Improving the Load Balancing Performance of Reliable Server Pooling in Heterogeneous Capacity Environments*

Xing Zhou[1], Thomas Dreibholz[2], and Erwin P. Rathgeb[2]

[1] Hainan University
College of Information Science and Technology
Renmin Road 58, 570228 Haikou, Hainan, China
Tel: +86 898 6625-0584, Fax: +86 898 6618-7056
xing.zhou@uni-due.de
[2] University of Duisburg-Essen
Institute for Experimental Mathematics
Ellernstrae 29, D-45326 Essen, Germany
Tel: +49 201 183-7637, Fax: +49 201 183-7673
dreibh@iem.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. But is this approach useful – in particular if the server capacities increase in terms of a heterogeneous capacity distribution? Applying failover handling mechanisms of RSerPool, in this case, could choose a more appropriate server instead.
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.
Key words: Reliable Server Pooling, Redundancy, Load Balancing, Heterogeneous Pools, Performance Evaluation

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 [1] – there had not been any generic, standardized approaches for the availability of Internet-based services. Each application had to realize its own solution and

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

therefore to re-invent the wheel. This deficiency – once more arisen for the availability of SS7 (Signalling System No. 7 [2]) 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 [3,4]), but their combination into one application-independent framework is.

The Reliable Server Pooling (RSerPool) architecture [5] currently under standardization by the IETF RSerPool WG is an overlay network framework to provide server replication [6] and session failover capabilities [7, 8] to its applications. Server redundancy leads to load distribution and load balancing [9], which are also covered by RSerPool [10, 11]. But in strong contrast to already available solutions in the area of GRID and high-performance computing [12], the fundamental property of RSerPool is to be "light-weight", 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 [13]. A generic classification of load distribution algorithms can be found in [9]; 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. On the other hand, non-adaptive algorithms do not require such status data. More details on such algorithms can be found in [14, 15].

There has already been some research on the performance of RSerPool usage for applications like SCTP-based endpoint mobility [16], VoIP with SIP [17], web server pools [18], IP Flow Information Export (IPFIX) [19], real-time distributed computing [6, 7] and battlefield networks [20]. A generic application model for RSerPool systems has been introduced by [6, 10], 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 [6, 10, 11].

2 "Reject and Retry" – Our Performance Improvement Approach

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 in case of pool capacity changes. The merit of our approach is that the policies themselves are not modified: they are widely supported and their performance is well-known [10]. Furthermore, implementing only a very limited number of policies is quite easy [13, 21] (which is clearly beneficial for a "light-weight" system). That is, applying a specialised new policy to only improve a temporary capacity extension may be unsuitable ("Never change a running system!"). Therefore, we focus on the session layer instead: if a request gets rejected, the failover mechanisms provided by RSerPool could choose a possibly better server instead. For a pool of homogeneous servers, we have already shown in [22] that our approach works quite well. Even when the capacity distribution within the pool changes – while the overall pool capacity remains constant – it is useful to apply "reject and retry" (see our paper [23]). But what happens when the pool capacity temporarily increases, e.g. due to spare capacity on some servers? The goal of this paper is to evaluate the performance of our strategy in such situations, with respect to the resulting protocol overhead. We also identify critical configuration parameter ranges in order to provide a guideline for designing and configuring efficient RSerPool systems.

3 The RSerPool Protocol Framework

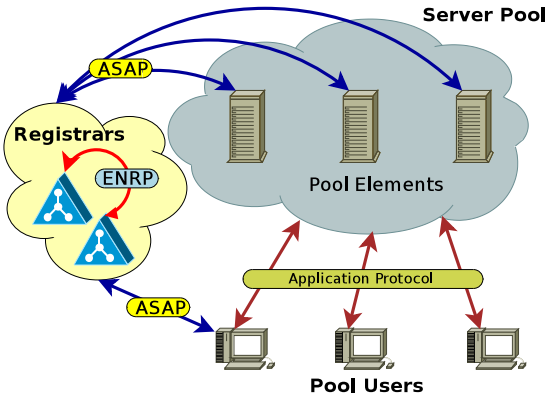

Fig. 1. The RSerPool Architecture

Figure 1 illustrates the RSerPool architecture [6]. 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 [24]), transported via SCTP [25,26]. 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 [12] – in order to keep the management complexity [13] reasonably low. Nevertheless, it is assumed that PEs can be distributed globally for their service to survive localized disasters [27].

