Implementing the Reliable Server Pooling Framework

Thomas Dreibholz
University of Duisburg-Essen
Ellernstrasse 29
45326 Essen, Germany
Email: dreibh@exp-math.uni-essen.de
Telephone: +49 201 183-7637
Fax: +49 201 183-7373

Erwin P. Rathgeb
University of Duisburg-Essen
Ellernstrasse 29
45326 Essen, Germany
Email: rathgeb@exp-math.uni-essen.de
Telephone: +49 201 183-7670
Fax: +49 201 183-7373

***Abstract*— The Reliable Server Pooling (RSerPool) protocol suite currently under standardization by the IETF is designed to build systems providing highly available services by mechanisms and protocols for establishing, configuring, accessing and monitoring pools of server resources. But RSerPool is not only able to manage pools of redundant servers and facilitate service failover between servers: it also includes sophisticated mechanisms for server selections within the pools. These mechanisms make RSerPool useful for applications in load balancing and distributed computing scenarios.**

As part of our RSerPool research and to verify results of our simulation model in real-life scenarios, we have created a complete implementation prototype of the RSerPool framework. In this paper, we will give a detailed description of the concepts, ideas and realizations of our prototype. Furthermore, we will show performance issues raised by the management of large servers pools, as it is necessary for load balancing or distributed computing scenarios. We will explain the algorithms and data structures we designed to solve these challenges and finally present a rough performance evaluation that verifies our concept.

Keywords: Internet applications, IPv6 deployment and applications, SS7, server pools

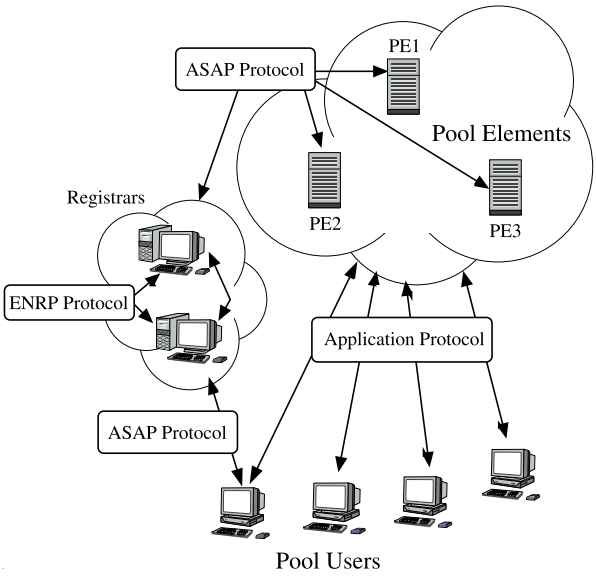

Fig. 1. The RSerPool Architecture

I. Reliable Server Pooling

A. Motivation

The convergence of classical circuit-switched networks (i.e. PSTN/ISDN) and data networks (i.e. IP-based) is rapidly progressing. This implies that PSTN signalling via the SS7 protocol is 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 relationship between two signalling endpoints; for more information see [1]), all links and components of the network devices must be redundant. When transporting signalling over IP networks, such redundancy 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 [2], [3], details follow in section II); redundancy of network device components is supported by the SGP/ASP (signalling gateway process/application server process) concept [1]. However, this concept has some limitations: no support of dynamic addition and removal of components, limited ways of server selection, no specific failover procedures and inconsistent application to different SS7 adaptation layers.

B. Introduction

To cope with the challenge of creating a unified, lightweight, real-time, scalable and extendable redundancy solution (see [4] 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.

Multiple server elements providing the same service belong to a *server pool* to provide both redundancy and scalability. Server pools are identified by a unique ID called *pool handle* (PH) within the set of all server pools, the *handlespace*. A server in a pool is called a *pool element* (PE) of the respective pool. The handlespace is managed by redundant *registrars* (PR). The registrars synchronize their view of the handlespace using the Endpoint haNdlespace Redundancy Protocol (ENRP [5]). PRs

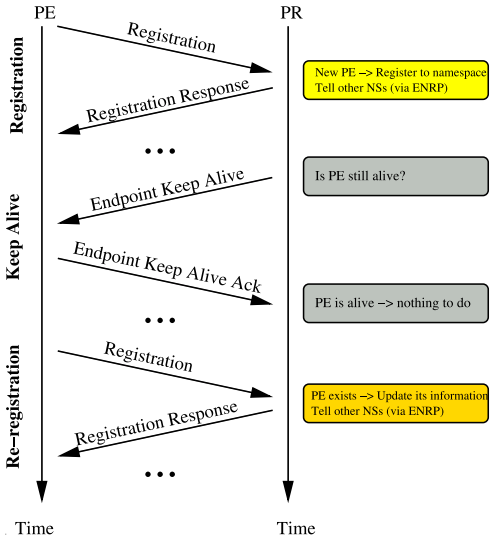

Fig. 2. Registration and Monitoring

announce themselves using multicast mechanisms, i.e. it is not necessary (although possible) to pre-configure PR addresses into the components described in the following.

