A New Server Selection Strategy for Reliable Server Pooling in Widely Distributed Environments*

Xing Zhou
Hainan University
College of Information Science and Technology
570228 Haikou, Hainan, China
xing.zhou@uni-due.de

Thomas Dreibholz, Erwin P. Rathgeb
University of Duisburg-Essen
Institute for Experimental Mathematics
Ellernstrasse 29, 45326 Essen, Germany
{dreibh,rathgeb}@exp-math.uni-essen.de

Abstract

In order to provide a generic, application-independent and resource-efficient framework for server redundancy and session failover, the IETF RSerPool WG is currently standardizing the Reliable Server Pooling (RSerPool) framework. Server redundancy has to take load distribution and load balancing into consideration since these issues are crucial for the system performance.

There has already been some research on the server selection strategies of RSerPool for different application scenarios. In particular, it has been shown that the adaptive Least Used selection usually provides the best performance. This strategy requires up-to-date load information of the services, which has to be propagated among distributed pool management components. But network delay (which is realistic for systems being widely distributed to achieve availability in case of regional servers failures) as well as caching of information may both lead to obsolete load information. Therefore, the purpose of this paper is to analyse and evaluate the performance of a new server selection rule to cope with update latencies. Especially, we will also analyse the impact of different workload parameters on the performance of the new server selection strategy.

Keywords: *Reliable Server Pooling, Load Balancing, Least-Used Selection, Latency*

1 Introduction and Scope

In today's Internet, the availability of services (e.g. e-commerce) gets increasingly crucial. But in contrast with the telecommunications world – where availability is ensured by redundant links and devices [21] – no standards approaches have been available for Internet services. If required, each application had to realize its home-grown approach, re-inventing the wheel again and again. This has been the motivation of the IETF for the foundation of the RSerPool WG to define the Reliable Server Pooling (RSerPool) framework [9,20] – a generic, application-independent framework for pool management and session handling.

RSerPool provides a framework for server replication [7] as well as session management[1] including session failover capabilities [2,12] to its applications. Efficiently handling server redundancy requires load distribution and load balancing [18], which are also covered by RSerPool [7,8,16,28–30]. Two classes of load distribution algorithms [18] are supported by RSerPool: non-adaptive and adaptive algorithms. Adaptive strategies base their assignment decisions on the current status of the servers (which requires up-to-date information); non-adaptive algorithms do not need such data.

In strong contrast to existing frameworks for GRID and high-performance computing (see [17] for details), the fundamental property of RSerPool is intended to be "light-weight", i.e. it must be usable on equipment providing only scarce CPU power and memory resources (in particular: routers and other telecommunications equipment). Of course, its application is not restricted to low-end hardware [3]. The "light-weight"-property restricts the scope of the RSerPool architecture to the session handling and pool management, but on the other hand it allows for highly efficient realization [6,9].

Up to now, there has already been research on the usage of RSerPool for various scenarios: SCTP-based mobility [5], VoIP with SIP [1], web server pools [3], IP Flow Information Export (IPFIX) [4], real-time distributed computing [3,7,16,27,28] and battlefield networks [24]. A generic application model for RSerPool systems has been defined in [7]. This model 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 [3,7,16].

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

[1]RSerPool will be the IETF's first Session Layer standard.

