Reliable Server Pooling –
A Novel IETF Architecture for Availability-Sensitive Services*

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

Abstract

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

The goal of our paper is – after presenting a short overview of the RSerPool architecture and its application cases – to provide a quantitative, application-independent performance analysis of RSerPool's server failure handling capabilities with respect to important adaptive and non-adaptive load balancing strategies. We will also analyse the impact of RSerPool protocol parameters on the performance of the server failure handling functionalities and the network overhead.

Keywords: *RSerPool, Availability, Redundancy, Failover, Server Selection*

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 [21] – 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) 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 [1,23]), but their combination into one application-independent framework is.

Server redundancy leads to the issues of load distribution and load balancing [17], which are also covered by RSerPool [5, 9]. But unlike solutions in the area of GRID and high-performance computing [16], the RSerPool architecture is intended to be lightweight. That is, RSerPool may only introduce a small computation and memory overhead for the management of pools and sessions [9, 13]. In particular, this means the limitation to a single administrative domain and only taking care of pool and session management – but not for higher-level tasks like data synchronization, locking and user management. These tasks are considered to be application-specific. On the other hand, these restrictions allow for RSerPool components to be situated on low-end embedded devices like routers or telecommunications equipment.

While there has already been some research on the applicability of RSerPool for applications like SCTP-based mobility [7], web server pools [5], real-time distributed computing [10, 12, 15, 28–30], Voice over IP [2], IP Flow Information Export (IPFIX) [6], and battlefield networks [25], a generic, application-independent performance analysis of its failover handling capabilities is still missing. In particular, it is necessary to evaluate the different RSerPool mechanisms for session monitoring, server maintenance and failover support – as well as the corresponding system parameters – in order to show how to achieve a good system performance at a reasonably low maintenance overhead. The goal of our work is an application-independent quantitative characterization of RSerPool systems, as well as a generic sensitivity analysis on changes of workload and system parameters. Especially, we intend to identify the critical parameter ranges in order to provide guidelines for design and configuration of efficient RSerPool-based services.

2 The RSerPool Architecture

Figure 1 illustrates the RSerPool architecture, as defined in [19]. It consists of three major component classes: servers of a pool are called *pool elements* (PE). Each pool is identified by a unique *pool handle* (PH) in the handlespace, i.e. the set of all pools. The handlespace is managed by *registrars* (PR). PRs of an operation scope synchronize their view of the handlespace using the Endpoint haNdlespace

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

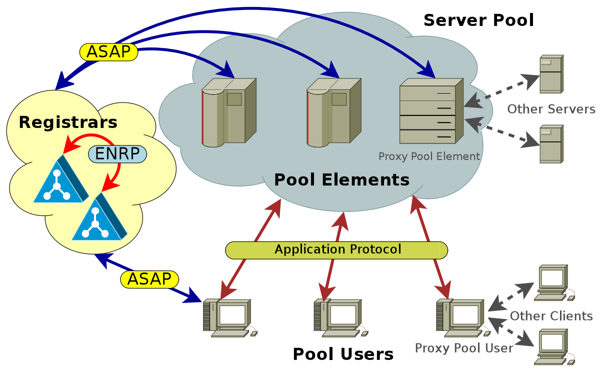


Figure 1. The RSerPool Architecture

Redundancy Protocol (ENRP [27]), transported via SCTP[1]. An operation scope has a limited range, e.g. a company or organization; RSerPool does not intend to scale to the whole Internet. This restriction results in a very small pool management overhead (see also [9, 13]), which allows to host a PR service on routers or embedded systems. Nevertheless, it is assumed that PEs can be distributed worldwide, for their service to survive localized disasters [14, 30].

A client is called *pool user* (PU) in RSerPool terminology. To use 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]). The PR selects the requested list of PE identities using a pool-specific selection rule, called *pool policy*. Adaptive and non-adaptive pool policies are defined in [24], relevant for this paper are the non-adaptive policies Round Robin (RR) and Random (RAND) and 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 [9].