PEs providing a specific service can register for a corresponding pool at an arbitrary PR using the Aggregate Server Access Protocol (ASAP [6]) as shown in figure 2. The *home PR* (PR-H) is the PR which was chosen by the PE for initial registration. It monitors the PE using SCTP heartbeats (layer 4, not shown in figure; see section II) and ASAP Endpoint Keep Alives. RSerPool does not rely on the layer 4 heartbeat mechanism of SCTP here: the application itself could e.g. hang in an infinite loop while the system's kernel is still responding to the SCTP heartbeats. Using additional keep alives above SCTP therefore improves the monitoring reliability. 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 handlespace by its home PR. A PE can also intentionally de-register from the handlespace by an ASAP de-registration allowing for dynamic reconfiguration of the server pools. PR failures are handled by requiring PEs to re-register regularly (and therefore choosing a new PR 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 PR, which registers, re-registers or de-registers a PE, propagates this information to all other PRs via ENRP. Therefore, it is not necessary for the PE to use any specific PR. In case of a failure of its home PR, a PE can simply use another arbitrarily chosen one.

C. Server Selection

When a client requests a service from a pool, it first asks an *arbitrary* PR 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 (we show examples in section V-B; the standards policies are defined in [7], a quantitative policy performance comparison can be found in [8]). The PU adds this list of PE identities to its local cache (denoted as PU-side cache) and again selects *one* entry from its cache by policy. 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:

1) at the PR when compiling the list of PEs and
2) in the local PU-side cache where the target PE is selected from the list.

If the connection to the selected PE fails, e.g. due to overload or failure of the PE, the PU selects another PE (i.e. directly from cache or by asking a PR first) and tries again. The PU may report a PE failure to a PR, which may decide to remove this PE from the handlespace.

D. Failover Procedure

RSerPool supports optional client-based state synchronization [9] for failover: 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.

E. The Protocol Stack

Figure 3 illustrates the RSerPool protocol stack. All components are based on SCTP over IPv4 and/or IPv6. For the PR, the application layer consists of ENRP and ASAP. While ENRP provides handlespace redundancy between multiple PRs, ASAP is used for registration, re-registration, de-registration and monitoring of PEs as well as for handle resolutions and failure reports by PUs.

Between a PU and PE, ASAP becomes a session layer protocol that provides the client-based state synchronization as described in section I-D. This session layer communication, called *control channel*, is multiplexed with the application's protocol, called *data channel*, over the same SCTP association.

Optionally, PRs can announce themselves via ASAP and ENRP via multicast so that other PRs, PEs and PUs may be fully auto-configuring. This functionality has been omitted in the figure to enhance its readability.

F. Applications

The lightweight, real-time, scalable and extendable architecture of RSerPool is not only applicable to the 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].

Finally, load balancing using RSerPool is currently under discussion by the IETF RSerPool Working Group: due to its flexible server selection policies and pool management functionalities, it has many similarities to load