PEs choose an arbitrary PR of the operation scope to register into a pool by using the Aggregate Server Access Protocol (ASAP [28]), 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, again using ASAP [28] 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 [29], 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 [13]. Detailed discussions of pool policies can be found in [6, 10].

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 time-

out, 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 [10].

4 Quantifying a RSerPool System

In order to evaluate the behaviour of a RSerPool system, it is necessary to quantify RSerPool systems. The system parameters relevant to this paper can be divided into two groups: RSerPool system parameters and server capacity distributions.

4.1 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[3]. 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 is limited by the parameter $\mathrm{MinCapPerReq}$. 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

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

simultaneously handled requests. Note, that the limit is rounded to the nearest integer, in order to support arbitrary capacities. If a PE's requests limit is reached, a new request gets rejected. For example, if the PE capacity is 10^6 calculations/s and $\mathrm{MinCapPerReq}$=$2.5*10^5$, 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 [30]).

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):

$$d_{\mathrm{Handling}} = d_{\mathrm{Queuing}} + d_{\mathrm{Startup}} + d_{\mathrm{Processing}}. \tag{2}$$

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

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

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:

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

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 also [6, 10] for more details on this subject.

4.2 Heterogeneous Server Capacity Distributions

In order to present the effects introduced by heterogeneous servers, we have considered three different and realistic capacity distributions (based on [6]) for increasing the pool capacity: a single powerful server, multiple powerful servers and a linear capacity distribution. Clearly, the goal of our "reject and retry" approach (see section 2) is to make best use of the additional capacity for increasing the request handling speed.

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 ($\mathrm{PoolCapacity}_{\mathrm{New}}$) and the original capacity ($\mathrm{PoolCapacity}_{\mathrm{Original}}$) of the pool:

$$\varphi = \frac{\mathrm{PoolCapacity}_{\mathrm{New}}}{\mathrm{PoolCapacity}_{\mathrm{Original}}}. \tag{4}$$

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 $\Delta_{\mathrm{Pool}}(\varphi)$ of the whole pool can be calculated as follows:

$$\Delta_{\mathrm{Pool}}(\varphi) = \underbrace{(\varphi * \mathrm{PoolCapacity}_{\mathrm{Original}})}_{\mathrm{PoolCapacity}_{\mathrm{New}}} - \mathrm{PoolCapacity}_{\mathrm{Original}}. \tag{5}$$

Then, the capacity of the i-th PE can be deduced using equation 5 by the following formula (where NumPEs denotes the number of PEs):

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

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

Multiple Powerful Servers If using multiple powerful servers ($\mathrm{NumPEs}_{\mathrm{Fast}}$) instead of only one at one time, the capacity of the i-th PE can be calculated as follows (according to equation 5):

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

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

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 $\mathrm{PoolCapacity}_{\mathrm{New}}$. Therefore, the capacity of the i-th PE can be obtained using the following equations (again, using $\Delta_{\mathrm{Pool}}(\varphi)$ as defined in equation 5):

$$\Delta_{\mathrm{FastestPE}}(\varphi) = \frac{2 * \Delta_{\mathrm{Pool}}(\varphi)}{\mathrm{NumPEs}},$$

$$\mathrm{Capacity}_i(\varphi) = \underbrace{\underbrace{\frac{\Delta_{\mathrm{FastestPE}}(\varphi)}{\mathrm{NumPEs}-1}}_{\text{Capacity Gradient}} * (i-1)}_{\text{Additional Capacity for PE } i} + \frac{\mathrm{PoolCapacity}_{\mathrm{Original}}}{\mathrm{NumPEs}}.$$

