An Evaluation of the Pool Maintenance Overhead in Reliable Server Pooling Systems[1]

Thomas Dreibholz
Institute for Experimental Mathematics, University of Duisburg-Essen
Ellernstrasse 29, 45326 Essen, Germany
dreibh@iem.uni-due.de

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

Abstract

Currently, the IETF RSerPool WG is standardizing a generic protocol framework for server redundancy and session failover: Reliable Server Pooling (RSerPool). 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. Up to now, 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.

The aim of this article is therefore – after giving 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 in a real network setup.

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

1. Introduction and Scope

In today's Internet, service availability is getting increasingly important. But – in strong contrast to the telecommunications world, where availability is ensured by redundant devices [1] and links – 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 again. This deficiency – once more arisen for the availability of SS7 (Signalling System No. 7 [2]) 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 (e.g. [3,4] present approaches for TCP connection migration), but their combination into a single, 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

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

session failover capabilities to its applications [5,6]. In particular, server redundancy leads to the issues of load distribution and load balancing [7], which are also covered by RSerPool [8,9,10,11]. But in contrast to already available solutions in the area of GRID and high-performance computing [12], 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 [13,14]. This particularly 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 telecommunications equipment or routers.

There has already been some research on the performance of RSerPool for applications like SCTP-based mobility [15,16], VoIP with SIP [17], e-commerce scenarios [18], web server pools [19], IP Flow Information Export (IPFIX) [20,21], management of virtual systems [22], real-time distributed computing [6,8,10,23,11,24,25] and battlefield networks [26]. Furthermore, some ideas and rough performance estimations for the pool management have been described in our paper [14]. 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

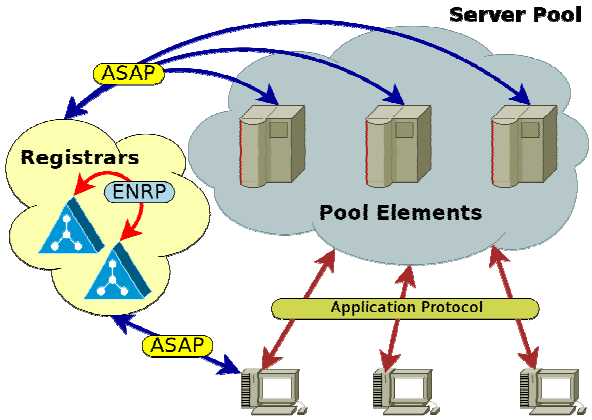

Figure 1. The RSerPool Architecture

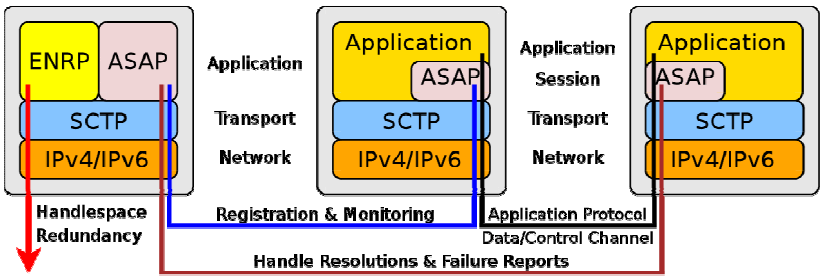

Figure 2. The RSerPool Protocol Stack

Figure 1 provides an illustration of the RSerPool architecture, as defined in [27]; 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 haNdlespace Redundancy Protocol (ENRP [28,29]). 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 [23,6].

A PE can register into a pool at an arbitrary PR of the operation scope, using the Aggregate Server Access Protocol (ASAP [30,29]). 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 [31]. Adaptive and non-adaptive pool policies are defined in [32,33]; for a detailed discussion of these policies, see [8,9,10,11,34,35]. Relevant for this article 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 [14].

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 [5,18]). All associations among the three RSerPool component types (see also figure 2) are usually based on the Stream Control Transmission Protocol (SCTP [36]), which in particular allows for path multi-homing (see [37,38] for details).

3. Our Solution for an Efficient Handlespace Management

Managing a handlespace is the crucial duty of the PR. Figure 3 presents an example for a handlespace. The pool with the PH "e-Shop Database" contains 3 PEs with the IDs 71, 144 and 7466. Each PE has two IP addresses (one IPv4, one IPv6). Since the pool policy is LU, each PE also provides its load state as policy information.

