RSerPool – Providing Highly Available Services using Unreliable Servers

Thomas Dreibholz
University of Duisburg-Essen
Institute for Experimental Mathematics
Ellernstraße 29, D-45326 Essen, Germany
dreibh@exp-math.uni-essen.de
Tel: +49 201 183-7637

Erwin P. Rathgeb
University of Duisburg-Essen
Institute for Experimental Mathematics
Ellernstraße 29, D-45326 Essen, Germany
rathgeb@exp-math.uni-essen.de
Tel: +49 201 183-7670

Abstract

The Reliable Server Pooling (RSerPool) protocol suite currently under standardization by the IETF is designed to build systems providing highly available services by mechanisms and protocols for establishing, configuring, accessing and monitoring pools of server resources. Using RSerPool, critical infrastructure services like SS7 telecommunication systems, e-commerce transaction processing or distributed computing can be provided highly available using pools of unreliable servers.

In this paper, we first give an overview of the RSerPool framework. In the following, we quantitatively show performance impacts of varying RSerPool parameters to failover handling, server selection efficiency and overhead traffic under server failure conditions.

1 Introduction

The Reliable Server Pooling (RSerPool) architecture [20] currently under standardization by the IETF RSerPool WG is an overlay network framework to provide server replication and session failover capabilities to its applications. These functionalities themselves are not new, but their combination into one application-independent framework is.

While there has already been some research on the applicability of RSerPool for applications like SCTP-based mobility [5], reliable SIP-based telephony [1], battlefield networks [22] and distributed computing [7, 8, 6, 3, 9, 24], an analysis of its session failover capabilities is still missing. Therefore, the goal of this paper is to quantitatively examine RSerPool's session failover performance in case of unreliable servers. That is, how should the parameters of RSerPool be tuned to provide a reliable service to its users despite of error-prone servers?

This paper is structured as follows: in section 2, we give an overview of the RSerPool architecture. Our OMNeT++-based simulation model is presented in section 3; the results of our performance analysis are shown in section 4.

2 The Reliable Server Pooling Architecture

2.1 Motivation

The convergence of classical circuit-switched networks (i.e. PSTN/ISDN) and data networks (i.e. IP-based) is rapidly progressing. This implies that SS7 PSTN signalling [11] has to be transported over IP networks. Since SS7 signalling networks offer a very high degree of availability (e.g. at most 10 minutes downtime per year for any signalling relation between two signalling endpoints; for more information see [10]), all links and components of the network devices must be fault-tolerant, and this is achieved through having multiple links, and using the link redundancy concept of the Stream Control Transmission Protocol (SCTP [17]).

When transporting signalling over IP networks, such concepts also have to be applied to achieve the required availability. Link redundancy in IP networks is supported using SCTP providing multiple network paths and fast failover [12, 13]; redundancy of network device components is supported by the SGP/ASP (signalling gateway process/application server process) concept. However, this concept has some limitations:

- no support of dynamic addition and removal of components;
- limited ways of server selection;
- no specific failover procedures and inconsistent application to different SS7 adaptation layers.

To cope with the challenge of creating a unified, lightweight, real-time, scalable and extendable redundancy

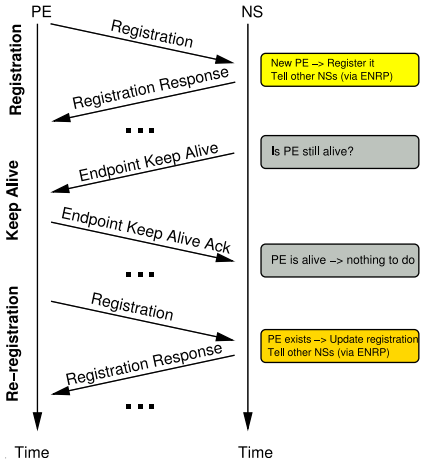

Figure 2. PE Registration and Monitoring

solution (see [21] for details), the IETF Reliable Server Pooling Working Group was founded to specify and define the Reliable Server Pooling concept.

2.2 Architecture

An overview of the RSerPool architecture currently under standardization and described by several Internet Drafts is shown in figure 1.

