On Robustness and Countermeasures of Reliable Server Pooling Systems against Denial of Service Attacks*

Thomas Dreibholz[1], Erwin P. Rathgeb[1], and Xing Zhou[2]

[1] University of Duisburg-Essen
Institute for Experimental Mathematics
Ellernstrae 29, D-45326 Essen, Germany
Tel: +49 201 183-7637, Fax: +49 201 183-7673
dreibh@iem.uni-due.de
[2] Hainan University
College of Information Science and Technology
Renmin Road 58, 570228 Haikou, Hainan, China
Tel: +86 898 6625-0584, Fax: +86 898 6618-7056
xing.zhou@uni-due.de

Abstract. The Reliable Server Pooling (RSerPool) architecture is the IETF's novel approach to standardize a light-weight protocol framework for server redundancy and session failover. It combines ideas from different research areas into a single, resource-efficient and unified architecture. While there have already been a number of contributions on the performance of RSerPool for its main tasks – pool management, load distribution and failover handling – the robustness of the protocol framework has not yet been evaluated against intentional attacks.
The first goal of this paper is to provide a robustness analysis. In particular, we would like to outline the attack bandwidth necessary for a significant impact on the service. Furthermore, we present and evaluate our countermeasure approach to significantly reduce the impact of attacks.
Key words: Reliable Server Pooling, Attacks, Denial of Service, Robustness, Countermeasures

1 Introduction and Scope

The Reliable Server Pooling (RSerPool) architecture [1] is a generic, application-independent framework for server pool [2] and session management, based on the Stream Control Transmission Protocol (SCTP) [3]. While there have already been a number of publications on the performance of RSerPool for load balancing [4] and server failure handling [5], there has not yet been any research on security and robustness. SCTP already protects against simple flooding attacks [6] and the Internet Drafts [7] of RSerPool mandatorily require using mechanisms like TLS [8] or IPSEC [9] in order to ensure authenticity, integrity and confidentiality. However, this approach is not sufficient: as for every distributed system, there is a chance that an attacker may compromise a legitimate component and obtain the private key. So, how robust are the RSerPool protocols under attack?

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

The goal of this paper is therefore to first analyse the robustness of RSerPool against a denial of service (DoS) attack by compromised components, in order to show the impact of different attack scenarios on the application performance. Using these analyses as a baseline performance level, we will present our countermeasure approach which can significantly reduce the impact of attacks at a reasonable effort.

2 The RSerPool Architecture

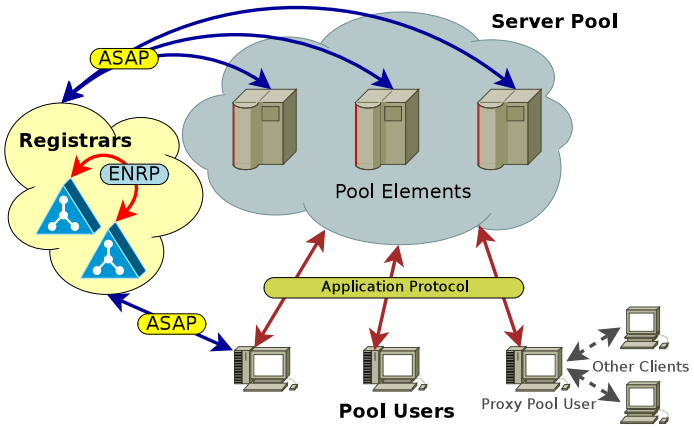

Fig. 1. The RSerPool Architecture

Figure 1 illustrates the RSerPool architecture [1, 10] which consists of three types of components: servers of a pool are called *pool elements* (PE), a client is denoted as *pool user* (PU). The *handlespace* – which is the set of all pools – is managed by redundant *pool registrars* (PR). Within the handlespace, each pool is identified by a unique *pool handle* (PH).

2.1 Components and Protocols

PRs of an *operation scope* synchronize their view of the handlespace by using the Endpoint haNdlespace Redundancy Protocol (ENRP) [11], transported via SCTP [12] and secured e.g. by TLS [8] or IPSEC [9]. In contrast to GRID computing [13], an operation scope is restricted to a single administrative domain. That is, all of its components are under the control of the same authority (e.g. a company or an organization). This property results in a small management overhead [2, 14], which also allows for RSerPool usage on devices providing only limited memory and CPU resources (e.g. embedded systems like telecommunications equipment or routers). Nevertheless, PEs may be distributed globally for their service to survive localized disasters [15].