5 Setup Simulation Model

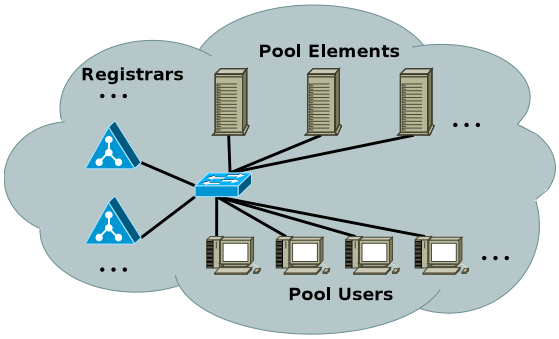

Fig. 2. The Simulation Setup

For the performance analysis, the RSerPool simulation model RSPSIM [6, 10] has been used. This model is based on the OMNET++ [31] simulation environment and

contains the protocols ASAP [28] and ENRP [24], a PR module as well as PE and PU modules for the request handling scenario defined in section 4. 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 [13].

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% for φ=1.
- 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; in the basic setup, each one is providing a capacity of 10^6 calculations/s.
- The heterogeneity parameter φ is 3 (we analyse variations in subsection 6.2).
- A PU:PE ratio of 3 is used (this parameter is analysed in subsection 6.1).
- The default request size:PE capacity ratio is 5 (i.e. a size of $5 * 10^6$ calculations; subsection 6.1 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 [30] 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 [10] for the impact of multiple PRs).
- No network latency is used (we will examine the impact of delay in subsection 6.4).
- The simulated real-time is 60m; 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 corresponding 95% confidence intervals.

6 Performance Analysis

[10] 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 case under different workload parameters.

6.1 General Behaviour on Workload Changes

The PU:PE ratio r has been found the most critical workload parameter [10]: 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 φ=3 and a single fast server (we will examine distributions and settings of φ in detail in subsection 6.2). The handling speed result is shown on the left-hand side of figure 3 and clearly reflects the expectation from [10]: 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: for $q = 10^6$, it is even comparable to LU.

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.

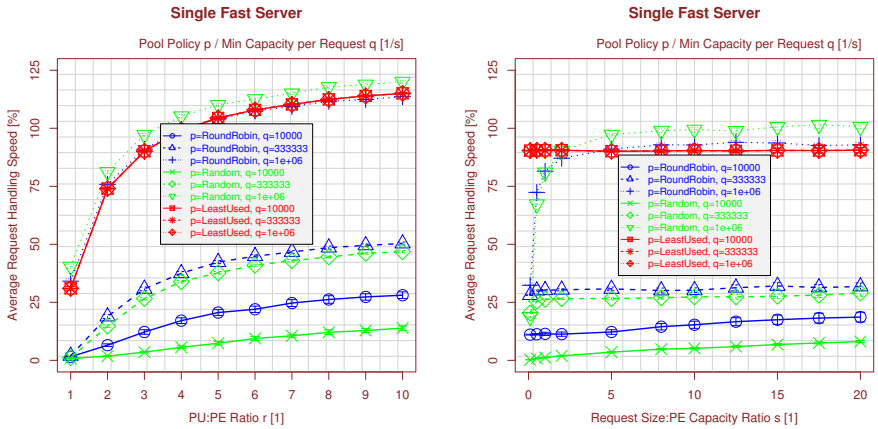

Fig. 3. Varying the Workload Parameters

That is, while the workload keeps constant, there are 100 times more requests in the system for s=10^5 compared with s=10^7. However, comparing the results for different settings of $\mathrm{MinCapPerReq}$ q in this critical parameter range, a significant impact can be observed: the handling speed for using a high setting of q significantly drops. The reason is that each rejection leads to an average penalty of 100ms (in order to avoid overloading the network with unsuccessful requests [30]). 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 also the utilization plots have been omitted, since the larger φ, the higher the pool capacity. Consequently, at φ=3, the utilization is already in a non-critical range.