Multiple server elements providing the same service belong to a *server pool* to provide redundancy on one hand and scalability on the other. Server pools are identified by a unique ID called *pool handle* (PH) within the set of all server pools, the *handlespace*. A server in a pool is called a *pool element* (PE) of the respective pool. The handlespace is managed by redundant *registrars* (PR). The registrars synchronize their view of the handlespace using the Endpoint haNdlespace Redundancy Protocol (ENRP [23]). PRs announce themselves using multicast mechanisms, i.e. it is not necessary (but still possible) to pre-configure any PR address into the other components described in the following.

PEs providing a specific service can register for a corresponding pool at an arbitrary PR using the Aggregate Server Access Protocol (ASAP [18]) as shown in figure 2. The *home PR* (PR-H) is the PR which was chosen by the PE for initial registration. It monitors the PE using ASAP Endpoint Keep Alives. The frequency of monitoring messages depends on the availability requirements of the provided service. When a PE becomes unavailable, it is immediately removed from the handlespace by its home PR. A PE can also intentionally de-register from the handlespace by an ASAP de-registration allowing for dynamic reconfiguration of the server pools. PR failures are handled by requiring PEs to re-register regularly (and therefore choosing a new PR when necessary). Re-registration also makes it possible for the PEs to update their registration information (e.g. transport addresses or selection policy states).

The home PR, which registers, re-registers or de-registers a PE, propagates this information to all other PR via ENRP. Therefore, it is not necessary for the PE to use any specific PR. In case of a failure of its home PR, a PE can simply use an arbitrarily chosen other one.

When a client, called *pool user* (PU) in RSerPool terminology, requests a service from a pool, it:

1. Asks an *arbitrary* PR to translate the pool handle to a list of PE identities selected by the pool's selection policy (*pool policy*), e.g round robin or least used (to be explained in detail in section 2.4). The PR does not return the total number of identities in the pool, instead it has a constant value, *MaxHResItems*, which dictates how many PE identities should be returned. For example, if there were 5 PEs and *MaxHResItems* was set to 3, then the PR would select 3 of the 5; conversely, if *MaxHResItems* were set to 5, and there were only 3 PEs, then all 3 PE identities would be returned.
2. The PU adds this list of PE identities to its local cache (denoted as PU-side cache) and again selects *one* entry by policy from its cache.
3. To this selected PE, a connection is established, using the application's protocol, to actually use the service. The client then becomes a PU of the PE's pool.

It has to be emphasized, that there are two locations where a selection by pool policy is applied during this process:

- at the PR when compiling the list of PEs and
- in the local PU-side cache where the target PE is selected from the list.

The timeout of the PU-side cache is called *stale cache value*. Within this time period, subsequent handle resolutions of the PU may be satisfied directly from the PU-side cache, saving the effort and bandwidth of asking the PR.

If the connection to the selected PE fails, e.g. due to overload or failure of the PE, the PU selects another PE from its list and tries again. The PU may report a PE failure to its PR; the PR can increment a counter of the given PE's unreachability reports and remove it from the handlespace if it has reached the limit *MaxBadPEReports* (its default value is 3 [23]). If the PE failure occurs during an active session, a new connection to another available PE is established and an application-specific failover procedure is invoked.

2.3 Session Failover

RSerPool supports optional client-based state synchronization [2] for session failover. An example is shown in

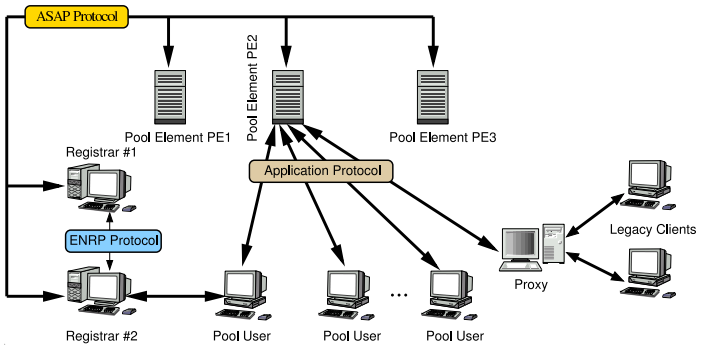

Figure 1. The RSerPool Architecture

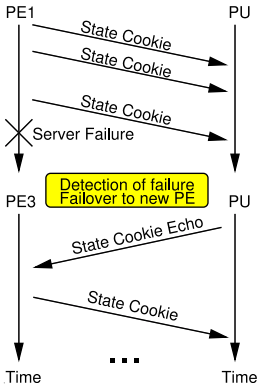

Figure 3. Failover with Client-Based State Synchronization

figure 3: A PE can store its current state with respect to a specific session in a *state cookie* which is sent to the corresponding PU. The PU only has to store the latest state cookie, i.e. the server's latest state information for the session. When a failover to a new PE is necessary, the PU can send the state cookie to the new PE, which can then restore the saved state and resume service at this point. However, RSerPool is not restricted to client-based state synchronization; any other application-specific failover procedure can be used as well.