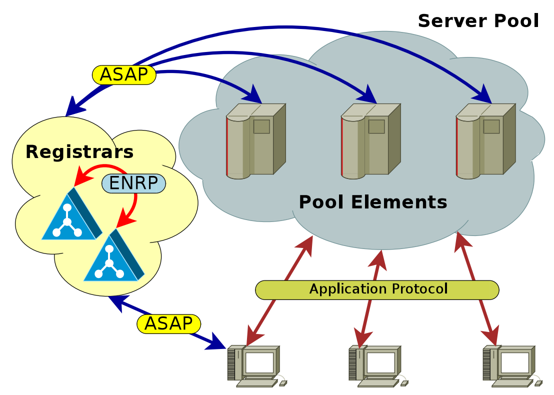

Figure 1. The RSerPool Architecture

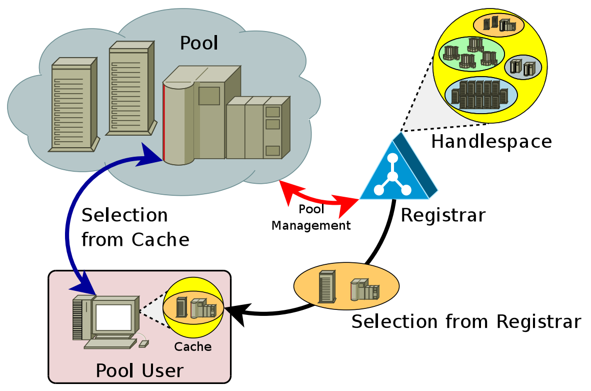

Figure 2. The Server Selection by PR and PU

2 The RSerPool Architecture

An overview of the RSerPool architecture [3, 11, 20] is illustrated in figure 1: servers of a pool are called *pool elements* (PE), a client is denoted as *pool user* (PU). Proxy PEs and PUs may provide a migration path for non-RSerPool components. 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 [26]), transported via SCTP [19]. An operation scope has a limited range, e.g. a whole organization or only a wiring closet. Unlike GRID computing [17], it is restricted to a single administrative domain, in order to allow an efficient management [6, 9]. Nevertheless, PEs may be distributed globally for their service to survive localized disasters [10, 15].

PEs choose an arbitrary PR of the operation scope to register into a pool by using the Aggregate Server Access Protocol (ASAP [22]), 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 [22]), transported via SCTP. As illustrated in figure 2, the PR selects the requested list of PE identities by applying a pool-specific selection rule, called *pool policy*. Adaptive and non-adaptive pool policies are defined in [23]; 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].

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 Least-Used with Degradation Policy

Due to the latency on the connections between PE and PR-H as well as among PRs, the load state information of a PE may already be out of date when it has reached all handlespace copies. And this deprecation gets even worse because of the usage of the PU-side cache. Since all PE selections for LU are based on the load state information in different copies of the handlespace (PRs and PU-side caches), the load balancing quality may suffer.

In [13], we have considered the so-called Least Used with Degradation (LUD) policy as part of our future work. For this policy, each PE may specify a *load increment* constant providing the load increase by a newly accepted request. In each selection instance (i.e. PR or PU-side cache), the actual load value is incremented by this constant on each selection of the corresponding PE. An update by the PE again resets the load information to the most up-to-date value. For example, let the load of a PE be 50% and the load increment 10%. On each of the following three selections, the load value is increased by another 10%, i.e. it will be 80% after that. Upon the next re-registration (and therefore load information update), the load value will be reset to the reported value (i.e. the latest known load state).

The LUD policy has yet to be examined in detail. Therefore, the goal of this paper is the performance analysis and evaluation of LUD. We also intend to identify the workload and system parameter configuration for which LUD provides a benefit over plain LU.

4 Quantifying a RSerPool System

In order to evaluate the performance of a RSerPool system, it is necessary to quantify it. We therefore use the model of [7], in which the service provider side of a RSerPool system consists of a pool of PEs. Each PE has a request handling *capacity*, which we define in the abstract unit of calculations per second[2]. 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.

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 $d_{\text{Processing}}$ (acceptance until finish):

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

That is, d_{Handling} not only incorporates the time required for processing the request, but also the latencies of queuing, server selection and protocol message transport. The *handling speed* is defined as: $\text{handlingSpeed} = \frac{\text{requestSize}}{d_{\text{handling}}}$. 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 PoolCapacity) as:

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

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 [3, 7] for details).

