Overview and Evaluation of the Server Redundancy and Session Failover Mechanisms in the Reliable Server Pooling Framework*

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

Abstract

The number of availability-critical Internet applications is steadily increasing. To support the development and operation of such applications, the IETF has recently defined a new standard for a common server redundancy and session failover framework: Reliable Server Pooling (RSerPool). The basic ideas of the RSerPool framework are not entirely new, but their combination into a single, resource-efficient and unified architecture is. Service availability achieved by the redundancy of servers 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 service application.

In this article – which is an extended version of our paper [1] – we first present an overview of the RSerPool architecture with a focus on the component failure detection and handling mechanisms. We will also shortly introduce the underlying SCTP protocol and its link redundancy features. After that, we will present a quantitative, application-independent performance analysis of the failure detection and session failover mechanisms provided by RSerPool, with respect to important adaptive and non-adaptive load balancing strategies.

Keywords: *Reliable Server Pooling, Service Availability, Redundancy, Session Failover, Server Selection*

1 Introduction and Related Work

Service availability is becoming increasingly important in today's Internet. But – in contrast to the telecommunications world, where availability is ensured by redundant links and devices [2] – 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 [3, 4]), but their combination into one application-independent framework is.

Server redundancy [5] leads to the issues of load distribution and load balancing [6], which are also covered by RSerPool [7, 8]. But unlike solutions in the area of GRID and high-performance computing [9], 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 [8, 10–12]. 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 have already been a number of publications on applicability and performance of RSerPool (see e.g. [7, 13–17]), a generic, application-independent performance analysis of its failover handling capabilities was 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

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

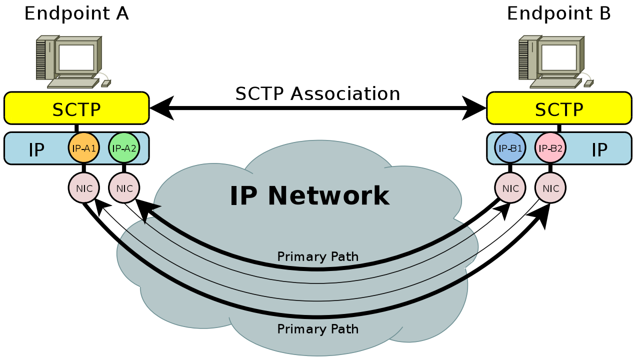

Figure 1. A Multi-Homed SCTP Association

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.

This article is an extended version of our conference paper [1]. It contains a broader overview of the redundancy mechanisms provided by RSerPool and the underlying SCTP protocol as well as a more detailed analysis of the session failover mechanisms.

The document is structured as follows: in Section 2, we present an overview of RSerPool and the underlying SCTP protocol. A generic quantification of RSerPool systems is introduced in Section 3. Using the RSPSIM simulation model and setup described in Section 4, we evaluate the server failure detection and handling features of RSerPool in Section 5.

2 The RSerPool Architecture

RSerPool is based on the SCTP transport protocol. Therefore, it is necessary to shortly introduce this protocol and its link failure handling features first.

2.1 Data Transport and Motivation

The Stream Control Transmission Protocol (SCTP, see [18, 19]) is a connection-oriented, general-purpose, unicast transport protocol which provides reliable transport of user messages. An SCTP connection is denoted as *association*. Each SCTP endpoint can use multiple IPv4 and/or IPv6 addresses to provide network fault tolerance. The addresses used by the endpoints are negotiated during association setup, a later update is possible using dynamic address reconfiguration (Add-IP, see [20]). Add-IP can also be applied to allow for endpoint mobility. This link redundancy feature is called *multi-homing* [21, 22] and illustrated in Figure 1. An endpoint sees each remote address as unidirectional *path*. In each direction, one of the paths is selected as so-called *primary path*. This path is used for the transport of user data. The other paths are backup paths, which are used for retransmissions only. Upon failure of the primary path, SCTP selects a new primary path from the set of possible backup paths. That is, as long as there is at least one usable path in each direction, the association remains usable despite of link failures. However, multi-homing cannot protect a service against endpoint failures. To cope with this problem, the IETF has defined the RSerPool architecture on top of SCTP.

2.2 Architecture

Figure 2 illustrates the RSerPool architecture, as defined in [23]. 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 *pool registrars* (PR), which are also shortly denoted as *registrars*. PRs of an operation scope synchronize their view of the handlespace using the Endpoint haNdlespace Redundancy Protocol (ENRP [24]), transported via SCTP. 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 [8, 10, 25]), 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 [26, 27].

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 [28, 29]). 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 [30]; relevant for this article are the non-adaptive policies Round Robin and Random and the adaptive policy Least Used. Least Used selects the least-used PE, according to up-to-date load information. The actual definition of *load* is application-specific: for each pool the corresponding application has to specify the actual meaning of *load* (e.g. CPU utilization or storage space usage) and present it to RSerPool in form of a numeric value. Among multiple least-loaded PEs, Least Used applies Round Robin selection (see also [8]). Some more policies are evaluated in [26, 27, 31].

A PE can register into a pool at an arbitrary PR of the operation scope, again using ASAP transported via SCTP. The

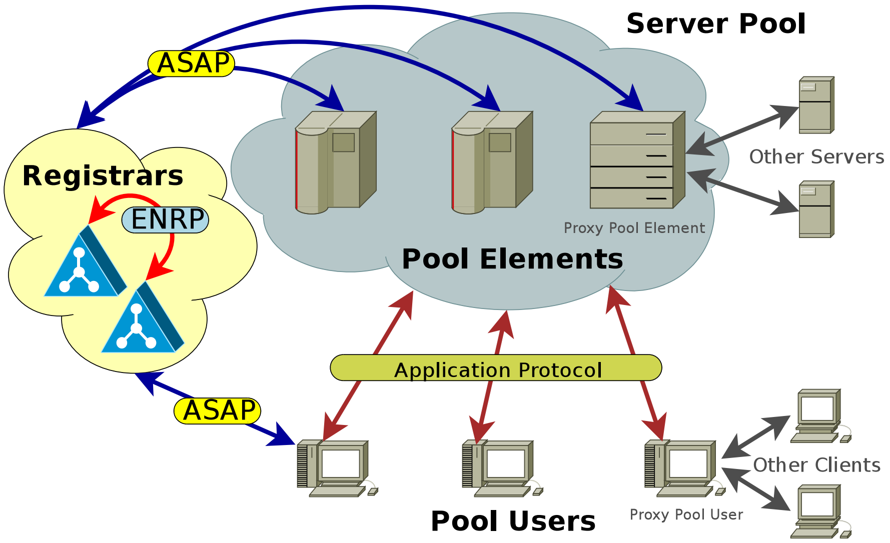

Figure 2. The RSerPool Architecture

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.