A PE can register into a pool at an arbitrary PR of the operation scope, again using ASAP transported via SCTP. The chosen PR becomes the *Home PR* (PR-H) of the PE and is also responsible for monitoring the PE's health by *endpoint keep-alive* messages. If not acknowledged, the PE is assumed to be dead and removed from the handlespace. Furthermore, PUs may report unreachable PEs; if the threshold MAX-BAD-PE-REPORT of such reports is reached, a PR may also remove the corresponding PE. The PE failure detection mechanism of a PU is application-specific.

While RSerPool allows use of arbitrary mechanisms to realize the application-specific resumption of an interrupted session on a new server, it contains only one built-in mechanism: client-based state sharing [3, 11]. Using this feature, a PE can send its current session state to the PU in form of a state cookie. The PU stores the latest state cookie and provides it to a new PE upon failover. Then, the new PE simply restores the state described by the cookie. Cryptographic methods can ensure the integrity, authenticity and confidentiality of the state information.

[1] Stream Control Transmission Protocol, see [18].

3 Quantifying a RSerPool System

The service provider side of a RSerPool-based service consists of a pool of PEs, using a certain server selection policy. Each PE has a request handling *capacity*, which we define in the abstract unit of calculations per second. Depending on the application, an arbitrary view of capacity can be mapped to this definition, e.g. CPU cycles or memory usage. Each request consumes a certain number of calculations, we call this number the *request size*. A PE can handle multiple requests simultaneously, in a processor sharing mode (multi-tasking principle).

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.

Clearly, the user-side performance metric is the handling speed – which should be as fast as possible. The total delay for handling a request d_{handling} is defined as the sum of queuing delay, startup delay (dequeuing until reception of acceptance acknowledgement) and processing time (acceptance until finish) as illustrated in figure 2. The *handling speed* (in calculations/s) is defined as:

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

For convenience reasons, the handling speed can be represented in % of the average PE capacity. Clearly, in case of a PE failure, all work between the last checkpoint and the failure is lost and has to be re-processed later. A failure has to be detected by an application-specific mechanism (e.g. keep-alives) and a new PE has to be chosen and contacted for session resumption.

Using the definitions above, the system utilization – which is the provider-side performance metric – can be calculated:

$$\text{systemUtilization} = \text{puToPERatio} * \frac{\frac{\text{requestSize}}{\text{requestInterval}}}{\text{peCapacity}}$$

In practice, a well-designed RSerPool system is dimensioned for a certain *target system utilization*. [5, 10] provide a detailed discussion of this subject.

4 The Simulation Scenario Setup

For our performance analysis, we have developed our simulation model RSPSIM [5] using OMNeT++ [26], containing full implementations of the protocols ASAP [22] and ENRP [27], a PR module and PE and PU modules modelling the request handling scenario defined in section 3. The scenario setup is shown in figure 3: all components are interconnected by a switch. Network delay is introduced by link latency only. Component latencies are neglected, since they are not significant (as shown in [9]). We further assume

