# On Improving the Performance of Reliable Server Pooling Systems for Distance-Sensitive Distributed Applications

Thomas Dreibholz and Erwin P. Rathgeb

University of Duisburg-Essen, Ellernstrasse 29, 45326 Essen, Germany,
dreibh@iem.uni-due.de,
http://www.exp-math.uni-essen.de/~dreibh

**Abstract.** Reliable Server Pooling (RSerPool) is a protocol framework for server redundancy and session failover, currently under standardization by the IETF RSerPool WG. While the basic ideas of RSerPool are not new, their combination into a single, unified architecture is. Server pooling becomes increasingly important, because there is a growing amount of availability-critical applications. For a service to survive localized disasters, it is useful to place the servers of a pool at different locations. However, the current version of RSerPool does not incorporate the aspect of component distances in its server selection decisions.
In our paper, we present an approach to add distance-awareness to the RSerPool architecture, based on features of the SCTP transport protocol. This approach is examined and evaluated by simulations. But to also show its usefulness in real life, we furthermore validate our proposed extension by measurements in a PlanetLab-based Internet scenario.

## 1 Introduction

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. These functionalities themselves are not new, but their combination into a single, unified and application-independent framework is.

While the initial motivation and main application of RSerPool is the telephone signalling transport over IP using the SS7 protocol [1], there has already been some research on the applicability and performance of RSerPool for other applications like VoIP with SIP [2, 3], IP Flow Information Export (IPFIX) [4], SCTP-based mobility [5], real-time distributed computing [6–10] and battlefield networks [11]. But a detailed examination of an important application scenario is still missing: short transactions in widely distributed pools. Due to their short processing duration, network transport latency significantly contributes to their overall handling time. The goal of this paper is to optimize RSerPool's support for such transactions by extending RSerPool with an awareness for distances (i.e. latency) between clients and servers, as well as to define an appropriate server selection strategy trying to minimize this distance.

In section 2, we present the scope of RSerPool and related work, section 3 gives a short overview of the RSerPool architecture. A quantification of RSerPool

systems including the definition of performance metrics is given in section 4. This is followed by the description of our distance-sensitive server selection approach. Our approach is simulatively examined in section 6; experimental results in the PlanetLab – showing the usefulness of our approach also in real life – are finally presented in section 7.

## 2 Scope and Related Work

A basic method to improve the availability of a service is server replication. Instead of having one server representing a single point of failure, servers are simply duplicated. In case of a server failure, a client's communication session can perform a failover to another server of the pool [7, 12, 13].

The existence of multiple servers for redundancy automatically leads to the issues of load distribution and load balancing. While load distribution [14] only refers to the assignment of work to a processing element, load balancing refines this definition by requiring the assignment to maintain a balance across the processing elements. This balance refers to an application-specific parameter like CPU load or memory usage. A classification of load distribution algorithms can be found in [15]; the two most important classes are non-adaptive and adaptive algorithms. Adaptive strategies base their assignment decisions on the current status of the processing elements (e.g. CPU load) and therefore require up-to-date information. On the other hand, non-adaptive algorithms do not require such status data. An analysis of adaptive load distribution algorithms can be found in [16]; performance evaluations for web server systems using different algorithms are presented in [17, 18].

The scope of RSerPool [1] is to provide an open, application-independent and highly available framework for the management of server pools and the handling of a logical communication (session) between a client and a pool. Essentially, RSerPool constitutes a communications-oriented overlay network, where its session layer allows for session migration comparable to [19, 20]. While server state replication is highly application-dependent and out of the scope of RSerPool, it provides mechanisms to support arbitrary schemes [7,12]. The pool management provides sophisticated server selection strategies [6, 8, 13, 21] for load balancing, both adaptive and non-adaptive ones. Custom algorithms for new applications can be added easily [22].

