporates almost the pool's complete capacity (e.g. 4,100% of a slow server's capacity for φ=5, see figure 5), the additional capacity is now divided up among three servers (i.e. 3 servers having 1,300% of a slow server's capacity for φ=5, see the left-hand side of figure 7). In this case, the top handling speed is lower (e.g. 1,250% vs. 3,250% for PLU at φ=5), since the maximum possible request handling speed is limited by the processing speed of a fast PE. For LU and also for RR and RAND, the fact that now one third of the servers are powerful ones becomes beneficial: their handling speed is increased, since the probability of mapping a request to a powerful PE is increased significantly (e.g. 160% vs. 125% for LU at φ=5 and r=10).

After it has been shown that an appropriate policy can make use of additional capacity to improve the handling speed in scenarios containing a set of powerful PEs, it is furthermore necessary to have a look at a scenario containing a less extreme capacity distribution.

5.3 Results for a Linear Capacity Distribution

In the final capacity distribution scenario, we have increased the PE capacities linearly. That is, while the capacity of the first PE remains constant, the capacities of the following PEs are increased with a linear gradient. Figure 8 shows the simulation results for the linear capacity distribution. All other configuration parameters have remained as described in section 4. Again, the left-hand side shows the capacity for each of the 10 PEs for having varied φ; the right-hand side presents handling speed results. A system utilization plot has been omitted again, since it would not provide any new insights.

While the general ranking behaviour of the policies remains as observed for the two scenarios of fast servers (see section 5.1 and section 5.2), it is clearly visible that a linear capacity distribution results in smaller performance differences among the policies: for φ=5, the capacity of the fastest PE is only 900% of the slowest PE's one (see the left-hand side of Figure 8), while it is 1,300% in the three-fast-servers scenario and even 4,100% for the dedicated powerful server (see the left-hand side of figure 7 and figure 5). That is, while the top request handling speed for requests is significantly lower (e.g. only about 800% for PLU and r=10), the chance that a less-performing policy can reach a high handling speed is significantly improved. At φ=5 and r=10, the RAND policy already achieves a handling

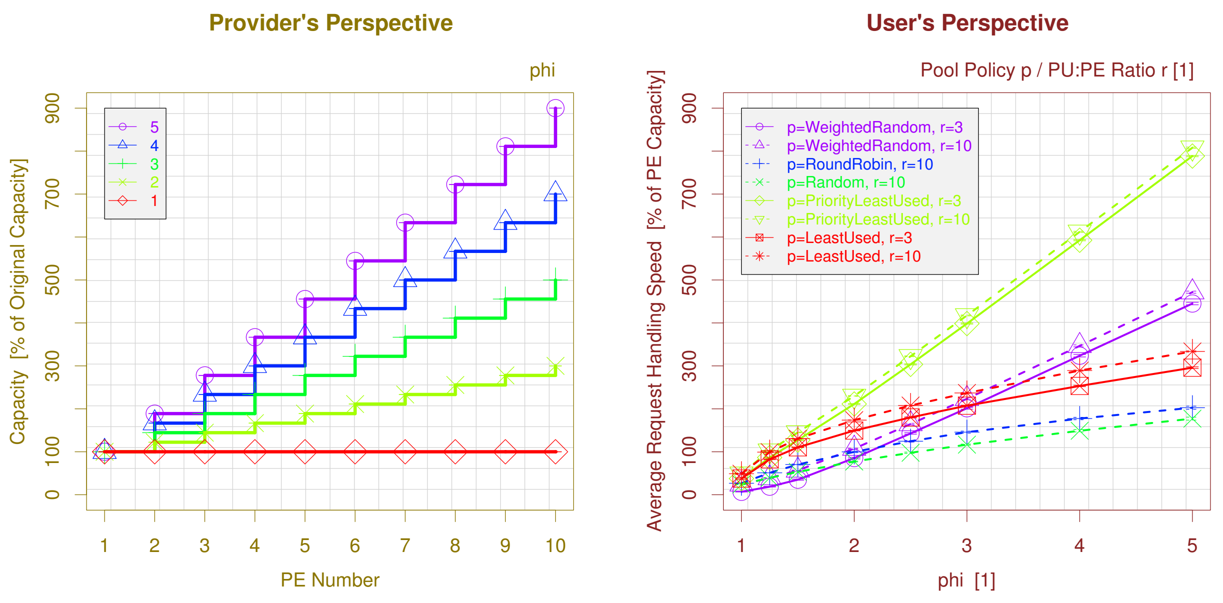

Figure 8. Increasing the Server Capacities Linearly

speed of about 175%, while RR even exceeds 200%. LU is able to reach about 330%, but is still being outperformed by WRAND with about 450%. Note, that the handling speed of WRAND only outperforms LU for $\varphi > 3$ – for smaller values of φ, its performance is significantly lower. Compared to one dedicated server (LU is already outperformed at φ=2), the performance of LU for a linear capacity distribution is significantly better.