PEs choose an arbitrary PR of the operation scope to register into a pool by using the Aggregate Server Access Protocol (ASAP) [16], again transported via SCTP and using TLS or IPSEC. Within its pool, a PE is characterized

by its PE ID, which is a randomly chosen 32-bit number. 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 by 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 updates. PEs re-register regularly (in an interval denoted as *registration lifetime*) and for information updates.

In order to access the service of a pool given by its PH, a PU requests a PE selection from an arbitrary PR of the operation scope, again using ASAP. The PR selects the requested list of PE identities by applying a pool-specific selection rule, called *pool policy*. RSerPool supports two classes of load distribution policies: non-adaptive and adaptive algorithms [4]. While adaptive strategies base their assignment decisions on the current status of the processing elements (which of course requires up-to-date states), non-adaptive algorithms do not need such data. A basic set of adaptive and non-adaptive pool policies is defined in [17]. Relevant for this paper are the non-adaptive policies Round Robin (RR) and Random (RAND) as well as the adaptive policies Least Used (LU) and Least Used with Degradation (LUD). LU selects the least-used PE, according to up-to-date application-specific load information. Round robin selection is applied among multiple least-loaded PEs. LUD [18] furthermore introduces a *load decrement* constant which is added to the actual load each time a PE is selected. This mechanism compensates inaccurate load states due to delayed updates. An update resets the load value to the actual load again.

PUs may report unreachable PEs to a PR by using an ASAP Endpoint Unreachable message. A PR locally counts these reports for each PE. If the threshold MaxBadPEReports [5] is reached, the PR may decide to remove the PE from the handlespace. The counter of a PE is reset upon its re-registration.

2.2 Application Scenarios

Although the main motivation to define RSerPool has been the availability of SS7 (Signalling System No. 7 [19]) services over IP networks, it is intended to be a generic framework. There has already been some research on the performance of RSerPool usage for applications like SCTP-based mobility [20], VoIP with SIP [21], web server pools [10], IP Flow Information Export (IPFIX) [22], real-time distributed computing [4, 10] and battlefield networks [23]. A generic application model for RSerPool systems has been introduced by [4], including performance metrics for the provider side (pool utilization) and user side (request handling speed). Based on this model, the load balancing quality of different pool policies has been evaluated [4, 10].

3 Quantifying a RSerPool System

The service provider side of a RSerPool system consists of a pool of PEs. Each PE has a request handling *capacity*, which we define in the abstract unit of calculations per second[3]. Each request consumes a certain number of calculations; we call this number *request size*. A PE can handle multiple requests simultaneously – in a processor sharing mode as provided by multitasking operating systems.

[3] An application-specific view of capacity may be mapped to this definition, e.g. CPU cycles or memory usage.

On the service user side, there is a set of PUs. The number 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.

The total delay for handling a request d_{Handling} is defined as the sum of queuing delay d_{Queuing}, startup delay d_{Startup} (dequeuing until reception of acceptance acknowledgement) and processing time $d_{\text{Processing}}$ (acceptance until finish):

$$d_{\text{Handling}} = d_{\text{Queuing}} + d_{\text{Startup}} + d_{\text{Processing}}. \tag{1}$$

That is, d_{Handling} not only incorporates the time required for processing the request, but also the latencies of queuing, server selection and protocol message transport. The *handling speed* is defined as: $\text{handlingSpeed} = \frac{\text{requestSize}}{d_{\text{handling}}}$. For convenience reasons, the handling speed (in calculations/s) is represented in % of the average PE capacity. Clearly, the user-side performance metric is the handling speed – which should be as fast as possible.

Using the definitions above, it is possible to delineate the average system utilization (for a pool of NumPEs servers and a total pool capacity of PoolCapacity) as:

$$\text{systemUtilization} = \text{NumPEs} * \text{puToPERatio} * \frac{\frac{\text{requestSize}}{\text{requestInterval}}}{\text{PoolCapacity}}. \tag{2}$$

Obviously, the provider-side performance metric is the system utilization, since only utilized servers gain revenue. In practise, a well-designed client/server system is dimensioned for a certain *target system utilization*, e.g. 80%. That is, by setting any two of the parameters (PU:PE ratio, request interval and request size), the value of the third one can be calculated using equation 2 (see [4, 10] for details).