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

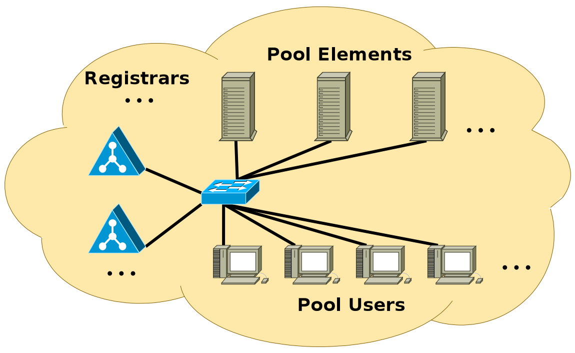

Figure 3. The Simulation Setup

5 The Simulation Setup

For the performance analysis, the RSerPool simulation model RSPSIM [3, 7] has been used. This model is based on the OMNeT++ [25] simulation environment and contains the protocols ASAP [22] and ENRP [26], a PR module and PE as well as 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 [6, 9].

Unless otherwise specified, the basic simulation setup – which is also presented in figure 3 – uses the following parameter settings:

- The target system utilization is 80%; the 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; each providing a capacity of 10^6 calculations/s (i.e. we use a homogeneous capacity distribution). We have set that each PE can handle up to 4 requests simultaneously. Therefore, the load increment of LUD is 25%. Further requests get rejected [28].
- The inter-component network delay is 500ms, which is realistic for satellite-based connections (we will analyse latency variation in subsection 6.1).
- We use a single PR only, since we do not examine failure scenarios here (we will analyse the PR number variation in subsection 6.3).
- 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 our results – including the computation of

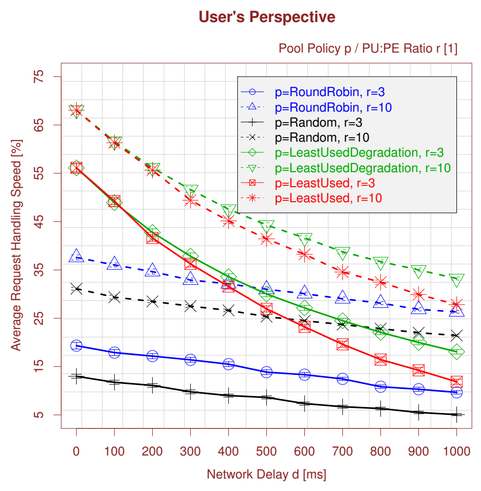

Figure 4. Variation of the Network Delay

95% confidence intervals – and plotting. Each resulting plot shows the average values and their corresponding confidence intervals.

6 Performance Analysis

Clearly, the intention of the LUD policy is to be able to perform better in case of synchronization latency. Therefore, our first simulation in subsection 6.1 provides a proof of concept under varying network delay for selected workload settings. An analysis of changing the workload parameter settings will be presented in subsection 6.2. The performance impact of modifying the number of PRs will be shown in subsection 6.3 and the cache usage will be analysed in subsection 6.4.

6.1 Variation of the Network Delay

The handling speed results of varying the inter-component latency are presented in figure 4. Since the utilization always remains at 80% (target utilization), a plot has been omitted. As already observed in [7], it can be observed that a smaller setting of the PU:PE ratio r is more critical and therefore leads to a slower handling speed: the smaller r, the higher the per-PU load put on the system. Therefore, inappropriate scheduling leads to queuing and increases the overall request handling time (see equation 1). Furthermore, the ranking of the policies (LU is better than RR, RR is better than RAND) can be observed.

Clearly, the adaptive LU policy has a better performance than the non-adaptive RR and RAND. However, the higher the network latency, the more out-of-date the load information in the handlespace when it actually gets used for PE selection. In this case, the LUD policy achieves a significant benefit: e.g. at 500ms (which is quite realistic for inter-continental and satellite-based connections) for r=10, the speed increases from 41% to 45% (a gain of ca. 10%); for r=3, it increases from 27% to 31% – which is even a gain of 15%. That is, the LUD policy significantly improves the service performance for the user.

In summary, the behaviour of the new LUD policy is as expected for the selected workload settings in the proof-of-concept simulation. But how will a variation of the workload parameters (PU:PE ratio, request interval and request size) influence its performance?