2.4 Pool Policies

Whilst reliability is one of the obvious aspects of RSerPool, load distribution is another important one: the choice of the pool element selection policy (*pool policy*) controls the way in which PUs are mapped to PEs when they request a service. An appropriate strategy here is to balance the load among the PEs to avoid excessive response times due to overload in some servers, while others run idle.

For RSerPool, *load* only denotes a constant in the range from 0% (not loaded) to 100% (fully loaded) which describes a PE's actual normalized resource utilization. The definition of a mapping from resource utilization to a load value is application-dependent. Formally, such a mapping function is defined as

$$m(u) := \frac{u}{U_{max} - U_{min}}, u \in \{U_{min}, ..., U_{max}\} \subset \mathbb{R},$$

where U_{min} denotes the application's minimum and U_{max} the maximum possible resource utilization.

A file transfer application could define the resource utilization as the number of users currently handled by a server. Under the assumption of a maximum amount of 20 simultaneous users: $U_{min} = 0$ and $U_{max} = 20$. Therefore, $m(u) := \frac{u}{20}$.

For an e-commerce transaction processing system, response times are crucial; e.g. a customer should get a response in less than 5 seconds. In this case, utilization can be defined as a server's average response time. Then, $U_{min} = 0s$ and $U_{max} = 5s$ and $m(u) := \frac{u}{5s}$. Other schemes can be defined as well, based e.g. on CPU usage, memory utilization etc.

Depending on the used pool element selection policy, RSerPool can try to achieve a balanced *load* of the PEs within a pool. That is, if the application defines its load as a function of the amount of users, RSerPool will balance the amount of users. And if load is defined as average response time, RSerPool will balance response times.

Currently, the standards document [19] defines the following policies: Round Robin (RR), Random (RAND) and Least Used (LU). The first two policies, RR and RAND, are called *non-adaptive*, because they do not require and incorporate any information on the actual load state of the active PEs when making the selection. However, policies may be "stateful" in a sense that the current selection depends on the selection made in the previous request. This can – if carelessly implemented - lead to a severe performance degradation in some situations we show in our paper [8].

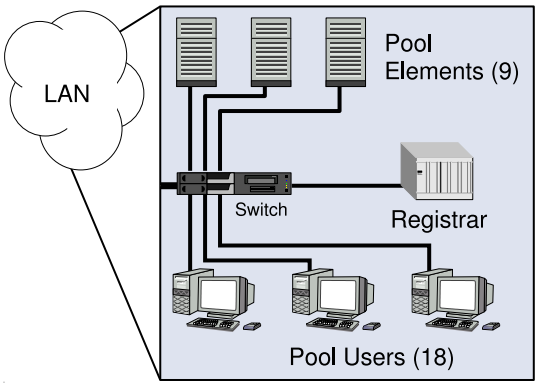

Figure 4. RSerPool Simulation Scenario

Unlike the non-adaptive policies, the LU policy tries to select the PEs which currently carry the least load. Therefore, the PEs are required to propagate their load information into the handlespace (by doing a re-registration) regularly or upon changes. These required dynamic policy information changes lead to the term *adaptive policy*. It is obvious that the adaptive policies have the potential to provide a better load sharing resulting in a better overall performance. However, the trade-off is that these policies require additional signalling overhead in order to keep the load information sufficiently current.

3 Our Simulation Model

To quantitatively evaluate the RSerPool concept, we have developed a simulation model which is based on the discrete event simulation system OMNeT++ [14]. Currently, it includes implementations of the two RSerPool protocols – ASAP [18] and ENRP [23] – and a PR module. Furthermore, it also includes models for PE and PU components of the distributed fractal graphics computation application described in [24]. This application was originally created using our RSerPool prototype *rsplib* [4, 9] and tested in a lab testbed emulating a LAN scenario. Basic performance simulations of homogeneous and heterogeneous server scenarios using our simulation model can be found in our paper [8].