4 The Simulation Setup

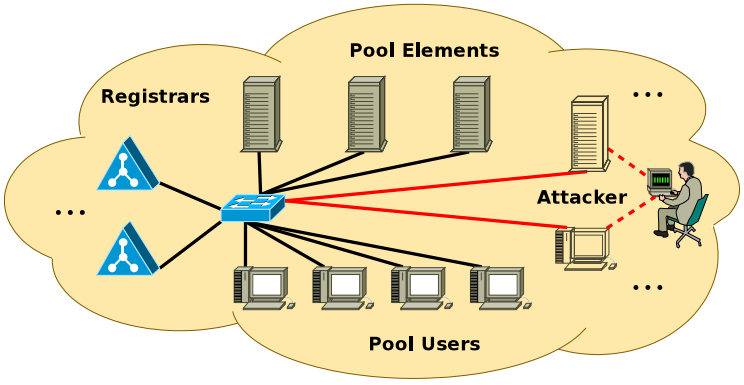

Fig. 2. The Simulation Setup

For our performance analysis, the RSerPool simulation model RSPSIM [4, 10, 24] has been used. This model is based on the OMNeT++ [25] simulation environment and contains the protocols ASAP [16] and ENRP [11], a PR module, an attacker module and PE as well as PU modules for the request handling scenario defined in section 3. Network latency is introduced by link delays only. Therefore, only the network delay is significant. The latency of the pool management by PRs is negligible [2].

Unless otherwise specified, the basic simulation setup – which is also presented in figure 4 – uses the following parameter settings:

- The target system utilization is 80%. Request size and request interval are randomized using a negative exponential distribution (in order to provide a generic and application-independent analysis [10]). There are 10 PEs; each one provides a capacity of 10^6 calculations/s.
- A PU:PE ratio of 10 is used (i.e. a non-critical setting as shown in [4]). The default request size:PE capacity is 10 (i.e. the request size is 10^7 calculations; being processed exclusively, the processing of an average request takes 10s – see also [4]).
- We use a single PR only, since we do not examine PR failure scenarios here (see [4] for the impact of multiple PRs). PEs re-register every 30s (registration lifetime) and on every load change for the adaptive policies.
- MaxBadPEReports is set to 3 (default value defined in [11]). A PU sends an Endpoint Unreachable if a contacted PE fails to respond within 10s.
- The system is attacked by a single attacker node.
- The simulated real-time is 120m; each simulation run is repeated at least 24 times with a different seed in order to achieve statistical accuracy.

GNU R has been used for the statistical post-processing of the results. Each resulting plot shows the average values and their 95% confidence intervals.

5 Robustness Analysis and Attack Countermeasures

5.1 Introduction

The attack targets of RSerPool are the PRs, PEs and PUs; the goal of an attack is to affect the services built on top of RSerPool. Due to mandatory connection security by TLS or IPSEC (see subsection 2.1), an attacker has to compromise a component itself. Clearly, an attacker being able to compromise a PR is able to propagate arbitrary misinformation into the handlespace via ENRP. However, since the number of PRs is considered to be quite small [2] (e.g. less than 10) in comparison with the number of PEs and PUs (up to many thousands [2, 10]) and due to the restriction of RSerPool to a single administrative domain, it is assumed to be feasible to protect the small number of PRs rather well. Instead, the most probable attack scenario is an attacker being able to compromise a PE or PU. These components are significantly more numerous [10] and may be distributed over multiple, less controllable locations [15]. For that reason, ASAP-based attacks are in the focus of our study.

For our analysis, we use a single attacker node only. Assuming that there is a certain difficulty in compromising a PE or PU, the number of attackers in a system is typically small. If a powerful attacker is able to compromise a large number of RSerPool components, protocol-internal countermeasures are obviously difficult and the effort should be spent to increase the system security of the components. However, as we will show, even a single attacker can cause a DoS without further countermeasures.