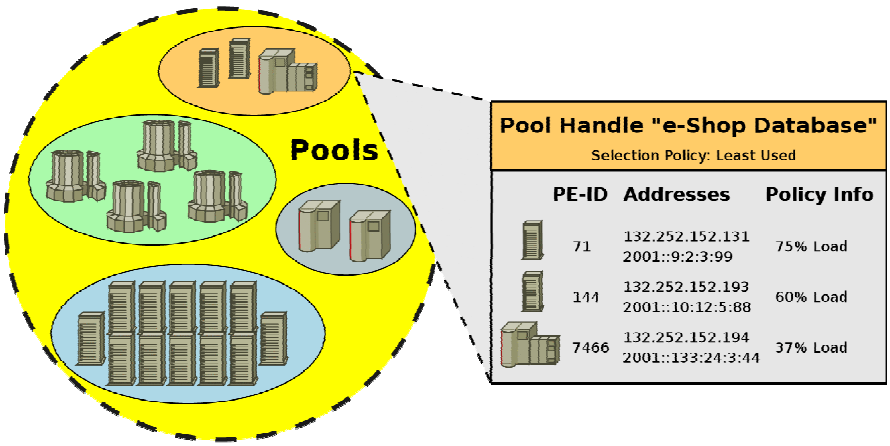

Figure 3. A Handlespace Example

3.1. Requirements to the Handlespace Management

The handlespace management must provide two important properties, with regard to the "lightweight" requirement of the RSerPool architecture:

- server pools may get large (up to many thousands of PEs [24,6]) and
- there may be various pools, each one using its own policy for server selection [8,9] (and new applications may even introduce additional policies [23,10,11]).

In order to keep such handlespaces maintainable, it is obviously necessary to use a unified storage structure (i.e. being usable for all policies) that can be realized efficiently. This handlespace data structure has to support the following six operations:

1. Registration is the registration of a new PE.
2. Deregistration denotes the removal of a PE entry.
3. Re-Registration is an information update for an existing PE entry. In particular, a re-registration is necessary to update the policy information of an adaptive policy (e.g. changing the load state for LU).
4. Handle Resolution means the selection of PEs according to the pool's policy (see [32] for details).
5. Timer denotes scheduling and expiry of a handlespace timer (see also [6]). 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 support the block-wise transfer of the handlespace contents to another PR via ENRP (see [6] for a detailed example).

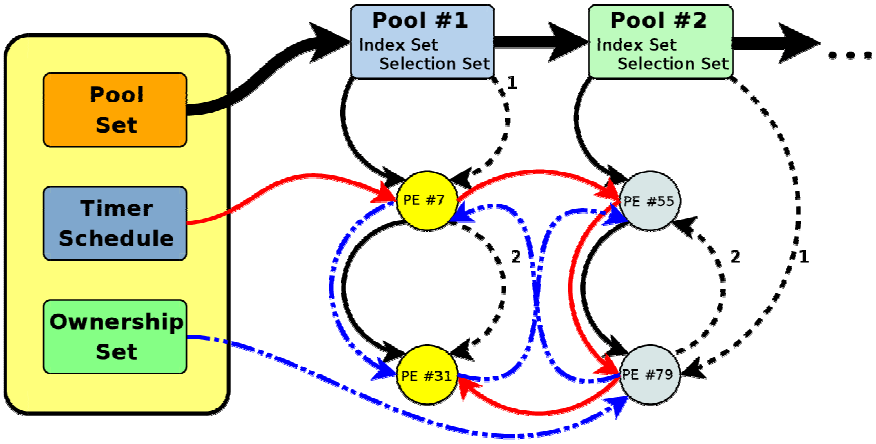

Figure 4. The Handlespace Structure

3.2. Handlespace Structure and Realization of Pool Policies

In [14], we have already proposed to realize the handlespace in form of multiple sets. Figure 4 illustrates our approach: a handlespace consists of a set of pools (Pools Set). Each pool contains a set of PE references sorted by PE ID (Index Set) and a set of these references sorted by a policy-specific sorting order (Selection Set). A policy is simply realized by specifying a sorting order for the Selection Set as well as defining a corresponding selection procedure. Usually, the selection procedure simply takes the first PE from the Selection Set. On selection of a PE entry, its position in the Selection Set is updated. In order to simplify the policy definition, we introduce two helper constructs:

