Evaluation of a Simple Load Balancing Improvement for Reliable Server Pooling with Heterogeneous Server Pools*

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

The IETF is currently standardizing a light-weight protocol framework for server redundancy and session failover: Reliable Server Pooling (RSerPool). It is the novel combination of ideas from different research areas into a single, resource-efficient and unified architecture. Server redundancy directly leads to the issues of load distribution and load balancing. Both are important and have to be considered for the performance of RSerPool systems. While there has already been some research on the server selection policies of RSerPool, an interesting question is still open: Is it possible to further improve the load balancing performance of the standard policies without modifying the policies – which are well-known and widely supported – themselves? Our approach places its focus on the session layer rather than the policies and simply lets servers reject inappropriately scheduled requests. Applying failover handling mechanisms of RSerPool, in this case, could choose a more appropriate server instead. In [26], we have already shown that our approach is useful for homogeneous server pools. But is it also useful for heterogeneous pools?

In this paper, we first present a short outline of the RSerPool framework. Afterwards, we analyse and evaluate the performance of our new approach for different server capacity distributions. Especially, we are also going to analyse the impact of RSerPool protocol and system parameters on the performance of the server selection functionalities as well as on the overhead.

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

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

1 Introduction and Scope

Service availability is getting increasingly important in today's Internet. But – in contrast to the telecommunications world, where availability is ensured by redundant links and devices [19] – there had not been any generic, standardized approaches for the availability of Internet-based services. Each application had to realize its own solution and therefore to re-invent the wheel. This deficiency – once more arisen for the availability of SS7 (Signalling System No. 7 [16]) services over IP networks – had been the initial motivation for the IETF RSerPool WG to define the Reliable Server Pooling (RSerPool) framework. The basic ideas of RSerPool are not entirely new (see [1,21]), but their combination into one application-independent framework is.

The Reliable Server Pooling (RSerPool) architecture [5, 11,18] which is currently under standardization by the IETF RSerPool WG is an overlay network framework to provide server replication [3] and session failover capabilities [2] to its applications. Server redundancy leads to load distribution and load balancing [15], which are also covered by RSerPool [7, 13]. But in strong contrast to already available solutions in the area of GRID and high-performance computing [14], the fundamental property of RSerPool is to be "lightweight", i.e. it must be usable on devices providing only meagre 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 [6, 9]. Two classes of load distribution algorithms [15] are supported by RSerPool: non-adaptive and adaptive algorithms. Adaptive strategies base their assignment decisions on the current status of the processing elements (which requires up-to-date information); non-adaptive algorithms do not need such data.