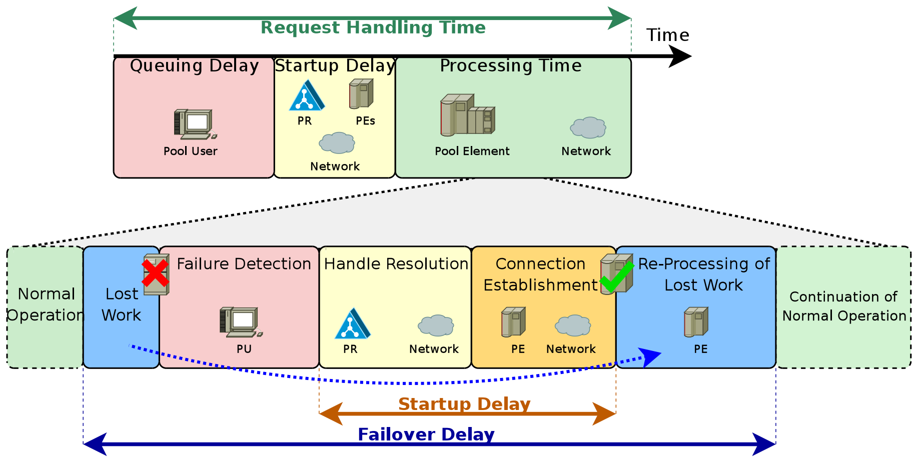


Figure 2. Request Handling Delays

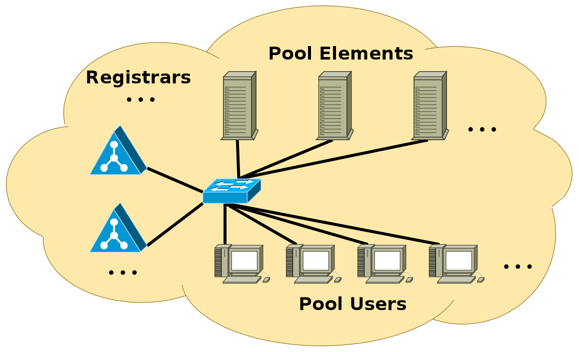


Figure 3. The Simulation Setup

sufficient network bandwidth for pool management and applications. Since an operation scope is limited to a single administrative domain, QoS mechanisms may be applied.

Unless otherwise specified, the used target system utilization is 60%, i.e. there is sufficient over-capacity to cope with PE failures. For the LU policy, we define *load* as the current number of simultaneously handled requests. The capacity of a PE is 10^6 calculations/s, the average request size is 10^7 calculations. Both parameters use negative exponential distribution – for a generic parameter sensitivity analysis being independent of a particular application [12]. We use 10 PEs and 100 PUs, i.e. the PU:PE ratio is 10. This is a non-critical setting for the examined policies, as shown in [10].

Session health monitoring is performed by the PUs using keep-alive messages in a *session keep-alive interval* of 1s, to be acknowledged by the PE within a *session keep-alive timeout* of 1s (parameters evaluated in subsection 5.3). Upon a simulated failure, the PE simply disappears and reappears immediately under a new transport address, i.e. the overall pool capacity remains constant. Client-based state sharing is applied for failovers; the default cookie interval is 10% of the request size (i.e. 10^6 calculations; parameter is evaluated in subsection 5.5). Work not being protected by a checkpoint has to be re-processed on a new PE.

In this paper, we neglect PR failures and therefore use a single PR only. All failure reports by PUs are ignored (i.e. MAX-BAD-PE-REPORT=∞) and the endpoint keep-alive interval and timeout are 1s (parameters evaluated in subsection 5.4). The inter-component network delay is 10ms (realistic for connections within a limited geographical area, see [14]). The simulated real-time is 60 minutes; each simulation is repeated 24 times with different seeds to achieve statistical accuracy. The post-processing of results, i.e. computation of 95% confidence intervals and plotting, has been performed using GNU R [20].

5 Results

As shown in figure 2, two components contribute to the failure handling time: the failure detection delay and the re-processing effort for lost work.

5.1 Dynamic Pools

In the ideal case, a PE informs its PU of an oncoming shutdown, sets a checkpoint for the session state (e.g. by a state cookie [3]) and performs a de-registration at a PR. Then, no re-processing effort is necessary. This situation, as shown for the handling speed in figure 4, becomes critical only for a very low PE MTBF (Mean Time Between Failure; here: given in average request handling times) in combination with network delay (i.e. the failover to a new PE is not for free). As being observable for failure-free scenarios (see [10,12]), the best performance is again provided by the adaptive LU policy, due to PE load state knowledge. However, the RR performance converges to the RAND result for a low MTBF: in this case, there is no stable list of PEs to select from in turn – the selection choices become more and more random. Since the system utilization results are similar to the handling speed behaviour, a plot has been omitted.