Proxy Pool Elements (PPE) allow for the usage of non-RSerPool servers in RSerPool-based environments. Respectively, non-RSerPool clients can use Proxy Pool Users (PPU) to access RSerPool-based services.

2.3 Protocol Stack

The protocol stack of the three RSerPool components is illustrated in Figure 3: a PR provides ENRP and ASAP services to PRs and PEs/PUs respectively. But between PU and PE, ASAP provides a Session Layer protocol in the OSI model[1]. From the perspective of the Application Layer, the PU side establishes a *session* with a pool. ASAP takes care of selecting a PE of the pool, initiating and maintaining the underlying transport connection and triggering a failover procedure when the PE becomes unavailable.

2.4 Session Failover Handling

While RSerPool allows the usage 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. This mechanism has been proposed by us in our paper [32] and it is now part of the ASAP standard [28]. Using this feature, which is illustrated in Figure 4, 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. For RSerPool itself, the cookie is opaque, i.e. only the PE-side application has to know about its structure and contents. The PU can simply handle it as a vector of bytes (However, as we will describe in Subsubsection 5.2.2, a more complex handling concept may improve application efficiency). Cryptographic methods can ensure the integrity, authenticity and confidentiality of the state information. In the usual case, this can be realized easily by using a pool key which is known by all PEs (i.e. a "shared secret").

2.5 Applications

The initial motivation of RSerPool has been to ensure the availability of SS7 (Signalling System No. 7, see [33]) services over IP networks. However, since component availability is a very common problem for computer network applications, RSerPool has been designed for application independence. The current research on applicability and performance of RSerPool includes application scenarios (described in detail by [7, Section 3.6]) like VoIP with SIP [17], SCTP-based mobility [34], web server pools [7, Section 3.6], e-commerce systems [32], video on demand [15], battlefield networks [16], IP Flow

[1] ASAP [28] is the first IETF standard for a Session Layer protocol.

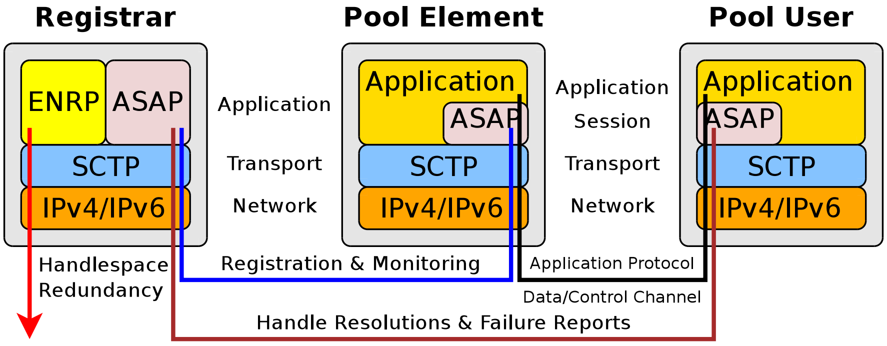

Figure 3. The RSerPool Protocol Stack

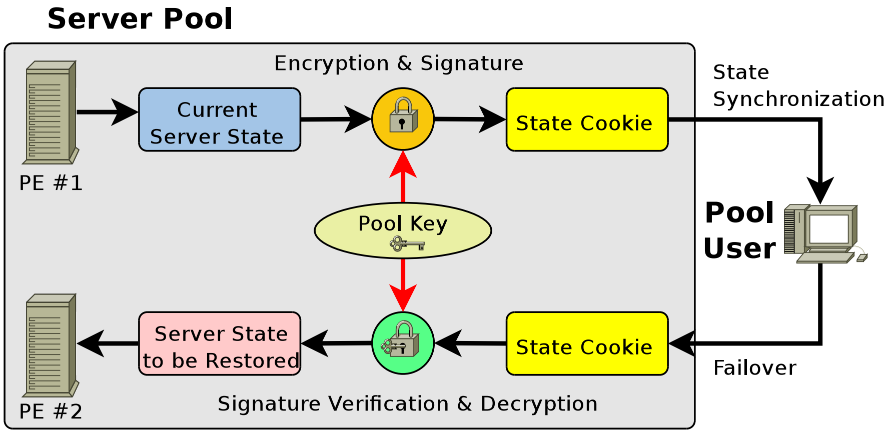

Figure 4. Client-Based State Sharing

Information Export (IPFIX) [35] and workload distribution [7, 13, 14, 26, 27, 31, 36–39].

Since RSerPool has just reached a major milestone by the publication of its basic protocol documents as RFCs in September 2008, a wide deployment of RSerPool-based applications is expected for the future [40].

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 5. 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.