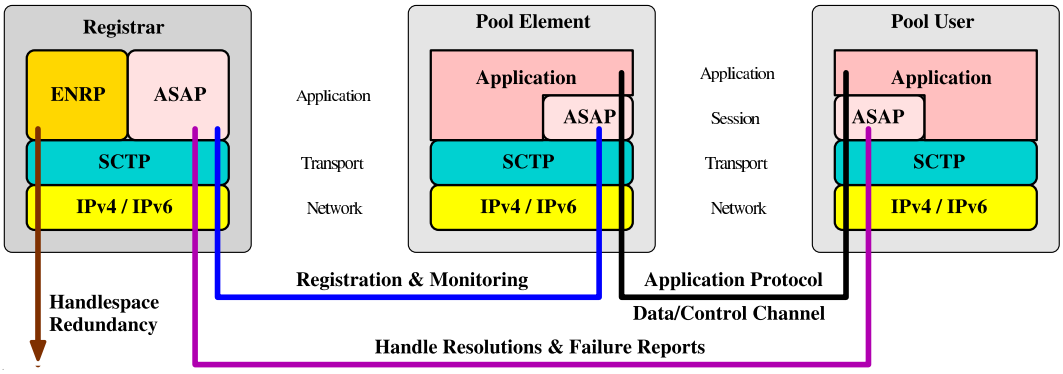

Fig. 3. The RSerPool Protocol Stack

balancer protocols. A very common application for such load balancing systems is to distribute HTTP requests in web server farms. There is an ongoing effort to merge both the RSerPool framework and the Server/Application State Protocol (SASP [14], a contribution of IBM) for load balancers into one common architecture for highly-available server pool management and load distribution.

II. The SCTP Protocol

While the duty of RSerPool is to provide fault-tolerance against component failures, it relies on the SCTP transport protocol [2] to provide fault-tolerance against network failures. As explained in the introduction, SCTP allows multi-homing to fulfil the fault tolerance requirements of SS7. That is, two SCTP endpoints can be connected via two or more networks. When there are multiple disjoint paths between the two endpoints, SCTP can use another one when its primary path becomes unavailable. Such unavailability can occur by network component and link failures or simply due to long convergence times of inter-domain routing protocols (e.g. in the range of several minutes for BGP).

SS7 requires a failover time of at most 800ms and SCTP is able to satisfy this requirement [15]; from an endpoint's view, each destination address is considered as a possible path – denoted as *SCTP path* [2] – to transmit data over. SCTP uses *path monitoring* to check these paths for availability: in configurable intervals, SCTP sends control messages, called *heartbeats*, over each possible path. The peer endpoint, when receiving such a heartbeat, acknowledges it by sending a *heartbeat acknowledgement*. Paths on which acknowledgements are received, are considered to be usable paths for data transport. When the actual data transport path (called *primary path*) becomes unavailable, a working one is selected and the data transmission is continued. The whole process of path monitoring and selection of a new primary path is transparent to the application layer. For details on the configuration of suitable heartbeat intervals and path selection parameters, see [15].

SCTP has been designed to be independent of the underlying network layer protocol, i.e. it is not only possible to use IPv4 and IPv6 but also adapt it to other or future protocols. In the view of SCTP, network layer protocols appear as SCTP paths to the multi-homing functionality. For example, an endpoint supports IPv4 and IPv6 and the peer endpoint is reachable via IPv4 and IPv6. Then, an association between these endpoints has two SCTP paths: one via IPv4 and one via IPv6. If there is a failure e.g. on the IPv4 path, it is therefore still possible to use the IPv6 path. Such multi-protocol setups are very likely in today's networks, due to the growing IPv6 deployment in formerly IPv4-only networks.

In the area of telecommunications, associations are established for durations in the range of months or even years. Therefore, it has been necessary to define a dynamic address reconfiguration extension (abbreviated *Add-IP*, see [16]) allowing for the dynamic addition to and removal of transport addresses from an SCTP association without connection interruption. This especially allows interruption-free IPv6 site renumbering, i.e. changing the address prefix on a provider change to keep BGP routing tables small or even add an additional provider for redundancy reasons. Furthermore, it even allows an association to be established in an IPv4-only network, being upgraded to IPv4+IPv6 and finally turned into IPv6 only – interruption-free and transparent to the upper layers.

III. The RSerPool API

The programming API for RSerPool is currently actively being discussed by the IETF RSerPool WG. It will consist of two styles: the *basic mode* and the *enhanced mode*.

A. Basic Mode API

The basic mode provides only the fundamental RSerPool function calls for PEs to register, re-register or de-register and for PUs to resolve a pool handle and select a PE by policy. All session layer functionalities between PE and PU – especially failure detection and failover – have to be provided by the application programs themselves. That is, a control channel is not supported here.

The reason for having the basic mode API is to provide easy deployment of RSerPool functionality to existing applications, e.g. a FTP service application that supports download continuation using FTP's reget functionality.

An example for using the basic mode API can be found in [12].

B. Enhanced Mode API

Unlike the basic mode API, the enhanced mode API offers a complete session layer between PE and PU, including optional failover handling using client-based state synchronization.

That is, a PU establishes a session to a pool providing its desired service. The session layer provided by the enhanced mode API transparently handles pool handle resolutions, PE selections, association establishments, failure detection on the association using SCTP heartbeats, selecting a new PE when the former one becomes unreachable and optionally failover handling using state cookies via the control channel. For the application itself, this session layer can be completely transparent[1]. In fact, the pool appears to the user as one highly available server.

