Load Distribution Performance of the Reliable Server Pooling Framework

Thomas Dreibholz[1] and Erwin P. Rathgeb[1] and Michael Tüxen[2]

[1] University of Duisburg-Essen
Institute for Experimental Mathematics
Ellernstraße 29, D-45326 Essen, Germany
Tel: +49 201 183-7637, Fax: +49 201 183-7673
dreibh@exp-math.uni-essen.de
[2] University of Applied Sciences, Münster
Fachbereich Elektrotechnik und Informatik
Stegerwaldstraße 39, D-48565 Steinfurt, Germany
Tel: +49 2551 962550
tuexen@fh-muenster.de

Abstract. The Reliable Server Pooling (RSerPool) protocol suite currently under standardization by the IETF is designed to build systems providing highly available services by providing mechanisms and protocols for establishing, configuring, accessing and monitoring pools of server resources.
While availability is one main aspect of RSerPool, load distribution is another. Since most of the time a server pool system runs without component failures, optimal performance is an extremely important issue for the productivity and cost-efficiency of the system. In this paper, we therefore focus especially on the load distribution performance of RSerPool in scenarios without failures, presenting a quantitative performance comparison of the different load distribution strategies (called pool policies) defined in the RSerPool specifications. Based on the results, we propose some new pool policies providing significant performance enhancements compared to those currently defined in the standards documents.

1 The Reliable Server Pooling Architecture

The convergence of classical circuit-switched networks (i.e. PSTN/ISDN) and data networks (i.e. IP-based) is rapidly progressing. This implies that SS7 PSTN signalling [1] has to also be transported over IP networks. Since SS7 signalling networks offer a very high degree of availability (e.g. at most 10 minutes downtime per year for any signalling relation between two signalling endpoints; for more information see [2]), all links and components of the network devices must be fault-tolerant, and this is achieved through having multiple links, and using the link redundancy concept of SCTP [3]. When transporting signalling over IP networks, such concepts also have to be applied to achieve the required availability. Link redundancy in IP networks is supported using the Stream Control Transmission Protocol (SCTP) providing multiple network paths and fast failover [4,5]; redundancy of network device components is supported by the SGP/ASP (signalling gateway process/application server process) concept. However, this concept has some limitations: there is no support of dynamic addition and removal of components; it has only limited ways of server selection and no specific failover procedures and inconsistent application to different SS7 adaptation layers.

To cope with the challenge of creating a unified, lightweight, realtime, scalable and extendable redundancy solution (see [6] for details), the IETF Reliable Server Pooling

Working Group was founded to specify and define the Reliable Server Pooling Concept. An overview of the architecture currently under standardization and described by several Internet Drafts is shown in figure 1.

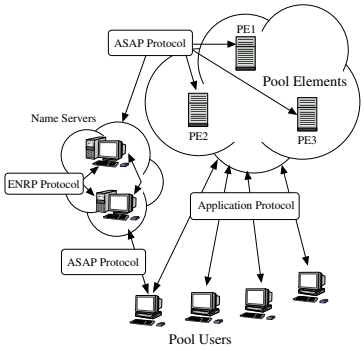

Fig. 1. The RSerPool Architecture

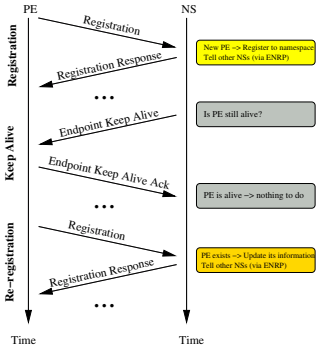

Fig. 2. PE Registration and Monitoring

Multiple server elements providing the same service belong to a *server pool* to provide redundancy on one hand and scalability on the other. Server pools are identified by a unique ID called *pool handle* (PH) within the set of all server pools, the *namespace*. A server in a pool is called a *pool element* (PE) of the respective pool. The namespace is managed by redundant *name servers* (NS). The name servers synchronize their view of the namespace using the Endpoint Name Resolution Protocol (ENRP [7]). NSs announce themselves using broadcast/multicast mechanisms, i.e. it is not necessary (but still possible) to pre-configure any NS address into the other components described in the following.

