An Evaluation of the Pool Maintenance Overhead in Reliable Server Pooling Systems*

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 protocol framework for server redundancy and session failover, currently still under standardization by the IETF RSerPool WG. An important property of RSerPool is its lightweight architecture: server pool and session management can be realized with small CPU power and memory requirements. That is, RSerPool-based services can also be managed and provided by embedded systems. Currently, there has already been some research on the performance of the data structures managing server pools. But a generic, application-independent performance analysis – in particular also including measurements in real system setups – is still missing.

Therefore, the aim of this paper is – after an outline of the RSerPool framework, an introduction to the pool management procedures and a description of our pool management approach – to first provide a detailed performance evaluation of the pool management structures themselves. Afterwards, the performance of a prototype implementation is analysed in order to evaluate its applicability under real network conditions.

Keywords: *RSerPool, Server Pools, Handlespace Management, SCTP, Performance*

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 [27] – 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 [23]) 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 [1,32]), but their combination into one application-independent framework is.

The Reliable Server Pooling (RSerPool) architecture currently under standardization by the IETF RSerPool WG is an overlay network framework to provide server replication and session failover capabilities to its applications [9]. In particular, server redundancy leads to the issues of load distribution and load balancing [22], which are also covered by RSerPool [13, 15, 19]. But in full contrast to already available solutions in the area of GRID and high-performance computing [20], 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 [6, 12]. Especially, this means the limitation to a single administrative domain and only taking care of pool and session management – but not for tasks like data synchronization, locking and user management (which are considered to be application-specific). On the other hand, these restrictions allow for RSerPool components to be situated on embedded devices like routers or telecommunications equipment.

There has already been some research on the performance of RSerPool for applications like SCTP-based mobility [11], VoIP with SIP [4], web server pools [28], IP Flow Information Export (IPFIX) [10], real-time distributed computing [9,13,19] and battlefield networks [34,35]. Furthermore, some ideas and rough performance estimations for the pool management have been described in our paper [12]. But up to now, a detailed performance analysis of these data structures, as well as an evaluation of the pool management overhead in a real system setup, are still missing. The goal of our work is therefore to provide these analyses. In particular, we intend to identify critical parameter spaces to provide guidelines for designing and provisioning efficient RSerPool systems.

2 The RSerPool Architecture

Figure 1 provides an illustration of the RSerPool architecture, as defined in [17,26]; the protocol stack is presented in figure 2. RSerPool consists of three component classes: servers of a pool are called *pool elements* (PE). A pool is identified by a unique *pool handle* (PH) in the handlespace, which is the set of all pools. The handlespace is managed by *pool registrars* (PR). PRs of an *operation scope* synchronize their view of the handlespace using the Endpoint haNdle-

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

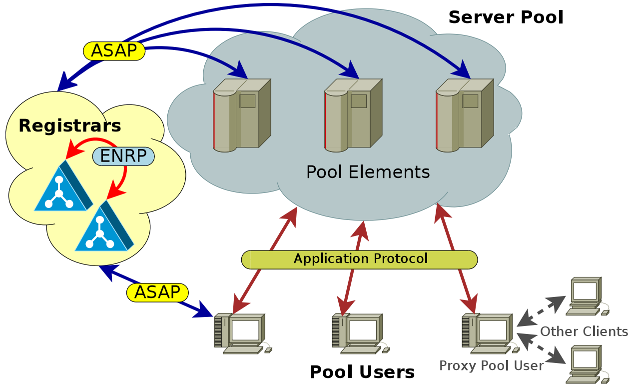

Figure 1. The RSerPool Architecture

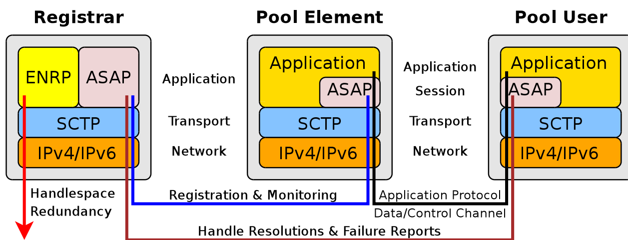

Figure 2. The RSerPool Protocol Stack

space Redundancy Protocol (ENRP [36]). In the operation scope, each PR is identified by a PR ID. An operation scope has a limited range, e.g. a company or organization; RSerPool does not intend to scale to the whole Internet. Nevertheless, it is assumed that PEs can be distributed globally, for their service to survive localized disasters [16].