To provide easy adaptation of existing and new applications to RSerPool's session layer functionality, the API mimics the Unix socket API to provide session layer functionality. A pseudo-code example is shown in algorithm 1: similarly to creating a TCP socket, connecting it to a remote server and finally using the application's protocol to do something, a RSerPool session is created, connected to a pool and the application's protocol is used over the session. But unlike a simple TCP connection, RSerPool provides seamless service continuation in case of server failure – transparent to the application.

The PE side of the enhanced mode API also looks similar to TCP-based servers, but instead of binding a socket to a port number, it is registered as PE under the service's pool handle.

Note, that applications do not have to care about any transport address when using the enhanced mode API. A PE is by default registered under all of its transport addresses – regardless of whether they are IPv4 or IPv6. Furthermore, using the Add-IP [16] extension of SCTP as described in section II, transport addresses may change at runtime, e.g. due to IPv6 prefix change. At the PU side, transport association management and therefore handling of addresses is completely transparent to the application layer.

Currently, there are only two existing implementations of RSerPool: a closed source version by Motorola [17] and the authors' own GPL-licensed Open Source prototype *rsplib*. This prototype will be explained in detail in the following section.

IV. The *rsplib* Prototype

As part of our RSerPool research and to verify the results of our simulation model [8], [18] in real-life scenarios, we have created a complete implementation [12], [19] prototype of the RSerPool framework. It consists of a PR and a library – the *rsplib* – providing the PE and PU functionalities. Our implementation package, called the *rsplib prototype*, has been released [19] as Open Source under the GPL license.

[1]If the application uses a custom failover procedure, some interaction may be required.

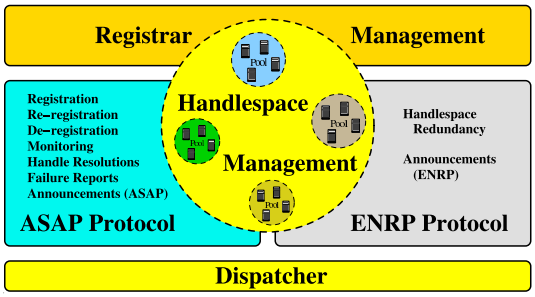

Fig. 4. The *rsplib* Registrar

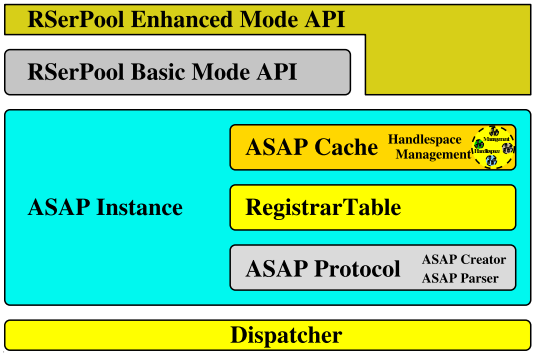

Fig. 5. The *rsplib* PU/PE Library

Elementary design criteria of our prototype have been platform independence and the support of both IPv4 and IPv6. To ensure platform independence, we have chosen C instead of C++ as implementation language, because C is more common on exotic devices. Although currently only Linux, FreeBSD and Darwin (MacOS X) are supported by our prototype, our long-term goal is to make it also available on embedded devices like PDAs and smartphones. A short-term goal is to extend our support to the Windows and Solaris platforms.

When we started the development of our prototype in 2002, the only stable SCTP implementation on our three main platforms (Linux, FreeBSD, Darwin) has been our own Open Source userland SCTP implementation *sctplib* [20]. Meanwhile, the native SCTP support of these platforms has improved so that we also support the built-in kernel SCTP of Linux, FreeBSD (KAME stack) and Darwin. All afore-mentioned SCTP implementations, including our own *sctplib*, support the Add-IP extension for dynamic address reconfiguration.

The *rsplib* prototype is a complete implementation of RSerPool, also including the features being optional in the standards documents. In particular, we support both the basic and enhanced mode APIs, full auto-configuration by PR announcements via multicast – both, for ASAP and ENRP – and all optional policies defined in the draft [7]. This draft is one contribution, based on our RSerPool research on policy performance [8], and has become a working group draft of the IETF RSerPool WG.