5.2 A Compromised Pool Element

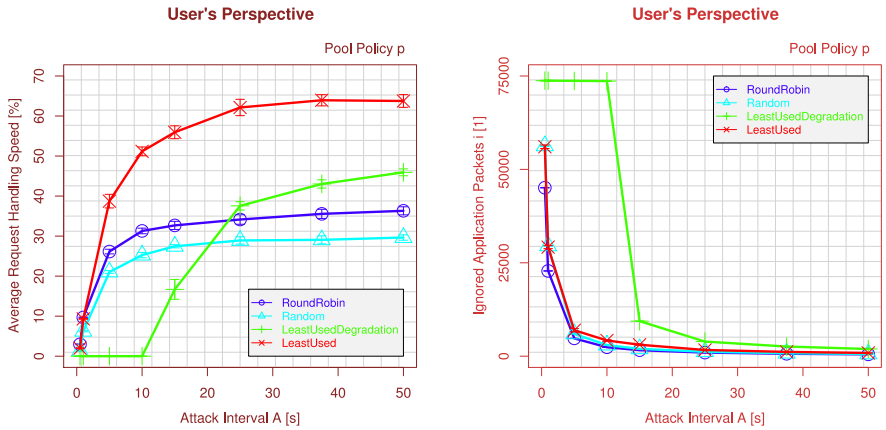

Fig. 3. The Impact of a Compromised Pool Element

The goal of an attacker being able to perform PE registrations is clearly to execute as many fake registrations as possible. That is, each registration request simply has to contain a new PE ID (randomly chosen). The policy parameters may be set appropriately, i.e. a load of 0% (LU, LUD) and a load increment of 0% (LUD), to get the fake PE selected as often as possible.

It is important to note that address spoofing is already avoided by the SCTP protocol [3, 6]: each network-layer address under which a PE is registered must be part of the SCTP association between PE and PR. The ASAP protocol [16] requires the addresses to be validated by SCTP. However, maintaining the registration association with the PE and silently dropping all incoming PU requests is already sufficient for an attacker.

In order to illustrate the impact that even a *single* attacker can cause on the pool performance, we have performed simulations using the parameters described in section 4. Each PE handles up to 4 requests simultaneously, i.e. the load increment of LUD is 25% for a real PE. Figure 3 presents the results; the left-hand side shows the request handling speed, the right-hand one the number of PU requests ignored by the attacker (requests sent to fake PEs).

Obviously, the smaller the attack interval (i.e. the delay between two fake PE registrations), the more fake entries go into the handlespace. This leads to an increased probability for a PU to select a non-existing PE and therefore to a decreased overall request handling speed. In particular, this effect is the strongest for LUD: real PEs get loaded and provide their real load increment (here: 25%), while the fake PEs always claim to be unloaded with a load increment of 0%. It takes only a single registration every 10s (A=10) to cause a complete DoS (i.e. a handling speed of 0%). The other policies are somewhat more robust, since they do not allow an attacker to manipulate the PE selection in such a powerful way.

Clearly, using a larger setting of MaxBadPEReports would lead to an even worse performance: the longer it takes to get rid of a fake PE entry, the more trials of PUs to use these PEs (see also the right-hand side of figure 3). As a

summary, it is clearly observable that even a single attacker with small attack bandwidth can cause a complete DoS.

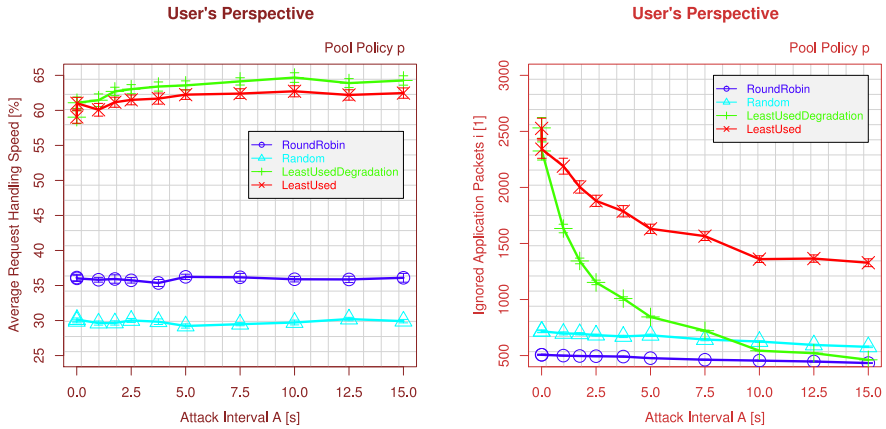

Fig. 4. Applying the Countermeasure against the Pool Element Attack

The key problem of the described threat is attacker's power to create a new fake PE with each of its registration messages. That is, only a few messages per second (i.e. even a slow modem connection) is sufficient to cause a severe service degradation. For that reason, an effective countermeasure is clearly to restrict the number of PE registrations that a single PE identity is authorized to create. However, in order to retain the "light-weight" property of RSerPool [2] and to avoid synchronizing such numbers among PRs, we have put forward a new approach and introduce the concept of a *registration authorization ticket*. Such a ticket consists of