PEs providing a specific service can register for a corresponding pool at an arbitrary NS using the Aggregate Server Access Protocol (ASAP [8]) as shown in figure 2. The *home NS* is the NS which was chosen by the PE for initial registration. It monitors the PE using SCTP heartbeats (layer 4, not shown in figure) and ASAP Endpoint Keep Alives. The frequency of monitoring messages depends on the availability requirements of the provided service. When a PE becomes unavailable, it is immediately removed from the namespace by its home NS. A PE can also intentionally de-register from the namespace by an ASAP de-registration allowing for dynamic reconfiguration of the server pools. NS failures are handled by requiring PEs to re-register regularly (and therefore chosing a new NS when necessary). Re-registration also makes it possible for the PEs to update their registration information (e.g. transport addresses or policy states).

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

When a client requests a service from a pool, it first asks an *arbitrary* NS to translate the pool handle to a list of PE identities selected by the pool's selection policy (*pool policy*), e.g round robin or least used (to be explained in detail in section 2). The NS does not return the total number of identities in the pool, instead it has a constant value, *MaxNResItems*, which dictates how many PE identities should be returned. For example, if there were 5 PEs and *MaxNResItems* was set to 3, then the NS would select 3 of the 5; conversely, if *MaxNResItems* were set to 5, and there were only 3 PEs, then all 3

PE identities would be returned. The PU adds this list of PE identities to its local cache (denoted as PU-side cache) and again selects *one* entry by policy from its cache. To this selected PE, a connection is established, using the application's protocol, to actually use the service. The client then becomes a *pool user* (PU) of the PE's pool.

It has to be emphasized, that there are two locations where a selection by pool policy is applied during this process: at the NS when compiling the list of PEs and in the local PU-side cache where the target PE is selected from the list.

The default timeout of the PU-side cache, called *stale cache value* is 30s [8]. That is, within this time period, subsequent name resolutions of the PU may be satisfied directly from the PU-side cache, saving the effort and bandwidth for asking the NS.

If the connection to the selected PE fails, e.g. due to overload or failure of the PE, the PU selects another PE from its list and tries again. Optionally, the PU can report a PE failure to a NS, which may then decide to remove this PE from the namespace. If the PE failure occurs during an active connection, a new connection to another available PE is established and an application-specific failover procedure is invoked.

RSerPool supports optional client-based state synchronization [9] for failover. That is, a PE can store its current state with respect to a specific connection in a *state cookie* which is sent to the corresponding PU. When a failover to a new PE is necessary, the PU can send this state cookie to the new PE, which can then restore the state and resume service at this point. However, RSerPool is not restricted to client-based state synchronization, any other application-specific failover procedure can be used as well.

The lightweight, realtime, scalable and extendable architecture of RSerPool is not only applicable to transport of SS7-based telephony signalling. Other application scenarios include reliable SIP based telephony [10], mobility management [11] and the management of distributed computing pools [12,13]. Furthermore, additional application scenarios in the area of load distribution and balancing are currently under discussion within the IETF RSerPool Working Group.

Currently, there are two existing implementations of RSerPool: the authors' own GPL-licensed Open Source prototype *rsplib* [12] and a closed source version, by Motorola [14]. The standards documents are currently Internet Drafts, having some open issues. These open questions are: evaluation of reliability aspects, state synchronization between PEs and load distribution among PEs.

In this paper, we focus our attention on the third point above – the open topic of load distribution among PEs, which is crucial for the efficient operation of server pools.

2 Load Distribution and Balancing

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

For RSerPool, *load* only denotes a constant in the range from 0% (not loaded) to 100% (fully loaded) which describes a PE's actual normalized resource utilization. The definition of a mapping from resource utilization to a load value is application-dependent. Formally, such a mapping function is defined as $m(u) := \frac{u}{U_{max}-U_{min}}$, $u \in \{U_{min}, ..., U_{max}\} \subset \mathbb{R}$, where U_{min} denotes the application's minimum and U_{max} the maximum possible resource utilization.