6.2 Variation of the Workload Parameters

The PU:PE ratio is the most critical workload parameter (see [7] for an analysis for RR, RAND and LU), since it defines the parallelism in request handling. Since the RR and RAND results are already known, we only show LU and LUD in the following. Figure 5 presents the utilization (left-hand side) and handling speed (right-hand side) results for varying the PU:PE ratio r from 1 to 10 for different request interval settings. Clearly, smaller settings of r lead to a reduced handling speed. However, it also sinks for larger settings: according to equation 2, the request size decreases with a rising PU:PE ratio for a fixed request interval setting. That is, a small request interval in combination with large PU:PE ratio results in very short requests. Clearly, this leads to significant overhead for transmission in equation 1 (i.e. the handling speed gets low). Comparing LUD and LU, a significant gain can be observed: while e.g. a setting of r=7 and a request interval of i=10 is critical for LU and leads to an unusable performance (a handling speed of almost 0%; utilization is less than 80%), the system performance is acceptable for LUD (handling speed of 19% at the target utilization).

Taking a look at the performance results for varying the request interval (as shown in figure 6), it can be observed that the gain achieved by LUD is highest between a request interval of about i=5 (for smaller i, the request size gets too critical – the utilization drops below 80%) and about i=20 (for larger settings of i, the network delay gets negligible) for r=3 and r=10. However, for r=1 (see also the left-hand plot), the results of LU and LUD are equal: here, each PU should be mapped to an exclusive PE. This is already achieved by the round-robin selection among least-loaded PEs [6], which has – for r=1 – an effect similar to LUD.

Plots for the request size variation have been omitted, since the results can be derived from the previous results (using equation 2) and therefore provide no new insights.

In summary, the LUD policy is especially useful for situations where a small PU:PE ratio occurs in combination with a small request interval: this leads to a significant penalty on requests being scheduled inappropriately due to deprecated load information.