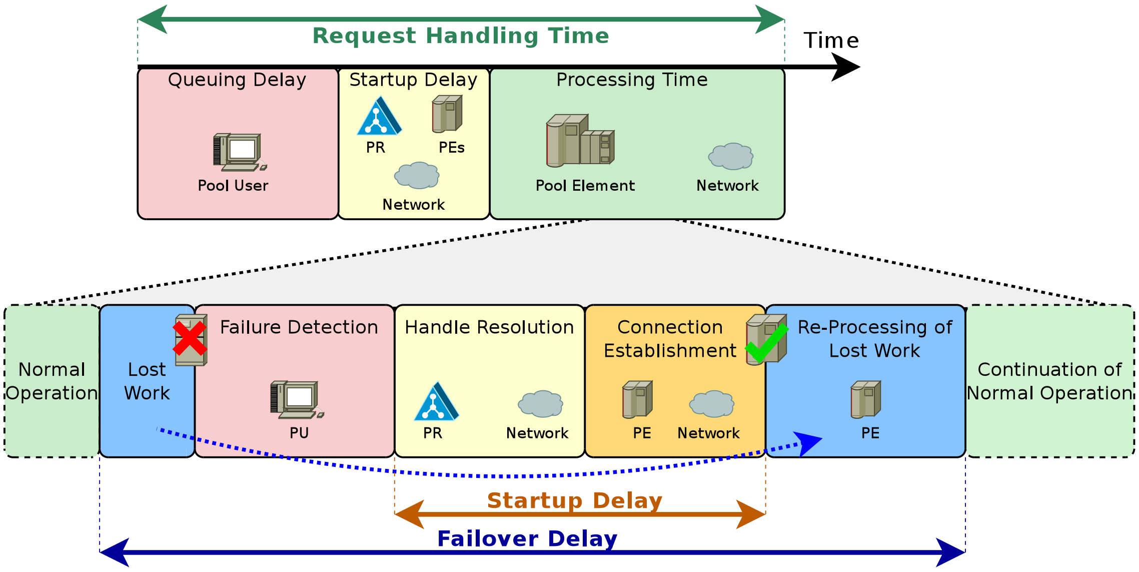

Figure 5. Request Handling Delays

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:

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

In practice, a well-designed RSerPool system is dimensioned for a certain *target system utilization*. [7, 14] provide a detailed discussion of this subject.

4 The Simulation Scenario Setup

For our performance analysis, we have developed our simulation model RSPSIM [7, 13] using OMNeT++ [41] and the SimProcTC [36] tool-chain, containing full implementations of the protocols ASAP [28, 29] and ENRP [24], a PR module and PE and PU modules modelling the request handling scenario defined in Section 3. The scenario setup is shown in Figure 6: 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 [8]). We further assume 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 Least Used 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 [14]. 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 [14].

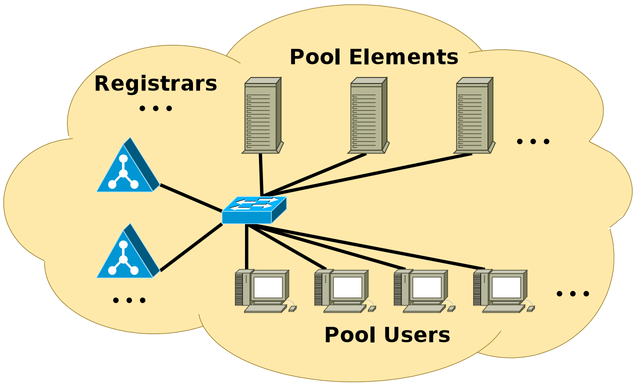

Figure 6. The Simulation Setup

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 Subsubsection 5.1.3). 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 Subsubsection 5.2.2). Work not being protected by a checkpoint has to be re-processed on a new PE upon session failover.

In this article, we neglect PR failures and therefore use a single PR only (for details on PR failures, see [11, 12]). 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 Subsubsection 5.1.4). The inter-component network delay is 10ms (realistic for connections within a limited geographical area, see [26]). 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 [42].

5 Results

As shown in Figure 5, two components contribute to the failure handling time:

1. The failure detection delay and
2. The failure handling delay (i.e. re-processing effort for lost work).

Therefore, we will examine the failure detection procedures in the following Subsection 5.1 and failover handling procedures in Subsection 5.2.

5.1 Failure Detection

5.1.1 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 [1, 32]) and performs a de-registration at a PR. Then, no re-processing effort is necessary. This situation, as shown for the handling speed on the right-hand side of Figure 7, 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 [14]), the best performance is again provided by the adaptive Least Used policy, due to PE load state knowledge. However, the Round Robin performance converges to the Random 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.

The results for the system utilization, shown on the left-hand side of Figure 7, are very similar to the handling speed behaviour: except for extremely low MTBF settings, the utilization remains at the expected target system utilization of 60%.

An approach to utilize the failover capabilities of RSerPool-based systems for improving the server selection performance is described by [38, 39]: “Reject and Retry”. When a PE is highly loaded due to non-optimal server selection (e.g. due to temporary capacity changes), it may simply reject a new request. The failover capabilities of RSerPool then retry at another PE. Since the failover handling of RSerPool is quite efficient, this can lead – despite of the failure handling – to a significant performance improvement in certain scenarios.

5.1.2 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 8. 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 Least Used (due to superior load balancing) and highest for Random. 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.

5.1.3 Session Health Monitoring

To emphasize the impact of the session health monitoring granularity, Figure 9 shows the utilization (left-hand side) and handling speed (right-hand side) 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 for the handling speed – regardless of the policy used. However, the policy has a significant impact on the utilization: as shown in [14], the selection quality of Least Used is better than Round Robin, and Round Robin is better than Random. A better selection quality results in better handling