There has already been some research on the perfor-

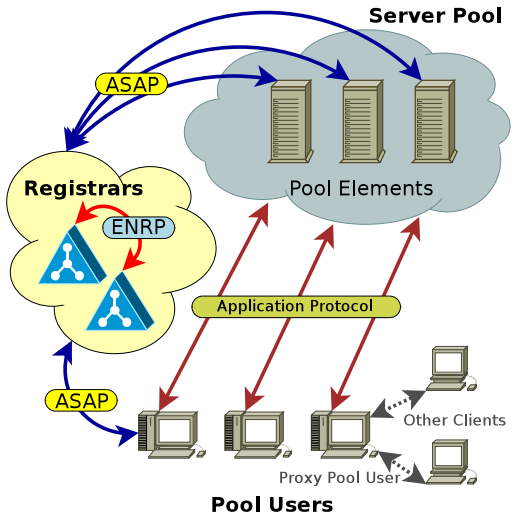

Figure 1. The RSerPool Architecture

mance of RSerPool usage for applications like SCTP-based endpoint mobility [4], VoIP with SIP, web server pools, IP Flow Information Export (IPFIX), real-time distributed computing and battlefield networks (see [3] for detailed application scenario descriptions). A generic application model for RSerPool systems has been introduced by [7], which includes performance metrics for the provider side (pool utilization) and user side (request handling speed). Based on this model, the load balancing quality of different pool policies has been evaluated [3, 7, 13]. The question arisen from these results is whether it is possible to improve the load balancing performance of the standard policies by allowing servers to reject requests, especially for scenarios with heterogeneous server capacities. In particular, it is intended to leave the policies themselves unchanged: they are widely supported and their performance is well-known [7], so that applying a specialised non-standard policy to only improve the performance during a temporary capacity distribution change may be unsuitable. Therefore, we focus on the session layer: if a request gets rejected, the failover mechanisms provided by RSerPool could choose a possibly better server instead. For this reason, the goal of this paper is to evaluate the performance of this strategy, with respect to the resulting protocol overhead. We also identify critical configuration parameter ranges in order to provide a guideline for design and configuration of efficient RSerPool systems.

2 The RSerPool Protocol Framework

Figure 1 illustrates the RSerPool architecture [3, 11]. It contains three types of components: 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 [24]), transported via SCTP [17]. An operation scope has a limited range, e.g. an organization or only a building. In particular, it is restricted to a single administrative domain – in contrast to GRID computing [14] – in order to keep the management complexity [6,9] at a minimum. Nevertheless, PEs may be distributed globally for their service to survive localized disasters [10].

PEs choose an arbitrary PR of the operation scope to register into a pool by using the Aggregate Server Access Protocol (ASAP [20]), 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 [20]), transported via SCTP. 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 [22]; relevant for 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 application-specific load information. Round robin selection is applied among multiple least-loaded PEs [6]. Detailed discussions of pool policies can be found in [3, 7, 8, 12].

The PU writes the list of PE identities selected by the PR into its local cache (denoted as *PU-side cache*). From this 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 [7].

3 Quantifying a RSerPool System

The system parameters relevant for this paper can be divided into two groups: RSerPool system parameters and server capacity distributions.

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[1]. 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 provided by multitasking operating systems. The maximum number of simultaneously handled requests (MaxRequests) is limited by the parameter

[1] An application-specific view of capacity may be mapped to this definition, e.g. CPU cycles or memory usage.

$\mathrm{MinCapPerReq}$. This parameter defines the minimum capacity share which should be available to handle a new request. That is, a PE providing the capacity ($\mathrm{peCapacity}$) only allows at most

$$\mathrm{MaxRequests} = \mathrm{round}(\frac{\mathrm{peCapacity}}{\mathrm{MinCapPerReq}}) \qquad (1)$$

simultaneously handled requests. Note, that the limit is rounded to the nearest integer, in order to support arbitrary capacities. If a PE's limit is reached, a new request gets rejected. For example, the PE capacity is 10^6 calculations/s and $\mathrm{MinCapPerReq}$=$2.5 * 10^5$. Then, there is only room for $\mathrm{MaxRequests} = \mathrm{round}(\frac{10^6}{2.5*10^5}) = 4$ simultaneously processed requests. After the time $\mathrm{ReqRetryDelay}$, it is tried to find another PE for a rejected request. Such a delay is necessary to avoid request-rejection floods [25].

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.

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):

$$d_{\mathrm{Handling}} = d_{\mathrm{Queuing}} + d_{\mathrm{Startup}} + d_{\mathrm{Processing}}. \qquad (2)$$

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* is defined as: $\mathrm{handlingSpeed} = \frac{\mathrm{requestSize}}{d_{\mathrm{handling}}}$. For convenience reasons, the handling speed (in calculations/s) is represented in % of the average PE capacity. Clearly, the user-side performance metric is the handling speed – which should be as fast as possible.

Using the definitions above, it is possible to delineate the average system utilization (for a pool of NumPEs servers and a total pool capacity of $\mathrm{PoolCapacity}$) as:

$$\mathrm{systemUtilization} = \mathrm{NumPEs} * \mathrm{puToPERatio} * \frac{\frac{\mathrm{requestSize}}{\mathrm{requestInterval}}}{\mathrm{PoolCapacity}}. \qquad (3)$$

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 3 (see [3, 7] for details).