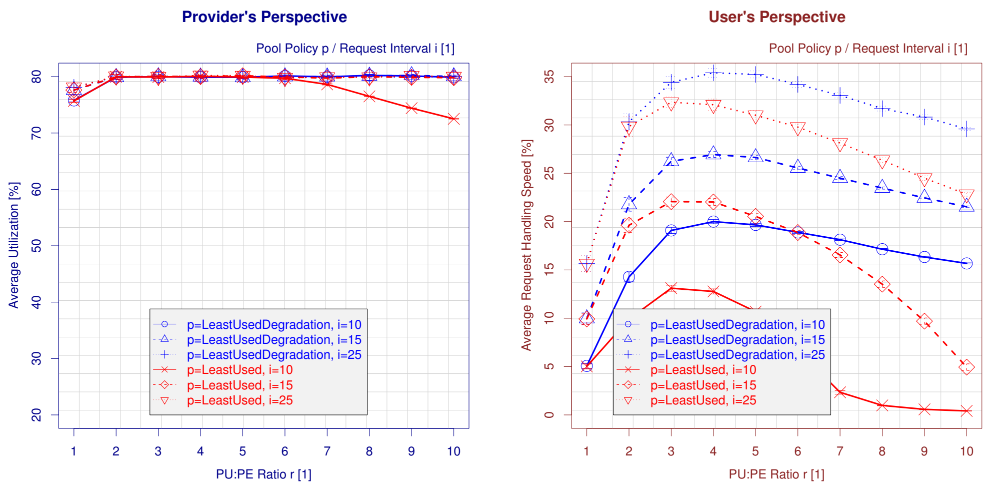

Figure 5. Variation of the PU:PE Ratio

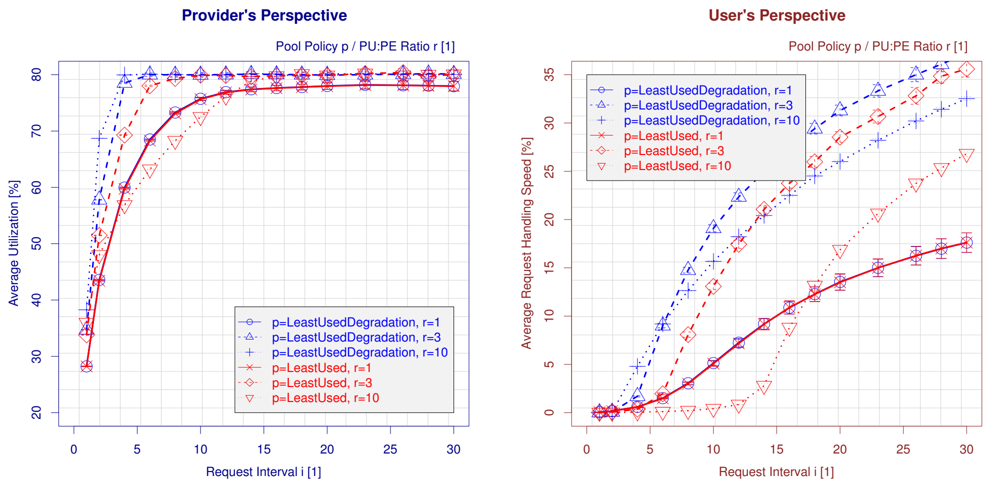

Figure 6. Variation of the Request Interval

6.3 Varying the Number of Registrars

For redundancy reasons, a RSerPool system has to contain multiple PRs[3]. Since each PR is an independent selection component, the number of PRs has an impact on the performance. In order to demonstrate this effect, figure 7 shows the handling speed for varying the number of PRs. An utilization plot has been omitted, since it is 80% (target utilization) for the shown values. The PUs have been distributed equally among the PRs.

As already observed in [7], the number of PRs has no impact on the RAND policy: this policy is "stateless" [13], i.e. the content of the next selection does not depend on the current one. On the other hand, RR and LU as well as LUD are "stateful" [13]. That is, since there is no synchronization among the PRs on which

[3]A single PR would be a single point of failure.

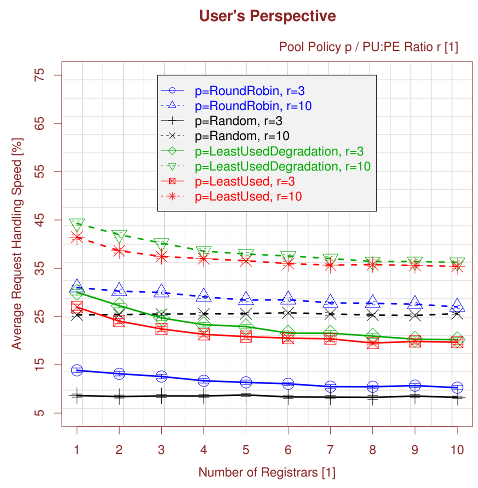

Figure 7. Scaling the Number of Registrars

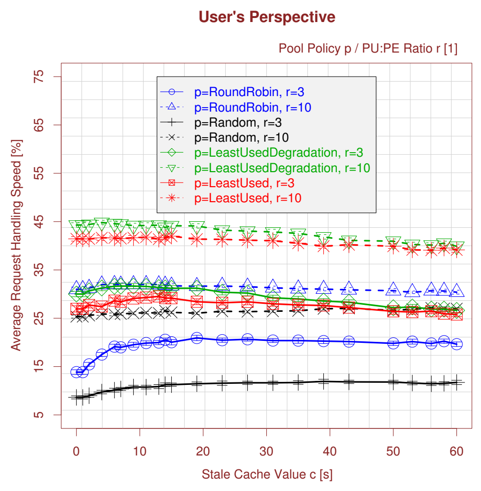

Figure 8. Using the Handle Resolution Cache

PEs to select next, the performance slightly decreases: e.g. PR #1 and PR #2 could simultaneously select the same least-loaded PE for LU. Nevertheless, for a realistic number of PRs – e.g. 2 or 3 – the LUD policy still achieves a significant performance gain over LU. But what about the PU-side cache – which, if used, also acts as an independent selection component? In particular, the number of caches is much higher than the number of PRs!

6.4 Using the Handle Resolution Cache

While the general recommendation on cache configuration is to turn it off [7], it gets useful in scenarios having a significant network delay (which is realistic for critical services being distributed globally [10]) and short requests. In such situations, the cache can avoid time-consuming PR queries. This is especially useful when requests get rejected by their selected PE during the startup phase (see section 4) and further trials at other PEs become necessary [7].