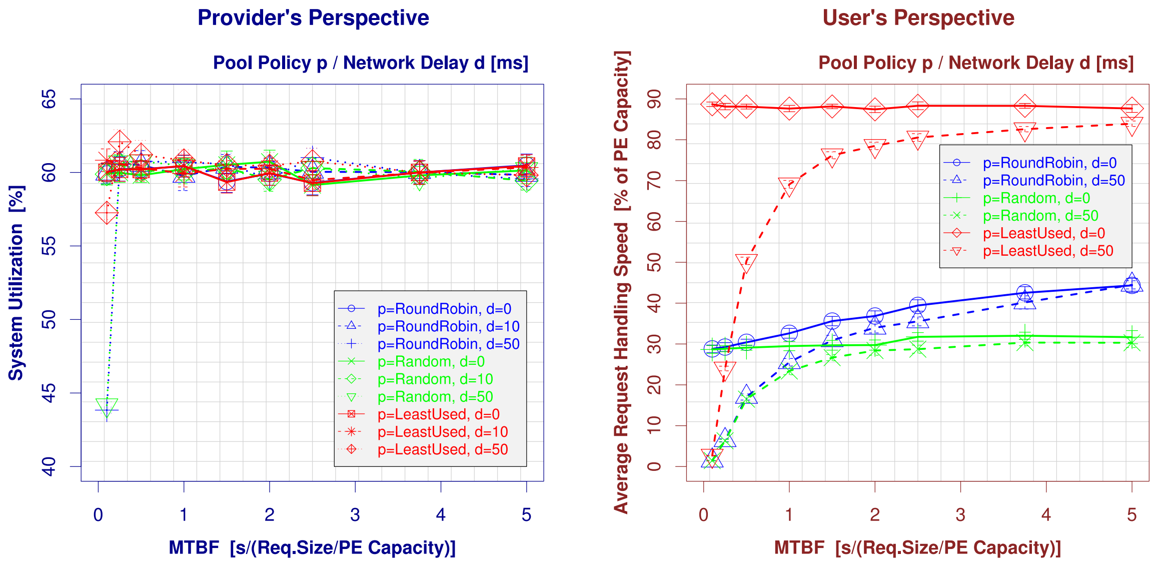

Figure 7. The Performance for Dynamic Pools

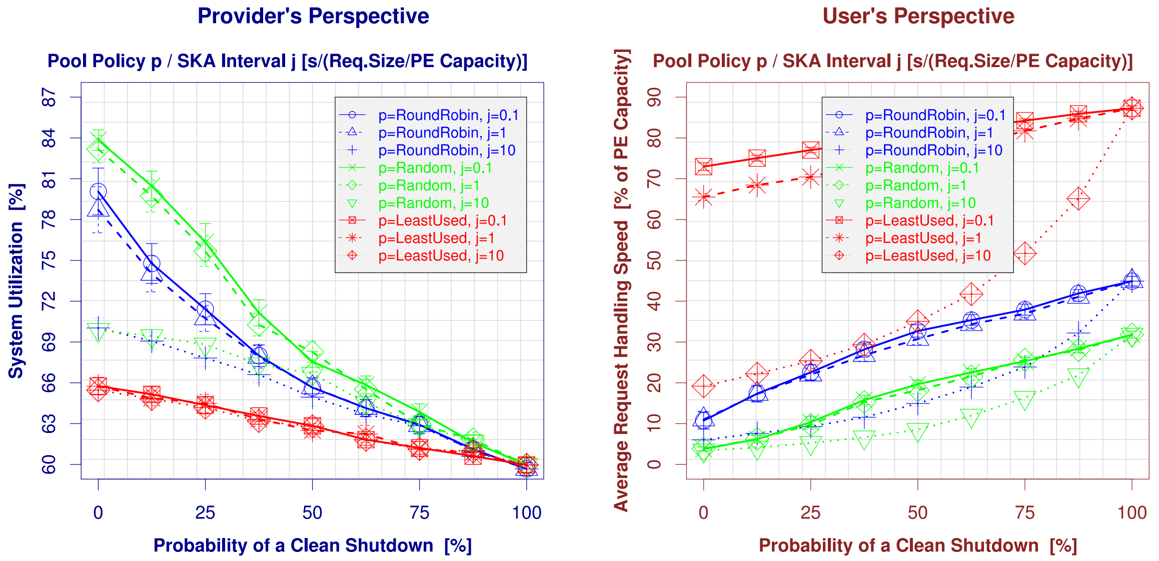

Figure 8. The Impact of Clean Shutdowns

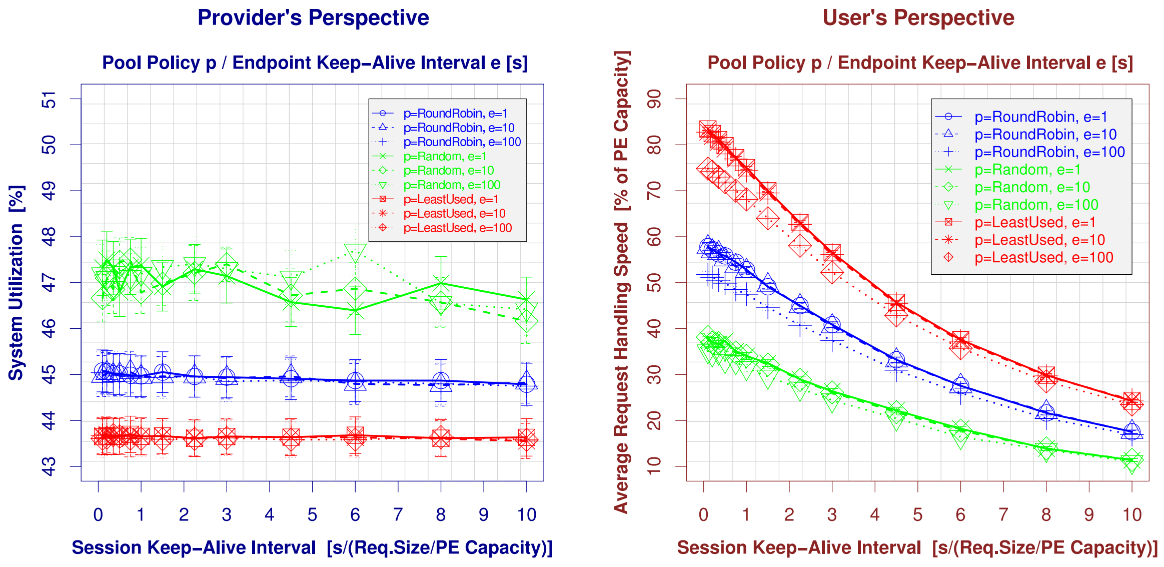

Figure 9. The Impact of Session Monitoring

speeds. Since the handling speed for Random is lowest, the amount of lost work is highest here. This results in an increased amount of re-processing, which can be observed by the higher utilization: about 47% for Random, about 45% for Round Robin but only about 43.5% for Least Used at a target system utilization of 40%.

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, 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%.