3.2 Server Capacity Distributions

In order to present the effects introduced by heterogeneous servers, we keep the total capacity of the pool constant and only vary the heterogeneity within the pool. Based on [3], we have defined three different normalized capacity distributions: a single powerful server, multiple powerful servers and a linear capacity distribution.

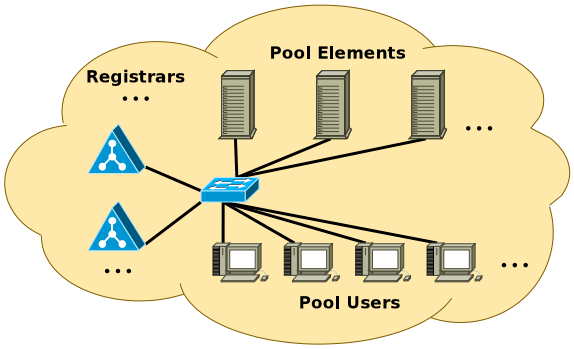

Figure 2. The Simulation Setup

In the case of one or more dedicated powerful servers, the parameter κ defines the capacity ratio between a fast PE ($\mathrm{Capacity}_{\mathrm{Fast}}$) and a slow PE ($\mathrm{Capacity}_{\mathrm{Slow}}$). That is, for κ=3, a fast PE has three times the capacity of a slow one. Using a fixed setting of $\mathrm{PoolCapacity}$ for the total capacity of a pool containing NumPEs servers, of which $\mathrm{NumPEs}_{\mathrm{Fast}}$ are fast ones, the PE capacities can be calculated as follows:

$$\mathrm{Capacity}_{\mathrm{SlowPE}}(\kappa) = \frac{\mathrm{PoolCapacity}}{\mathrm{NumPEs}_{\mathrm{Fast}} * (\kappa - 1) + \mathrm{NumPEs}}, \qquad (4)$$

$$\mathrm{Capacity}_{\mathrm{FastPE}}(\kappa) = \kappa * \mathrm{Capacity}_{\mathrm{SlowPE}}. \qquad (5)$$

For the linear distribution, the parameter γ specifies the capacity ratio between the fastest PE ($\mathrm{Capacity}_{\mathrm{Fastest}}$) and the slowest PE ($\mathrm{Capacity}_{\mathrm{Slowest}}$). That is, for γ=4 the fast PE has four times more capacity than the slowest one; the other PEs' capacities are linearly distributed between the limits $\mathrm{Capacity}_{\mathrm{Slowest}}$ and $\mathrm{Capacity}_{\mathrm{Fastest}}$. Therefore, given a fixed value of $\mathrm{PoolCapacity}$, $\mathrm{Capacity}_{\mathrm{SlowestPE}}$ can be calculated as follows:

$$\mathrm{Capacity}_{\mathrm{SlowestPE}}(\gamma) = \frac{\mathrm{PoolCapacity}}{\mathrm{NumPEs} * (\frac{\gamma-1}{2} + 1)}.$$

Finally, the capacity of the i-th PE is:

$$\mathrm{Capacity}_i(\gamma) = \underbrace{\frac{(\gamma - 1) * \mathrm{Capacity}_{\mathrm{SlowestPE}}}{\mathrm{NumPEs} - 1}}_{\text{Capacity Gradient}} * (i - 1) + \underbrace{\mathrm{Capacity}_{\mathrm{SlowestPE}}}_{\text{Base Capacity}}.$$

4 Setup Simulation Model

For our performance analysis, the RSerPool simulation model RSPSIM [3, 7] has been used. This model is based on the OMNeT++ [23] simulation environment and contains the protocols ASAP [20] and ENRP [24], 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 [6, 9].