A PE can register into a pool at an arbitrary PR of the operation scope, using the Aggregate Server Access Protocol (ASAP [30]). In its pool, the PE will be identified by a random 32-bit identifier which is denoted as PE ID. The PR chosen for registration becomes the *Home-PR* (PR-H) of the PE and is in particular 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 a certain threshold of such reports is reached, a PR may also remove the corresponding PE. The PE failure detection mechanism of a PU is application-specific. A non-PR-H only sets a lifetime expiration timer for each PE (owned and monitored by another PR). If not updated by its PR-H in time, a PE is simply removed from the local handlespace.

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 – which is called *handle resolution* – from an arbitrary PR of the operation scope, again using ASAP [30]. The PR selects the requested list of PE identities using a pool-specific selection rule, called *pool policy*. The maximum number of selected entries per request is defined by the constant *MaxHResItems*. Adaptive and non-adaptive pool policies are defined in [33]; for a detailed discussion of these policies, see [13, 15, 19, 37, 38]. 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 [12].

The ASAP protocol also provides an optional Session Layer between a PU and a PE. That is, a PU establishes a logical session with a pool; ASAP takes care of the transport connection establishment, for the connection monitoring and for triggering a failover to a new PE in case of a failure (see [5, 14]). All associations among the three RSerPool component types (see also figure 2) are usually based on the Stream Control Transmission Protocol (SCTP [29]), which in particular allows for path multi-homing (see [24, 25] for details).

3 The Handlespace Management Approach

3.1 Requirements

The challenge of the handlespace management is to fulfil two important properties, with particular regard of the "lightweight" requirement of the RSerPool architecture: (1) Server pools may get large (up to many thousands of PEs [8]) and (2) A handlespace may contain various pools, each one may use a different policy for server selection [15] (and new applications may even introduce further policies [16, 19]). Clearly, in order to keep such a handlespace maintainable, it is necessary to use an unified storage structure (which is usable for all policies) and realize it in an efficient way. Furthermore, the handlespace data structure has to support the following six operations: (1) *Registration* denotes the registration of a new PE. (2) *Deregistration* means the removal of a PE entry. (3) *Re-Registration* is an information update for an exiting PE entry. In particular, a re-registration is necessary to update the policy information of an adaptive policy (e.g. the load state for LU [13]). (4) *Handle Resolution* denotes a PE selection operation. (5) *Timer* denotes scheduling and expiry of a handlespace timer. For a PR-H, this means scheduling a keep-alive transmission time, its timeout, scheduling a timeout for the keep-alive and cancelling it (on acknowledgement reception). For a non-PR-H, it denotes the scheduling of a registration's lifetime expiration and its cancellation (for an update). (6) *Synchronization* is the step-wise traversal of the complete handlespace, in order to obtain a block-wise copy for another PR.

3.2 Policy Realization

On the topic of supporting different policies, we have already proposed in [12] to realize the handlespace in form of multiple *sets* (as illustrated in figure 3): a handlespace is simply a set of pools (Pools Set); each pool contains a set of PE references sorted by PE ID (Index Set) and a second set of these references sorted by a policy-specific sorting order

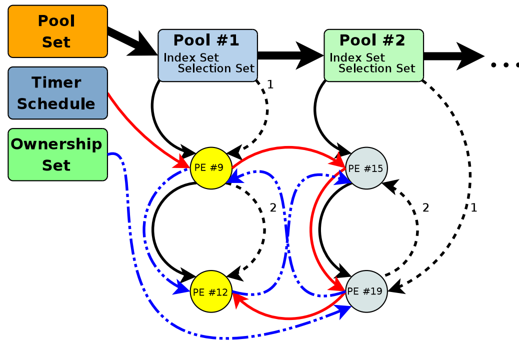

Figure 3. The Handlespace Structure

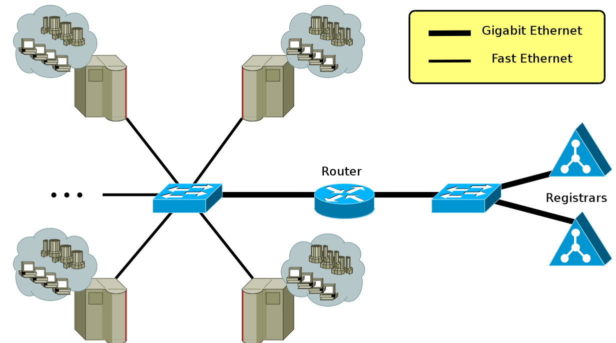

Figure 4. The Measurement Setup

(Selection Set). In order to realize different policies, it is simply necessary to specify a sorting order for the Selection Set, as well as a selection procedure (which is usually to take the first PE). Upon selection of a PE entry, its position in the Selection Set is updated.