## 3 The RSerPool Architecture

An illustration of the RSerPool architecture defined in [1] is shown in figure 1. It consists of three component classes: servers of a pool are called *pool elements* (PE). Each pool is identified by a unique *pool handle* (PH) in the handlespace, i.e. the set of all pools; the handlespace is managed by *pool registrars* (PR). PRs of an *operation scope* synchronize their view of the handlespace using the Endpoint haNdlespace Redundancy Protocol (ENRP [23]), transported via SCTP [24, 25]. 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 (e.g. earthquakes or floodings).

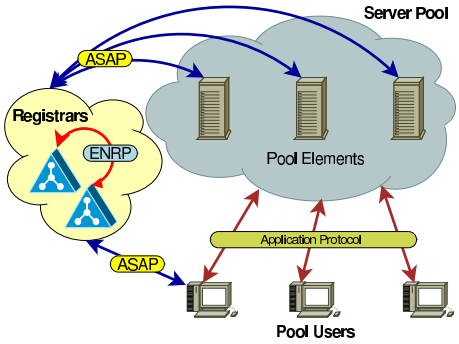


**Fig. 1.** The RSerPool Architecture

PEs choose an arbitrary PR to register into a pool using the Aggregate Server Access Protocol (ASAP [26]). Upon registration at a PR, the chosen PR becomes the Home-PR (PR-H) of the newly registered PE. A PR-H is responsible for monitoring its PEs' availability using ASAP Endpoint Keep-Alive messages (to be acknowledged by the PE within a given timeout) and propagates the information about its PEs to the other PRs of the operation scope via ENRP Update messages.

A client is called *pool user* (PU) in RSerPool terminology. To access the service of a pool given by its PH, a PE has to be selected. This selection – called *handle resolution* in RSerPool terminology – is performed by an arbitrary PR of the operation scope. A PU can request a handle resolution from a PR using the ASAP protocol. The PR selects PE identities using a pool-specific selection rule, called *pool policy*. A set of adaptive and non-adaptive pool policies is defined in [21]; for a detailed discussion of these policies, see [6, 8, 13, 22]. For this paper, only the adaptive Least Used (LU) policy is relevant. LU selects the least-used PE, according to up-to-date load information. The definition of *load* is application-specific and could e.g. be the current number of users, bandwidth or CPU load. For further information on RSerPool, see also [6–8, 13, 22, 27, 28].

## 4 Quantifying a RSerPool System

The service provider side of a RSerPool system consists of a pool of PEs, using a certain server selection policy. Each PE has a request handling *capacity*, which we define in the abstract unit of calculations per second. Depending on the application, an arbitrary view of capacity can be mapped to this definition, e.g. CPU cycles, bandwidth or memory usage. Each request consumes a certain amount of calculations, we call this amount *request size*. A PE can handle multiple requests simultaneously, in a processor sharing mode as commonly used in multitasking operating systems.

On the service user side, there is a set of PUs. The amount of PUs can be given by the ratio between PUs and PEs (PU:PE ratio), which defines the parallelism of the request handling. Each PU generates a new request in an interval denoted as *request interval*. The requests are queued and sequentially assigned to PEs.

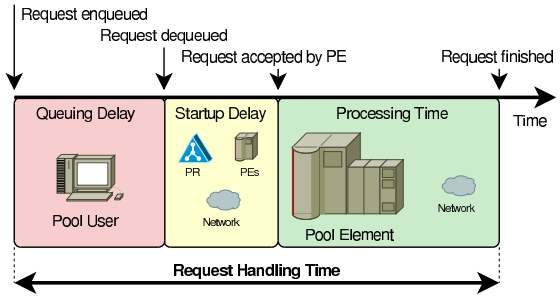


**Fig. 2.** Request Handling Delays

The total delay for handling a request $d_{\text{handling}}$ is defined as the sum of queuing delay, startup delay (dequeuing until reception of acceptance acknowledgement) and processing time (acceptance until finish) as illustrated in figure 2. The *handling speed* (in calculations/s) is defined as:

$$\text{handlingSpeed} = \frac{\text{requestSize}}{d_{\text{handling}}}. \tag{1}$$

Clearly, the user-side performance metric is the handling speed – which should be as fast as possible.