In order to show the impact of the cache usage on the system performance, figure 8 presents the handling speed of an example simulation. A plot for the system utilization has been omitted, since the target utilization of 80% is reached. Therefore, it is important that small settings of the stale cache value even increase the handling speed of all policies, in particular if r is also small: according to equation 2, the request interval rises proportional to the PU:PE ratio for a fixed request size. That is, the smaller r, the more critical the workload settings. In this case, the cache reduces the startup time of requests, which results in a reduction of "request jams" due to excessive waiting in the queue.

For the adaptive policies, the load states clearly get deprecated the higher the setting of the stale cache value c. For sufficiently small settings (here: $c \leq 20$) – which in particular cover the startup phase and therefore the main use case of the PU-side cache – the LUD policy achieves almost a constant gain of about 10%. Clearly, for higher settings of c, LUD becomes more and more incapable to compensate the deprecated state information and the performance gain shrinks.

7 Conclusions

In this paper, we have analysed the performance of the Least Used with Degradation (LUD) pool policy. This new policy extends the plain Least Used selection with load increment information, which is used by selection instances to estimate the deviation from the latest known load state in case of delayed information updates. The LUD policy gets useful in case of network delay, which is very realistic when PEs are distributed over a large geographical area for redundancy reasons. Our simulations have shown that LUD can achieve a significant gain over the plain LU policy – particularly in case of critical workload parameter settings.

We are currently validating our simulative performance results in real-life scenarios by using our RSerPool prototype implementation RSPLIB [3, 14] in the PLANETLAB. First results of our PLANETLAB-based evaluations and performance optimizations can be found in [3, 10]. Furthermore, we are actively promoting the IETF standardization process of RSerPool: the LUD policy is now part of the Internet Draft defining the pool policies [23].

References

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

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

[5] T. Dreibholz, A. Jungmaier, and M. Tüxen. A new Scheme for IP-based Internet Mobility. In *Proceedings of the 28th IEEE Local Computer Networks Con-*

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

[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 Generation 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 and E. P. Rathgeb. Reliable Server Pooling – A Novel IETF Architecture for Availability-Sensitive Services. In *Proceedings of the 2nd IEEE International Conference on Digital Society (ICDS)*, Sainte Luce/Martinique, Feb. 2008.

[13] 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.

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

[15] T. Dreibholz and X. Zhou. Definition of a Delay Measurement Infrastructure and Delay-Sensitive Least-Used Policy for Reliable Server Pooling. Internet-Draft Version 00, IETF, Individual Submission, June 2007. draft-dreibholz-rserpool-delay-00.txt, work in progress.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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.

[22] R. Stewart, Q. Xie, M. Stillman, and M. Tüxen. Aggregate Server Access Protcol (ASAP). Internet-Draft Version 17, IETF, RSerPool Working Group, Sept. 2007. draft-ietf-rserpool-asap-17.txt, work in progress.

[23] M. Tüxen and T. Dreibholz. Reliable Server Pooling Policies. Internet-Draft Version 06, IETF, RSerPool Working Group, Sept. 2007. draft-ietf-rserpool-policies-06.txt, work in progress.

[24] 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.

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

[26] Q. Xie, R. Stewart, M. Stillman, M. Tüxen, and A. Silverton. Endpoint Handlespace Redundancy Protocol (ENRP). Internet-Draft Version 17, IETF, RSerPool Working Group, Sept. 2007. draft-ietf-rserpool-enrp-17.txt, work in progress.

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

[28] 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.

[29] X. Zhou, T. Dreibholz, and E. P. Rathgeb. 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, Dec. 2007.

[30] X. Zhou, T. Dreibholz, and E. P. Rathgeb. Improving the Load Balancing Performance of Reliable Server Pooling in Heterogeneous Capacity Environments. In *Proceedings of the 3rd Asian Internet Engineering Conference (AINTEC)*, Phuket/Thailand, Nov. 2007.