Algorithm 1 A PU Pseudo-Code Example

```
sd = rsp_socket(...);
rsp_connect(sd, "MyDownloadServerPool", ...);
rsp_write(sd, "GET MyMovie.mpeg HTTP/1.0\r\n\r\n");
while((length = rsp_read(sd, buffer, ...)) > 0) {
   doSomething(buffer, length);
}
rsp_close(sd);
```

The building blocks of the *rsplib* prototype are shown in figure 4 (registrar) and figure 5 (PU/PE library). Both parts contain the *Dispatcher* component encapsulating the platform-dependent timer and file/socket event management as well as thread-safety functionality. On top of this component, the registrar realizes the ASAP and ENRP protocols. Their functionality is controlled by the *Registrar Management*, which consists of the binding layer between protocols and the registrar's central component: the *Handlespace Management*. This component takes care of storing the handespace's content and providing access functionality for both ASAP (registration, re-registration, deregistration and monitoring of PEs, handle resolutions for PUs) and ENRP (handlespace synchronization between PRs). Authenticating and authorizing requests to the handlespace management is the duty of the Registrar Management. It also takes care of the optional transmission of ASAP and ENRP announcements via multicast.

For PUs and PEs, the *Dispatcher* is the foundation of the *ASAP Instance* component. The *ASAP Instance* consists of three sub-components: *ASAP Protocol* is the implementation of ASAP for communication to PRs and between PE and PU via the multiplexed control/data channel. For creating and parsing ASAP messages, it contains the sub-components *ASAP Creator* and *ASAP Parser*. In the *Registrar Table*, addresses of usable PRs – either statically configured or learned by the PRs' announcements via multicast – are managed. When communication to a PR is necessary, this component also takes care of establishing a connection to a PR. The last sub-component of the *ASAP Instance* is the *ASAP Cache*, i.e. the PU-side cache for handle resolutions. For the implementation, the data structures and algorithms necessary to manage the cache are equal to the PR's handlespace management. Therefore, its code can be reused here.

In an early version of our prototype, we realized the handlespace management using linear lists and provided only round robin as pool policy. This worked fine for simple lab scenarios; however, there has been a growing demand to realize additional policies like random selection or least used for the research on load distribution performance. Furthermore, pools of load balancing scenarios can become very large (hundreds of elements) and efficiency becomes crucial. Therefore, our simple approach became unsuitable and a more sophisticated handlespace management concept was necessary. We will explain our concept in the following section.

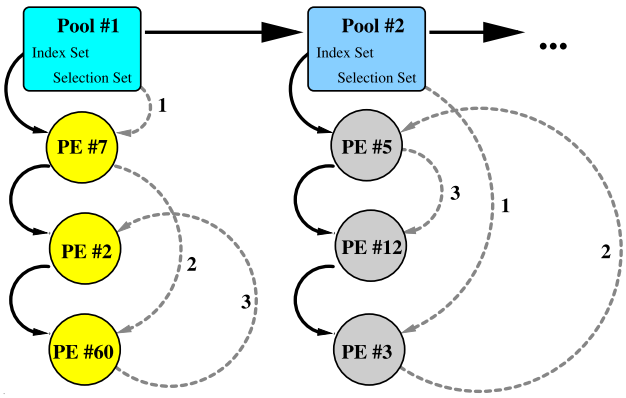

Fig. 6. Handlespace Management

V. Handlespace Management

Before we describe our implementation of the handlespace management, we first define it as abstract datatype: the *handlespace* is a set of n pools ($n \in \mathbb{N}$), denoted by PH h_1 to h_n. Each pool π contains a non-empty set of PE entries, denoted by their PE ID

$$i_{\pi k} \in \{0, ..., 2^{32} - 1\} \subset \mathbb{N}_0.$$

A PE entry includes the PE's policy information $y_{\pi k}$ (e.g. the PE's load in case of LU policy) and the PE's non-empty set of transport addresses $a_{\pi k}$.

The following operations must be possible on the handlespace datatype:

1) Insertion, lookup and removal of pools by pool handle;
2) Insertion, lookup and removal of PEs within a pool by PE ID;
3) Selection of PEs within a pool by policy;
4) Traversal of the handlespace for ENRP synchronization purposes.

Furthermore, it should be easily possible to add new selection policies for new applications.