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 if the server capacity distribution and heterogeneity are changed?

6.2 Varying the Heterogeneity of the Pool

In order to show the effect of varying the heterogeneity of different server capacity distributions (φ; denoted as *phi* in the plots), simulations have been performed for the scenarios defined in subsection 4.2. 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 are obvious and 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. 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. Interestingly, the handling speed for RR and RAND at $q = 10^6$ calculations/s even slightly exceeds the speed of LU for $\varphi > 2$ in the case of the "fast servers" scenarios! The reason for

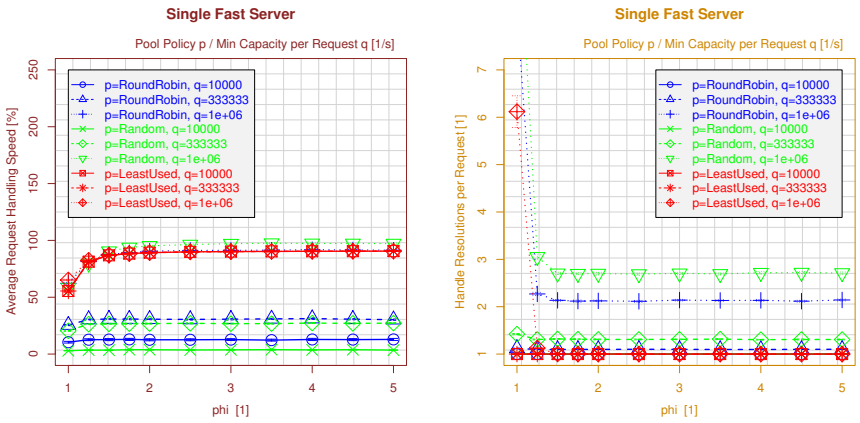

Fig. 4. The Impact of the Pool Heterogeneity for a Single Fast Server

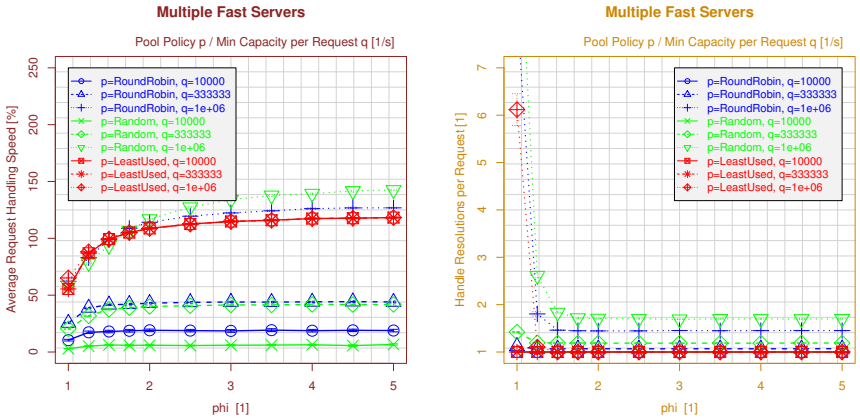

Fig. 5. The Impact of the Pool Heterogeneity for a Multiple Fast Servers

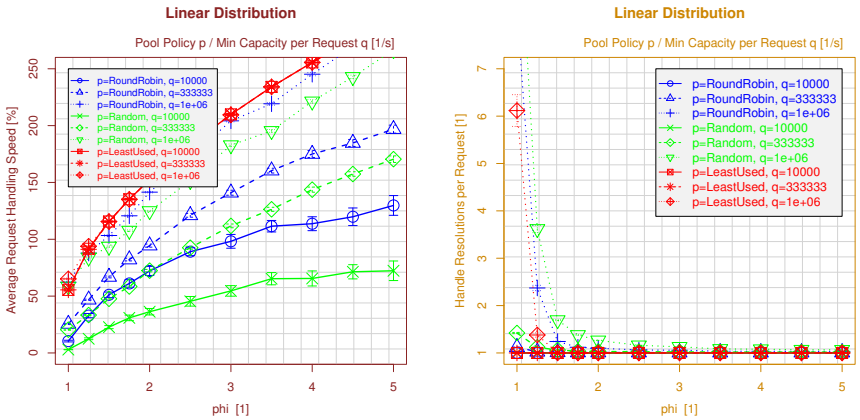

Fig. 6. The Impact of the Pool Heterogeneity for a Linear Capacity Distribution

this behaviour is that LU selects the least loaded PE rather than the fastest one. That is, if all faster PEs have a higher load than a slow one, the slow one is always chosen by definition. In contrast, the linear distribution is much less critical (except for the first PE, all other PEs are "faster" ones) and LU performs better here. Another interesting observation is that the performance of RAND becomes better than RR for large q (here: $q = 10^6$ calculations/s): RR deterministically selects the PEs in turn. In the worst case, if all 9 slow PEs are already loaded up to their limit, there will be 9 rejections until the fast one gets selected again. Clearly, selecting randomly will provide a better result here.