5.1.4 Server Health Monitoring

The endpoint keep-alive interval gains increasing importance when the request size becomes small. Then, the startup delay becomes significant, as illustrated in Figure 5. In order to show the general effects of the PE health monitoring based on endpoint keep-alives, Figure 10 presents the utilization (left-hand side) and handling speed results (right-hand side) for a request size:PE capacity ratio of 1 and a target system utilization of 25% (otherwise, the system becomes unstable too quickly).

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 Subsubsection 5.1.3) and trigger a new PE selection. The result is a significantly reduced request handling speed. Furthermore, for a very low MTBF, the utilization decreases: instead of letting PEs re-process lost work, the PUs spend more and more time on waiting for request timeouts. For these reasons, 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 Subsubsection 5.1.3) 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 11 (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 the power to impeach PEs! Approaches for improving the security of RSerPool systems against Denial of Service attacks are presented by [43–46].

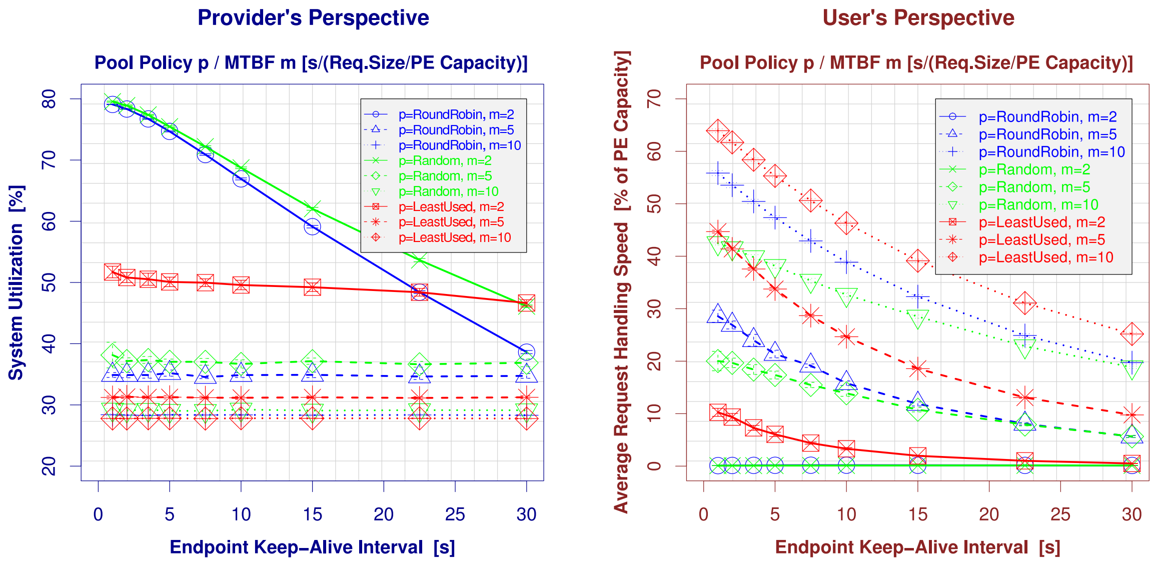

Figure 10. The Impact of Pool Element Health Monitoring

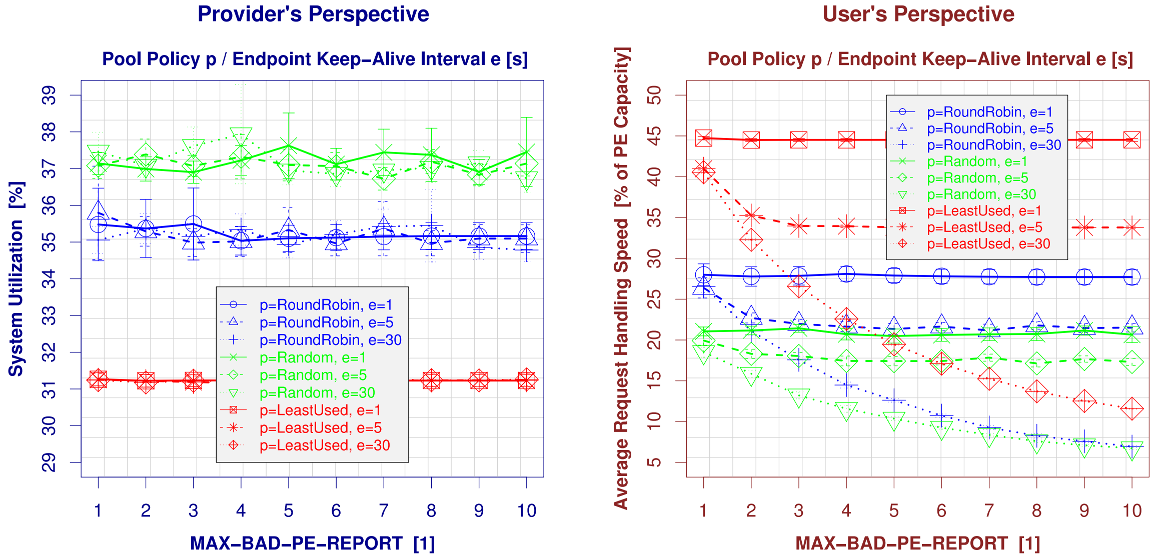

Figure 11. Utilizing Failure Reports

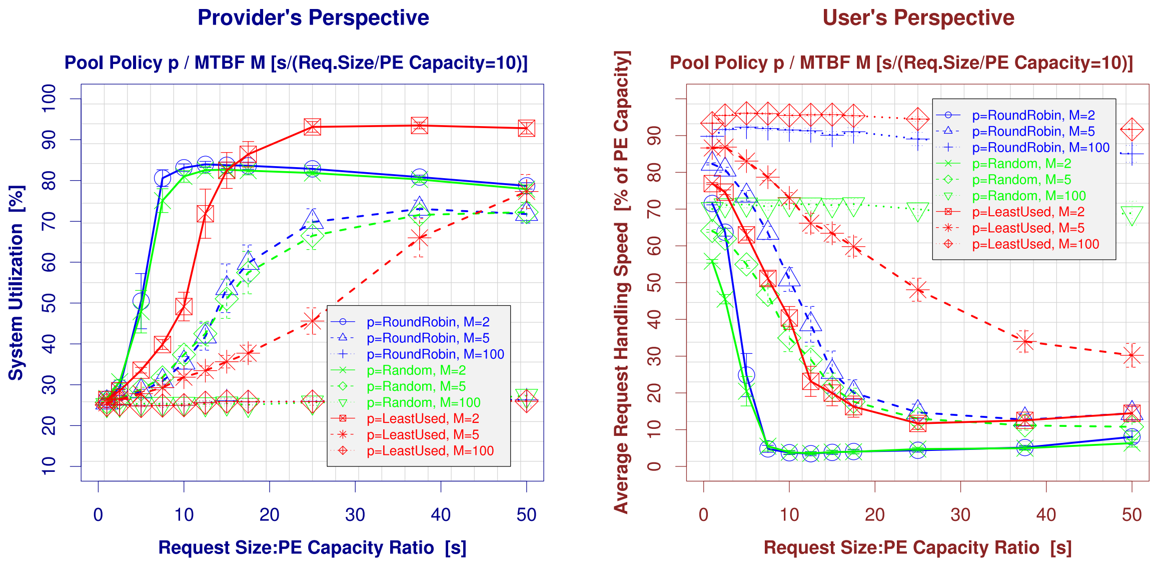

Figure 12. Using "Abort and Restart"

5.2 Failover Mechanisms

5.2.1 "Abort and Restart"

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" [7]: the session is restarted from scratch. The essential effects of applying this mechanism are presented in Figure 12: As expected, the impact of using "Abort and Restart" on the average system utilization (left-hand side of Figure 12) is small, as long as the PEs remain sufficiently available: for a MTBF of 100 times the time required to exclusively process a request having a request size:PE capacity of 10, the utilization increment is almost invisible. Furthermore, the decrement of the handling speed (right-hand side of Figure 12) is also small. Clearly, the rare necessity to restart a session has no significant performance impact.