In [12], we have already shown the scalability of this approach for a specific example application scenario. However, a performance analysis for a broader parameter range has still been missing. Furthermore, our handlespace management approach had to be extended by more features, which are described in the following.

3.3 Timer Schedule

Scheduling and expiration of timers for PE entries is an additional task of the handlespace management. There are three types of timers: a keep-alive transmission timer schedules the transmission of an ASAP keep-alive to a PE; the keep-alive timeout timer schedules the timeout for the PE's answer. A lifetime expiry timer schedules the expiration of a PE entry on a non-PR-H. An important observation for these three timers is that at any given time exactly one of them is scheduled for each PE. That is, each PE entry only has to contain the type of the timer and the expiration time stamp. Then, the timer schedule is simply another set of PE entries (sorted by time stamp, of course), as shown in figure 3.

3.4 Checksum and Ownership Set

The ENRP protocol takes care of the handlespace synchronization. In order to detect discrepancies in the handlespace views of different PRs, each PR calculates a checksum of its own PE entries (i.e. the PEs for which it is in the role of a PR-H). These checksums can be transmitted to other PRs, which can compare the value expected from their own handlespace view with the announced value. In case of a difference, a synchronization is necessary. The checksum algorithm used by ENRP is the 16-bit Internet Checksum [3], which allows for incremental updates [9].

The synchronization procedure requires to traverse all PE entries belonging to a certain PR. This functionality can be realized by introducing the so called Ownership Set – containing the PE references sorted by PR-H (see figure 3).

4 The Measurement Setup

In [12], the pool management workload of a PR has already been examined for different implementation strategies of the Set datatype – but only for a very specific setup.

4.1 Data Structure Performance

However, a detailed analysis of the handlespace operations throughput is still missing. Therefore, this will be the first part of this paper. Our program for the corresponding measurements simply performs as much operations of the requested type as possible, in the pool built up in advance. Since registrations and deregistrations cannot be examined separately (the pool would either grow or shrink), these operations are examined combinedly: a *Registration/Deregistration* operation simply performs the deregistration of a randomly selected element if the pool has the configured size; otherwise, a new PE is registered. The system used for the performance measurements uses a 1.3 GHz AMD Athlon CPU – which has been state of the art in early 2001 (i.e. almost seven years ago) and whose performance seems to be realistic for upcoming router or embedded device generations (which could host a PR service). All measurements are repeated 18 times in order to provide statistical accuracy.

4.2 Real System Performance

While the operations throughput is useful to estimate the scalability of the handlespace management, the resulting question is clearly how a real system performs. In order to evaluate such a system, i.e. including real components, protocol stacks and network overhead, we have set up a lab scenario as shown in figure 4: it consists of a set of 10 PCs (each having a 2.4 GHz Pentium IV CPU and 1 GB of memory) connected by a gigabit switch to a Linux-based router. Two PRs (using the same CPU as for the data structure performance evaluation, see subsection 4.1) are connected to the router by Gigabit Ethernet. On each of the hosts, a configurable number of test PEs, PUs and PRs can be started.

All systems run Kubuntu Linux 6.10 “Edgy Eft”, using kernel 2.6.17-11 and the kernel SCTP module provided

by the distribution. Our RSerPool implementation RSPLIB [7, 9, 18], version 2.2.0 has been installed on all machines. Each measurement run is repeated 12 times to achieve statistical accuracy.

GNU R has been used for the statistical post-processing of our results – including the computation of 95% confidence intervals – and plotting. All results plots show the average values and their confidence intervals.

5 Performance Analysis

Our performance evaluation is subdivided into two parts. The first part in subsection 5.1 provides a performance analysis of the handlespace management structure itself and constitutes the foundation of the real system evaluation in subsection 5.2.