Using the definitions above, it is now possible to give a formula for the system's utilization:

$$\text{systemUtilization} = \text{puToPERatio} * \frac{\frac{\text{requestSize}}{\text{requestInterval}}}{\text{peCapacity}} \tag{2}$$

Obviously, the provider-side performance metric is the system utilization, since only utilized servers gain revenue.

In summary, the workload of a RSerPool system is given by the three dimensions – PU:PE ratio, request interval and request size. In a well-designed client/server system, the amount and capacities of servers are provisioned for a certain *target system utilization*, e.g. 60%. That is, by setting any two of the parameters, the value of the third one can be calculated using equation 2. See [13] for a detailed discussion of the workload parameters.

## 5 A Distance-Aware Least Used Policy

As explained in section 1, PEs may be distributed over a large geographical area to survive localized disasters like an earthquake or tsunami. However, distributing PEs globally could e.g. result in PUs in Europe using PEs in Asia while PUs in America use PEs in Australia. Clearly, for transactions of short duration (compared to the network latency), this results in an increased overall request handling time. Currently, there are no distance-aware pool policies defined. Therefore, our goal is to adapt the Least Used policy to not only take care of PE load but also take the distance between PU and PE into consideration when selecting a server.

### 5.1 How to Quantify Distance?

Two approaches have been considered to actually quantify *distance*: using geographical position information and measuring the delay. Since geographically near endpoints do not necessarily have a low-delay connection (e.g. if using a satellite link), this approach is not useful. Instead, measuring the up-to-date network delay is preferable. Clearly, this implies the need for a measurement component. But in case of SCTP connections, this can be realized quite easily: the SCTP protocol [24] – used for the RSerPool communication – already calculates a smoothed round-trip time (RTT) for its paths. This RTT only has to be queried via the standard SCTP API [29]. Using the RTT, the end-to-end delay between two associated components is approximately $\frac{\mathrm{RTT}}{2}$.

In real networks, there may be negligible delay differences: for example, the delay between a PU and PE #1 is 5ms and the latency between the PU and PE #2 is 6ms. From the service user's perspective, such minor delay differences are negligible and furthermore unavoidable in Internet scenarios. Therefore, the distance parameter between two components $A$ and $B$ can be defined as follows:

$$\mathrm{Distance}_{A \leftrightarrow B} = \mathrm{DistanceStep} * \mathrm{round}\left(\frac{\frac{\mathrm{RTT}}{2}}{\mathrm{DistanceStep}}\right) \tag{3}$$

That is, the distance parameter is defined as the nearest integer multiple of the constant DistanceStep for the measured delay (i.e. $\frac{\mathrm{RTT}}{2}$).

### 5.2 An Environment for Distance-Aware Policies

In order to define a distance-aware policy, it is first necessary to define a basic rule: PEs and PUs choose "nearby" PRs. Since the operation scope of RSerPool is restricted to a single organization, this condition can be met easily by appropriately locating PRs. A PR-H can measure the delay of the ASAP associations to each of its PEs. As part of its ENRP updates to other PRs, it can report this measured delay together with the PE information. A non-PR-H receiving such an update simply adds the delay of the ENRP association with the PR-H to the PE's reported delay. Now, each PR can approximate the distance to every PE in the operation scope using equation 3. Note, that delay changes are propagated to all PRs upon PE re-registrations, i.e. the delay information (and the approximated distance) dynamically adapts to the state of the network.