Unless otherwise specified, the basic simulation setup – which is also presented in figure 2 – 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 [7]).
- There are 10 PEs; in the basic setup, each one provides a capacity of 10^6 calculations/s. The heterogeneity parameters κ (fast servers) and γ (linear) are set to 3 (we analyse variations in subsection 5.2).
- A PU:PE ratio of 3 is used (this parameter is analysed in subsection 5.1). The default request size:PE capacity is 5 (i.e. a size of $5*10^6$ calculations; subsection 5.1 contains an analysis of this parameter).
- $\mathrm{ReqRetryDelay}$ is uniformly randomized between 0ms and 200ms. That is, a rejected request is distributed again after an average time of 100ms. This timeout is recommended by [25] in order to avoid overloading the network with unsuccessful trials.
- We use a single PR only, since we do not examine failure scenarios here (see [7] for the impact of multiple PRs). No network latency is used (we will examine the impact of delay in subsection 5.4).
- The simulated real-time is 60 minutes; each simulation run is repeated at least 25 times with a different seed in order to achieve statistical accuracy.

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

5 Performance Analysis

[7] shows that an inappropriate load distribution of the RR and RAND policies leads to low performance in homogeneous capacity scenarios. Therefore, the first step is to examine the behaviour in the heterogeneous cases (as defined in subsection 3.2) under different workload parameters.

5.1 Workload Changes

The PU:PE ratio r has been found the most critical workload parameter [7]: e.g. at r=1 and a target utilization of 80%, each PU expects an exclusive PE during 80% of its runtime. That is, the lower r, the more critical the load distribution. In order to demonstrate the policy behaviour in a heterogeneous capacity scenario, a simulation has been performed varying r from 1 to 10 for κ=3 and a single fast server (we will examine distributions and settings of κ in detail in subsection 5.2). The handling speed result is shown on the left-hand side of figure 3 and clearly reflects the expectation from [7]: the lower r, the slower the request handling.

Applying our idea of ensuring a minimum capacity $\mathrm{MinCapPerReq}$ q for each request in process by a PE, it is clearly shown that the performance of RR and RAND is significantly improved: from less than 10% at $q = 10^4$ to about 32% at q=333,333 and even about 47% for $q = 5*10^5$. LU remains almost unaffected – due to state knowledge, "bad" selections (and therefore rejections) are very rare.

Varying the request size:PE capacity ratio s for a fixed setting of r=3 (the handling speed results are presented on the right-hand side of figure 3), the handling speed slightly sinks with a decreasing s: the smaller s, the higher the frequency of requests. For example, when decreasing the request size from s=10 to s=0.1 and keeping the number of PEs and other workload parameters constant, there will be 100 times more requests in the system (according to equation 3 and equation 1). The resulting speed reduction is strongest for RR and RAND, due to the lower load balancing qualities of these policies in comparison to LU (see also [7]). However, comparing the results for different settings of $\mathrm{MinCapPerReq}$ q, a significant increase of the handling speed can be observed when s gets larger at a higher setting of q. That is, small request sizes s are the most critical. The reason is that each rejection leads to an average penalty of 100ms (in order to avoid overloading the network with unsuccessful requests [25]). But for smaller s, the proportion of the startup delay gains an increasing importance in the overall request handling time of equation 2. For larger requests, the delay penalty fraction of the request handling time becomes negligible.

The results for varying the request interval can be derived from the previous results (see also equation 3) and have therefore been omitted. Note that the utilization plots have also been omitted, since there is no significant difference unless the setting of κ leads to a large number of extremely slow PEs.

In summary, it has been shown that our idea of using $\mathrm{MinCapPerReq}$ for rejecting inappropriately distributed requests can lead to a significant performance improvement. But what happens when the server capacity distribution and heterogeneity are changed?

5.2 Varying the Pool Heterogeneity

In order to show the effect of varying the heterogeneity of different server capacity distributions (κ/γ; denoted as *kappa* and *gamma* in the plots), simulations have been performed for the scenarios defined in subsection 3.2. The results are presented for a single fast server out of 10 (figure 4), 3 fast servers out of 10 (figure 5) and a linear capacity distribution (figure 6). For each figure, the left-hand side shows the handling speed, while the right-hand side presents the overhead in form of handle resolutions at the PR. We have omitted utilization plots, since they would not provide any new insights.