Figure 4 shows the simulation scenario. The modelled RSerPool network consists of a LAN; the links within this LAN introduce an average delay of 10ms (both settings are based on the testbed LAN scenario). We have chosen a LAN scenario instead of a more complicated WAN scenario since network delay becomes only significant when the duration of requests has the network delay's order of magnitude. In this case, the client-based state synchronization to be examined in this paper is useless – it would be much easier to simply restart an interrupted session. Each LAN contains of 1 PR, 9 PEs (the local PR is their home PR) and 18 PUs. Since we do not examine PR failures in this paper, PR redundancy (i.e. ENRP) is unnecessary here.

Each PE has an average computation capacity of $C_{\mathrm{avg}} = 10^6$ calculations per second (negative exponentially distributed). A PE can process several computation requests simultaneously in a processor sharing mode as commonly used in multitasking operating systems. At most, 4 simultaneous requests are allowed on a server to avoid overloading and excessive response times. The *load* of a server in our scenario has been defined as the number of currently running requests, divided by its request limit of 4. A PE rejects an additional request if it is fully loaded. In this case, the PU will try another PE (selected by pool policy, of course) after an average timeout of 1s to avoid overloading the PR and network with unsuccessful ASAP Handle Resolution requests (recommendation based on the results from [24]).

The keep-alive and re-registration configuration is based on the testbed scenario [24]: PEs re-register at the PR in intervals of 5s; for the LU policy, load information changes result in immediate re-registration. As home PR, the PR monitors the PEs by endpoint keep-alives in intervals of 5 seconds. The PE is considered dead if no answer is received within 5 seconds. For request associations between PEs and PUs, keep-alive intervals of 5 seconds with a timeout of 5 seconds are used. *MaxHResItems* has been set to 4, *MaxBadPEReports* has been set to the standard document's default 3 [23].

For state synchronization, client-based state sharing [2] using state cookies with state approximation is used: After a given amount of completed calculations, a state cookie is sent to the PU. This state cookie simply contains the number of calculations already processed. This allows the PE to resume the session at the recorded point in the last state cookie. Computations made after the last cookie before a failure are lost, the new PE has to process them again.

For each PE, an availability $\alpha = \frac{\mathrm{total\,uptime}}{\mathrm{total\,runtime}}$ can be specified. In case of a failure, the PE becomes unreachable for an average duration of 60s, negative exponentially distributed. We explicitly do not examine the case of a clean shutdown here (PE explicitly deregisters from pool and sends cookies to all its PUs; therefore failover is quick and cheap). While a clean shutdown is likely for server pools in a computing centre, this is not the case for highly dynamic distributed computing scenarios (see [24]) where e.g. home users provide computation power. Such PEs may provide its service only for very short durations, immediately stop service when their computation power is requested by their owner, simply disappear because they are turned off, modem connections break, etc. Therefore, we will examine a broad range of server availabilities from 10% to 100%.

Unless otherwise specified, PUs sequentially request the processing of requests by the pool, having an average re-

quest size of 10^7 calculations and a negative exponential distribution (approximation of the real system behaviour, see [24]). After receiving the result of a request, a PU waits for an average of 10 seconds (again, negative exponentially distributed) to model the reaction time of a user. The stale cache value for the PU-side cache is set to 0s (i.e. the cache is turned off) for the first two simulations. Impacts of the cache are examined in section 4.3.

The length of the simulation runs has been set to 20 minutes simulated real-time. All simulation runs have been repeated 50 times using different seeds to be able to compute confidence intervals. For the statistical post-processing and plotting, *R Project* [16] has been used. The plots show mean values and their 95% confidence intervals.

4 Simulation Results

4.1 Performance of Client-Based State Synchronization

In our first simulation, we show the effect of client-based state synchronization using cookies on both system performance and required overhead traffic. Figure 5 shows the total amount of completed requests (upper part), the amount of ASAP packets (middle part, only results for lowest and highest parameter setting) and the total amount of cookies sent (lower part) during the 20 minutes of system runtime. As policy, LU has been used. Since client-based state synchronization is most useful when request runtimes are sufficiently long[1] the average request size for this simulation has been set to 10^8 calculations, negative exponentially distributed. That is, processed as the only request on a PE, it takes an average period of 100s to complete it.