### 5.3 The Policy Definition

As shown in [13], the Least Used policy provides the best performance and therefore becomes the obvious candidate to be extended with distance sensitivity: instead of only taking the load value into account for server selection, the new load value $\mathrm{Load}^*$ is computed by increasing the PE's reported value Load by a distance-dependent *Distance Penalty Factor* (DPF) as follows:

$$\mathrm{Load}^* = \mathrm{Load} + \underbrace{\mathrm{Distance} * \mathrm{LoadDPF}}_{\text{Distance Penalty Factor}} . \tag{4}$$

The constant LoadDPF describes the load units per millisecond the actual load value is increased for every millisecond of the network delay. That is, the unit for LoadDPF is $ms^{-1}$. Due to the DPF parameter, the new policy is denoted as Least Used with DPF (LU-DPF). It simply selects the PE with the lowest value of Load$^*$. If there are multiple lowest-valued PEs, round robin selection is applied among them.

Note, that the sorting of the PEs for selection is still per-PR rather than per-PU. This property is crucial for the efficiency of the handlespace management, since it allows for maintaining a LU-DPF pool by a set of PE identities sorted by Load$^*$ values. As shown in [22], this is very efficiently realizable.

## 6 Simulative Results

In order to examine the new policy, a simulative proof of concept has been performed first.

### 6.1 The Simulation Model

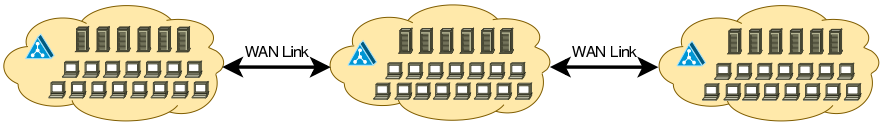


**Fig. 3.** The Simulation Setup

For our performance analysis, we have developed a simulation model using OMNeT++ [30], containing the protocols ASAP [26] and ENRP [23], a PR module and PE and PU modules modelling the request handling scenario defined in section 4. The simulation setup as shown in figure 3 consists of LANs interconnected by WAN links. Each LAN contains one PR and a variable amount of PEs and PUs – all in the same operation scope. As shown in [22], the component latencies are negligible and therefore have been omitted. As in [13], a negative exponential distribution is used for request intervals and sizes. For the policies, the *load* of a PE is defined as the current amount of simultaneously handled requests. The capacity of a PE is $10^6$ calculations/s, the simulation runtime is 15 minutes; each simulation has been repeated 24 times with different seeds to achieve statistical accuracy. All results plots show the average values and their 95% confidence intervals.

### 6.2 A Proof of Concept

In our first simulation, we provide a proof of concept for the LU-DPF policy in a scenario consisting of 3 LANs, each containing 10 PEs. We have used a fixed inter-component LAN delay of 10ms and have varied the WAN delay from 0ms to 500ms (these settings are based on PlanetLab measurements and will be motivated in detail in subsection 7.1). The PU:PE amount ratio $r$ varies from 1

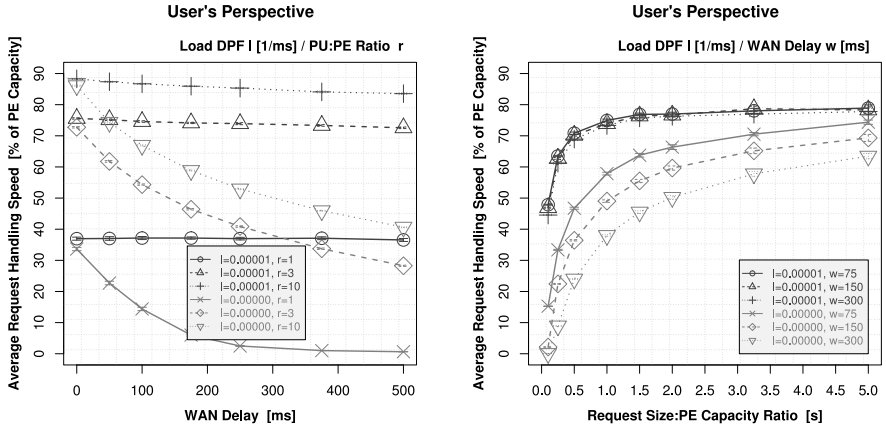


**Fig. 4.** A Proof of Concept

to 10 for DPF settings of 0 (i.e. the LU-DPF policy is equal to plain LU) and $1*10^{-5}$ (this parameter will be examined in detail in subsection 6.3); the average request size is $10^6$ calculations (i.e. if processed exclusively, the processing time of an average request is 1s) and the target system utilization is 60% (the request interval is calculated using equation 2). The setting of DistanceStep is 1ms.