In general, the LU policy already provides a good load balancing, leading to no significant room for improvement by our $\mathrm{MinCapPerReq}$ approach unless the heterogeneity of the pools gets very high. However, a significant performance gain can be achieved for RR and RAND for all three capacity distributions: the higher $\mathrm{MinCapPerReq}$ q, the better the handling speed.

Two interesting observations can be made when comparing the results of RR and RAND: (1) For a higher heterogeneity of the pool, the performance of RR con-

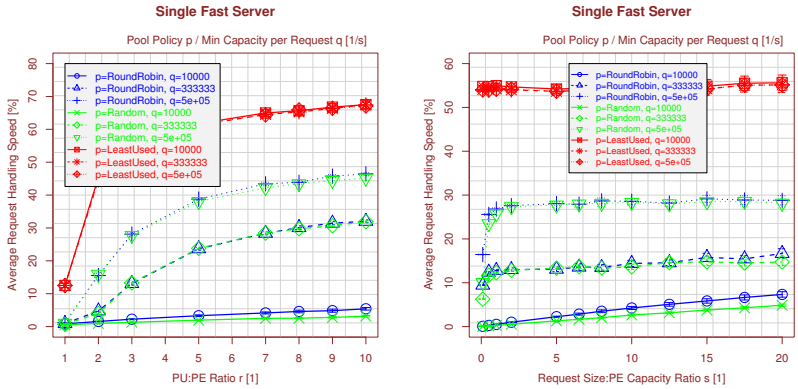

Figure 3. Varying the Workload Parameters

verges to the results of RAND. Since RR deterministically selects the PEs in turn, it will try all slow PEs in each round. If they are highly loaded (which gets more probable if their capacity is small, i.e. for a high setting of κ), the handling speed suffers. Here, RR's assumption that the next PE in its list is probably less utilized is wrong. In result, the RR performance gets more and more random. (2) There are performance jumps after κ=3 for q=333,333 and after κ=4 for $q = 5 * 10^5$. The reason for these jumps is the calculation of the maximum number of simultaneously processed requests $\mathrm{MaxRequests}$ in equation 1. For example, $\mathrm{MaxRequests}$ for a slow PE at q=333,333 sinks from 3 at κ=3 to 2 at κ=3.5. Given a pool of 10 PEs with one fast PE, and a total pool capacity of 10^7 calculations/s, $\mathrm{MaxRequests}$ is calculated as follows: For κ=3 and $\mathrm{MinCapPerReq}$ q=333,333: according to equation 4, $\mathrm{Capacity}_{\mathrm{SlowPE}}$=833,333 $\Rightarrow$ $\mathrm{MaxRequests}$=round($\frac{833{,}333}{333{,}333}$)=3. For κ=3.5 and $\mathrm{MinCapPerReq}$ q=333,333: $\mathrm{Capacity}_{\mathrm{SlowPE}}$=800,000 $\Rightarrow$ $\mathrm{MaxRequests}$=round($\frac{800{,}000}{333{,}333}$)=2. That is, loaded PEs accept less additional requests and the usage of better servers is enforced by rejection. This leads to an improved handling speed.

Comparing the results of the different capacity distributions, it is clear that the "single fast server" scenario is the most critical one: for higher settings of κ, most of the pool's capacity is concentrated at a single PE. Therefore, this dedicated PE has to be selected in order to achieve a better handling speed. If three of the PEs are fast ones, the situation changes: a larger share of the total pool capacity is provided by fast PEs. That is, the "selection in turn" strategy of RR becomes more successful – the "next" PE is not almost certainly a slow one. Since LU selects by load without incorporating capacity, its handling speed decays with a higher heterogeneity of the pool: selecting a slow but currently least-loaded PE results in a decreased handling speed. In this case, the rejection approach also gets useful for LU: 48% at $q = 5 * 10^5$ vs. 40% at q=333,333 for κ=5.