Obviously, cookies provide no benefit when there are no failures, but still substantially cause overhead traffic. Especially for the setting of a cookie after every 10^6 calculations, cookie packets exceed the ASAP traffic by more than two times. On the other hand, while high cookie rates create an excessive amount of cookie traffic, there is only a small benefit of this additional traffic: comparing the transmission of a cookie after 10^6 and 10^8 calculations, the overhead traffic is reduced by a factor of 100 while the amount of completed requests is decreased by only 8% at 90% availability.

For lower availabilities however, more frequent cookies become beneficial. At 50% availability, sending a cookie after 10^7, 10^8 or 10^9 calculations already reduces the amount of completed requests by 14%, 53% and 57%. For 10% availability, the reduction increases to 46%, 63% and 63%. The reason for higher cookie settings having (almost) equal reductions is, that at such low availabilities, the PE mostly does not even reach the first cookie transmission. Therefore, the request has to be started from the beginning. The

[1]For small request sizes, it is most efficient to simply start the request again instead of adding effort to enable session resumptions.

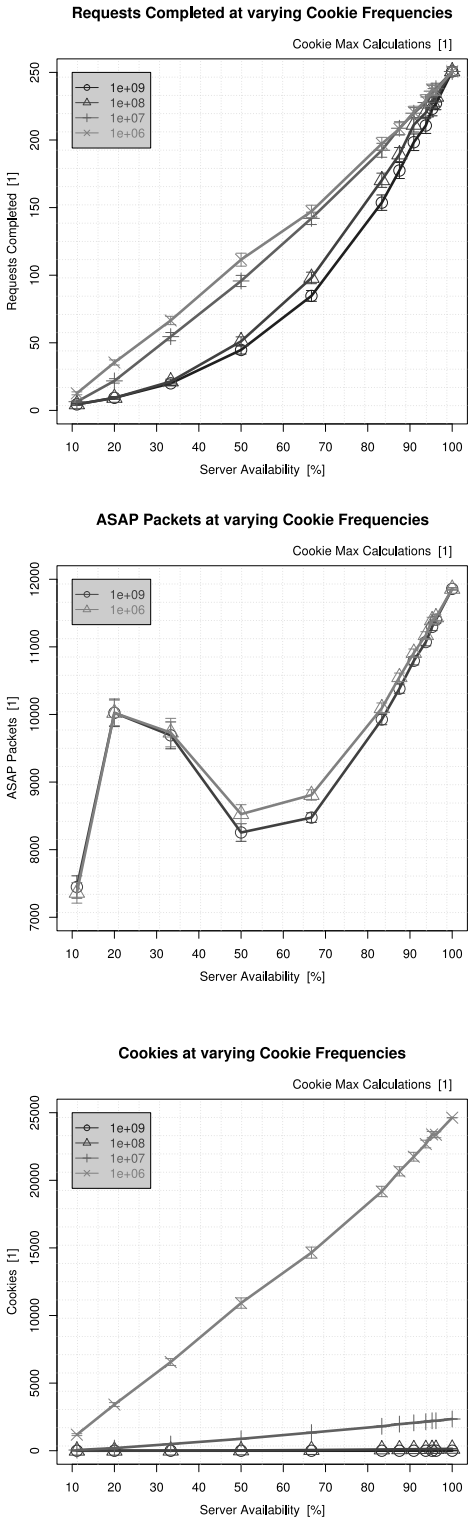

Figure 5. Client-Based State Synchronization Performance Results

results for the cookie amount curve make this effect clear: at low availabilities, there are almost no cookies sent.

The middle part of figure 5 shows the amount of ASAP packets during the simulation scenario. Since the differences between the curves for the four used cookie intervals are small, we only provide the plots for sending a cookie after 10^6 and 10^9 calculations, i.e. a change of three orders of magnitude. The difference between the two curves is a result of the amount of requests processed: since more requests are handled when the cookie interval is smaller, more handle resolutions are necessary.

Obviously, the amount of ASAP packets at first decreases when the availability goes down from 100% to about 50%. Here, the amount of PE re-registrations and keep-alives decreases since the PEs are unavailable. However, for availabilities below 50%, the amount of ASAP packets starts increasing. The reason is, that now failures are more likely, causing PUs to send more failure reports to the PR (1 message) and requesting additional handle resolutions (1 request and 1 response). For an availability below 20% however, the reduction of PE traffic dominates the result again (PUs are pausing for about 1s after each unsuccessful connection establishment, to avoid overloading the network and PR), implying a decrease in ASAP traffic.