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

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

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

Currently, the drafts [8] and [15] define the following four pool policies: Round Robin (RR), Weighted Round Robin (WRR), Least Used (LU) and Leased Used with Degradation (LUD). The RR and WRR policies are called *static* policies because they do not require and incorporate any information on the actual load state of the active PEs when making the selection. However, they are "stateful" in a sense that the current selection depends on the selection made in the previous request. This can – if carelessly implemented - lead to a severe performance degradation in some situations as shown in section 5.1. The LU and LUD policies try to select the PEs which currently carry the least load. Therefore, the PEs are required to propagate their load information into the namespace (by doing a re-registration) regularly or upon changes. These required dynamic policy information changes lead to the term *dynamic policy*. It is obvious that the dynamic policies have the potential to provide a better load sharing resulting in a better overall performance. However, the tradeoff is that these policies require additional signalling overhead in order to keep the load information sufficiently current. These effects will be quantified in section 5.2.

3 Detailed Definition of Pool Policies

Even though the pool policies are mentioned in the current standards documents, their definitions are not sufficient for a consistent implementation. E.g. it was not defined how to perform the load degradation in the LUD policy. Therefore, to be able to do the implementation as well as the simulation study, we refined the definition of these policies in a first step as described below and introduced these definitions into the standardization process as Internet Draft [16]. Furthermore, based on our quantitative evaluation described in section 5 we propose modifications as well as additional, more efficient policies.

3.1 Policies defined in the standards documents

Round Robin (RR) and Weighted Round Robin (WRR) Using this policy, the elements in a list of PEs should be used in a round robin fashion, starting with the first PE. If all elements of the list have been used, selection starts again from the beginning of the list.

The RR policy does not take into account the fact that servers may have different capacities. Therefore, WRR tries to improve the overall performance by selecting more powerful servers more often. The capacity of a PE is reflected by its integer *weight* constant. This constant specifies how many times per round robin round a PE should be selected. For example, this can be realized using a round robin list where each PE gets as many entries as its *weight* constant specifies. Obviously, RR can be viewed as a special case of WRR with all weight factors set to identical values.

Least Used (LU) The effort to serve a request may – in some application scenarios – vary significantly. Therefore, the LU policy tries to incorporate the actual load value of a server into the selection decision. When selecting a PE under the LU policy, the least loaded PE is chosen. That is, a NS executing the selection has to know the current load states of all PEs. For the case that there are multiple PEs of the same load, round robin selection between these equal-loaded PEs should be applied.

Least Used with Degradation (LUD) When using the LU policy, load information updates are propagated among the NSs using ENRP Peer Update messages [7]. However, they are not propagated immediately to the PU-side caches as these are only updated by ASAP Name Resolutions [8]. To keep the resulting namespace inconsistencies small, the LUD policy extends LU by a per-PE load degradation constant (this was not defined by the original ASAP draft, it had to be added by us to make this policy useful [16]). This load degradation constant specifies in units of the load how much a new request to the PE will increase its load. For the file transfer example above, a new request means a new user on this PE. Therefore, its load increases by $m(1) = \frac{1}{20} = 5\%$. That is, the load degradation constant should be set to 5%.

Each selecting component, i.e. NS or PU-side cache, has to keep a local per-PE *degradation counter* which is initialized with 0%. Whenever a PE is selected, this local counter is incremented by the load degradation constant. On update, i.e. the PE re-registers with its up-to-date load information and the information is spread via ENRP, the local degradatation counter is reset. For selection, the PE having the lowest sum of load value and degradation counter is selected. If there are PEs having equal sums, round robin selection is applied.

For example, there is a PE of load 50% and load degradation 5% in a PU-side cache. At first, its degradation counter is 0%. When it is selected for the first time, it is incremented to 5%, and then to 10% for the second time. Now, a new selection in the PE's pool is invoked. In this case, it is only selected when its sum of load and degradation counter ($50\% + 10\% = 60\%$) is lowest within the pool (or if there are PEs of equal sum, by round robin selection among them).

First experiments have shown that the LUD performance is highly dependent on the scenario and generally rather unpredictable. Since no generally applicable results for LUD have been obtained so far, this policy is not recommended for use and not included in the quantitative comparison.

3.2 Modified and New Policies

Modified Round Robin (RRmod) and Modified Weighted Rd. Robin (WRRmod) In some situations, the RR selection degenerates due to its statefulness as shown in section 5.1. To avoid this, the cyclic pattern has to be broken up. Therefore, we propose to modify the RR policy by instead of incrementing the round robin pointer by the number of items actually selected, simply to increment it by one. The same modification should also be applied to WRR, which is a generalization of RR, resulting in the modified policy WRRmod.