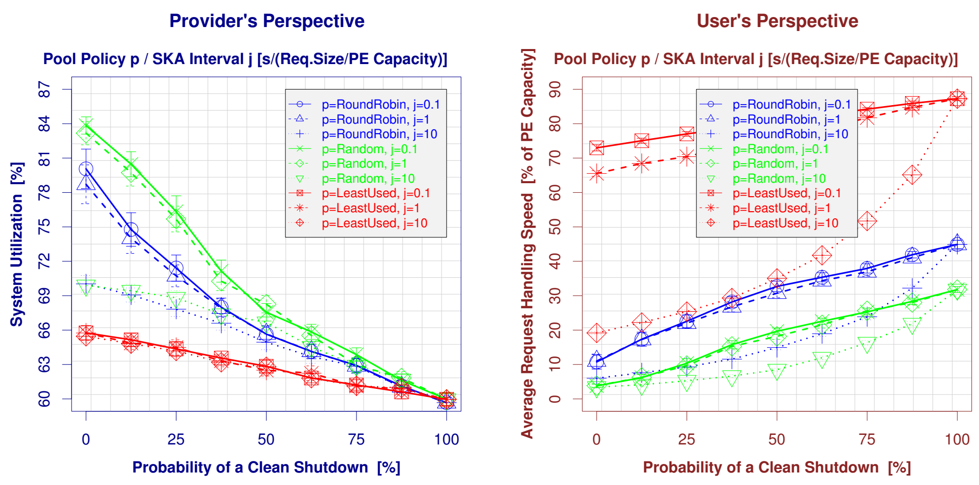


Figure 5. The Impact of Clean Shutdowns

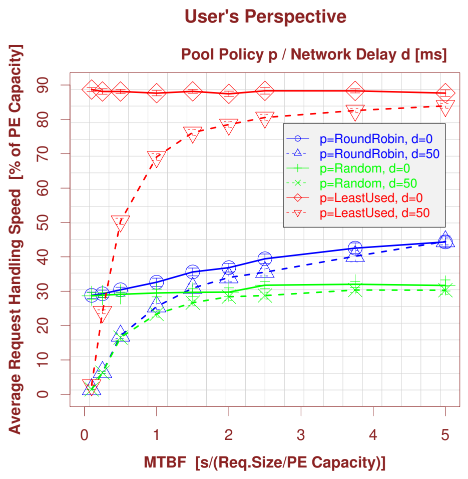


Figure 4. The Performance for Dynamic Pools

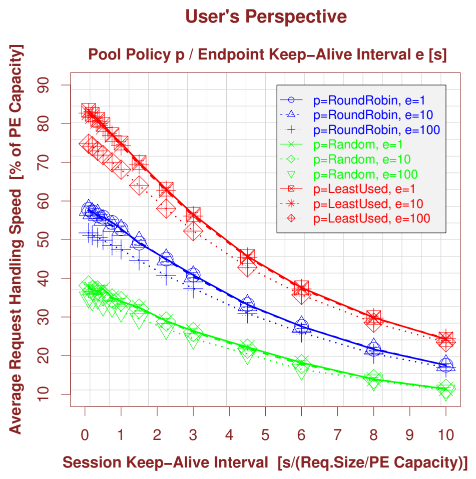


Figure 6. The Impact of Session Monitoring

5.2 De-Registrations and Failures

In real scenarios, PEs may fail without warning. That is, a PU has to detect the failure of its PE in order to trigger a failover. For the simulated application, this detection mechanism has been realized by keep-alive messages. The general effects of a decreasing amount of "clean" shutdowns (i.e. the PE simply disappears) are presented in figure 5. Clearly, the less "clean" shutdowns, the higher the re-processing effort for lost work: this leads to a higher utilization and lower handling speed. As expected, this effect is smallest for LU (due to superior load balancing) and lowest for RAND. There is almost no reaction of the utilization to an increased session keep-alive interval (given in average request handling times): a PU does not utilize resources while it waits for a timeout. However, the impact on the handling speed is significant: waiting increases the failover handling time and leads to a lower handling speed. For that reason, a tight session health monitoring interval is crucial for the system performance.