Summary As a result of this simulation, it is obviously useful to send cookies that enable a better performance in failure situations. However, when failures are not extremely likely, the amount of cookies should be kept sufficiently small. In this scenario, sending a cookie after 10^7 calculations causes low overhead traffic (about up to 2500 cookies in 20 minutes at 100% availability) but still reduces the request amount at 50% availability by only 14%.

4.2 Performance of Pool Policies

In this simulation, we examine the impact of different pool policies on the system's failover performance. The left side of figure 6 shows the total amount of completed requests for LU, RR and RAND polices under varying server availabilities. The cookie interval for this simulation has been set to $5 * 10^6$ calculations.

As shown in our paper [8], the adaptive LU policy always outperforms the non-adaptive RR and RAND policies for failure-less scenarios. This can also be verified for failover scenarios: The performance gap of about 43% at 100% PE availability (1260 requests for LU vs. 715/714 for RR/RAND) slightly shrinks to about 37% at 50% availability (903 requests for LU vs. 556/571 for RR/RAND) and then quickly reduces to only about 10% at 10% availability (91 requests for LU vs. 75/77 for RR/RAND).

At the cost side, i.e. the overhead amount of ASAP packets in the 20 minutes of runtime as shown on the right side of figure 6, the non-adaptive policies significantly reduce the amount of ASAP packets since there is no need to update policy information (i.e. server load). But while the reduction is almost constantly 25% for availabilities from 100% to 66%, the amount of ASAP packets for RR and RAND almost keeps constant for availabilities from 66% to 33%. The reason is, that the descent of the amount of completed requests in this area for LU is much higher than for the non-adaptive policies: 48% for LU (1104 requests to 573 requests) vs. 36% for RR (660 requests to 420 requests) and 34% for RAND (651 requests to 429 requests). When a currently low-loaded PE under the LU policy fails, this one will be selected by PUs until its failure is detected; this causes additional delays resulting in a reduced amount of completed requests. For availabilities below 33%, the overhead packets curve again becomes almost parallel.

Summary Like in failure-less scenarios, the adaptive LU policy also outperforms the non-adaptive RR and RAND policies in failure scenarios. However, the performance gain of LU compared to RR and RAND shrinks with decreasing server availability.

4.3 Performance of the PU-Side Cache

Finally, we examine the system's behaviour when we activate the PU-side cache. If it is used (*stale cache value* greater than 0s), handle resolutions may be satisfied from cache instead of querying the PR. This saves some amount of ASAP traffic but the inaccurate cache content[2] may reduce the system's performance. For this simulation, the LU policy is used. The non-adaptive policies are not significantly affected by the cache, therefore we do not show the results here.

The left side of figure 7 shows the amount of requests completed within 20 minutes of system runtime. The performance penalty of using the cache remains quite constant for availabilities from 100% to 87% – the amounts of completed requests are reduced by 3%, 8% and 13% (stale cache values of 7.5s, 15s, 30s) compared to the case without cache. But reducing the availability further, the performance penalty of the cache becomes smaller: 3%, 4% and 8% for 50% availability. For availabilities below 33%, even no significant performance loss due to the cache can be observed.

On the overhead side – the right side of figure 7 shows the amount of ASAP packets – the traffic reduction remains quite constant for availabilities above 50%: the amount of overhead packets reduces by about 5%, 9% and 13% for stale cache values of 7.5s, 15s and 30s compared to the scenario without cache. But in the availability range from 20% to 50%, a significantly increased cost reduction by the cache can be observed: about 9%, 13% and 18%. Here, the chance

[2]The policy information (load) may have changed or PEs may be already dead.