Sequence Numbers: Each pool element PE_i of a pool containing PE_1 to PE_m gets a PE sequence number s_i, which is unique within the pool. The pool sequence number S is defined as:

$$S = 1 + \max\{s_i \mid i \in \{1,...,m\}\}, \qquad (1)$$

i.e. the largest PE sequence number of the pool plus one. Upon registration, re-registration and selection of an element PE_j, its sequence number s_j is set to S and therefore S is increased by one according to its definition in equation 1 (i.e. uniqueness is preserved). Obviously, this operation can be realized in $O(1)$[2] time.

Weights and Weight Sum: Furthermore, each pool element PE_i gets a weight constant $w_i > 0$. Then, the weight sum W of a pool is defined as

$$W = \sum_{1 \le i \le m} w_i.$$

Now, for any number $r \in [1,\ldots,W]$, exactly one PE j fulfils the condition

$$\sum_{1 \le i \le j-1} w_i < r < w_j + \sum_{1 \le i \le j-1} w_i. \qquad (2)$$

Obviously, the weight sum maintenance can also be realized in O(1) time.

[2] $O(f) := \{g : \aleph \to \aleph \mid \exists c > 0, n0\ \forall n \geq n0 : g(n) \leq cf(n)\}$

Using our helper constructs, the definition of policies gets rather easy. For example, the Selection Set sorting order for the LU policy is:

1. The load state in ascending order and
2. The sequence number in ascending order.

The selection procedure is simply to take the first PE of the Selection Set. While the load state obviously ensures that a least-utilized PE is selected, the sequence number not only ensures uniqueness of the composed sorting key, but also provides a round robin selection among multiple least-loaded PEs.

For RAND selection, the weight constant w_i of each PE i corresponds to the PE's proportional selection probability[3]. The selection procedure is to choose a random number $r \in_R [1,\ldots,W]$ and take the element PE_i that uniquely satisfies equation 2. Using a uniform distribution for the choice of r, the selection provides the desired behaviour of the RAND policy. For further policy examples, see [14,39].

3.3. Timer Schedule

Next to policy realization, the handlespace management also has to maintain the PE timers:

- A keep-alive transmission timer schedules the transmission of an ASAP keep-alive to a PE.
- A 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.

At any given time, exactly one of these timers is scheduled for each PE. This means that 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 illustrated in figure 4.

3.4. Checksum and Ownership Set

Handlespace synchronization is the duty of ENRP [28]. In order to detect discrepancies in the handlespace views of different PRs, each PR calculates a checksum[4] 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, 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 all PE references sorted by PR-H (see figure 4).

4. The Performance Evaluation Setup

In [14], 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. A detailed analysis of the handlespace operations throughput was missing.

4.1. Setup for the Performance Measurements of the Handlespace Operations

Therefore, a performance analysis of the handlespace operations themselves will be the first part of this article. Our program for the corresponding measurements simply performs as many operations of the requested type as possible, on a pool being set up in advance. Since

[3] Which is – for the RAND policy – the same for all PEs. Supporting Weighted Random [16] is obvious.
[4] 16-bit Internet Checksum [3]; see also [8, section 3.10.5] for some additional information on the checksum.

registrations and deregistrations cannot be examined separately (the pool would either grow or shrink), these operations are examined in combination: 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 being 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. 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. Setup for the Performance Measurements in a Real System

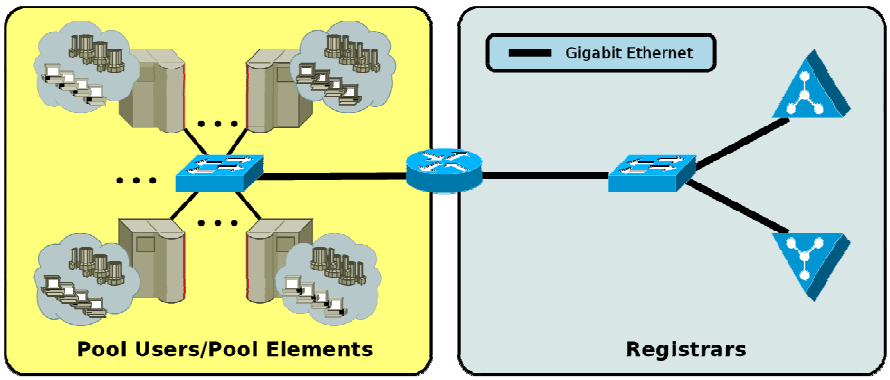

Figure 5. The Measurement Setup

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 5: 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 1) are connected to the router over 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 this distribution. Our RSerPool implementation RSPLIB [6,41,42], 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