5.3 Session Health Monitoring

To emphasize the impact of the session health monitoring granularity, figure 6 shows the handling speed results for varying this parameter in combination with the endpoint keep-alive interval, for a target utilization of 40% (higher settings become critical too quickly). The utilization results have been omitted, since they are obvious. Again, the performance results for varying the policy and session keep-alive interval reflect the importance of a quick failure detection – regardless of the policy used. Furthermore, it has to be noted that a small monitoring granularity does not necessarily increase overhead: e.g. a PU requesting transactions by a PE could simply set a transaction timeout. In this case,

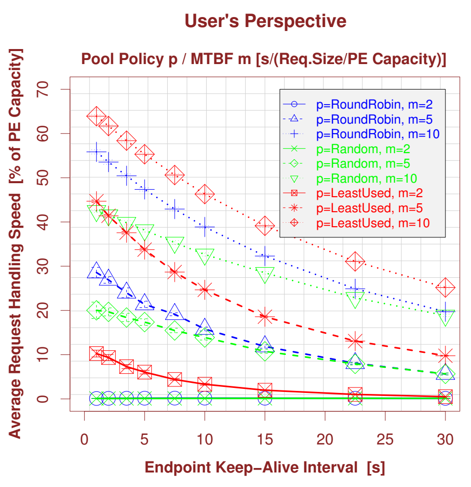


Figure 7. The Impact of PE Health Monitoring

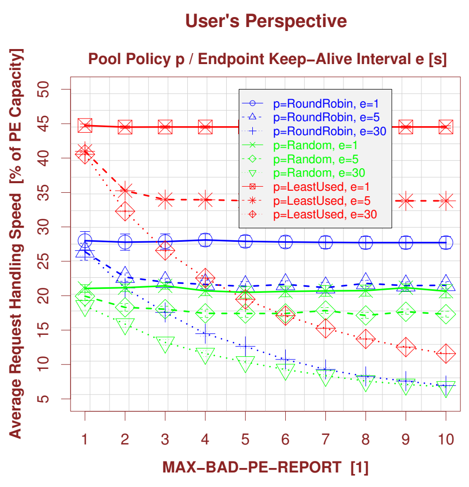


Figure 8. Utilizing Failure Reports

session monitoring even comes for free. Unlike the session monitoring, the impact of the PR's endpoint keep-alive interval is quite small here: even a difference of two orders of magnitude only results in at most a performance difference of 10%.

5.4 Server Health Monitoring

The endpoint keep-alive interval gains increasing importance if the request size gets small. Then, the startup delay becomes significant, as illustrated in figure 2. In order to show the general effects of the PE health monitoring based on endpoint keep-alives, figure 7 presents the handling speed results for a request size:PE capacity ratio of 1 and a target system utilization of 25% (otherwise, the system becomes unstable too quickly). Since the results for the system utilization are as expected, a plot has been omitted.

While the policy ranking remains as expected, it is clearly observable that the higher the endpoint keep-alive interval and the smaller the MTBF, the more probable is the selection of an already failed PE. That is, the PU has to detect the failure of its PE by itself (by session monitoring, see subsection 5.3) and trigger a new PE selection. The result is a significantly reduced request handling speed. Therefore, a PR-based PE health monitoring becomes crucial for such scenarios. But this monitoring results in network overhead for the keep-alives and acknowledgements as well as for the SCTP transport. So, is there a possibility to reduce this overhead?