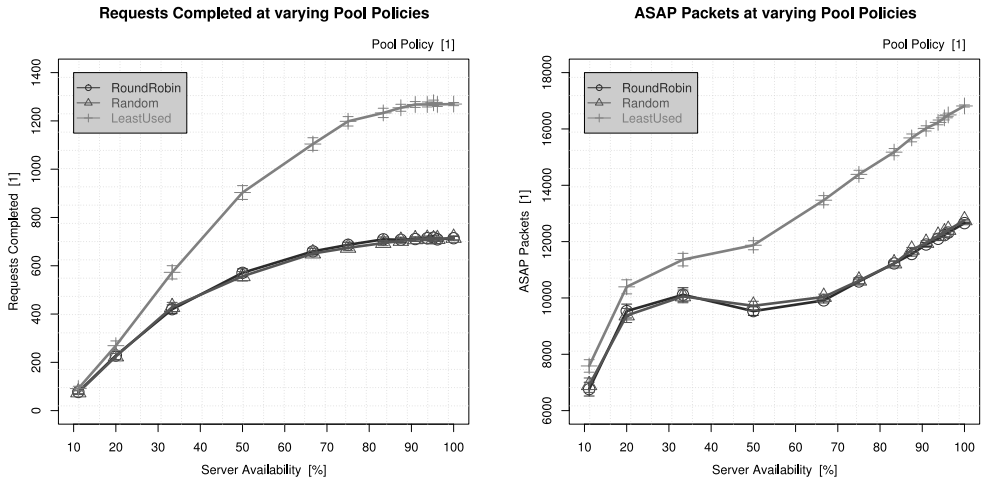

Figure 6. Pool Policy Performance Results

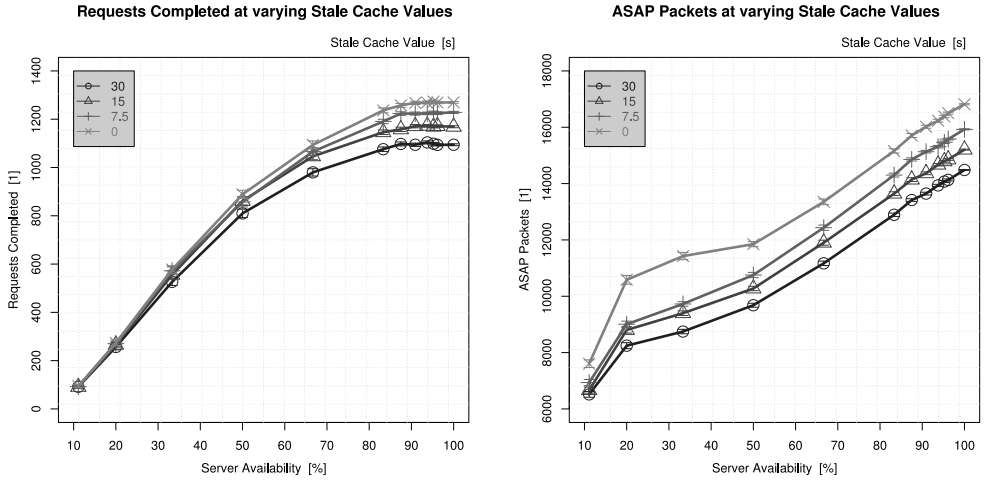

Figure 7. PU-Side Cache Performance Results

that a just selected PE is already dead when the PU sends a request becomes significant. In such a case, an additional handle resolution plus an unreachability report to the NS are necessary (3 messages). When there is a cache, the PU can skip the handle resolution at the PR and directly choose a new PE from its cache.

Summary While the PU-side cache reduces the amount of requests completed by the system due to inaccurate content (PE availability and policy information), it also reduces the amount of overhead traffic (handle resolutions at the PR). At low server availabilities, this traffic reduction significantly increases and furthermore the performance loss decreases. That is, when servers are likely to fail, a cache becomes beneficial.

5 Conclusion and Outlook

In this paper, we first presented the Reliable Server Pooling (RSerPool) architecture, the new server pool and session management framework currently under standardization by the IETF. While some research has already been made on RSerPool's applicability for certain applications, an analysis of its failover capabilities was still missing. Therefore, the analysis of these functionalities has been the subject of our paper.

We have shown that the client-based state synchronization mechanism of RSerPool can achieve efficient failover performance at the cost of small overhead when choosing the cookie interval appropriately. As for a failure-less scenario, adaptive server selection policies lead to improved performance compared to non-adaptive policies, but this performance gain is reduced when server failures become

more likely.

Finally, we have shown the impact of the PU-side server selection cache under failure conditions: while the cache reduces overhead traffic at the cost of some performance reduction in scenarios of low failure probability, it achieves overhead reduction without significant performance loss in failure-likely scenarios.