5.4 Results Summary

As the main result, it has been observed that the linear scenario is significantly less critical in comparison to the fast servers scenarios – even inappropriate policies like RR and RAND are able to make use of the improved capacity. While it is still possible to map a request to a slower PE, the probability to map the next request to a faster one is the same (due to the linear capacity distribution). Nevertheless, the WRAND policy (in the class of non-adaptive policies) and in particular the new adaptive PLU policy provide a superior performance.

6 Conclusions

In summary, we have answered an important question on the performance of heterogeneous RSerPool systems: Which selection policy is able to make best use of additional PE capacity? The answer to this question is crucial for the cost-benefit ratio of such systems, since additional capacity should provide a performance improvement. As main results, we have shown that the non-adaptive WRAND policy provides good results for sufficiently heterogeneous pools. However, the results for plain LU in this case are underperforming. The reason for this low performance is the lack of PE capacity knowledge for this policy. This problem has been overcome by the PLU policy. We have shown that this new policy – although very simple and efficiently realizable – provides superior performance results in different capacity distribution scenarios.

As part of our future research, we are going to evaluate the policies in real-life scenarios, using our RSerPool prototype implementation RSPLIB [5, 7, 15, 17] in the PLANETLAB. In particular, we will also analyse the effects of network delay on the handlespace synchronization (which leads to the deprecation of policy information like the load states of LU/PLU) and evaluate the resulting system performance impacts, in order to find approaches for further performance improvement.

References

[1] E. Berger and J. C. Browne. Scalable Load Distribution and Load Balancing for Dynamic Parallel Programs. In *Proceedings of the International Workshop on Cluster-Based Computing 99*, Rhodes/Greece, June 1999.

[2] 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.

[3] 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 2002, Mobile/Wireless Computing and Communication Systems II*, volume X, Orlando, Florida/U.S.A., July 2002. ISBN 980-07-8150-1.

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

[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 02, IETF, Individual Submission, Aug. 2006. draft-dreibholz-rserpool-applic-distcomp-02.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 pool use in IP flow information exchange. Internet-Draft Version 02, IETF, Individual Submission, Feb. 2006. draft-coene-rserpool-applic-ipfix-02.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*, 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 01, IETF, Individual Submission, Sept. 2006. draft-dreibholz-rserpool-applic-mobility-01.txt, work in progress.

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

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

[13] 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.

[14] 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.

[15] 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*, Bern/Switzerland, Feb. 2007.

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

[17] T. Dreibholz and M. Tüxen. High Availability using Reliable Server Pooling. In *Proceedings of the Linux Conference Australia*, 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 *Proceedings of the IEEE Infocom 2000*, volume 3, pages 1361–1370, Tel Aviv/Israel, Mar. 2000. ISBN 0-7803-5880-5.

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

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

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

[22] 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*, pages 141–148, Heidelberg/Germany, June 2000.

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

[24] 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.

[25] R. Stewart, Q. Xie, M. Stillman, and M. Tüxen. Aggregate Server Access Protcol (ASAP). Technical Report Version 15, IETF, RSerPool Working Group, Jan. 2007. draft-ietf-rserpool-asap-15.txt, work in progress.

[26] M. Tüxen and T. Dreibholz. Reliable Server Pooling Policies. Internet-Draft Version 04, IETF, RSerPool Working Group, Mar. 2007. draft-ietf-rserpool-policies-04.txt, work in progress.

[27] M. Tüxen, Q. Xie, R. Stewart, M. Shore, J. Loughney, and A. Silverton. Architecture for Reliable Server Pooling. Technical Report Version 12, IETF, RSerPool Working Group, Nov. 2006. draft-ietf-rserpool-arch-12.txt, work in progress.

[28] 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.

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

[30] Q. Xie, R. Stewart, M. Stillman, M. Tüxen, and A. Silverton. Endpoint Handlespace Redundancy Protocol (ENRP). Internet-Draft Version 15, IETF, RSerPool Working Group, Jan. 2007. draft-ietf-rserpool-enrp-15.txt, work in progress.