The mechanism for overhead reduction is to utilize the session health monitoring (which is necessary anyway, as shown in subsection 5.3) for PE monitoring by letting PUs report the failure of PEs. If MAX-BAD-PE-REPORT failure reports have been received, the PE is removed from the handlespace. The effectiveness of this mechanism is demonstrated by the results in figure 8 (for the same parameters as above): even if the endpoint keep-alive overhead is reduced to $\frac{1}{30}$th, there is only a handling speed decrease of about 4% for MAX-BAD-PE-REPORT=1. The higher MAX-BAD-PE-REPORT, the more important the endpoint keep-alive granularity. However, while the failure report mechanism is highly effective for all three policies, care has to be taken for security: trusting in failure reports gives PUs to power to impeach PEs!

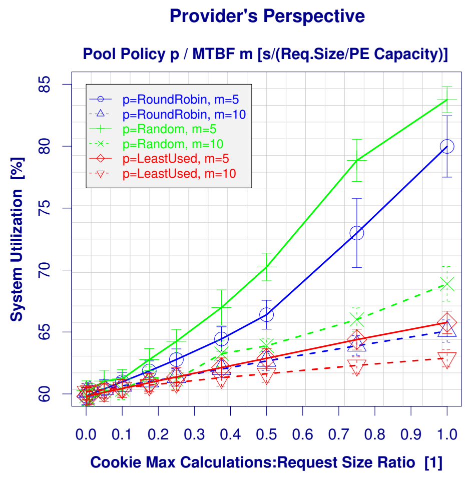
5.5 Failover Handling

After detecting a PE failure and contacting a new server, the session state has to be restored for the re-processing of lost work and the application resumption. The simplest mechanism is "abort and restart" [5]: the session is restarted from scratch. Of course, this mechanism is only useful if the requests are small and the PE MTBF is sufficiently large.

Client-based state sharing [3, 11] using state cookies offers a simple but effective solution for the state transfer. It is applicable as long as the state information remains sufficiently small[2]. To show the general effects of using this mechanism, figure 9 presents the performance results for varying the cookie interval $\mathrm{CookieMaxCalcs}$ (given as the ratio between the number of calculations and the average request size) for different policy and MTBF settings. The larger to cookie transmission interval and the smaller the PE MTBF, the lower the system performance: work (i.e. processed calculations) not being conserved by the cookie is lost. This results in an increased utilization, due to reprocessing effort. Furthermore, this additional workload leads to a reduction of the request handling speed. Clearly, the better a policy's load balancing capabilities, the better the system utilization and request handling speed (LU better than RR better than RAND, as for failure-free scenarios [10, 12]).

In order to configure an appropriate cookie interval, the overhead of the state cookie transport has to be taken into account – which may range from a few bytes up to kilobytes (subsubsection 9.4.2.2 of [5] contains some estimations). The average loss of calculations per failure can be

[2]The maximum state cookie size is less than 64K [22].