Random Selection (RAND) and Weighted Random Selection (WRAND) Another solution to avoid the degeneration problem is to use a static and completely stateless selection mechanism. To achieve this, PEs are randomly selected from the pool with equal probability (RAND) or with a probability proportional to the weight constant of a PE (WRAND). RAND can be viewed as a special case of WRAND with all weight factors set to identical values (as for RR and WRR).

Priority Least Used (PLU) PLU is a dynamic policy based on LU with the difference that PEs can provide a load increment constant similar to LUD (see section 3.1). Then, the PE having the lowest value of load + load increment constant is selected. But unlike LUD, no local incrementation is applied to the load information by the selecting component (NS or PU-side cache) itself. This makes the policy simpler and avoids its sensitivity to variances of update timing and fraction of selected PEs actually used by the PU for service.

4 The Simulation Model

To quantitatively evaluate the RSerPool concept, we have developed a simulation model which is based on the discrete event simulation system OMNeT++ [17]. Currently, it includes implementations of the two RSerPool protocols – ASAP [8] and ENRP [7] – and a NS module. Furthermore, it also includes models for PE and PU components of the distributed fractal graphics computation application described in [13]. This application was originally created using our RSerPool prototype *rsplib* and tested in a lab testbed emulating a LAN/WAN scenario.

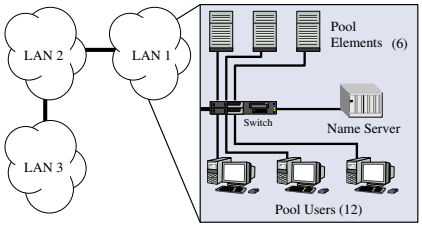

Fig. 3. RSerPool Simulation Scenario

Figure 3 shows the simulation scenario. The modelled RSerPool network consists of 3 LANs, interconnected via WAN links. The LAN links introduce an average delay of 10ms, WAN links an average one of 100ms (both settings are based on the testbed LAN/WAN scenario). Each LAN contains 1 NS, 6 PEs (the local NS is their home NS) and 12 PUs (using the local NS for name resolutions).

Unless otherwise specified, a PE has a default computation capacity of $C = 10^6$ calculations per second. A PE can process several computation jobs simultaneously in a processor sharing mode as commonly used in multitasking operating systems. At most, $MaxJobs = \left\lfloor \frac{\mathrm{C}}{2.5*10^5} \right\rfloor$ simultaneous jobs are allowed on a server to avoid overloading and excessive response times. Therefore, for a server with the default capacity of 10^6 calculations per second, at most 4 jobs can be processed simultaneously. The *load* of a server in our scenario has been defined as the number of currently running jobs, divided by its respective job limit *MaxJobs*. That is, if server B has twice the computation capacity of A, B may have a load of 50% with 4 jobs while A is already loaded 50% with 2 jobs. A PE rejects an additional job if it is fully loaded. In this case, the PU will try another PE (selected by pool policy, of course) after an average timeout of 100ms to avoid overloading the NS and network with unsuccessful ASAP Name Resolution requests (recommendation based on the results from [13]).

In our scenario, PUs sequentially request the processing of jobs by the pool, having an average job size of 10^7 calculations and a negative exponential distribution (approximation of the real system behavior, see [13]). After receiving the result of a job, a PU waits for an average of 10 seconds (again, negative exponentially distributed) to model the reaction time of a user. The stale cache value for the PU-side cache is set to the default of 30s (see [8] and [15]), i.e. a stale cache period contains about 2 to 3 service times.

The length of the simulation runs was set to 20 minutes simulated realtime. All simulation runs have been repeated 4 times using different seeds to be able to compute confidence intervals. For the statistical post-processing and plotting, GNU Octave

and GNU Plot have been used. The plots show mean values and their 95% confidence intervals.

5 Simulation Results

5.1 RR Policy Degeneration

To show the effects of inappropriately implemented stateful policies (see section 2), we first examine a homogeneous scenario where all PEs have the same capacity of $C = 10^6$ calculations per second.

Figure 4 shows the total number of completed jobs for different values of *MaxNResItems*, i.e. for a different number of PE identities returned per name resolution request.