As first part of our handlespace performance evaluation, we analyse the throughput of the six handlespace operations (as defined in subsection 1) in a “dry run”.

5.1. Performance of the Handlespace Operations

5.1.1. Registration/Deregistration: The registration/deregistration is the most important operation provided to PEs (see subsection 1) by the PR. As we have already shown in [14,6],

deterministic policies can lead to systematic insertion and removal operations in the Selection Set (see subsection 2). For example, a RR selection always takes the first PE from the Selection Set and re-inserts it as the last one. While this is obviously the worst case for a linear list, the performance of a simple (i.e. non-balanced) binary tree is even worse: the tree degenerates in fact to a linear list. But operations on a tree are slightly more complex than on an actual linear list (see [6] for a detailed performance analysis). Due to these problems, 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 balanced binary trees: the red-black tree [43] (a deterministic approach) and the treap [44] (a randomized approach).

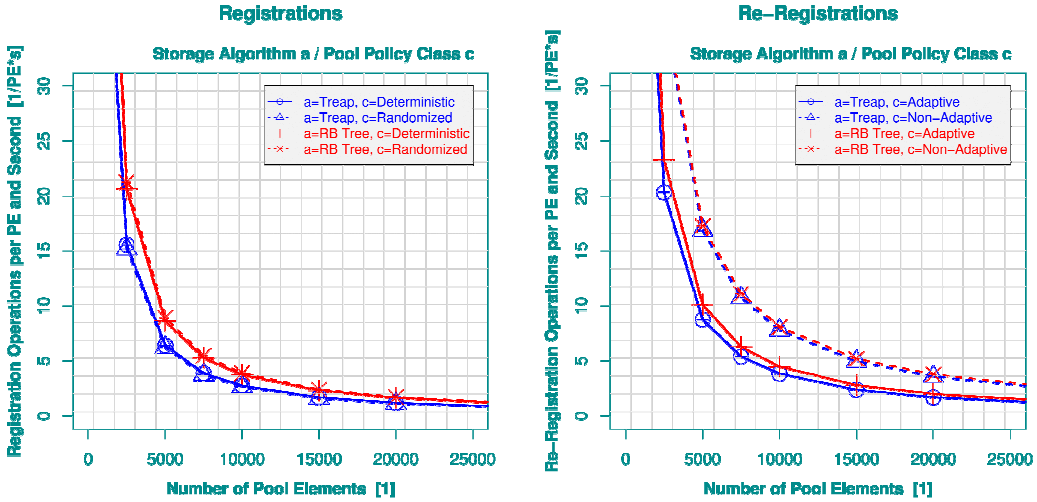

Figure 6. The Scalability of the Registration/Deregistration and Re-Registration Operations

The left-hand side of figure 6 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 of the higher complexity of the insertion and removal algorithms [43] – 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).

5.1.2. Re-Registration: Obviously, the provided registration/deregistration performance 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 when using an adaptive policy. For such a policy (e.g. LU), the position of the PE entry within the Selection Set changes (see also subsection 2). In order to show the impact on the re-registration operations performance, the right-hand side of figure 6 presents the re-registrations throughput per PE and second. For the adaptive policy (here: LU), each re-registration 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.