However, for a sufficiently small MTBF, the results change: at a MTBF of 5, a significant utilization rise – as well as a handling speed decrease – can be observed for Round Robin and Random if the request size:PE capacity ratio s is increased. The effect on Least Used is smaller: as expected from the dynamic pool performance results of Subsubsection 5.1.1, this policy is able to provide a better processing speed due to superior request load distribution. That is, the probability for a request of a fixed size to be affected by PE failures is smaller if using Least Used instead of Round Robin and Random. Note that the utilization of Least Used almost reaches 95% for a MTBF of 2 and larger request size:PE capacity ratios s, due to its better load distribution capabilities. In the same situation – i.e. high overload, of course – the Round Robin and Random policies only achieve an utilization of less than 85%.

In summary, "Abort and Restart" is fairly simple and useful in case of short transactions on sufficiently available PEs. But in all other cases, it is instead useful to define checkpoints to allow for session resumption from the latest checkpoint.

5.2.2 Client-Based State Sharing

Client-Based State Sharing (see Subsection 2.4) 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 13 presents the performance results for varying the cookie interval CookieMaxCalcs (given as the ratio between the number of calculations and the average request size) for different policy and PE MTBF settings.

The larger the 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 re-processing 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 (Least Used better than Round Robin better than Random, as for failure-free scenarios [7, 14]).

[2] The maximum state cookie size is less than 64 KiB [28].