After these first promising results, we are currently continuing the evaluation of reliability aspects by examining the parameter sensitivity with respect to a broad range of system parameters including, e.g., request sizes and intervals, keep-alive timer settings, stale cache values, cookie intervals, settings of MaxBadPEReports and policies in homogeneous and heterogeneous server capacity scenarios. Furthermore, we are going to to verify our simulation results in real-life network scenarios. Based on our prototype implementation [4, 6, 3, 9] of RSerPool we are going to build a lab test scenario and finally also want to analyse large-scale scenarios using the PLANETLAB [15]. Our goal is to transfer the theoretical insights of our simulations to reality, providing guidelines for designing and tuning RSerPool systems and promoting standardization and deployment of RSerPool.

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 SCI 2002, Mobile/Wireless Computing and Communication Systems II*, volume X, Orlando/U.S.A., Jul 2002.

[2] T. Dreibholz. An efficient approach for state sharing in server pools. In *Proceedings of the 27th IEEE Local Computer Networks Conference*, Tampa, Florida/U.S.A., Oct 2002.

[3] T. Dreibholz. An Overview of the Reliable Server Pooling Architecture. In *Proceedings of the 12th IEEE International Conference on Network Protocols 2004*, Berlin/Germany, Oct 2004.

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

[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 Conference*, Königswinter/Germany, Nov 2003.

[6] T. Dreibholz and E. P. Rathgeb. An Application Demonstration of the Reliable Server Pooling Framework. In *Proceedings of the 24th IEEE Infocom 2005*, Miami, Florida/U.S.A., Mar 2005.

[7] T. Dreibholz and E. P. Rathgeb. Implementing of the Reliable Server Pooling Framework. In *Proceedings of the 8th IEEE International Conference on Telecommunications 2005*, Zagreb/Croatia, Jun 2005.

[8] 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 2005*, Saint Gilles Les Bains/Reunion Island, Apr 2005.

[9] T. Dreibolz and M. Tüxen. High availability using reliable server pooling. In *Proceedings of the Linux Conference Australia 2003*, Perth/Australia, Jan 2003.

[10] K. D. Gradischnig and M. Tüxen. Signaling transport over IP-based networks using IETF standards. In *Proceedings of the 3rd International Workshop on the design of Reliable Communication Networks*, pages 168–174, Budapest, Hungary, 2001.

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

[12] A. Jungmaier, E. Rathgeb, and M. Tüxen. On the Use of SCTP in Failover-Scenarios. In *Proceedings of the SCI 2002, Volume X, Mobile/Wireless Computing and Communication Systems II*, volume X, Orlando/U.S.A., Jul 2002.

[13] A. Jungmaier, M. Schopp, and M. Tüxen. Performance Evaluation of the Stream Control Transmission Protocol. In *Proceedings of the IEEE Conference on High Performance Switching and Routing*, Heidelberg/Germany, June 2000.

[14] OMNeT++ Discrete Event Simulation System. http://www.omnetpp.org.

[15] PlanetLab: Home. http://www.planet-lab.org.

[16] 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.

[17] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. Standards Track RFC 2960, IETF, Oct 2000.

[18] R. Stewart, Q. Xie, M. Stillman, and M. Tüxen. Aggregate Server Access Protcol (ASAP). Internet-Draft Version 11, IETF, RSerPool WG, Feb 2005. draft-ietf-rserpool-asap-11.txt, work in progress.

[19] M. Tüxen and T. Dreibholz. Reliable Server Pooling Policies. Internet-Draft Version 00, IETF, RSerPool WG, Oct 2004. draft-ietf-rserpool-policies-00.txt, work in progress.

[20] M. Tüxen, Q. Xie, R. Stewart, M. Shore, J. Loughney, and A. Silverton. Architecture for Reliable Server Pooling. Internet-Draft Version 09, IETF, RSerPool WG, Feb 2005. draft-ietf-rserpool-arch-09.txt, work in progress.

[21] M. Tüxen, Q. Xie, R. Stewart, M. Shore, L. Ong, J. Loughney, and M. Stillman. Requirements for Reliable Server Pooling. Informational RFC 3227, IETF, Jan 2002.

[22] U. 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.

[23] Q. Xie, R. Stewart, M. Stillman, M. Tüxen, and A. Silverton. Endpoint Name Resolution Protcol (ENRP). Internet-Draft Version 11, IETF, RSerPool WG, Feb 2005. draft-ietf-rserpool-enrp-11.txt, work in progress.

[24] Y. Zhang. Distributed Computing mit Reliable Server Pooling. Masters thesis, Universität Essen, Institut für Experimentelle Mathematik, Apr 2004.