The linear distribution is the least critical one: even if randomly selecting one of the slower PEs, the next handle resolution will probably return one of the faster PEs. For RR, this behaviour will even be deterministic and LU again improves it by PE load state knowledge.

In summary, it has been shown that our request rejection approach is working in all three heterogeneous capacity distribution scenarios. But what about its overhead? The handle resolutions overhead is significantly increased for a large setting of $\mathrm{MinCapPerReq}$ q: the higher the rejection probability, the more server selections. Again, as already explained above, the overhead varies with κ (fast servers) and γ (linear) depending on the setting of q, due to the $\mathrm{MaxRequests}$ calculation in equation 1. Obviously, the probability of a rejection is highest for RAND and lowest for LU (due to the different load balancing qualities of these policies). Comparing the results of the different capacity distributions, it can also be observed that the overhead is highest for the "single fast server" setup and lowest for the linear distribution. Clearly, the more critical the distribution, the higher the chance to choose an inappropriate PE. Therefore, the resulting question is: how to reduce this overhead – without a significant penalty on the handling speed improvement?

5.3 Reducing the Network Overhead

In order to present the impact of the PU-side cache on performance and overhead, we have performed simulations using a setting of κ=3 – i.e. a not too critical setting – and varying the stale cache value c (given as ratio between the actual stale cache value and the request size:PE capacity ratio) from 0.0 to 1.0. This cache value range has been chosen to allow for cache utilization in case of retries and to also support dynamic pools (PEs may register or deregister, i.e. the entries may not get too old). Figure 7 presents the results for a single fast server, i.e. for the most critical distribution.

Taking a look at the results, it is clear that even a small setting of c results in a significantly reduced overhead while the handling speeds of RR and RAND are not negatively affected. For LU, however, the load state information becomes more out of date the higher c. This leads to a slightly decreasing handling speed if $\mathrm{MinCapPerReq}$ q is low (here: q=333,333 calculations/s) – using a larger setting, inappropriate choices are "corrected" by the reject-and-retry approach. The results for multiple fast servers or