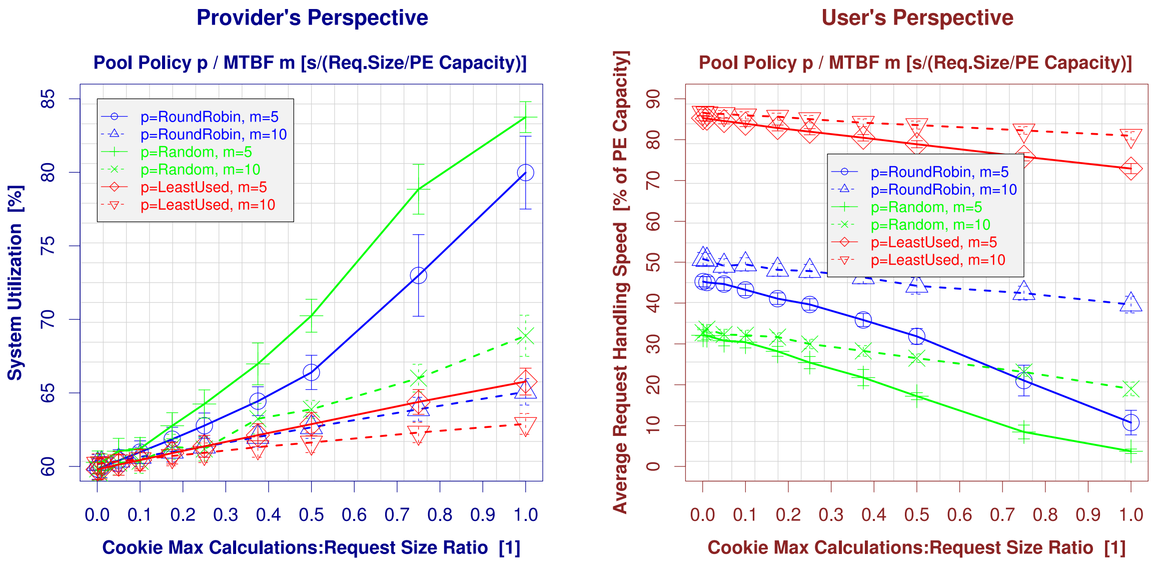

Figure 13. Using Client-Based State Sharing for the Session Failover

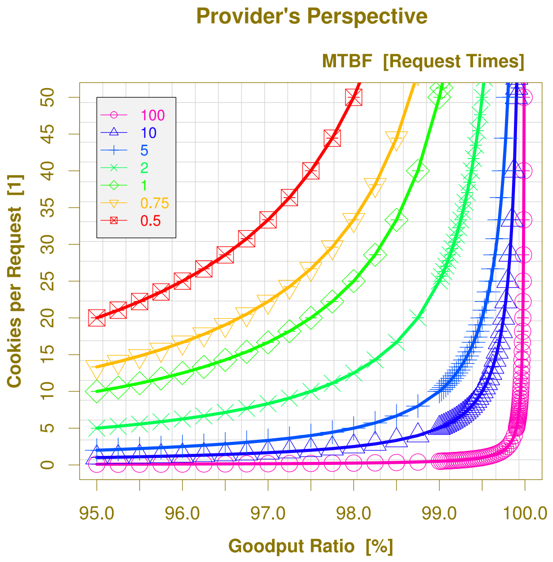

Figure 14. The Number of Cookies

In order to configure an appropriate cookie interval, the overhead of the state cookie transport has to be taken into account. The average loss of calculations per failure can be estimated as the half cookie interval $\mathrm{CookieMaxCalcs}$ (in calculations, as multiple of the average request size):

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

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:

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

Then, the cookie interval $\mathrm{CookieMaxCalcs}$ for a given goodput ratio is:

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

Figure 14 illustrates the cookies per request (i.e. $\frac{1}{\mathrm{CookieMaxCalcs}}$) for varying the goodput ratio and MTBF in equation 1. As shown for realistic MTBF values (i.e. MTBF $\gg$ 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 reasonable overhead at good system performance.

The size of a state cookie is usually in the range of a few bytes up to multiple kilobytes (some examples are provided by [7, Subsubsection 9.4.2.2]). In order to further reduce the overhead, it is useful to examine the contents of a session state [32]. Usually, it consists of some *long-term sub-states* (which mostly remain constant) and some *short-term sub-states* (which change frequently). A useful strategy is to only transmit the changed fractions of the state as *partial state cookie*. Then, the PU can combine them with the already known long-term part sent in a *full state cookie*. Of course, the need to combine long-term and short-term parts in order to apply this so-called *state splitting* technique requires the PU to be aware of the cookie structure.

Knowing the state cookie structure allows for a further optimization: by using so-called *state derivation* [32], the

PU can derive parts of the session state from the application protocol data. Then, the PU itself is able to fill in parts of the cookie, avoiding transmission overhead. This mechanism requires further differentiation of the cookie parts:

- *public* parts may be read and modified,
- *immutable* parts may only be read (a PE can verify the integrity by signature; see Subsection 2.4) and
- *private* parts (which are encrypted and therefore understandable by the PEs only; see Subsection 2.4).

An application example is an audio on demand service: a state cookie could contain some authentication information, the name of the media file and the current playback position. While the authentication information is clearly private, the media name could be immutable (it is known by the PU-side application anyway) and the playback position could be public. Since the playback position is transmitted in each RTP frame [47], it can be filled in by the PU – there is no need to transmit it in form of a new (partial) state cookie.

6 Conclusions

RSerPool is the new IETF standard for server redundancy and session failover. In this article, we have introduced RSerPool – including the underlying SCTP protocol – with a focus on server failure handing. After that, we have provided a quantitative performance analysis of its server failure handling performance. This failover performance is influenced by two factors: (1) the failure detection speed and (2) the failover mechanism.

In any case, it is crucial to detect server failures as soon as possible (e.g. by session keep-alives or an application-specific mechanism). The PR-based server health monitoring is becoming important when the request size becomes small. Failure reports may be used to reduce its overhead significantly – if taking care of security. Using the "Abort and Restart" failover mechanism, a reasonable performance is already achieved with sufficiently reliable PEs at minimal costs. A more sophisticated but still very simple and quite effective failover strategy is Client-Based State Sharing. Configured appropriately, a good performance is achieved at small overhead.

The goal of our ongoing RSerPool research is to provide configuration and optimization guidelines for application developers and users of the new IETF RSerPool standard. As part of our future work, we are going to validate our results in real-life scenarios. That is, we are going to perform PLANETLAB experiments by using our RSerPool prototype implementation RSPLIB [7, 12, 26, 43]. Furthermore, we are going to analyse the failure handling features of the ENRP protocol in detail [11, 12].

References

[1] 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)*, pages 150–156, Sainte Luce/Martinique, February 2008. ISBN 978-0-7695-3087-1.

[2] 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, March 1999.

[3] 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., April 2001. ISBN 0-7803-7016-3.

[4] 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.

[5] K. Echtle. *Fehlertoleranzverfahren*. Springer-Verlag, Heidelberg/Germany, 1990. ISBN 3-540526-80-3.

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

[7] T. Dreibholz. *Reliable Server Pooling – Evaluation, Optimization and Extension of a Novel IETF Architecture*. PhD thesis, University of Duisburg-Essen, Faculty of Economics, Institute for Computer Science and Business Information Systems, March 2007.

[8] T. Dreibholz and E. P. Rathgeb. An Evaluation of the Pool Maintenance Overhead in Reliable Server Pooling Systems. *SERSC International Journal on Hybrid Information Technology (IJHIT)*, 1(2):17–32, April 2008.

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

[10] T. Dreibholz and E. P. Rathgeb. An Evaluation of the Pool Maintenance Overhead in Reliable Server Pooling Systems. In *Proceedings of the IEEE International Conference on Future Generation Communication and Networking (FGCN)*, volume 1, pages 136–143, Jeju Island/South Korea, December 2007. ISBN 0-7695-3048-6.

[11] X. Zhou, T. Dreibholz, F. Fa, W. Du, and E. P. Rathgeb. Evaluation and Optimization of the Registrar Redundancy Handling in Reliable Server Pooling Systems. In *Proceedings of the IEEE 23rd International Conference on Advanced Information Networking and Applications (AINA)*, Bradford/United Kingdom, May 2009.

[12] X. Zhou, T. Dreibholz, E. P. Rathgeb, and W. Du. Takeover Suggestion – A Registrar Redundancy Handling Optimization for Reliable Server Pooling Systems. In *Proceedings of the 10th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2009)*, Daegu, South Korea, May 2009.