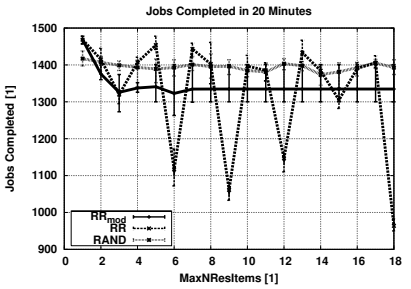

Fig. 4. Round Robin Policy Behaviour

For the original RR policy, a significant periodic degeneration can be observed. If the number of PEs in the pool is an integer multiple of the number of entries in the list sent back by the NS, specific PEs will be systematically overloaded while the others will be hardly used. Assume, e.g. that the pool consists of the pool elements PE1 to PE6 and that the configured amount of PE identities delivered by the NS in a name resolution response is 3. Now, the first resolution query to the NS returns the set {PE1, PE2, PE3}, the following one will return {PE4, PE5, PE6}. Then, new resolutions again start with {PE1, PE2, PE3} and so on. In the worst case, the pool size is lower than the configured amount. Then, the reply is always the same (that is, the complete pool).

From the list received from the NS, the PU selects again one PE to establish the application connection to (see section 2). Using round robin, this will always be the first PE of the list after a refresh of the PU-side cache. The result is of course that some PEs will be systematically overloaded. Subsequent service requests within the stale cache period also select subsequent elements of the list with decreasing probability.

The curve for the RRmod policy shows the behaviour if the RR policy is modified as described in section 3.2. Obviously, the problem of periodic variation has been solved.

The RAND policy shows a slightly higher performance as RRmod for this scenario. Only if a single PE identity is returned per request, the stateful policies RR and RRmod perform better, because PU-side caching always results in using the one PE selected by the NS first. For higher values, the PU-side cache contains a list of multiple elements, which are used for local round robin selection. For example, the NS replies with the list PE1, PE2, PE3 to PU1 and PE2, PE3, PE4 to PU2. Since the elements are ordered for round robin selection, the probability of simultaneous requests from PU1 and PU2 to PE2 is higher than for random selection.

5.2 Performance in Heterogeneous Scenarios

In real-world application scenarios, servers usually do not have equal capacities. Therefore, we examine two types of heterogeneous scenarios. In the first case, a specific server has a significantly higher capacity. The second case models an evolutionary scenario, where capacities vary linearly over a certain range due to several server generations in use.

Fast Server Scenario In this scenario, one server ("fast server") in each LAN has a capacity which is scaled up from 1 to 15 million calculations per second while all the others have the standard capacity of 1 million calculations per second. *MaxNResItems* has been set to 5.

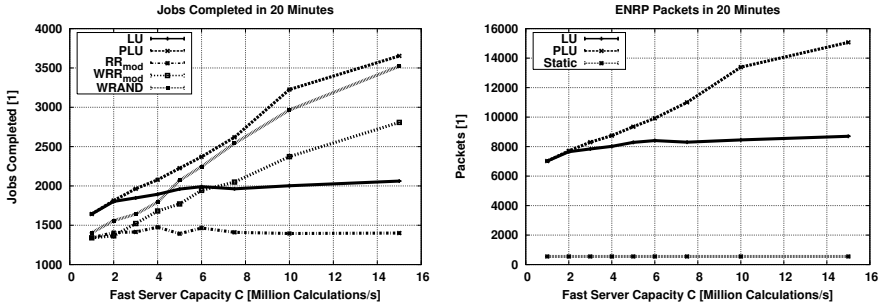

Fig. 5. Fast Server Scenario: Jobs Completed and ENRP Packets

Figure 5, shows how well the increased total processing capacity can be used by various policies (left side) and how much control overhead (number of ENRP packets) is required (right side). It can be observed that for a homogeneous or nearly homogeneous scenario (fast server capacity up to 4×10^6), the dynamic policies generally perform better than the static ones. However, for more heterogeneous scenarios, the behaviour changes significantly.

As expected, the simple RRmod policy is obviously not able to exploit the increased capacity due to the fact that the policy selects all PEs with the same probability irrespective of their capacity. For WRRmod, a metric for the capacity is added in form of a weight constant per PE. The resulting performance for this policy is, therefore, significantly higher: up to 500 additional jobs can be completed within 20 minutes. The LU policy achieves a slightly better performance than WRRmod. However, it also requires about 7000 to 15000 ENRP control packets in 20 minutes, compared to 444 for the static policies.