Comparing the results of the different capacity distributions, it is clear that the "single fast server" scenario (see also subsubsection 4.2) is the most critical one: for higher settings of φ, 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 becomes better, leading to a significantly improved handling speed compared with the first scenario. Finally, 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 for RR and RAND in all three heterogeneous capacity distribution scenarios, while there is no significant benefit for LU. But what about its overhead? The handle resolutions overhead is significantly increased for a small setting of φ: here, the overall pool capacity is still small and the PEs are working at almost their target utilization. This means that the selection of an inappropriate PE becomes more probable and therefore the rejection probability higher. Obviously, the probability of a rejection is highest for RAND and lowest for LU.

From the results above, it also can 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 get an inappropriate PE. But interestingly, the overhead for RR and RAND at $\varphi > 2$ almost keeps constant for the two fast servers scenarios – while it decreases for the linear distribution as well as for using LU: the non-adaptive policies may try to use fully occupied PEs due to their lack of load state knowledge, which leads to rejections and therefore to increased overhead. However, even for the scenario of a single fast PE, this overhead keeps below 2.75 handle resolutions per request for $\varphi \geq 2$. But is it possible to reduce this overhead – without a significant penalty on the handling speed improvement?

6.3 Reducing the Network Overhead by Cache Usage

In order to present the impact of the PU-side cache on performance and overhead, we have performed simulations using a setting of φ=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). Figure 7 presents the results for a single fast server (i.e. the most critical distribution) and figure 8 the plots for a linear distribution. We have omitted results for multiple fast servers for space reasons, since they would not provide any new insights.

Taking a look at the "fast server" 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. Even better, the handling speed of RR slightly increases! The reason for this effect is that each cache constitutes an additional selection instance