The left-hand side of figure 4 shows the resulting handling speed (in % of the PE capacity). As it is already shown in [6, 13], the PU:PE ratio $r$ is the most critical load workload parameter. For smaller values of $r$, the per-PU load is highest, leading to higher performance degradation upon "bad" server selections. While the impact on the system utilization is negligible (therefore, the plot is omitted here), this effect is clearly visible for the handling speed – even if there is no WAN delay. As expected, the handling speed for a DPF of 0 (i.e. the policy behaves like plain LU) becomes smaller if the WAN delay increases. However, also taking the network delay into account (by setting the DPF to $1*10^{-5}$), the impact of the WAN delay becomes hardly visible. That is, our new LU-DPF policy has shown the desired effect: avoiding unnecessary delays by preferably using local PEs.

Obviously, the request handling speed for short transactions strongly depends on the network latency. To make this effect clearer, the right-hand side of figure 4 shows the speed results for varying the request size:PE capacity ratio for different WAN delay and DPF settings, using a PU:PE ratio of 3 and again a target system utilization of 60%. As expected, the handling speed for a DPF of 0 is significantly reduced by an increased WAN latency. However, if taking care of the delay by using a DPF of $1*10^{-5}$, the request handling speed becomes almost delay-independent. While it is obvious that the handling speed decreases with the request size for a DPF of 0, this effect can – in a smaller degree – also be observed for a DPF of $1*10^{-5}$: although the PUs preferably choose local PEs, there is still the inter-component LAN latency (10ms) which contributes to the startup delay of the requests. However, compared to the results of plain Least Used selection, the handling speed impact of small requests is significantly reduced (e.g. still a handling speed of 48% for a DPF of $1*10^{-5}$ vs. almost 0%

for a DPF of 0, at a WAN delay of 300ms for a request size:PE capacity ratio of 0.1). Therefore, the next question to be answered is: how to configure the DPF setting appropriately?

### 6.3 Configuring the Distance Penalty Factor

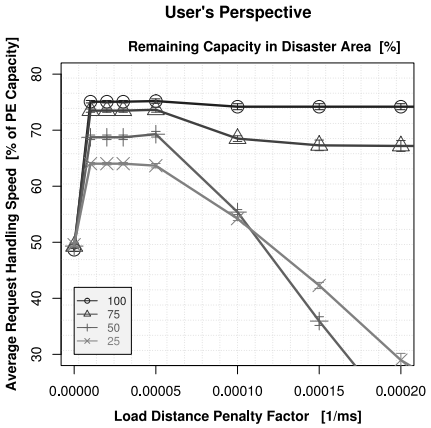


**Fig. 5.** Finding a Reasonable DPF Setting

After the first promising results from our proof-of-concept simulation, the following tasks have to be performed: (1) Find an appropriate DPF parameter setting and (2) verify that LU-DPF still provides a useful performance for the case that PEs in a region become unavailable (localized disaster) and remote PEs should be used instead.

To answer the questions, we have performed simulations for a large parameter space; the scenario presented here consists of a subset, carefully chosen to illustrate the essential effects. In this simulation, the DPF value is varied in a scenario consisting of 3 LANs with 12 PEs in each LAN, for a WAN delay of 150ms. Furthermore, the amount of PEs in the last LAN has been varied between 100% (i.e. all 12 PEs) and 25% (only 3 PEs) to simulate a localized disaster. To compensate the capacity loss in the last LAN, additional PEs have been equally distributed to the other 2 LANs. That is, the overall capacity of the pool always remains the same. All other parameters have been set as for the proof-of-concept simulation described in subsection 6.2.