1. a PH,
2. a fixed PE ID,
3. minimum/maximum policy information settings (e.g. a lower limit for the load decrement of LUD) and
4. a signature by a trusted authority (similar to a Kerberos service, see below).

Such a ticket, provided by a PE to its PR-H as part of the ASAP registration, can be easily verified by checking its signature. Then, if it is valid, it is only necessary to ensure that the PE's policy settings are within the valid range specified in the ticket. An attacker stealing the identity of a real PE would only be able to masquerade as *this* specific PE. A PR only has to verify the authorization ticket, i.e. no change of the protocol behaviour and especially no additional synchronization among the PRs are necessary. Therefore, the approach can be realized easily; the additional runtime required is in $O(1)$.

Clearly, the need for a trusted authority (e.g. a Kerberos service) adds an infrastructure requirement. However, since an operation scope belongs to a single authority only (see subsection 2.1), it is feasible at reasonable cost.

In order to show the effectiveness of our approach, figure 4 presents the results for applying the same attack as above, but using the new attack countermeasure. The other simulation parameters remain unchanged. Clearly, a significant

improvement can be observed. While the handling speed is only slightly sinking with a smaller attack interval (down to 0.001, which means 1,000 fake registrations per second), the number of ignored PU requests rises from about 1,000 to 2,200 at an attack interval of A=15s to only about 3,000 (LU, LUD) at A=0.001s. In particular, due to the lower limit for policy information, LUD now even achieves a performance benefit: only for very small attack intervals A, its performance converges to the results of LU. The stateful behaviour [10] of this policy now becomes beneficial: although the attacker registers a fake PE with a load of 0%, the PE entry's load increment is increased each time it gets selected. Therefore, frequent re-registrations of this fake PE are necessary in order to result in a significant performance degradation. In summary, a registration attack can be significantly diminished by our countermeasure.

5.3 A Compromised Pool User

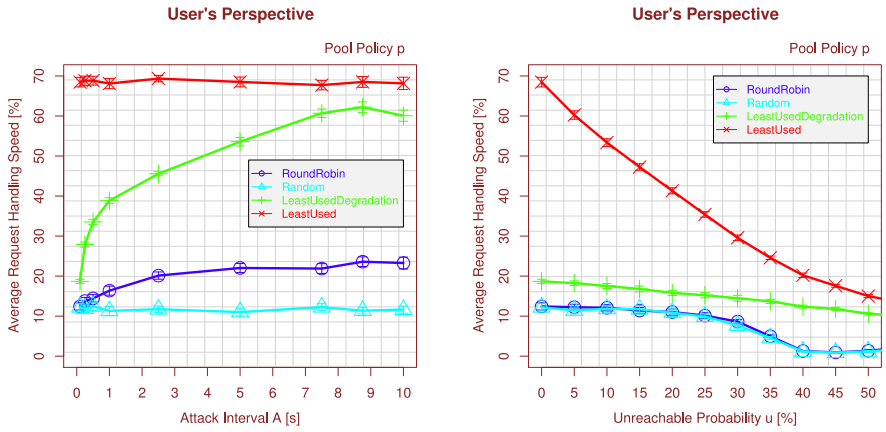

Fig. 5. The Impact of a Compromised Pool User

PUs are the other means of ASAP-based attacks, especially their handle resolution requests and unreachability reports. An obvious attack scenario on handle resolutions would be to flood the PR. However, the server selection can be realized very efficiently – as shown by [2] – but it is a fallacy to assume that simply performing some handle resolutions (without actually contacting any selected PE) cannot affect the service quality. In order to show the effects, the left-hand side of figure 5 presents the impact of a handle resolution attack on the pool performance. The right-hand side of figure 5 presents the results varying the probability for an unreachability report for A=0.1 (i.e. 10 handle resolutions per second).