The LU policy does not take into account the fact that a new job on a high-capacity PE increases the load less than a new job on a low-capacity one. The PLU policy proposed in section 3.2 does exactly that resulting in a significantly improved performance. However, since the number of re-registrations to update the load information in the NS is directly related to the number of accepted jobs, the overhead increases in a similar way[3].

Finally, the result for the WRAND policy proposed in section 3.2 is very promising. It achieves a performance close to that of the dynamic PLU policy with the minimum overhead of a static policy. Again, as explained in section 5.1 for the comparison of RRmod and RAND, the local selection in the PU-side cache leads to a higher probability of simultaneous requests to the same PE for WRRmod. WRAND therefore obviously performs much better.

Evolutionary Scenario While the scenario examined in section 5.2 used selected PEs with high capacities, this examination uses an evolutionary scenario using PEs with linearly varying capacities. For each LAN, the PEs have capacities of $c(n) = n * \vartheta * 10^6$

[3] The average size of an ENRP packet is usually less than 250 bytes, so even a slow 10 MBit/s Ethernet could handle this amount in less than 10 seconds.

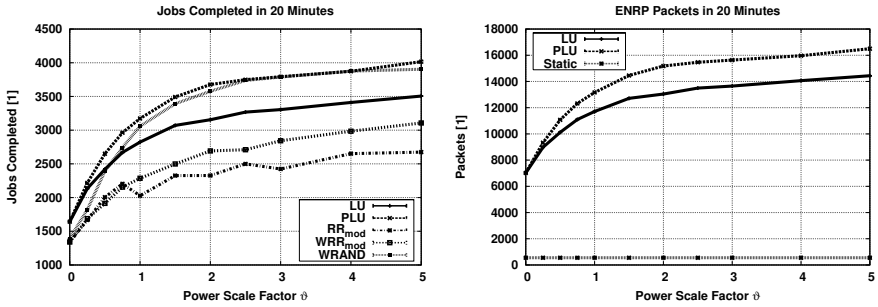

Fig. 6. Evolutionary Scenario: Jobs Completed and ENRP Packets

calculations per second, where n denotes the PE number and ϑ is a constant scale factor. That is, for $\vartheta = 1$, the first PE in a LAN has a capacity of 10^6 calculations per second, the second one 2×10^6 and so on. All other parameters remain unchanged.

As expected, the number of completed jobs for RRmod is lowest, since this policy does not take into account the different server capacities. As in the fast server scenario, the LU and WRRmod policies again show similar performance with the static WRRmod policy requiring significantly less control overhead. While the dynamic policies again generally perform slightly better in relatively homogeneous scenarios (scale factors up to 1), the picture changes for heterogeneous settings. The performance ranking among the policies is the same as for the fast server case although the differences are smaller. Again, the static WRAND policy performs remarkably well with minimum overhead.

6 Conclusion and Outlook

In this paper, we have presented the results of a study looking in detail at the issue of policy based load distribution in the Reliable Server Pooling concept currently under standardization in the IETF. Load distribution is one of the crucial aspects for RSerPool, as it significantly influences RSerPool's scalability and its capability to cope with realtime requirements while remaining "lightweight" in a sense that it keeps control overhead at an acceptable level.

As part of this work, we have detailed the incomplete specifications for the pool policies contained in the existing IETF RSerPool drafts [8] and [15] to a level which allows a consistent implementation.

Based on the results of our simulation studies, we proposed a modification of the Round Robin policy which avoids pathological patterns resulting in severe performance degradation in some cases. In addition, we proposed new static (RAND and WRAND) and dynamic (PLU) policies which perform significantly better than the original policies in realistic scenarios where the servers of a pool have different processing capacities.

As a consequence of this study, we have created the Internet Draft [16] containing the refinements and modifications for the already defined policies as well as the proposal to add our WRAND and PLU policies to the list of standard policies. This draft has been presented at the IETF RSerPool Working Group meeting at the 60th IETF meeting and has become a working group draft [18] of the IETF RSerPool Working Group.