A. Implementing the Handlespace

Implementing the abstract handlespace datatype becomes straightforward as illustrated in figure 6: there is a set of pools sorted by pool handle and each pool contains two sets of PE references – the first set sorted by PE ID (solid line), the second set sorted by a *sorting order* defined by the pool's policy (dotted line). We will explain later how to actually implement a *set*. A policy-specific *selection procedure* implements the selection of a PE. In the default case, this simply means to select

the first element from the set ordered by the *sorting order*. Using the structures above, it is only necessary to define a specific sorting order and selection procedure to implement a certain policy. Such definitions are the next step.

B. Implementing Policies

Before we define sorting order and selection procedure for some important policies defined in [7], we introduce two helper constructs for simplification:

For simplifying randomized selection, we define the following: to every PE entry i, a *value* $v_i \in \mathbb{R}$ may be mapped. It has to be possible to request the sum (called *value sum*)

$$V = \sum_i v_i$$

of all PE entries' values within the set. Then, randomized selection is possible by choosing a number

$$r \in_R \{0, ..., V\} \subset \mathbb{R}.$$

Since the set is ordered, r specifies the uniquely identifiable PE entry j that satisfies the condition

$$\sum_{i=1,...,j-1} v_i < r \leq v_j + \sum_{i=1,...,j-1} .$$

Furthermore, to guarantee uniqueness of sorting orders, we add sequence numbers to pools and PE entries: each pool gets a *pool sequence number* and each PE a *PE sequence number*. Every time a PE entry i is inserted into the selection set or being selected, its PE sequence number seq_i is set to the pool sequence number of its pool. Finally, this pool sequence number is then incremented by 1.

Now, we can define some example policies and show how the helper constructs are used:

a) Round Robin: The only sorting key is the PE entries' sequence number in ascending order and the selection procedure is the default one, i.e. getting the first element of the set. An example is given in table I: the upper block shows the pool "Example" before a selection. A selection returns PE entry ID-#1 (since it is the first entry of the set), its sequence number is set to the pool sequence number (4) and it is reinserted into the pool. Since it now has the highest sequence number, it is appended to the end of the set. Finally, the pool sequence number is incremented by one. A further selection will fetch PE ID-#2, then PE-ID-#3, again PE ID-#1 and so on, providing the desired round robin behaviour.

b) Weighted Random: Since random selection cannot take elements from the top of the selection set but has to use values v_i and their sum V, it is only necessary to ensure that the sorting keys in the set are unique. Using the PE sequence number as sorting key ensures this property. For the weighted random policy, the value v_i of PE entry i is set to the PE's given weight constant.

Table II shows an example: the pool consists of 3 PEs where PEs ID-#6 and ID-#7 have weight 1 (therefore $v = 1$). PE ID-#2 has weight 3 (and therefore $v = 3$) and PE ID-#8 has weight 2 (and therefore $v = 2$). The weight sum is therefore

$$V = 1 + 2 + 3 + 1 = 7.$$

For a selection, a random number

$$r \in_R \{0, ..., 7\} \subset \mathbb{R}$$

is chosen. Let $r = 5.75$. In this case, only $j = 3$ satisfies the condition

$$\sum_{i=1,...,j-1} v_i < 5.75 \leq v_j + \sum_{i=1,...,j-1} ,$$

that is

$$1 + 3 < 5.75 \leq 1 + 3 + 2.$$

Then, the third (j-th) PE of the set is selected: PE ID-#8. Using an uniform distribution for choosing r, weighted random results in the desired behaviour of selecting PEs at a probability proportional to their weight constant.

c) Least Used: Using the least used policy, each PE's policy information specifies the current server load as value from 0% to 100%. Clearly, the first part of the sorting key is this load value in ascending order. The second part is the PE sequence number in ascending order. We use the default selection procedure, i.e. taking the set's first element. Obviously, this will select the PE of the least load. And for the case that there are multiple PEs having the same least load, the PE sequence number as second part of the composite sorting key ensures round robin selection between these elements.

Clearly, arbitrary other policies can be expressed through definition of a sorting order and a selection procedure. That is, our policy implementation concept offers a solid foundation for future and application-specific extensions.

C. Performance

After definition of data structure and policies, the only remaining question of handlespace management is how to implement the datatype for the required *sets*. The naive solution is to simply use a linear list. A more efficient solution may be to use a binary tree, a red-black tree [21] (balanced tree) or a treap [22] (randomized tree). But does the effort for realistic pool sizes justify a more complicated structure?