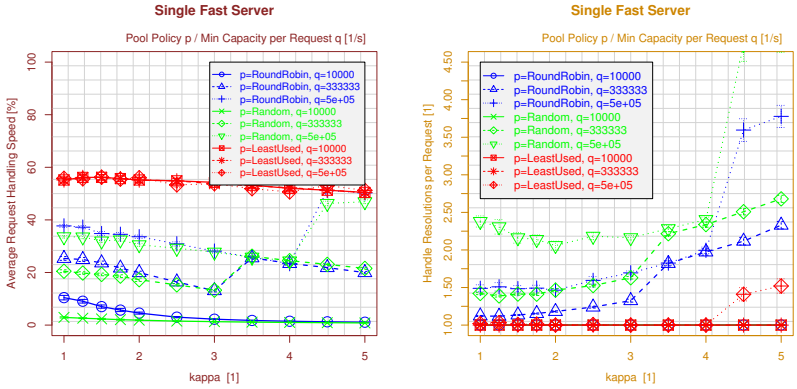

Figure 4. The Impact of the Pool Heterogeneity for a Single Fast Server

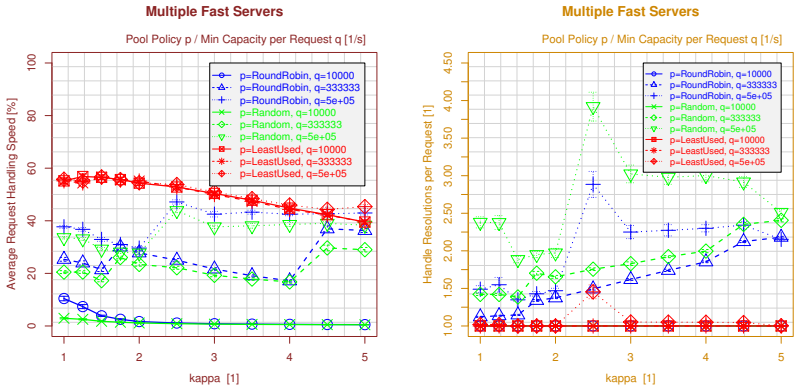

Figure 5. The Impact of the Pool Heterogeneity for a Multiple Fast Servers

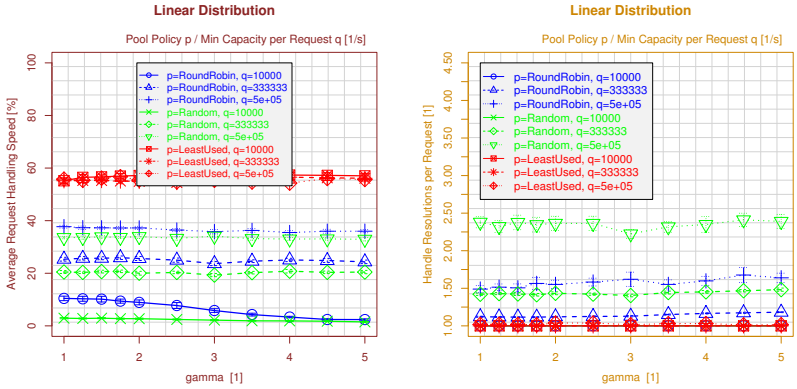

Figure 6. The Impact of the Pool Heterogeneity for a Linear Capacity Distribution

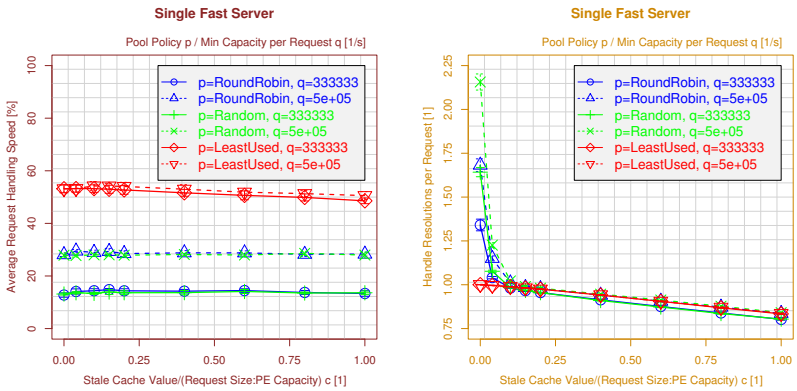

Figure 7. The Impact of the Stale Cache Value for a Single Fast Server

even a linear capacity distribution (for varying γ) are very similar, therefore plots have been omitted.

In summary, the PU-side cache can achieve a significant overhead reduction for the RR and RAND policies, while the performance does not decrease. However, care has to be taken of LU: its performance suffers for higher settings of c, at only a small achievable overhead reduction (LU already has a low rejection rate). In general, the cache should not be used with LU.

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 in the same building), there are some scenarios where components are distributed globally [10]. 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 8 presents the performance results of varying the delay in the three capacity distribution scenarios at κ=3 (fast servers) and γ=3 (linear) – i.e. in not too critical setups – for s=1.

As can be expected, the handling speed sinks with rising network delay: in equation 2, the startup delay gains an increasing importance in the overall handling time – due to the latency caused by PR queries and PE contacts.

Comparing the curves for the different settings of $\mathrm{MinCapPerReq}$, the achieved gain by a higher minimum capacity shrinks with the delay: while the request rejection rate of the PE keeps almost constant, the costs for a rejection increase: now, there is not only the penalty of $\mathrm{ReqRetryDelay}$ but also an additional latency for querying the PR and contacting another PE. The only exception is the LU policy: as adaptive policy, it relies on up-to-date load information. However, due to the latency, this information becomes more obsolete the higher the delay. That is, the latency increases the selection probability for inappropriate PEs. In this case, using a higher setting of $\mathrm{MinCapPerReq}$ (here: $5 * 10^5$ calculations/s) leads to a slightly improved handling speed.