[13] T. Dreibholz and E. P. Rathgeb. A Powerful Tool-Chain for Setup, Distributed Processing, Analysis and Debugging of OMNeT++ Simulations. In *Proceedings of the 1st ACM/ICST OMNeT++ Workshop*, Marseille/France, March 2008. ISBN 978-963-9799-20-2.

[14] 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, November 2005. ISBN 0-7695-2421-4.

[15] A. Maharana and G. N. Rathna. Fault-tolerant Video on Demand in RSerPool Architecture. In *Proceedings of the International Conference on Advanced Computing and Communications (ADCOM)*, pages 534–539, Bangalore/India, December 2006. ISBN 1-4244-0716-8.

[16] 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.

[17] M. Bozinovski, L. Gavrilovska, R. Prasad, and H.-P. Schwefel. Evaluation of a Fault-tolerant Call Control System. *Facta Universitatis Series: Electronics and Energetics*, 17(1):33–44, 2004.

[18] R. Stewart. Stream Control Transmission Protocol. Standards Track RFC 4960, IETF, September 2007.

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

[20] R. Stewart, , Q. Xie, M. Tüxen, S. Maruyama, and M. Kozuka. Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration. Standards Track RFC 5061, IETF, September 2007.

[21] 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.

[22] A. Jungmaier, E. P. Rathgeb, M. Schopp, and M. Tüxen. A multi-link end-to-end protocol for IP-based networks. *AEÜ - International Journal of Electronics and Communications*, 55(1):46–54, January 2001.

[23] P. Lei, L. Ong, M. Tüxen, and T. Dreibholz. An Overview of Reliable Server Pooling Protocols. Informational RFC 5351, IETF, September 2008.

[24] Q. Xie, R. Stewart, M. Stillman, M. Tüxen, and A. Silverton. Endpoint Handlespace Redundancy Protocol (ENRP). RFC 5353, IETF, September 2008.

[25] C. S. Chandrashekaran, W. L. Johnson, and A. Lele. Method using Modified Chord Algorithm to Balance Pool Element Ownership among Registrars in a Reliable Server Pooling Architecture. In *Proceedings of the 2nd International Conference on Communication Systems Software and Middleware (COMSWARE)*, pages 1–7, Bangalore/India, January 2007. ISBN 1-4244-0614-5.

[26] 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)*, pages 39–50, Bern/Switzerland, February 2007. ISBN 978-3-540-69962-0.

[27] 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)*, pages 171–177, Sainte Luce/Martinique, February 2008. ISBN 978-0-7695-3087-1.

[28] R. Stewart, Q. Xie, M. Stillman, and M. Tüxen. Aggregate Server Access Protcol (ASAP). RFC 5352, IETF, September 2008.

[29] T. Dreibholz. Handle Resolution Option for ASAP. Internet-Draft Version 04, IETF, Individual Submission, January 2009. draft-dreibholz-rserpool-asap-hropt-04.txt, work in progress.

[30] T. Dreibholz and M. Tüxen. Reliable Server Pooling Policies. RFC 5356, IETF, September 2008.

[31] 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, August 2007. ISBN 0-7695-2977-1.

[32] 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., October 2002. ISBN 0-7695-1591-6.

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

[34] 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, November 2003. ISBN 0-7695-2037-5.

[35] T. Dreibholz, L. Coene, and P. Conrad. Reliable Server Pooling Applicability for IP Flow Information Exchange. Internet-Draft Version 07, IETF, Individual Submission, January 2009. draft-coene-rserpool-applic-ipfix-07.txt, work in progress.

[36] T. Dreibholz, X. Zhou, and E. P. Rathgeb. SimProcTC – The Design and Realization of a Powerful Tool-Chain for OMNeT++ Simulations. In *Proceedings of the 2nd ACM/ICST OMNeT++ Workshop*, Rome/Italy, March 2009. ISBN 978-963-9799-45-5.

[37] 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, November 2005. ISBN 0-7803-9312-0.

[38] 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)*, pages 117–121, Guwahati/India, December 2007. ISBN 0-7695-3059-1.

[39] 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)*, volume 4866 of *Lecture Notes in Computer Science*, pages 125–140. Springer, November 2007. ISBN 978-3-540-76808-1.

[40] T. Dreibholz and E. P. Rathgeb. Towards the Future Internet – An Overview of Challenges and Solutions in Research and Standardization. In *Proceedings of the 2nd GI/ITG KuVS Workshop on the Future Internet*, Karlsruhe/Germany, November 2008.

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

[42] 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.

[43] T. Dreibholz and E. P. Rathgeb. A PlanetLab-Based Performance Analysis of RSerPool Security Mechanisms. In *Proceedings of the 10th IEEE International Conference on Telecommunications (ConTEL)*, Zagreb/Croatia, June 2009.

[44] P. Schöttle, T. Dreibholz, and E. P. Rathgeb. On the Application of Anomaly Detection in Reliable Server Pooling Systems for Improved Robustness against Denial of Service Attacks. In *Proceedings of the 33rd IEEE Conference on Local Computer Networks (LCN)*, pages 207–214, Montreal/Canada, October 2008. ISBN 978-1-4244-2413-9.

[45] X. Zhou, T. Dreibholz, W. Du, and E. P. Rathgeb. Evaluation of Attack Countermeasures to Improve the DoS Robustness of RSerPool Systems by Simulations and Measurements. In *Proceedings of the 16. ITG/GI Fachtagung Kommunikation in Verteilten Systemen (KiVS)*, pages 217–228, Kassel/Germany, March 2009. ISBN 978-3-540-92665-8.

[46] T. Dreibholz, E. P. Rathgeb, and X. Zhou. On Robustness and Countermeasures of Reliable Server Pooling Systems against Denial of Service Attacks. In *Proceedings of the IFIP Networking*, pages 586–598, Singapore, May 2008. ISBN 978-3-540-79548-3.

[47] T. Dreibholz. Management of Layered Variable Bitrate Multimedia Streams over DiffServ with Apriori Knowledge. Masters Thesis, University of Bonn, Institute for Computer Science, February 2001.