While there is no significant impact on the utilization (therefore, we omit a figure), the handling speed as shown in figure 5 is significantly increased even for a small DPF setting. As expected, a higher value of the DPF setting has no impact if all PEs in the last LAN are available. For this scenario, the server selection mainly behaves as for three separate pools. However, decreasing the amount of PEs in the last LAN and increasing the amount in the other LANs, the scenario becomes heterogeneous. For setting the DPF parameter too high, the handling speed decreases and – for a sufficiently large setting (e.g. $15*10^{-5}$ for $\leq 50\%$ PEs in the last LAN) – the handling speed is even exceeded by plain LU (i.e. a DPF of 0).

In summary, the resulting general guideline on setting the DPF parameter is rather simple: set it to a value slightly above 0 – e.g. $1*10^{-5}$. In case of having multiple least-loaded PEs, this setting gives the server selection a preference for the nearest PE (see equation 4). Furthermore, it also provides an improved performance in scenarios of localized disasters – by enabling the selection of remote PEs if necessary. That is, LU-DPF can achieve a significant performance benefit over LU – at least in simulations. But since our goal is to also apply our new policy in real life, the next step is to validate our results by performing experiments in the Internet.

## 7 Experimental Results

Experimental results are necessary, because simulating all effects of the real Internet – including temporary QoS variations and the SCTP protocol's reaction – is almost impossible.

### 7.1 The Measurement Setup

In order to perform realistic measurements, we have used the PlanetLab [31], a set of globally distributed hosts in the Internet. Based on our SCTP prototype implementation sctplib [32] and our RSerPool implementation rsplib [27, 28, 33], we have realized an application model which is compatible to the simulated one. The setup consists of components distributed into three regions: Europe (mainly Germany), America (U.S.A., mainly West Coast) and Asia (mainly Japan). Each region contains one PR, which is used by the region's 5 PEs and 15 PUs. As for the simulations, the PEs have a capacity of $10^6$ calculations/s; the PUs use a request size of $10^6$ calculations and an average request interval of 7.5s (both using negative exponential distribution).

Tests using `ping` and `traceroute` have shown latencies between 5ms to 15ms within the regions; the inter-region delay varies between about 75ms to 150ms between Europe and America and America and Asia, as well as about 250ms to 350ms between Europe and Asia (routed via the U.S.A.). The delays between any two endpoints have not shown a significant variation. That is, it can be assumed that there has been sufficient bandwidth available. This is also realistic for RSerPool scenarios, since all components belong to a single operation scope (e.g. a company) and QoS mechanisms can therefore be applied easily (e.g. WAN connections via DiffServ-based VPN links using an appropriate SLA). Based on the delay experiments, DistanceStep has been set to 75ms.

### 7.2 Measurements

Each measurement run has a runtime of 65 minutes, with the following actions: at $t_1$=15min, 2 of the 5 Asian PEs are turned off; at $t_2$=30min, two additional PEs are turned on – one in America, the other one in Europe. At $t_3$=45min, the failure in Asia is repaired. Both PEs are again added to the pool, increasing its total capacity. The two additional PEs in Europe and America are turned off at

| Interval | Network State | LU-DPF | LU | Improvement |
|---|---|---|---|---|
| 1m - 14m | Normal Operation I | 2.17s ± 0.05 | 2.63s ± 0.05 | 17.5% |
| 16m - 29m | Failure in Asia | 2.55s ± 0.02 | 2.78s ± 0.02 | 8.3% |
| 31m - 45m | Added Backup Capacity | 2.54s ± 0.02 | 2.71s ± 0.02 | 6.3% |
| 46m - 49m | Failure Resolved | 2.35s ± 0.06 | 2.55s ± 0.03 | 7.8% |
| 51m - 64m | Normal Operation II | 2.08s ± 0.02 | 2.47s ± 0.02 | 15.8% |

**Table 1.** Average Request Handling Time Results

$t_4$=50min. In order to achieve sufficient statistical accuracy, the measurement has been performed 22 times, covering a total runtime of about 35 hours.