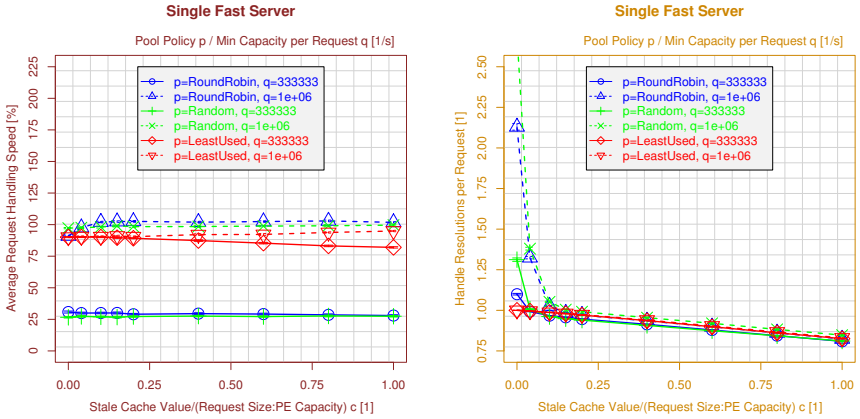

Fig. 7. The Impact of the Stale Cache Value for a Single Fast Server

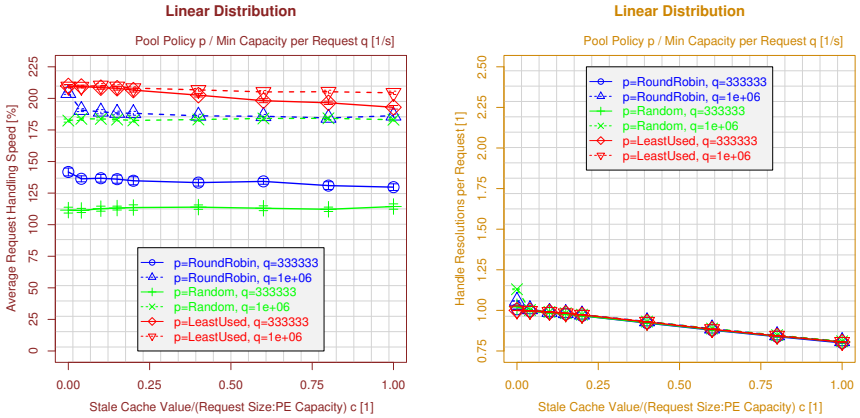

Fig. 8. The Impact of the Stale Cache Value for a Linear Capacity Distribution

performing round robin choices independently of the PR and other PU-side caches. That is, while each instance performs its selections in turn, the global view of the selections in the system differs from the desired round robin strategy. Instead, it gets more and more random – but the local selection still avoids that a rejected request is mapped to the same PE again (which may happen for RAND). For LU, the load state information gets the more out of date the higher c. This leads to a decreasing handling speed if $\mathrm{MinCapPerReq}$ q is low (here: q=333,333 calculations/s) – using a larger setting, inappropriate choices are "corrected" by our "reject and retry" approach.

Having multiple fast servers or even a linear capacity distribution, the number of rejections and retries – and therefore the number of handle resolutions – is significantly smaller (see also subsection 6.2). Therefore, the impact of the cache gets less significant in comparison with the scenario of a single fast server. The most interesting observation here is the behaviour of the RR policy: for a linear distribution, the cache leads to a slightly reduced handling speed. The reason is the caches which perform round robin selections independently. But here, the independent selections work counterproductively: almost all PEs can be considered to be fast ones (at least more powerful than the first one), so the selection order gets irrelevant – which is confirmed by the RR curve for $\mathrm{MinCapPerReq}$ $q = 10^6$ converging to the speed of RAND for increasing c.

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

6.4 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 [27]. 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. For that reason, figure 9 presents the performance results for varying the delay in the three capacity distribution scenarios at φ=3 (i.e. not too critical) for s=1.

As it can be expected, the handling speed sinks with rising network delay: in equation 2, the startup delay gains an increasing importance to the overall handling time due to the latency caused by querying a PR and contacting PEs.

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 of a rejection increase: now, there is not only the penalty of $\mathrm{ReqRetryDelay}$ but an additional latency for querying the PR and contacting another PE. This is particularly important for 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: 10^6 calculations/s) leads to a slightly improved handling speed for the critical "fast servers" setups – in particular for using a single fast server. However, for the linear distribution – which is much less critical (see subsection 6.2) – no significant change can be observed.

As a summary, the simulations have shown that our request rejection approach is also useful for scenarios having a significant network delay. In particular, it even gets useful for the adaptive LU policy.