5.1 Data Structure Performance

The most important operation for the PE side is the registration/deregistration (see subsection 3.1) at the PR. In [12], it has already been shown that deterministic policies can lead to systematic insertion and removal operations in the Selection Set (see subsection 3.2). On the other hand, randomized policies are not affected. Therefore, only a balanced tree structure is appropriate to base the Set datatype on. We have examined the scalability on the number of PEs for the two state-of-the-art representations of this datatype: the red-black tree [21] (a deterministic approach) and the treap [2] (a randomized approach).

The left-hand side of figure 5 shows the throughput of registration/deregistration operations per PE and second for both tree structures and classes of policies. While the performance difference between the two policy types is small, the treap has a slightly lower performance: using a deterministically balanced tree is – despite the greater complexity of the insertion and removal algorithms [21] – the faster solution. For a pool of 20,000 PEs, it would be possible to register or deregister each PE about 2 times per second (red-black tree).

Clearly, this is more than sufficient in realistic scenarios. But while the frequency of registration/deregistration operations (i.e. actual insertions of new or removals of existing PEs) is assumed to be rare, a re-registration (i.e. a registration update) of a PE occurs frequently, in particular if the policy is dynamic. For a dynamic policy (e.g. LU), the position of the PE entry within the Selection Set changes (see also subsection 3.2). In order to show the impact on the reregistration operations performance, the right-hand side of figure 5 presents the reregistrations throughput per PE and second. For the adaptive policy (here: LU), each reregistration updates the load value with a random value. As expected, a significant difference between adaptive and non-adaptive policies is shown: for 20,000 PEs, the non-adaptive policy still achieves a throughput of about 5 operations per PE and second (red-black tree), while it sinks to only about 3 in the adaptive case. That is, care has to be taken of the application behaviour – which actually has to decide when the policy information needs to be updated! Again, the performance for using a red-black tree is slightly better than using a treap.

The throughput of timer operations is depicted on the left-hand side of figure 6. Clearly, the two extreme cases for this operation are 0% and 100% of owned PEs. Therefore, the results of these two settings for both tree implementations are shown. However, the difference keeps very small: re-scheduling a timer is quite inexpensive – the CPU's cache helps to quickly re-insert the updated structure as described in subsection 3.3. As already expected, the performance for a red-black tree is slightly better than for a treap.

Handle resolution is the operation relevant for the PUs. Its performance is influenced by two factors: *MaxHResItems* and the type of policy – randomized or deterministic. For a randomized policy, it is necessary to move down the Selection Set tree (whose depth is $O(\log n)$ – n number of PEs – for red-black tree and treap) in order to obtain a random PE [12] – for each of the *MaxHResItems* entries. Deterministic policies, on the other hand, simply allow for taking a complete chain of PE entries from the list (since their order is deterministic and therefore already defined by the sorting order, see subsection 3.2), i.e. the overall runtime is $O(1)$ instead.

The throughput of handle resolution operations per PE and second is depicted on the right-hand side of figure 6. Clearly, it can be observed that the higher *MaxHResItems*, the lower the throughput: it sinks from 13 at *MaxHResItems* h=1 to about 7.5 at h=3 for 10,000 PEs (deterministic policy, red-black tree). Furthermore, the performance for a randomized policy is clearly lower: 7 at h=1 vs. about 4 at h=3 for 10,000 PEs (red-black tree). Again, the performance for the treap is somewhat lower than for the red-black tree. In a real system, the frequency of handle resolutions strongly depends on the application's PU workload. Having a PU with a high handle resolution frequency (e.g. a web proxy like [28]), it is possible to apply a handle resolution cache at the PU [13]. Furthermore, the handle resolution operation has an advantage over the previously examined operations: it can be performed independently of other PRs. That is, in case of a high handle resolution workload, the PUs could be distributed among multiple PRs.

The last operation is the synchronization, which only occurs when PRs detect an inconsistency or on PR startup. That is, the operation is quite rare (e.g. up to a few times per day only). However, the actual performance for a pool of 30,000 PEs allows for more than 100 operations per second, which is by orders of magnitude more than sufficient. Therefore, a plot has been omitted.