In result, our approach is also useful for scenarios with significant network delay – even for the adaptive LU policy.

6 Conclusions

We have indicated by our evaluations that it is possible to improve the request handling performance of the basic RSerPool policies under varying workload parameters in different server capacity scenarios of varying heterogeneity – without modifying the policies themselves – by setting a minimum capacity per request to limit the maximum number of simultaneously handled requests. This leads to a significant performance improvement for the RR and RAND policies, while – in general – it does not improve the performance of LU. However, in case of a significant network delay in combination with short requests, our approach also gets useful for LU. Request rejection leads to an increased overhead, in particular to additional handle resolutions. Using the PU-side cache can reduce this overhead while not significantly affecting the system performance – with care to be taken of the capacity distribution in case of LU.

We are currently also validating our simulative performance results in real-life scenarios by using our RSerPool prototype implementation RSPLIB in the PLANETLAB; first results can be found in [3, 10].

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*, volume 1, pages 329–337, Anchorage, Alaska/U.S.A., Apr. 2001. ISBN 0-7803-7016-3.

[2] 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.

[3] 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.

[4] 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.

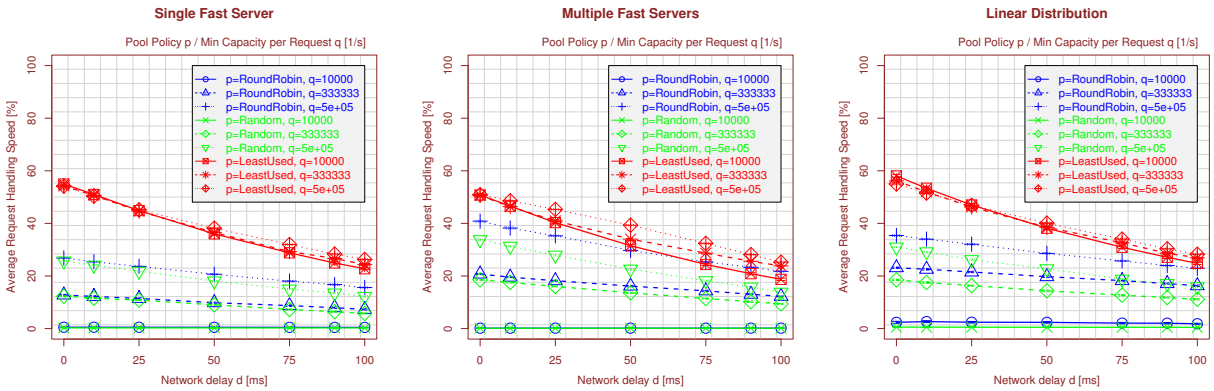

Figure 8. The Impact of the Network Delay on the Handling Speed

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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 Communication and Networking (FGCN)*, Jeju Island/South Korea, Dec. 2007.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] ITU-T. Introduction to CCITT Signalling System No. 7. Technical Report Recommendation Q.700, International Telecommunication Union, Mar. 1993.

[17] 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.

[18] 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.

[19] E. P. Rathgeb. The MainStreetXpress 36190: a scalable and highly reliable ATM core services switch. *International Journal of Computer and Telecommunications Networking*, 31(6):583–601, Mar. 1999.

[20] 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.

[21] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Highly available Internet services using connection migration. In *Proceedings of the ICDCS 2002*, pages 17–26, Vienna/Austria, July 2002.

[22] 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.

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

[24] 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.

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

[26] X. Zhou, T. Dreibholz, and E. P. Rathgeb. A New Approach of Performance Improvement for Server Selection in Reliable Server Pooling Systems. In *Proceedings of the 15th IEEE International Conference on Advanced Computing and Communication (ADCOM)*, Guwahati/India, Dec. 2007.