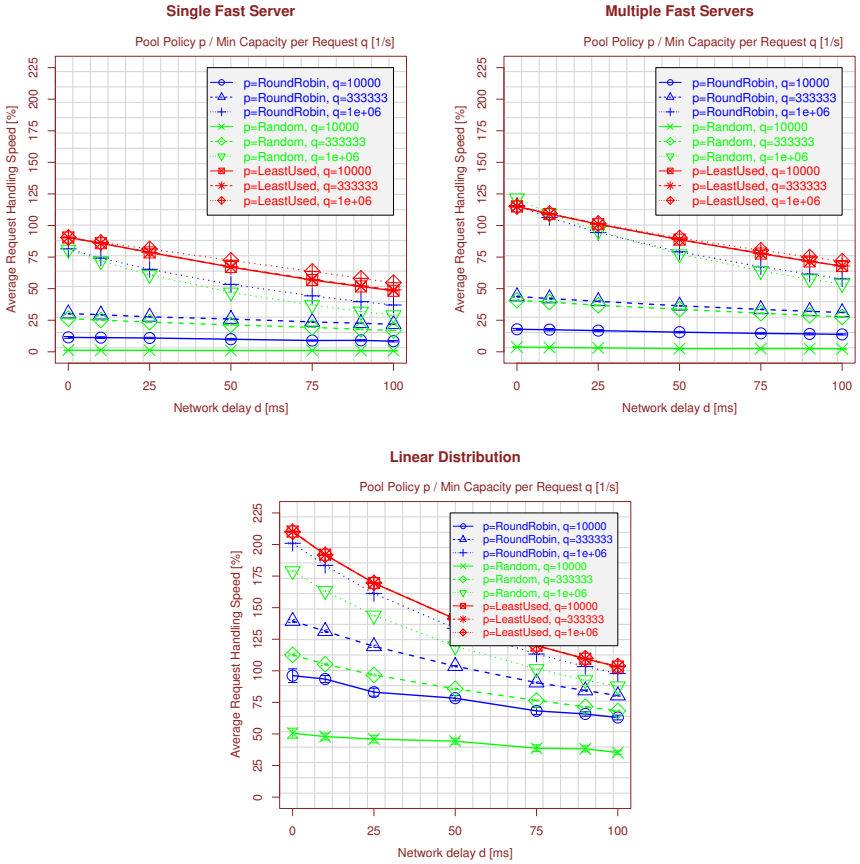

Fig. 9. The Impact of the Network Delay on the Handling Speed

7 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. Our "reject and retry" approach leads to a significant performance improvement for the RR and RAND policies, while – in general – it does not provide a benefit for 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 rejections lead to an increased overhead, in particular to additional handle resolutions. Usage of the PU-side cache can reduce this overhead while not significantly affecting the system performance – with care to be taken for the capacity distribution in case of RR.

As part of our future research, we are currently also validating our simulative performance results in real-life scenarios, using our RSerPool prototype implementation RSPLIB [6, 27] in the PLANETLAB; first results can be found in [6, 27].

References

1. Rathgeb, E.P.: The MainStreetXpress 36190: a scalable and highly reliable ATM core services switch. International Journal of Computer and Telecommunications Networking **31**(6) (March 1999) 583–601
2. ITU-T: Introduction to CCITT Signalling System No. 7. Technical Report Recommendation Q.700, International Telecommunication Union (March 1993)
3. Alvisi, L., Bressoud, T.C., El-Khashab, A., Marzullo, K., Zagorodnov, D.: Wrapping Server-Side TCP to Mask Connection Failures. In: Proceedings of the IEEE Infocom 2001. Volume 1., Anchorage, Alaska/U.S.A. (April 2001) 329–337 ISBN 0-7803-7016-3.
4. Sultan, F., Srinivasan, K., Iyer, D., Iftode, L.: Migratory TCP: Highly available Internet services using connection migration. In: Proceedings of the ICDCS 2002, Vienna/Austria (July 2002) 17–26
5. Lei, P., Ong, L., Tüxen, M., Dreibholz, T.: 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.
6. Dreibholz, T.: 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 (March 2007)
7. Dreibholz, T., Rathgeb, E.P.: RSerPool – Providing Highly Available Services using Unreliable Servers. In: Proceedings of the 31st IEEE EuroMirco Conference on Software Engineering and Advanced Applications, Porto/Portugal (August 2005) 396–403 ISBN 0-7695-2431-1.
8. Dreibholz, T.: An Efficient Approach for State Sharing in Server Pools. In: Proceedings of the 27th IEEE Local Computer Networks Conference (LCN), Tampa, Florida/U.S.A. (October 2002) 348–352 ISBN 0-7695-1591-6.
9. Gupta, D., Bepari, P.: Load Sharing in Distributed Systems. In: Proceedings of the National Workshop on Distributed Computing. (January 1999)
10. Dreibholz, T., Rathgeb, E.P.: On the Performance of Reliable Server Pooling Systems. In: Proceedings of the IEEE Conference on Local Computer Networks (LCN) 30th Anniversary, Sydney/Australia (November 2005) 200–208 ISBN 0-7695-2421-4.
11. Dreibholz, T., Zhou, X., Rathgeb, E.P.: 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, Lübeck/Germany (August 2007) 157–164 ISBN 0-7695-2977-1.
12. Foster, I.: What is the Grid? A Three Point Checklist. GRID Today (July 2002)