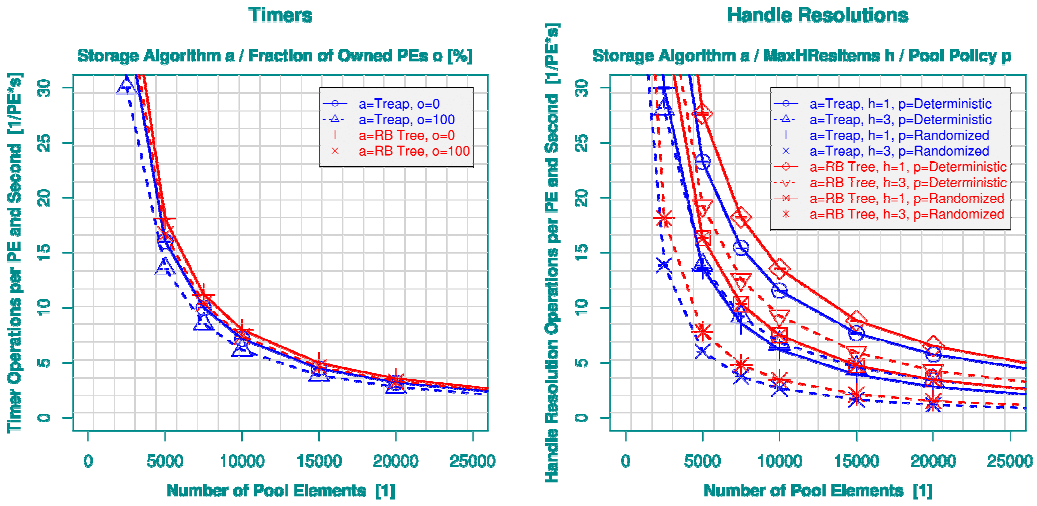

Figure 7. The Scalability of the Timer Handling and Handle Resolution Operations

5.1.3. Timer Handling: The left-hand side of figure 7 presents the timer operations throughput. Obviously, 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. As already expected, the performance for a red-black tree is slightly better than for a treap.

5.1.4. Handle Resolution: 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 is the number of PEs – for red-black tree and treap) in order to obtain a random PE [14] – 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 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 7. 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 [19]), it is possible to apply a handle resolution cache at the PU [8]. 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.

5.1.5. Synchronization: Synchronization – the last of the handlespace operations – only occurs on PR startup or when an inconsistency of the handlespace views has been detected. Clearly, this operation is quite rare (probably only up to a few times per day). However, the actual performance for a pool of 30,000 PEs allows for more than 100 operations per second. Since this is – by orders of magnitude – more than sufficient, a plot has been omitted. Detailed performance results are provided in [6].

5.1.6. Varying the Number of Pools: For the previous measurements, the number of PEs within a single pool has been increased. In fact, splitting up the PEs among a number of pools should not significantly change the performance of the handlespace operations. However, pools are identified by a PH, which is a byte vector. But comparing byte arrays takes more time than simply comparing numbers (e.g. PE IDs, policy information entries). For efficiency reasons, we have therefore decided to limit[5] the PH size to 32 bytes, which are stored inside the pool structure itself. This size is assumed to be sufficient for all current applications.

Our performance evaluations in [6] show that splitting up the number of PEs among different pools only results in some performance reduction when a really large number of PEs (more than 10,000) is distributed among many pools (more than 1,000). Assuming that the number of pools corresponds to the number of RSerPool-based applications in an operation scope (e.g. much less than 100), such cases are not very realistic. Therefore, no additional effort for a speed-up of the pool lookup has been taken yet. In future work on handlespace management, the usage of a hash table for the pool lookup may be considered.

5.2. Performance in the Real System Setup

As shown in subsection 1, our handlespace management approach using red-black trees can handle pools of 10,000 and more PEs. However, in realistic application scenarios, pools of up to a few hundreds of PEs seem to be most probable. Therefore, the following performance evaluation in the real system setup (as described in subsection 2) uses smaller pools, but with a high PR request frequency in order to show the throughput limits.