Clearly, even without unreachability reports, handle resolutions already result in a performance decay: the handling speed of RR converges to the value of RAND, since each selection leads to a movement of the Round Robin pointer [10]. That is, instead of trying to select PEs in turn, some servers are skipped and a

new round starts too early. This leads in fact to a random selection. The performance of LUD decreases due to the load incremented upon each selection. Until the next re-registration (on load change or registration lifetime expiration), the load value in the handlespace grows steadily and leads to a smaller selection probability. LU and RAND, on the other hand, are immune to a simple handle resolution attack: a PE's state is – unlike RR and LUD – not manipulable by the handle resolution itself. Therefore, a handle resolution attack (without Endpoint Unreachable) has no effect here.

Sending ASAP Endpoint Unreachables for the selected PEs (see the right-hand side of figure 5) has a significant negative impact on the pool performance if the attack interval is small enough (here: A=0.1). In this case, the attacker is able to impeach almost all PEs out of the handlespace. A PE comes back upon re-registration (i.e. 30s). Clearly, a larger re-registration interval setting leads to an even lower performance.

The key problem of the handle resolution/failure report attack is that a single PU is able to impeach PEs – even for MaxBadPEReport>1. Therefore, the basic idea of our countermeasure is to avoid counting multiple reports from the same PU. Like for the PEs, it is necessary to introduce a PU identification which is certified by a trusted authority and can be checked by a PR (see subsection 5.2 for details). After that, a PR simply has to remember (to be explained later) the PEs for which a certain PU has reported unreachability and to ignore multiple reports for the same PE. Note, that no synchronization among PRs is necessary, since the unreachability count for each PE is a PR-local variable. That is, sending unreachability reports for the same PE to different PRs does not cause any harm.

In order to remember the already-reported PE identities, we have considered the following hash-based approach of a per-PU message blackboard: instead of memorizing each reported PE (an attacker could exploit this by sending a large amount of random IDs), we simply define a function Ψ mapping each PE given by PE ID and PH to a bucket:

$$\Psi(\mathrm{PH}, \mathrm{ID_{PE}}) = h(\mathrm{PH}, \mathrm{ID_{PE}}) \text{ MOD Buckets.}$$

h denotes an appropriate hash function: an attacker may not easily guess its behaviour. This property is provided by so called universal hash functions [26], which are – unlike cryptographic hash functions like SHA1 [27] – also efficiently computable.

Each bucket contains the time stamps of the latest up to MaxEntries Endpoint Unreachables for the corresponding bucket. Then, the report rate can be calculated as:

$$\mathrm{Rate} = \frac{\mathrm{NumberOfTimeStamps}}{\mathrm{TimeStamp_{Last}} - \mathrm{TimeStamp_{First}}}. \tag{3}$$

Upon reception of an Endpoint Unreachable, it simply updates the reported PE's corresponding bucket entry. If the rate in equation 3 exceeds the configured threshold MaxEURate, the report is silently ignored. The effort for this operation is in $O(1)$, as well as the required per-PU storage space. Analogously, the same hash-based approach can be applied for Handle Resolutions with the threshold MaxHRRate, using only the PH of the requested pool as hash key.

In order to demonstrate the effectiveness of our approach, we have shown the results of two example simulations in figure 6. Both simulations have used 64 buckets with at most 16 entries. Assuming 1,000 PEs in the handlespace, each bucket would represent only about 16 PEs on average. The probability of a bucket collision for two really-failed PEs would therefore be rather small. The