13. Dreibholz, T., Rathgeb, E.P.: Implementing the Reliable Server Pooling Framework. In: Proceedings of the 8th IEEE International Conference on Telecommunications (ConTEL). Volume 1., Zagreb/Croatia (June 2005) 21–28 ISBN 953-184-081-4.
14. Kremien, O., Kramer, J.: Methodical Analysis of Adaptive Load Sharing Algorithms. IEEE Transactions on Parallel and Distributed Systems **3**(6) (1992)
15. Dykes, S.G., Robbins, K.A., Jeffery, C.L.: An Empirical Evaluation of Client-Side Server Selection Algorithms. In: Proceedings of the IEEE Infocom 2000. Volume 3., Tel Aviv/Israel (March 2000) 1361–1370 ISBN 0-7803-5880-5.
16. Dreibholz, T., Jungmaier, A., Tüxen, M.: A new Scheme for IP-based Internet Mobility. In: Proceedings of the 28th IEEE Local Computer Networks Conference (LCN), Königswinter/Germany (November 2003) 99–108 ISBN 0-7695-2037-5.
17. Conrad, P., Jungmaier, A., Ross, C., Sim, W.C., Tüxen, M.: 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.
18. Siddiqui, S.A.: 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 (November 2006)
19. Dreibholz, T., Coene, L., Conrad, P.: 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.
20. Uyar, Ü., Zheng, J., Fecko, M.A., Samtani, S., Conrad, P.: 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) (2004) 164–175
21. Dreibholz, T., Rathgeb, E.P.: 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 (December 2007)
22. Zhou, X., Dreibholz, T., Rathgeb, E.P.: 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 (December 2007)
23. Zhou, X., Dreibholz, T., Rathgeb, E.P.: Evaluation of a Simple Load Balancing Improvement for Reliable Server Pooling with Heterogeneous Server Pools. In: Proceedings of the IEEE International Conference on Future Generation Communication and Networking (FGCN), Jeju Island/South Korea (December 2007)
24. Xie, Q., Stewart, R., Stillman, M., Tüxen, M., Silverton, A.: 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. Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L., Paxson, V.: Stream Control Transmission Protocol. Standards Track RFC 2960, IETF (October 2000)
26. Jungmaier, A., Rathgeb, E.P., Tüxen, M.: 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. Dreibholz, T., Rathgeb, E.P.: 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 (February 2007)
28. Stewart, R., Xie, Q., Stillman, M., Tüxen, M.: Aggregate Server Access Protcol (ASAP). Internet-Draft Version 16, IETF, RSerPool Working Group (July 2007) draft-ietf-rserpool-asap-16.txt, work in progress.
29. Tüxen, M., Dreibholz, T.: Reliable Server Pooling Policies. Internet-Draft Version 05, IETF, RSerPool Working Group (July 2007) draft-ietf-rserpool-policies-05.txt, work in progress.
30. Zhang, Y.: Distributed Computing mit Reliable Server Pooling. Master's thesis, Universität Essen, Institut für Experimentelle Mathematik (April 2004)
31. Varga, A.: OMNeT++ Discrete Event Simulation System User Manual - Version 3.2, Technical University of Budapest/Hungary. (March 2005)