After these first promising results, we are currently continuing the evaluation of the load distribution mechanisms by examining the sensitivity with respect to a broad range of system parameters including, e.g., the stale cache timer value, the length of the PE list returned per name resolution request, the maximum number of simultaneous jobs per PE and the PE capacity. In addition, we also investigate the scalability with respect to PEs, NSs and PUs as well as the influence of different traffic and job patterns. Our goal is to provide recommendations to implementers and users of RSerPool with respect to

tuning of system parameters and guidelines for appropriate selection of pool policies in various application scenarios.

References

1. ITU-T. Introduction to CCITT Signalling System No. 7. Technical Report Recommendation Q.700, International Telecommunication Union, March 1993.
2. K. D. Gradischnig and M. Tüxen. Signaling transport over ip-based networks using ietf standards. In *Proceedings of the 3rd International Workshop on the design of Reliable Communication Networks*, pages 168–174, Budapest, Hungary, 2001.
3. 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. RFC 2960, IETF, Oct 2000.
4. A. Jungmaier, E.P Rathgeb, and M. Tüxen. On the Use of SCTP in Failover-Scenarios. In *Proceedings of the SCI 2002, Volume X, Mobile/Wireless Computing and Communication Systems II*, volume X, Orlando/U.S.A., Jul 2002.
5. A. Jungmaier, M. Schopp, and M. Tüxen. Performance Evaluation of the Stream Control Transmission Protocol. In *Proceedings of the IEEE Conference on High Performance Switching and Routing*, Heidelberg, Germany, June 2000.
6. M. Tüxen, Q. Xie, R. Stewart, M. Shore, L. Ong, J. Loughney, and M. Stillman. Requirements for Reliable Server Pooling. RFC 3227, IETF, Jan 2002.
7. Q. Xie, R. Stewart, and M. Stillman. Endpoint Name Resolution Protcol (ENRP). Internet-Draft Version 08, IETF, RSerPool WG, Jun 2004. draft-ietf-rserpool-enrp-07.txt, work in progress.
8. R. Stewart, Q. Xie, M. Stillman, and M. Tüxen. Aggregate Server Access Protcol (ASAP). Internet-Draft Version 09, IETF, RSerPool WG, Jun 2004. draft-ietf-rserpool-asap-08.txt, work in progress.
9. T. Dreibholz. An efficient approach for state sharing in server pools. In *Proceedings of the 27th Local Computer Networks Conference*, Tampa, Florida/U.S.A., Oct 2002.
10. P. Conrad, A. Jungmaier, C. Ross, W.-C. Sim, and M. Tüxen. Reliable IP Telephony Applications with SIP using RSerPool. In *Proceedings of the SCI 2002, Mobile/Wireless Computing and Communication Systems II*, volume X, Orlando/U.S.A., Jul 2002.
11. T. Dreibholz, A. Jungmaier, and M. Tüxen. A new Scheme for IP-based Internet Mobility. In *Proceedings of the 28th Local Computer Networks Conference*, Königswinter/Germany, Nov 2003.
12. T. Dreibolz and M. Tüxen. High availability using reliable server pooling. In *Proceedings of the Linux Conference Australia 2003*, Perth/Australia, Jan 2003.
13. Y. Zhang. Distributed Computing mit Reliable Server Pooling. Masters thesis, Universität Essen, Institut für Experimentelle Mathematik, Apr 2004.
14. Qiaobing Xie. Private communication at the 60th IETF meeting, San Diego/California, U.S.A., August 2004.
15. R. Stewart, Q. Xie, M. Stillman, and M. Tüxen. Aggregate Server Access Protocol (ASAP) and Endpoint Name Resolution Protocol (ENRP) Parameters. Internet-Draft Version 06, IETF, RSerPool WG, Jun 2004. draft-ietf-rserpool-common-param-05.txt, work in progress.
16. M. Tüxen and T. Dreibholz. Reliable Server Pooling Policies. Internet-Draft Version 00, IETF, RSerPool WG, Jul 2004. draft-tuexen-rserpool-policies-00.txt, work in progress.
17. OMNeT++ Discrete Event Simulation System. http://www.omnetpp.org.
18. M. Tüxen and T. Dreibholz. Reliable Server Pooling Policies. Internet-Draft Version 00, IETF, RSerPool WG, Oct 2004. draft-ietf-rserpool-policies-00.txt, work in progress.