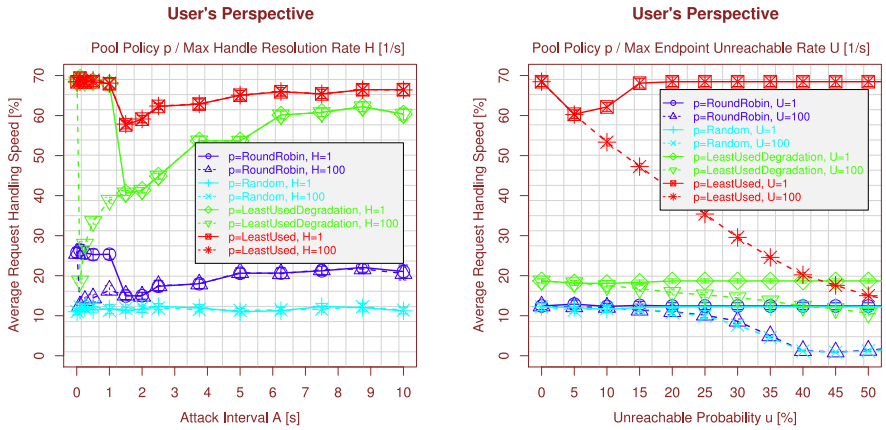

Fig. 6. Applying the Countermeasure against the Pool User Attack

left-hand plot presents the application of the handle resolution rate limit (without failure reports) for rate limits of MaxHRRate H=1 (solid lines) and H=100 (dotted lines) handle resolutions per second. An Endpoint Unreachable is sent for each selected PE, since this is the worst case and also affects LU and RAND. The actual average handle resolution interval for legitimate PUs for the workload (see section 4) is 125s (which is a rate of 0.008 operations/s), i.e. the limit still allows sufficient room for additional handle resolutions (e.g. due to PE failures, temporary workload peaks or hash collisions). As expected from the previous results, the handling speed slightly decreases with a smaller attack interval A for LU, LUD and RR, while RAND remains unaffected. However, when the attack interval exceeds the threshold, all subsequent handle resolutions of the attacker are blocked and the handling speed achieves its full level again. Clearly, the handle resolution threshold should not be set too large (here: H=100), since this gives the attacker too much room to cause service degradation.

The results for a failure report rate limit of MaxEURate U=1 (solid lines) and U=100 (dotted lines) are presented on the right-hand side of figure 6 for a handle resolution rate of A=0.1 at a PE registration life of 30s. In order to show the effect of failure reports only, the handle resolution rate has not been limited here. Obviously, as expected from the handle resolution results, a useful limit achieves a significant benefit: as soon as the failure report rate exceeds the limit, subsequent reports are simply ignored and the handling speed goes back to the original value. Setting a too-high limit (here: U=100) again gives the attacker room to degrade the service quality.

In summary, the handle resolution and failure report limits should be configured sufficiently above the expected rate (here: about two orders of magnitude) in order to allow for temporary peaks. Too high settings, on the other hand, result in a too-weak attack countermeasure.

6 Conclusions

In this paper, we have identified two critical attack threats on RSerPool systems: (1) PE-based attacks (registration) and (2) PU-based attacks (handle resolu-

tion/failure report). In order to reduce the attack impact on registrations, we have suggested to limit the number of PE registrations a single user authorization is allowed to perform by fixed PE IDs and to enforce upper/lower policy information values. This mechanism – denoted as registration authorization ticket – is applicable in time $O(1)$ and in particular requires no additional network overhead or protocol changes. Our solution for reducing the attack impact of handle resolutions/failure reports is a rate limitation based on hash tables. For each PU served by a PR, the rate limit can be realized in $O(1)$ time and space. We have provided simulation results for both attack countermeasures, demonstrating their ability to significantly reduce the impact of attacks on the service performance.

As part of our future work, it is also necessary to analyse the robustness of the ENRP protocol. Although the threat on the small number of PRs of an operation scope is significantly smaller, it is useful to obtain knowledge of possible attack scenarios. Furthermore, we intend to perform real-world security experiments in the PlanetLab, using our RSerPool prototype implementation rsplib. Finally, our goal is to also contribute our results into the IETF standardization process.