To answer this question, we made a rough performance evaluation of our handlespace management implementation on an Athlon 1.3GHz CPU. We have chosen this CPU since its power seems to be realistic for upcoming router CPU generations[2] – routers are devices on which a PR process could be started. For our handlespace performance evaluation, we are not interested in SCTP or network layer efficiency, therefore we omit it here.

As test scenario, we assume two large pools, using the least used policy, in which we scale the average amount of PEs from 1 to 1000. Since pools map to specific applications, it is realistic to assume a small amount

[2]The current Juniper ERX 1400, a 300,000 US$ router, only contains a Pentium-III at 500MHz

TABLE I
ROUND ROBIN POLICY EXAMPLE

Pool "Example"	Policy RR $seq = 4$	Pool Element ID-#1	$seq = 1$
		Pool Element ID-#2	$seq = 2$
		Pool Element ID-#3	$seq = 3$
Pool "Example"	Policy RR $seq = 5$	Pool Element ID-#2	$seq = 2$
		Pool Element ID-#3	$seq = 3$
		Pool Element ID-#1	$seq = 4$

TABLE II
WEIGHTED RANDOM POLICY EXAMPLE

Pool "Example"	Policy WRR $seq = 5$ $V = 7$	Pool Element ID-#7	$seq = 1,\ v = 1$
		Pool Element ID-#2	$seq = 2,\ v = 3$
		Pool Element ID-#8	$seq = 3,\ v = 2$
		Pool Element ID-#6	$seq = 4,\ v = 1$

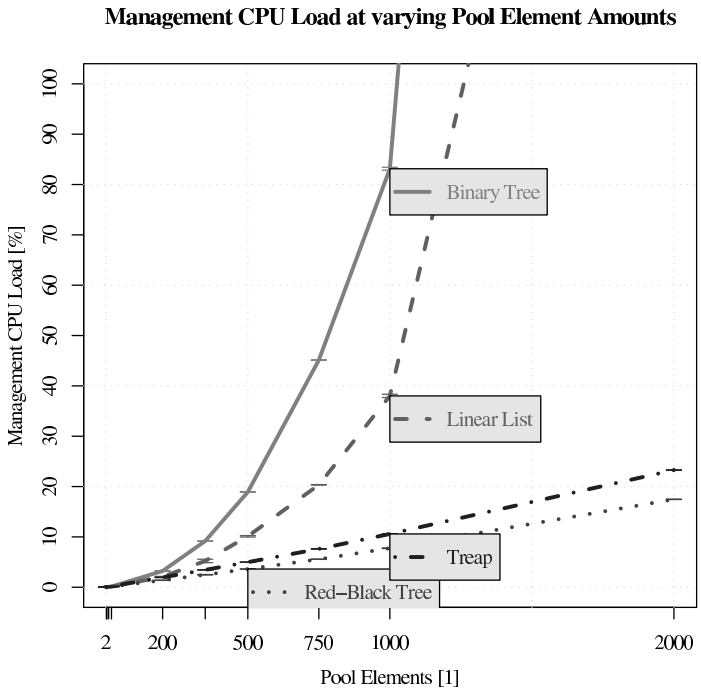

Fig. 7. Performance

(e.g. less than 20). Furthermore, applications requiring significantly large pools are assumed to be rare (e.g. a web server farm or a distributed computing service). Therefore, two large pools seem to be realistic. We omit adding additional small pools (e.g. 2 to 5 PEs) here, since this would not significantly affect the results.

Each PE is assumed to handle 10 PU requests/s (the more PEs, the more PU requests – adding servers only makes sense when there is more work to be done). That is, 10 handle resolutions per second and PE are required from the handlespace management. A PE stays registered for an average duration of 30m (uniform distribution) and then deregisters. During its runtime, a re-registration is made every 30s (default from [5]). When a PE is removed, a new PE is added to keep the average amount of PEs constant. Synchronization (this means traversal of the handlespace) is made every 5 minutes. The handlespace is *a priori* filled with the given amount of PEs; then, each test runs for 10m. The more components are in the scenario, the more handlespace operations have to be executed. For statistical accuracy, each test has been repeated 5 times; the shown results are the average values and their 95% confidence intervals, being computed by *R Project*.

Figure 7 shows the CPU's load as a fraction of the runtime (10m) required for handlespace operations in percent for the implementation of a *set* by linear list, binary tree, red-black tree and treap. On the x-axis, the total amount of PEs is shown (they divide up to the two pools). Obviously, balanced trees (red-black) and randomized trees (treap)

achieve the best results: at 2000 PEs (1000 per pool), they only occupy about 17% (red-black) and 21% (treap) of the CPU power. That is, assuming an efficient network stack (SCTP/IP, possibly implemented in hardware), the handling of large pools is even feasible on router CPUs.