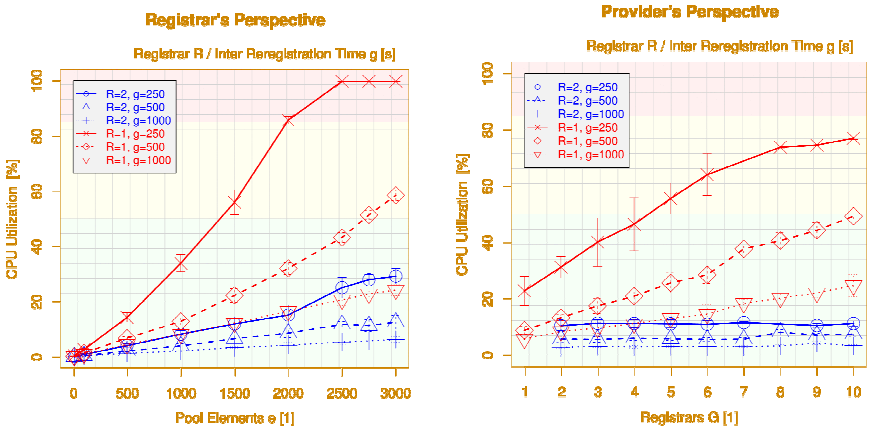

Figure 8. Registrar CPU Utilization for Pool Maintenance

[5] The Internet Draft [37] does not set any limit. In fact, it is only limited by the maximum message size of 64K.

5.2.1. Scalability on the Number of Pool Elements: 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-re-registration time between 250ms and 1000ms (such high rates may occur for adaptive policies). All ASAP (re-)registrations are performed on PR #1 (see figure 5), PR #2 is synchronized by ENRP only. That is, we observe the worst case here. The CPU utilization of PR #1 and PR #2 are shown on the left-hand side of figure 8. Randomized policy results have been omitted, since the results do not differ significantly (see also subsection 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-re-registration 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 [26]. 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?

5.2.2. Scalability on the Number of Registrars: 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 5). Further 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-re-registration 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 8.

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-re-registration 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 [45]). 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 [45], it has not been 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.

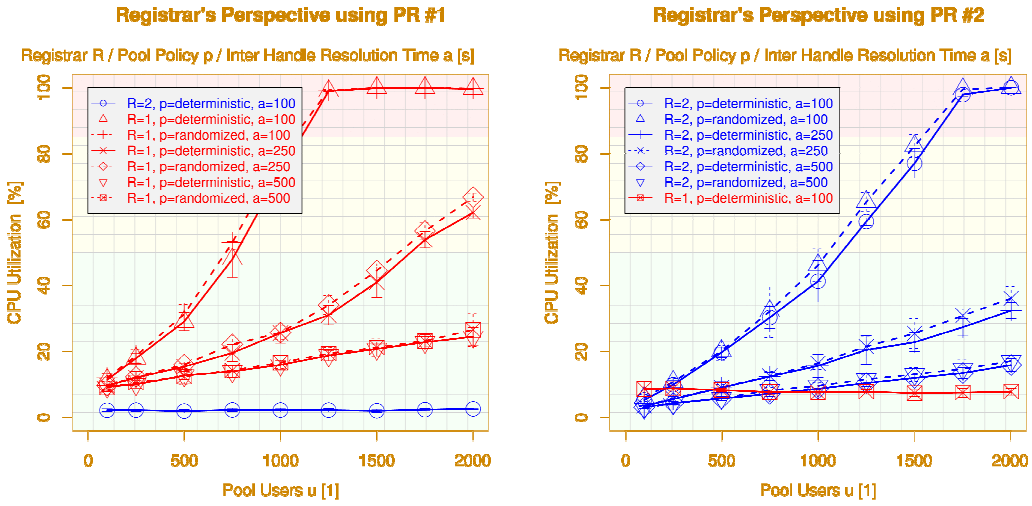

Figure 9. Registrar CPU Utilization for Handle Resolution

5.2.3. Scalability on the Number of Pool Users: 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 5) for a pool of 1,000 PEs using deterministic (solid lines) and randomized policies (dotted lines), an inter-re-registration 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 an 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 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 5) 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 below 100%!

5.3. 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 handlespace 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-re-registration and handle resolution frequency, also much larger scenarios are possible. A room for a further performance improvement will be the SCTP_SENDALL option of the SCTP stack, which will be realized in future SCTP implementations.

6. Conclusions

In this article, we have presented our handlespace management solution which uses red-black trees as base structure to store the handlespace contents. All operations on the handlespace 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 platform which is realistic for routers and embedded systems. Furthermore, we have also shown that our solution is applicable and efficient in a real-world system setup: a system based on the same low-performance CPU is still capable of handling the ASAP/ENRP protocol overhead and the maintenance of the necessary SCTP associations.

Now, we not only use our handlespace management implementation for our Open Source RSerPool implementation RSPLIB [6,41,42], but also for our RSerPool simulation model RSPSIM [46]. 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 such application scenarios. Our ongoing work will furthermore include performance evaluations of using our implementation RSPLIB on Linux-based embedded systems.