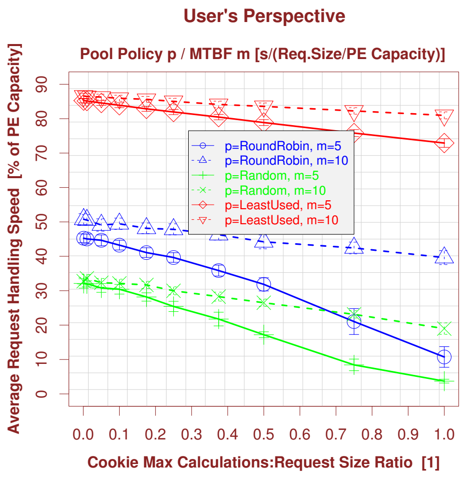


Figure 9. Using State Cookies for the Session Failover

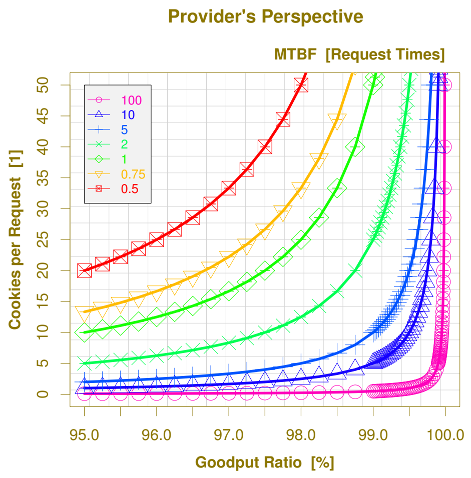


Figure 10. The Number of Cookies

estimated as the half cookie interval CookieMaxCalcs (in calculations, as multiple of the average request size):

$$\text{AvgLoss} = \frac{\text{CookieMaxCalcs}}{2}.$$

Given an approximation of the PE MTBF (in average request handling times) and AvgCap the average PE capacity, the goodput ratio can be estimated as follows:

$$\text{Goodput} = \frac{(\text{MTBF} * \text{AvgCap}) - \text{AvgLoss}}{\text{MTBF} * \text{AvgCap}}.$$

Then, the cookie interval CookieMaxCalcs for a given goodput ratio is:

$$\text{CookieMaxCalcs} = -2 * \text{MTBF} * \text{AvgCap} * (\text{Goodput} - 1). \quad (1)$$

Figure 10 illustrates the cookies per request (i.e. $\frac{1}{\text{CookieMaxCalcs}}$) for varying the goodput ratio and MTBF in equation 1. As shown for realistic MTBF values (i.e. $\gg$ a request time), the number of cookies per request keeps small unless the required goodput ratio becomes extremely high: accepting a few lost calculations (e.g. a goodput ratio of 98% for a MTBF of 10 request times) – and the corresponding re-processing effort on a new PE – leads to an acceptable overhead while still achieving a good system performance.

6 Conclusions

In this paper, we have provided a quantitative performance analysis of the server failure handling performance in RSerPool systems. Since server redundancy implies load distribution and balancing strategies, we have also analysed the behaviour of different server selection policies. In general, the adaptive LU policy provides the best performance, due to server load information. RR is better than random selection as long as there is a stable list of servers; for a too small MTBF, its performance converges to the results of RAND.

Two factors influence the failover performance: the failure detection and the failover mechanisms. In any case, it is crucial to detect server failures as quickly as possible (e.g. by session keep-alives or an application-specific mechanism). The PR-based server health monitoring only gets important when the request size becomes small. Failure reports may be used to reduce its overhead significantly – if taking care of security. A very simple and quite effective failover mechanism is client-based state sharing. Configured appropriately, a good performance is achieved at small overhead.

As part of our future work, it will be necessary to analyse and evaluate the handling of PR failures by the ENRP protocol. Furthermore, our goal is to also validate our results in real-life scenarios. That is, we are going to perform PLANETLAB experiments by using our RSerPool prototype implementation RSPLIB [4, 5, 8]. Particularly, we will also evaluate the impact of changing network conditions.

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

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] T. Dreibholz and E. P. Rathgeb. RSerPool – Providing Highly Available Services using Unreliable Servers. In *Proceedings of the 31st IEEE EuroMirco Conference on Software Engineering and Advanced Applications*, pages 396–403, Porto/Portugal, Aug. 2005. ISBN 0-7695-2431-1.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

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

[19] 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.

[20] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna/Austria, 2005. ISBN 3-900051-07-0.

[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] 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.

[24] 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.

[25] 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.

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

[27] 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.

[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. 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.

[30] X. Zhou, T. Dreibholz, and E. P. Rathgeb. A New Server Selection Strategy for Reliable Server Pooling in Widely Distributed Environments. In *Proceedings of the 2nd IEEE International Conference on Digital Society (ICDS)*, Sainte Luce/Martinique, Feb. 2008.