Using binary trees, handlespace management consumes already about 82% of the CPU power at 1000 PEs and overload (i.e. denial of service) for more. Interestingly, a linear list only requires 38% CPU power at the same amount of PEs. The reason is that the set ordered by sorting order causes systematic removal from the front and re-insertion at the end of the set due to the sequence number. The binary tree becomes in fact a linear list but causes significantly higher computation effort due to its increased management effort.

In summary, using balanced or randomized trees is mandatory for efficient handlespace management.

VI. Conclusion and Outlook

In our paper, we have given a detailed introduction to the Reliable Server Pooling (RSerPool) framework which is currently under standardization by the IETF. RSerPool does not only provide mechanisms for configuring, accessing and monitoring pools of server resources but also provides sophisticated methods for server selection (pool policies). These features make RSerPool also useful for load balancing and distributed computing applications.

Furthermore, we have presented the *rsplib* prototype, our Open Source implementation of the complete RSerPool framework; we have shown its basic ideas, concepts and its building blocks. Handlespace management is one of its main parts, and its efficiency becomes crucial when pool sizes grow. We have presented our concept and implementation of an efficient and extensible handlespace management using sorted sets and reducing the effort of specifying a new policy to the definition of a sorting order and selection procedure. In a rough performance evaluation, we have proven the usefulness of our ideas.

After these first promising results, we are currently continuing the performance evaluation of our implementation in lab scenarios including network traffic and protocol overhead. Our goal is to provide recommendations to implementers and users of RSerPool with respect to tuning of system parameters in various application scenarios.

References

[1] 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*, Budapest, Hungary, 2001, pp. 168–174.

[2] 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," IETF, Standards Track RFC 2960, Oct 2000.

[3] 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.

[4] M. Tüxen, Q. Xie, R. Stewart, M. Shore, L. Ong, J. Loughney, and M. Stillman, "Requirements for Reliable Server Pooling," IETF, Informational RFC 3227, Jan 2002.

[5] Q. Xie, R. Stewart, and M. Stillman, "Endpoint Name Resolution Protcol (ENRP)," IETF, RSerPool WG, Internet-Draft Version 10, Oct 2004, draft-ietf-rserpool-enrp-10.txt, work in progress.

[6] R. Stewart, Q. Xie, M. Stillman, and M. Tüxen, "Aggregate Server Access Protcol (ASAP)," IETF, RSerPool WG, Internet-Draft Version 10, Oct 2004, draft-ietf-rserpool-asap-10.txt, work in progress.

[7] M. Tüxen and T. Dreibholz, "Reliable Server Pooling Policies," IETF, RSerPool WG, Internet-Draft Version 00, Oct 2004, draft-ietf-rserpool-policies-00.txt, work in progress.

[8] T. Dreibholz, E. P. Rathgeb, and M. Tüxen, "Load Distribution Performance of the Reliable Server Pooling Framework," in *Proceedings of the International Conference on Networking 2005*, Saint Gilles Les Bains/Reunion Island, Apr 2005.

[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*, vol. 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] A. Bivens, "Server/Application State Protocol v1," IETF, Individual submission, Internet-Draft Version 01, Oct 2004, draft-bivens-sasp-01.txt, work in progress.

[15] A. Jungmaier, E. 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*, vol. X, Orlando/U.S.A., Jul 2002.

[16] M. Ramalho, Q. Xie, M. Tüxen, and P. Conrad, "Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration," IETF, Transport Area WG, Internet-Draft Version 09, Jun 2004, draft-ietf-tsvwg-addip-sctp-09.txt, work in progress.

[17] Q. Xie, "Private communication at the 60th IETF meeting, San Diego/California, U.S.A." August 2004.

[18] T. Dreibholz, "An Overview of the Reliable Server Pooling Architecture," in *Proceedings of the International Conference on Network Protocols 2004*, Berlin/Germany, Oct 2004.

[19] "Thomas Dreibholz's RSerPool Page," http://tdrwww.exp-math.uni-essen.de/dreibholz/rserpool.

[20] "The sctplib Prototype," http://www.sctp.de/sctp.html.

[21] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science*, 1978, pp. 8–21.

[22] C. Aragon and R. Seidel, "Randomized search trees," in *Proceedings of the 30th IEEE FOCS*, 1989.