References

[1] 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.
[2] ITU-T. Introduction to CCITT Signalling System No. 7. Technical Report Recommendation Q.700, International Telecommunication Union, March 1993.
[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] 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.
[6] 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.
[7] D. Gupta and P. Bepari. Load Sharing in Distributed Systems. In *Proceedings of the National Workshop on Distributed Computing*, January 1999.
[8] 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.

[9] 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.
[10] 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.
[11] 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.
[12] 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.
[13] T. Dreibholz and E. P. Rathgeb. An Evalulation of the Pool Maintenance Overhead in Reliable Server Pooling Systems. In *Proceedings of the IEEE International Conference on Future Generation Communication and Networking (FGCN)*, volume 1, pages 136–143, Jeju Island/South Korea, December 2007. ISBN 0-7695-3048-6.
[14] 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.
[15] 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.
[16] T. Dreibholz and J. Pulinthanath. Applicability of Reliable Server Pooling for SCTP-Based Endpoint Mobility. Internet-Draft Version 03, IETF, Individual Submission, January 2008. draft-dreibholz-rserpool-applic-mobility-03.txt, work in progress.
[17] 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.
[18] 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.
[19] Sohail Ahmed 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, November 2006.
[20] Jobin Pulinthanath. Zuverlässige Übertragung von IPFIX-Nachrichten mit der RSerPool-Architektur. Master's thesis, Universität Duisburg-Essen, Institut für Experimentelle Mathematik, November 2007.
[21] T. Dreibholz, L. Coene, and P. Conrad. Reliable Server Pooling Applicability for IP Flow Information Exchange. Internet-Draft Version 05, IETF, Individual Submission, January 2008. draft-coene-rserpool-applic-ipfix-05.txt, work in progress.
[22] Johannes Formann. *Verfügbarkeit und Verwaltung virtueller Server mit Xen und Reliable Server Pooling verbessern*. Universität Duisburg-Essen, Institut für Experimentelle Mathematik, September 2007. Projektseminararbeit.
[23] 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.
[24] T. Dreibholz. Applicability of Reliable Server Pooling for Real-Time Distributed Computing. Internet-Draft Version 04, IETF, Individual Submission, January 2008. draft-dreibholz-rserpool-applic-distcomp-04.txt, work in progress.
[25] Y. Zhang. Distributed Computing mit Reliable Server Pooling. Master's thesis, Universität Essen, Institut für Experimentelle Mathematik, April 2004.
[26] 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., April 2003. ISBN 0-7695-1938-5.
[27] P. Lei, L. Ong, M. Tüxen, and T. Dreibholz. An Overview of Reliable Server Pooling Protocols. Internet-Draft Version 04, IETF, RSerPool Working Group, January 2008. draft-ietf-rserpool-overview-04.txt, work in progress.
[28] Q. Xie, R. Stewart, M. Stillman, M. Tüxen, and A. Silverton. Endpoint Handlespace Redundancy Protocol (ENRP). Internet-Draft Version 18, IETF, RSerPool Working Group, November 2007. draft-ietf-rserpool-enrp-18.txt, work in progress.