5.2 Real System Performance

While our RSerPool handlespace management approach – based on red-black trees – handles pools of 10,000 and more PEs, pools of up to a few hundreds of PEs seem to be most realistic for the application cases of RSerPool. Therefore, the following measurements focus on smaller pools, but with a high PR request frequency in order to fathom the limits.

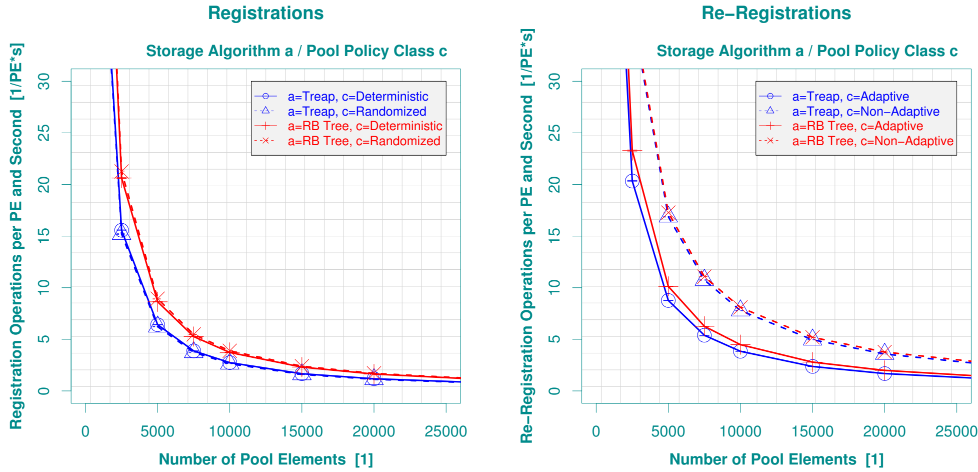

Figure 5. The Scalability of the Registration/Deregistration and Re-Registration Operations

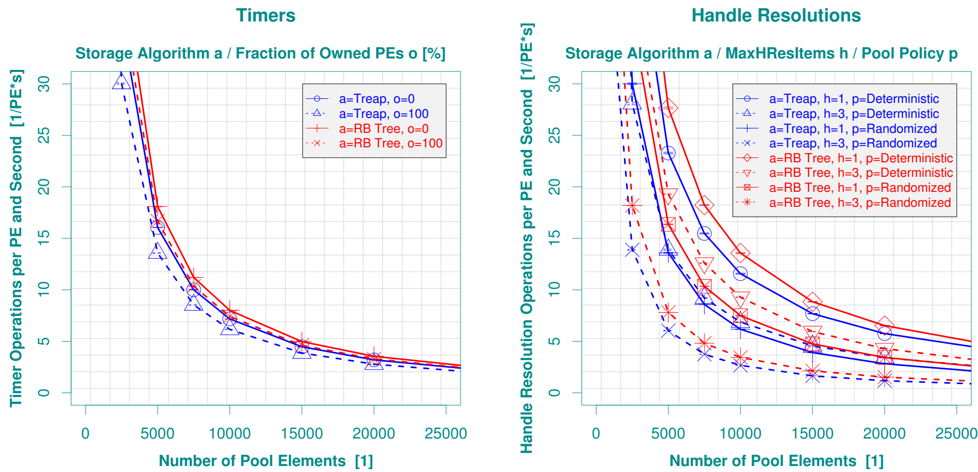

Figure 6. The Scalability of the Timer Handling and Handle Resolution Operations

5.3 Pool Elements Scalability

In order to show the scalability on PEs, the number of PEs has been varied. The pool is using the RR policy (i.e. deterministic) and an inter-reregistration time between 250ms and 1000ms (such high rates may occur for dynamic policies). All ASAP (re-)registrations are performed on PR #1 (see figure 4), PR #2 is synchronized by ENRP only. That is, we have used the worst case here. The CPU utilization of PR #1 and PR #2 are shown on the left-hand side of figure 7. Randomized policy results have been omitted, since the results do not differ significantly (see also subsection 5.1).

Clearly, the workload on PR #1 is highest: it not only has to handle up to 3,000 simultaneous SCTP associations to PEs (for ASAP), but also has to send out an ENRP update to the other PR on every update of a PE entry. This leads to a load of about 90% for 2,000 PEs at an inter-reregistration time of a=250ms. Extending this time to a=1000ms, it is already possible to manage 3,000 PEs at a load of only about 25%.