References

1. Lei, P., Ong, L., Tüxen, M., Dreibholz, T.: 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.
2. Dreibholz, T., Rathgeb, E.P.: 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., Jeju Island/South Korea (December 2007) 136–143 ISBN 0-7695-3048-6.
3. Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L., Paxson, V.: Stream Control Transmission Protocol. Standards Track RFC 2960, IETF (October 2000)
4. Dreibholz, T., Rathgeb, E.P.: On the Performance of Reliable Server Pooling Systems. In: Proceedings of the IEEE Conference on Local Computer Networks (LCN) 30th Anniversary, Sydney/Australia (November 2005) 200–208 ISBN 0-7695-2421-4.
5. Dreibholz, T., Rathgeb, E.P.: Reliable Server Pooling – A Novel IETF Architecture for Availability-Sensitive Services. In: Proceedings of the 2nd IEEE International Conference on Digital Society (ICDS), Sainte Luce/Martinique (February 2008) 150–156 ISBN 978-0-7695-3087-1.
6. Unurkhaan, E.: Secure End-to-End Transport - A new security extension for SCTP. PhD thesis, University of Duisburg-Essen, Institute for Experimental Mathematics (July 2005)
7. Stillman, M., Gopal, R., Guttman, E., Holdrege, M., Sengodan, S.: Threats Introduced by RSerPool and Requirements for Security. Internet-Draft Version 07, IETF, RSerPool Working Group (October 2007) draft-ietf-rserpool-threats-07.txt.
8. Jungmaier, A., Rescorla, E., Tüxen, M.: Transport Layer Security over Stream Control Transmission Protocol. Standards Track RFC 3436, IETF (December 2002)
9. Bellovin, S., Ioannidi, J., Keromytis, A., Stewart, R.: On the Use of Stream Control Transmission Protocol (SCTP) with IPsec. Standards Track RFC 3554, IETF (July 2003)
10. Dreibholz, T.: 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)

11. Xie, Q., Stewart, R., Stillman, M., Tüxen, M., Silverton, A.: Endpoint Handlespace Redundancy Protocol (ENRP). Internet-Draft Version 18, IETF, RSerPool Working Group (November 2007) draft-ietf-rserpool-enrp-18.txt, work in progress.
12. Jungmaier, A., Rathgeb, E.P., Tüxen, M.: 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.
13. Foster, I.: What is the Grid? A Three Point Checklist. GRID Today (July 2002)
14. Dreibholz, T., Rathgeb, E.P.: An Evalulation of the Pool Maintenance Overhead in Reliable Server Pooling Systems. SERSC International Journal on Hybrid Information Technology (IJHIT) **1**(2) (April 2008)
15. Dreibholz, T., Rathgeb, E.P.: On Improving the Performance of Reliable Server Pooling Systems for Distance-Sensitive Distributed Applications. In: Proceedings of the 15. ITG/GI Fachtagung Kommunikation in Verteilten Systemen (KiVS), Bern/Switzerland (February 2007) 39–50 ISBN 978-3-540-69962-0.
16. Stewart, R., Xie, Q., Stillman, M., Tüxen, M.: Aggregate Server Access Protcol (ASAP). Internet-Draft Version 18, IETF, RSerPool Working Group (November 2007) draft-ietf-rserpool-asap-18.txt, work in progress.
17. Tüxen, M., Dreibholz, T.: Reliable Server Pooling Policies. Internet-Draft Version 07, IETF, RSerPool Working Group (November 2007) draft-ietf-rserpool-policies-07.txt, work in progress.
18. Zhou, X., Dreibholz, T., Rathgeb, E.P.: 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), Sainte Luce/Martinique (February 2008) 171–177 ISBN 978-0-7695-3087-1.
19. ITU-T: Introduction to CCITT Signalling System No. 7. Technical Report Recommendation Q.700, International Telecommunication Union (March 1993)
20. Dreibholz, T., Jungmaier, A., Tüxen, M.: A new Scheme for IP-based Internet Mobility. In: Proceedings of the 28th IEEE Local Computer Networks Conference (LCN), Königswinter/Germany (November 2003) 99–108 ISBN 0-7695-2037-5.
21. Conrad, P., Jungmaier, A., Ross, C., Sim, W.C., Tüxen, M.: 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.
22. Dreibholz, T., Coene, L., Conrad, P.: 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.
23. Uyar, Ü., Zheng, J., Fecko, M.A., Samtani, S., Conrad, P.: 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) (2004) 164–175
24. Dreibholz, T., Rathgeb, E.P.: 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.
25. Varga, A.: OMNeT++ Discrete Event Simulation System User Manual - Version 3.2, Technical University of Budapest/Hungary. (March 2005)
26. Crosby, S.A., Wallach, D.S.: Denial of service via Algorithmic Complexity Attacks. In: Proceedings of the 12th USENIX Security Symposium, Washington, DC/U.S.A. (August 2003) 29–44
27. Eastlake, D., Jones, P.: US Secure Hash Algorithm 1 (SHA1). Informational RFC 3174, IETF (September 2001)