In order to compare the results for LU and LU-DPF (with a DPF setting of $1*10^{-5}$), we have performed both experiment runs simultaneously. That is, we have set up two pools – one for LU, the other one for LU-DPF. On each of the PE hosts, two PE instances have been started: one registering into the LU pool, the other one registering into the LU-DPF pool. Analogously, each PU host runs two PU instances: one using the LU pool, the other one using the LU-DPF pool. Due to the simultaneous execution, we ensure that both measurements are equally affected by temporal variations of the Internet's QoS conditions and keep the results comparable.

The resulting average request handling times and their 95% confidence intervals for both measurements are presented in table 1. Note, that we show the average over intervals beginning one minute after and ending one minute before a system condition change, since the latency to log into a PlanetLab node to start or stop a component may take up to about 30s. Furthermore, small deviations of the hosts' clocks may be possible. Comparing the results for LU and LU-DPF, the LU-DPF policy provides a significant handling speed gain: between 17.6% and 15.8% for the two phases of normal operation and still around 8% during the failure and its resolution in Asia.

It is important to note that the performance in the "failure resolved" state is lower than for the "normal operation" states, although there are additional PEs in America and Europe: the over-capacity in these regions attracts the assignment of requests from Asia. This effect – the assignment of requests to slightly-loaded servers – is a property of all load-based policies; trying to avoid it by simply using a high LoadDPF setting would not lead to a performance improvement, as described in subsection 6.3.

In summary, the measurements have shown that our new LU-DPF policy is also working as intended in the real Internet.

## 8 Conclusion and Future Work

In this paper, we have presented a new, efficiently implementable, adaptive, distance-aware pool policy for RSerPool, which bases its server selection decisions on delay (measured by a feature of the SCTP protocol) and server load. The goal of this policy is to minimize the request handling time in situations where the processing time on the server is in the range of the network's transport latency.

To show the usefulness of our new policy, we have provided simulation results first. Furthermore, in order to also validate the policy's applicability in real life, we have performed measurements in the Internet using the PlanetLab. Both

– simulations and measurements – have shown that our new policy can achieve a significant performance gain.

The future goal of our ongoing RSerPool research activities is to further examine the new policy under a broader range of network and application parameters – again, by simulations as well as measurements for validation. Furthermore, we intend to propose our new policy for standardization by the IETF RSerPool WG.

## References


1. M. Tüxen, Q. Xie, R. Stewart, M. Shore, J. Loughney, and A. Silverton. Architecture for Reliable Server Pooling. Technical Report Version 11, IETF, RSerPool Working Group, March 2006. draft-ietf-rserpool-arch-11.txt, work in progress.
2. M. Bozinovski. *Fault-tolerant platforms for IP-based Session Control Systems*. PhD thesis, Aalborg University, Aalborg/Denmark, June 2004.
3. 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 2002, Mobile/Wireless Computing and Communication Systems II*, volume X, Orlando, Florida/U.S.A., July 2002. ISBN 980-07-8150-1.
4. T. Dreibholz, L. Coene, and P. Conrad. Reliable Server pool use in IP flow information exchange. Internet-Draft Version 02, IETF, Individual Submission, February 2006. draft-coene-rserpool-applic-ipfix-02.txt, work in progress.
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*, pages 99–108, Königswinter/Germany, November 2003. ISBN 0-7695-2037-5.
6. 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.
7. 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, August 2005. ISBN 0-7695-2431-1.
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*, volume 2, pages 564–574, Saint Gilles Les Bains/Reunion Island, April 2005. ISBN 3-540-25338-6.
9. T. Dreibholz and E. P. Rathgeb. An Application Demonstration of the Reliable Server Pooling Framework. In *Proceedings of the 24th IEEE INFOCOM*, Miami, Florida/U.S.A., March 2005. Demonstration and poster presentation.
10. T. Dreibholz. Applicability of Reliable Server Pooling for Real-Time Distributed Computing. Internet-Draft Version 01, IETF, Individual Submission, February 2006. draft-dreibholz-rserpool-applic-distcomp-01.txt, work in progress.
11. 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.
12. T. Dreibholz. An Efficient Approach for State Sharing in Server Pools. In *Proceedings of the 27th IEEE Local Computer Networks Conference*, pages 348–352, Tampa, Florida/U.S.A., October 2002. ISBN 0-7695-1591-6.
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*