Obviously, the workload of PR #2 is significantly lower: it only has to maintain a single SCTP association to PR #1 to obtain the handlespace data. This results in a load of only about 15% for 2,000 PEs at a=250ms, and about 25% for 3,000 PEs at a=1000ms. It is therefore a clear recommendation to try to distribute the load among the PRs of the operation scope. In reality, this can be achieved using the automatic configuration feature of RSerPool [34]. However, care has to be taken of redundancy: in case of PR failure(s), there must be a sufficient number of other PRs! But what about the costs of the ENRP synchronization among PRs?

5.4 Registrars Scalability

In order to show the scalability on the number of PRs, we have again used PR #1 for the ASAP associations and PR #2 for ENRP synchronization only (as shown in figure 4). Fur-

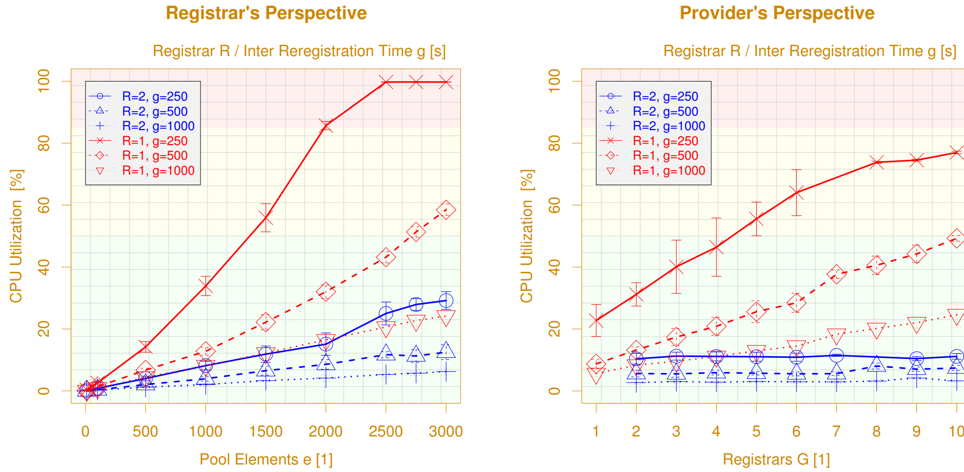

Figure 7. Registrar CPU Utilization for Pool Maintenance

ther PRs have been started on the other PCs (since only the utilizations of PR #1 and PR #2 are relevant). For our measurement, we have used a pool of 1,000 PEs and inter-reregistration times of a=250ms to a=1000ms. The CPU utilization results for PR #1 and PR #2 are presented on the right-hand side of figure 7.

Clearly, the number of PRs does not significantly affect PR #2. While it has to maintain an association with each other PR of the operation scope, the actual workload – which remains constant – is only transported via the association with PR #1. On the other hand, the utilization for PR #1 is significantly increased with the number of PRs, in particular if the inter-reregistration time is small: e.g. from about 20% for a single PR to slightly more than 60% for 6 PRs (at a=250ms). The bottleneck in this case is the interface between userland application (i.e. the PR) and the kernel's SCTP API. For each PR, a separate ENRP message has to be passed to the kernel's SCTP API. Clearly, the context switching and memory copying for this operation is time-consuming, while the actual message transport (IP packets via Ethernet interface) is quite efficient (a recent system can transport hundreds of thousands of packets per second).

The analysis of the described userland/kernel bottleneck has led to the suggestion of a SCTP API extension: the SCTP_SENDALL option (see subsection 5.2.2 of [31]). Using this option, a message to all PRs is passed to the kernel only once – and sent via all PR associations. But although the new option is already a part of the SCTP API standards document [31], it is not implemented for the current Linux kernel (version 2.6.20) yet. Therefore, a performance evaluation using this option has to be part of future work.

In summary, using a reasonably small number of PRs (e.g. two or three are usually sufficient to achieve redundancy), the ENRP overhead remains in an acceptable range – with room for future improvement on the SCTP layer.

5.5 Pool Users Scalability

Finally, we have evaluated the scalability on the number of PUs for handle resolution operations using two PRs. Again, we have observed the CPU utilization of PR #1 and PR #2 (see figure 4) for a pool of 1,000 PEs using deterministic (solid lines) and randomized policies (dotted lines), an inter-reregistration time of 1000ms and inter-handle-resolution times between 100ms and 500ms. For the first measurement, we have used PR #1 for both, registrations and handle resolutions (left-hand side), while we have put the burden of handle resolutions on PR #2 for the second measurement (right-hand side).