[29] R. Stewart, Q. Xie, M. Stillman, and M. Tüxen. Aggregate Server Access Protocol (ASAP) and Endpoint Handlespace Redundancy Protocol (ENRP) Parameters. Internet-Draft Version 15, IETF, RSerPool Working Group, December 2007. draft-ietf-rserpool-common-param-15.txt, work in progress.
[30] R. Stewart, Q. Xie, M. Stillman, and M. Tüxen. Aggregate Server Access Protcol (ASAP). Internet-Draft Version 18, IETF, RSerPool Working Group, November 2007. draft-ietf-rserpool-asap-18.txt, work in progress.
[31] T. Dreibholz. Handle Resolution Option for ASAP. Internet-Draft Version 01, IETF, Individual Submission, January 2008. draft-dreibholz-rserpool-asap-hropt-01.txt, work in progress.
[32] M. Tüxen and T. Dreibholz. Reliable Server Pooling Policies. Internet-Draft Version 07, IETF, RSerPool Working Group, November 2007. draft-ietf-rserpool-policies-07.txt, work in progress.
[33] T. Dreibholz and X. Zhou. Definition of a Delay Measurement Infrastructure and Delay-Sensitive Least-Used Policy for Reliable Server Pooling. Internet-Draft Version 01, IETF, Individual Submission, January 2008. draft-dreibholz-rserpool-delay-01.txt, work in progress.
[34] 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.
[35] X. Zhou, T. Dreibholz, and E. P. Rathgeb. Evaluation of a Simple Load Balancing Improvement for Reliable Server Pooling with Heterogeneous Server Pools. In *Proceedings of the IEEE International Conference on Future Generation Communication and Networking (FGCN)*, volume 1, pages 173–180, Jeju Island/South Korea, December 2007. ISBN 0-7695-3048-6.
[36] 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, October 2000.
[37] 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.
[38] A. Jungmaier. *Das Transportprotokoll SCTP*. PhD thesis, Universität Duisburg-Essen, Institut für Experimentelle Mathematik, August 2005.
[39] 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.
[40] R. Braden, D. Borman, and C. Partridge. Computing the Internet Checksum. Standards Track RFC 1071, IETF, September 1988.
[41] T. Dreibholz. Das rsplib–Projekt – Hochverfügbarkeit mit Reliable Server Pooling. In *Proceedings of the LinuxTag*, Karlsruhe/Germany, June 2005.
[42] T. Dreibholz and M. Tüxen. High Availability using Reliable Server Pooling. In *Proceedings of the Linux Conference Australia (LCA)*, Perth/Australia, January 2003.
[43] 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., October 1978.
[44] C. Aragon and R. Seidel. Randomized search trees. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pages 540–545, October 1989.
[45] 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 15, IETF, Transport Area Working Group, July 2007. draft-ietf-tsvwg-sctpsocket-15.txt, work in progress.
[46] 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 OMNeT++ Workshop*, Marseille/France, March 2008. ISBN 978-963-9799-20-2.

Authors

Thomas Dreibholz was born in 1976 in Bergneustadt/Germany. He studied computer science at the University of Bonn/Germany and received his diploma (Dipl.-Inform.) degree in 2001 for his thesis "Management of Layered Variable Bitrate Multimedia Streams over DiffServ with Apriori Knowledge". In 2007, he received his Ph.D. degree from the University of Duisburg-Essen for his thesis "Reliable Server Pooling – Evaluation, Optimization and Extension of a Novel IETF Architecture". Since 2001, he is a member of the scientific staff in the Computer Networking Technology group at the Institute for Experimental Mathematics, University of Duisburg-Essen/Germany.
Currently, his main research topic is Reliable Server Pooling (RSerPool). He is not only the author of various research papers – at international conferences and in journals – on this subject, but he also realized the first prototype implementation of the upcoming RSerPool standard as part of a research cooperation project with Siemens, Munich/Germany. Furthermore, he contributed multiple Working Group and Individual Submission Drafts to the IETF RSerPool Working Group's standardization process.

Erwin P. Rathgeb was born in Ulm/Germany in 1958. He received his diploma and Ph.D. degrees in electrical engineering from the University of Stuttgart/Germany in 1985 and 1991, respectively. From 1985 to 1990, he was member of the scientific staff at the Institute of Communication Networks and Computer Engineering (Prof. Paul J. Kühn) at the University of Stuttgart where he was head of a research group on design and analysis of distributed systems.
From 1990 to 1991, he was a member of technical staff at Bellcore, Morristown, NJ/U.S.A., before joining Bosch Telekom in Backnang/Germany. In 1993, he joined Siemens in Munich/Germany. In various positions in systems engineering and product planning, he contributed to concepts for commercial ATM nodes and ATM-based multi-service networks. Since January 1999, he holds the Alfried Krupp von Bohlen und Halbach-Chair for "Computer Networking Technology" at the Institute for Experimental Mathematics, University of Duisburg-Essen/Germany. He is the author of a book on ATM and has published more than 50 papers in journals and at international conferences. Professor Rathgeb is a senior member of IEEE and a member of GI, IFIP and ITG where he is chairman of the expert group on network security.
His current research interests include network security as well as concepts and protocols for next generation internets, in particular the Stream Control Transmission Protocol (SCTP) and Reliable Server Pooling.