*30th Anniversary*, pages 200–208, Sydney/Australia, November 2005. ISBN 0-7695-2421-4.
14. E. Berger and J. C. Browne. Scalable Load Distribution and Load Balancing for Dynamic Parallel Programs. In *Proceedings of the International Workshop on Cluster-Based Computing 99*, Rhodes/Greece, June 1999.
15. D. Gupta and P. Bepari. Load Sharing in Distributed Systems. In *Proceedings of the National Workshop on Distributed Computing*, January 1999.
16. O. Kremien and J. Kramer. Methodical Analysis of Adaptive Load Sharing Algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 3(6), 1992.
17. M. Colajanni and P. S. Yu. A Performance Study of Robust Load Sharing Strategies for Distributed Heterogeneous Web Server Systems. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):398–414, 2002.
18. S. G. Dykes, K. A. Robbins, and C. L. Jeffery. An Empirical Evaluation of Client-Side Server Selection Algorithms. In *Proceedings of the IEEE Infocom 2000*, volume 3, pages 1361–1370, Tel Aviv/Israel, March 2000. ISBN 0-7803-5880-5.
19. 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.
20. 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.
21. M. Tüxen and T. Dreibholz. Reliable Server Pooling Policies. Internet-Draft Version 02, IETF, RSerPool Working Group, February 2006. draft-ietf-rserpool-policies-02.txt, work in progress.
22. T. Dreibholz and E. P. Rathgeb. Implementing the Reliable Server Pooling Framework. In *Proceedings of the 8th IEEE International Conference on Telecommunications*, volume 1, pages 21–28, Zagreb/Croatia, June 2005. ISBN 953-184-081-4.
23. Q. Xie, R. Stewart, M. Stillman, M. Tüxen, and A. Silverton. Endpoint Handlespace Redundancy Protocol (ENRP). Internet-Draft Version 13, IETF, RSerPool Working Group, February 2006. draft-ietf-rserpool-enrp-13.txt, work in progress.
24. 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.
25. A. Jungmaier. *Das Transportprotokoll SCTP*. PhD thesis, Universität Duisburg-Essen, Institut für Experimentelle Mathematik, August 2005.
26. R. Stewart, Q. Xie, M. Stillman, and M. Tüxen. Aggregate Server Access Protcol (ASAP). Technical Report Version 13, IETF, RSerPool Working Group, February 2006. draft-ietf-rserpool-asap-13.txt, work in progress.
27. T. Dreibholz. Das rsplib–Projekt – Hochverfügbarkeit mit Reliable Server Pooling. In *Proceedings of the LinuxTag*, Karlsruhe/Germany, June 2005.
28. T. Dreibholz and M. Tüxen. High Availability using Reliable Server Pooling. In *Proceedings of the Linux Conference Australia*, Perth/Australia, January 2003.
29. 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, February 2006. draft-ietf-tsvwg-sctpsocket-12.txt, work in progress.
30. A. Varga. OMNeT++ Discrete Event Simulation System, 2005.
31. Larry Peterson and Timothy Roscoe. The Design Principles of PlanetLab. *Operating Systems Review*, 40(1):11–16, January 2006.
32. M. Tüxen. The sctplib Prototype, 2001.
33. T. Dreibholz. Thomas Dreibholz's RSerPool Page, 2006.