Clearly, if using PR #1 for all operations, PR #2 only has to synchronize and therefore its load keeps constant. But nevertheless, the CPU load of PR #1 only slightly exceeds 25% for 2,000 PUs and a inter-handle-resolution time of 500ms. For a higher handle resolution rate, however, the CPU utilization quickly grows: at 100ms, there is already a load of more than 80% for 1,000 PUs. The performance difference between the two types of policies is small – even at 2,000 PEs, the CPU utilization of a randomized policy is only by less than 5% higher (see subsection 5.1). That is, compared to the protocol overhead, the pool maintenance effort is small for this number of PEs.

So, with regard to these results, it is obviously a good idea to split up the workload of registration management and handle resolutions among the PRs. Therefore, PR #2 in the second measurement (right-hand side of figure 4) is responsible for all handle resolutions. Clearly, the system performance gets better now: at a CPU utilization below 80% (PR #2), it is now possible to serve 1,500 PUs with a handle-resolution rate of only 100ms – at a workload of about 10% for PR #1. Splitting up the workload of both operations between the two PRs would clearly result in an even better performance. However, a redundant system should always be provisioned for the worst case – which is a failure of $n-1$ of the n PRs. That is, the sum of the workloads of both PRs must remain significantly lower than 100%!

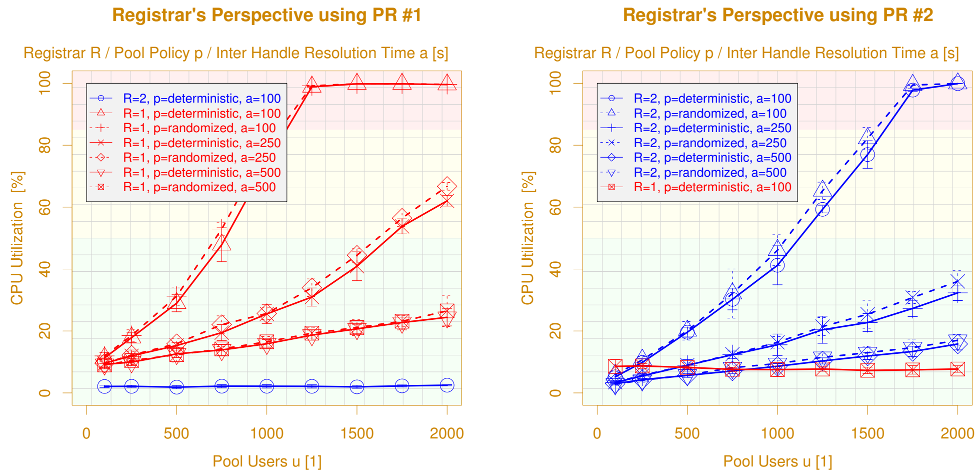

Figure 8. Registrar CPU Utilization for Handle Resolution

5.6 Results Summary

In summary, our handlespace performance analysis has shown that our approach of reducing the handlespace management to the storage of sets and operations on these sets is efficient if using a red-black tree to actually realize the sets. Critical operations are the re-registration (which may occur very frequently for adaptive policies) and the handle resolution. But in our real system performance analysis, we have shown that even a low-performance CPU is able to handle scenarios of significantly more than 1,000 PEs and PUs. As general recommendation, it is useful to distribute the PEs und PUs to different PRs of the operation scope to achieve the highest performance. However, care has to be taken of sufficient PR redundancy to cope with PR failures. Depending on the inter-reregistration and handle resolution frequency, also much larger scenarios are possible. A room for further performance improvement will be the SCTP_SENDALL option of the SCTP stack, which will be realized in future SCTP implementations.

6 Conclusions

The analyses of this paper have shown that our handlespace realization is efficient: using a red-black tree as base structure to store the handlespace content, all handlespace operations can be reduced to the management of balanced trees. The performance of this approach is sufficient to maintain handlespaces of many thousands of PEs – even on a low-performance CPU being realistic for upcoming routers and embedded systems.

In the second part of this paper, we have also proven that our approach is applicable and efficient in reality: a system based on the same CPU is also capable of handling the ASAP/ENRP protocol overhead and the maintenance of SCTP associations.

As part of our future research, we are going to further evaluate our approach for certain RSerPool-based application scenarios. Such real-world scenarios set requirements on pool size and policy type as well as on re-registration and handle resolution frequency. In particular, we intend to estimate a lower threshold for the CPU performance needed to handle these application scenarios. This also includes tests with our implementation on Linux-based embedded systems.

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] C. Aragon and R. Seidel. Randomized search trees. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pages 540–545, Oct. 1989.

[3] R. Braden, D. Borman, and C. Partridge. Computing the Internet Checksum. Standards Track RFC 1071, IETF, Sept. 1988.

[4] 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.

[5] 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.

[6] T. Dreibholz. Policy Management in the Reliable Server Pooling Architecture. In *Proceedings of the Multi-Service Networks Conference (MSN, Cosener's)*, Abingdon, Oxfordshire/United Kingdom, July 2004.

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

[8] T. Dreibholz. Applicability of Reliable Server Pooling for Real-Time Distributed Computing. Internet-Draft Version 03, IETF, Individual Submission, June 2007. draft-dreibholz-rserpool-applic-distcomp-03.txt, work in progress.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] T. Dreibholz and E. P. Rathgeb. Towards the Future Internet – A Survey of Challenges and Solutions in Research and Standardization. In *Proceedings of the Joint EuroFGI and ITG Workshop on Visions of Future Network Generations (EuroView)*, Würzburg/Germany, July 2007. Poster presentation.

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

[19] 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*, Lübeck/Germany, Aug. 2007.

[20] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. *Grid Service Infrastructure WG, Global Grid Forum*, June 2002.

[21] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science*, pages 8–21, New York/U.S.A., Oct. 1978.

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

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

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

[25] A. Jungmaier, E. P. Rathgeb, and M. Tüxen. On the Use of SCTP in Failover-Scenarios. In *Proceedings of the State Coverage Initiatives, Mobile/Wireless Computing and Communication Systems II*, volume X, Orlando, Florida/U.S.A., July 2002. ISBN 980-07-8150-1.

[26] 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.

[27] 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.

[28] S. A. Siddiqui. Development, Implementation and Evaluation of Web-Server and Web-Proxy for RSerPool based Web-Server-Pool. Master's thesis, University of Duisburg-Essen, Institute for Experimental Mathematics, Nov. 2006.

[29] 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.

[30] R. Stewart, Q. Xie, M. Stillman, and M. Tüxen. Aggregate Server Access Protcol (ASAP). Internet-Draft Version 16, IETF, RSerPool Working Group, July 2007. draft-ietf-rserpool-asap-16.txt, work in progress.

[31] R. Stewart, Q. Xie, Y. Yarroll, J. Wood, K. Poon, and M. Tüxen. Sockets API Extensions for Stream Control Transmission Protocol (SCTP). Internet-Draft Version 12, IETF, Transport Area Working Group, Feb. 2006. draft-ietf-tsvwg-sctpsocket-12.txt, work in progress.

[32] 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.

[33] M. Tüxen and T. Dreibholz. Reliable Server Pooling Policies. Internet-Draft Version 05, IETF, RSerPool Working Group, July 2007. draft-ietf-rserpool-policies-05.txt, work in progress.

[34] Ü. Uyar, J. Zheng, M. A. Fecko, and S. Samtani. Performance Study of Reliable Server Pooling. In *Proceedings of the IEEE NCA International Symposium on Network Computing and Applications*, pages 205–212, Cambridge, Massachusetts/U.S.A., Apr. 2003. ISBN 0-7695-1938-5.

[35] 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.

[36] Q. Xie, R. Stewart, M. Stillman, M. Tüxen, and A. Silverton. Endpoint Handlespace Redundancy Protocol (ENRP). Internet-Draft Version 16, IETF, RSerPool Working Group, July 2007. draft-ietf-rserpool-enrp-16.txt, work in progress.

[37] 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.

[38] X. Zhou, T. Dreibholz, and E. P. Rathgeb. Evaluation of a Simple Load Balancing Improvement for Reliable Server Pooling with Heterogeneous Server Pool. In *Proceedings of the IEEE International Conference on Future Communication and Networking (FGCN)*, Jeju Island/South Korea, Dec. 2007.