The Software Modeling and Implementation of Reliable Server Pooling and RSPLIB

Xing Zhou[†], Thomas Dreibholz[*], Martin Becke[*], Jobin Pulinthanath[*], Erwin P. Rathgeb[*] and Wencai Du[†]
[†]Hainan University, College of Information Science and Technology
Renmin Avenue 58, 570228 Haikou, Hainan, China
{zhouxing,wencai}@hainu.edu.cn
[*]University of Duisburg-Essen, Institute for Experimental Mathematics
Ellernstrasse 29, 45326 Essen, Germany
{dreibh,becke,jp,rathgeb}@iem.uni-due.de

***Abstract*—With the growing complexity of software applications, there is an increasing demand for solutions to distribute workload into server pools. Grid Computing provides powerful – but also highly complex – mechanisms to realize such tasks. Also, there is a steadily growing number of downtime-critical applications, requiring redundant servers to ensure service availability in case of component failures.**
To cope with the demand for server redundancy and service availability, the IETF has recently standardized the lightweight Reliable Server Pooling (RSerPool) framework, which is a common architecture for server pool and session management. In this paper, we first introduce the concept of RSerPool and then present the modeling thoughts of RSPLIB and the underlying general groupware design. Based on RSPLIB, we will illustratively show how to easily develop applications on top of RSerPool. We will also offer an application evaluation example for a proof-of-concept setup to distribute ray-tracing computation workload into a compute pool.[1]

Keywords: Reliable Server Pooling, RSPLIB, Service Availability, Software Modeling, Implementation

I. Introduction and Related Work

Service availability is becoming increasingly important in today's Internet. But – in contrast to the telecommunications world, where availability is ensured by redundant links and devices [1] – there had not been any generic, standardized approaches for the availability of Internet-based services. Each application had to realize its own solution and therefore to re-invent the wheel. This deficiency – once more arisen for the availability of SS7 (Signalling System No. 7) services over IP networks – had been the initial motivation for the IETF RSerPool WG to define the Reliable Server Pooling (RSerPool) framework. The basic ideas are not entirely new (see [2], [3]), but their combination into one application-independent framework is.

Server redundancy must lead to the issues of load distribution and load balancing [4], which are also covered by RSerPool [5]–[7]. But unlike solutions in the area of GRID and high-performance computing [8], the RSerPool architecture is designed into a lightweight system. That is, RSerPool may only introduce a small computation and memory overhead for the management of pools and sessions [6], [9]. In particular, this means the limitation is to a single administrative domain and only takes care of pool and session management – but not for higher-level tasks like data synchronization, locking and user management. These tasks are considered to be application-specific. On the other hand, these restrictions allow for RSerPool components to be situated on low-end embedded devices like routers or telecommunications equipment.

[1]Funded by the State Administration of Foreign Experts Affairs, P. R. China (funding number 20084600036) and the German Research Foundation (Deutsche Forschungsgemeinschaft).

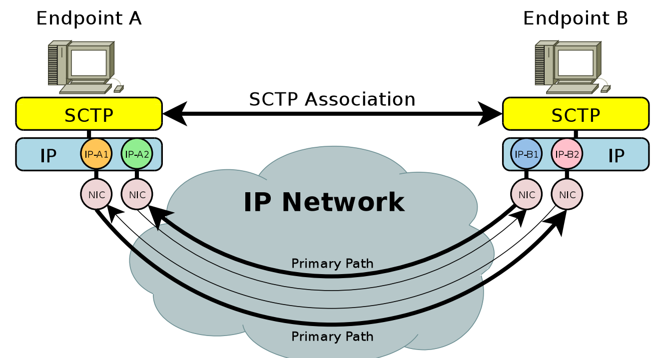

Figure 1. A Multi-Homed SCTP Association

The goal of this paper is to first introduce the protocols related to RSerPool – which include the underlying SCTP protocol, the RSerPool architecture and its protocol stack, session failover handling and application scenarios. Furthermore, we will present our software modeling's thought of RSerPool. Third, this paper analyzes the RSerPool API's functions – the basic mode API and the enhanced mode API – and shows a practical example how to respectively invoke the enhanced mode API of RSerPool at client side as well as server side in user applications. Finally, we show how to use the scripting service – a service provided by the RSerPool reference implementation RSPLIB – for distribution of POV-RAY-based ray-tracing image computations in a compute pool.

II. The RSerPool Architecture

RSerPool is based on the SCTP transport protocol. Therefore, it is necessary to first introduce this protocol and its link failure handling features.

A. The SCTP Protocol

The Stream Control Transmission Protocol (SCTP, see [10]) is a general purpose, reliable, connection-oriented, unicast transport protocol which provides reliable transport of user messages. A *connection* between two SCTP endpoints is called *association*. Each SCTP endpoint can use multiple IPv4 and/or IPv6 addresses to provide network fault tolerance. The addresses used by the endpoints are negotiated during association setup, a later update is possible by using dynamic address reconfiguration (Add-IP, see [11]). Add-IP can also be applied to allow for endpoint mobility. This link redundancy feature is called *multi-homing* [12] and illustrated in Figure 1. A *path* is a unidirectional network route between associated SCTP endpoints; SCTP endpoints can use multiple redundant paths through the network. One of the paths is selected as so-called *primary path*. This path is used for the transport of user data. The other paths are backup paths, which are used for retransmissions only. Upon failure of the primary path, SCTP selects a new primary path from the set of possible backup paths. That is, as long as there is at least one usable path in each direction, the association remains usable in spite of link failures.

CMT-SCTP [13], [14] is a Concurrent Multipath Transfer (CMT) extension for SCTP. Unlike standard SCTP [10], it utilizes *all* paths for data transport (not just a designated primary path). Combined with Resource Pooling (RP), the CMT/RP-SCTP [15] extension allows for TCP-friendly CMT transport over the Internet.

The *multi-streaming* feature of SCTP is the multiplexing of several independent message streams within one SCTP association. Since the message sequence integrity only has to be ensured within its own stream, SCTP multi-streaming avoids the "Head of Line Blocking" problem. Furthermore, SCTP also introduces security features like a 4-way handshake and a verification tag (see [10], [12], [16] for details), which make SCTP significantly less susceptible to Denial of Service attacks in comparison to TCP.

However, multi-homing and multi-streaming both cannot protect a service against endpoint failures. To cope with this problem, the IETF has defined the RSerPool architecture on top of SCTP.

B. The RSerPool Architecture

Figure 2 illustrates the RSerPool architecture, as defined in [17]. It consists of three major component classes: servers of a pool are called *pool elements* (PE). Each pool is identified by a unique *pool handle* (PH) in the handlespace, i.e. the set of all pools. The handlespace is managed by *pool registrars* (PR), which are also shortly denoted as *registrars*. PRs of an operation scope synchronize their view of the handlespace using the Endpoint haNdlespace Redundancy Protocol (ENRP [18]), transported via SCTP. An operation scope has a limited range, e.g. a company or organization; RSerPool does not intend to scale to the whole Internet. This restriction brings the benefit of a very small pool management overhead (see also [6]), allowing for hosting a PR service even on routers or embedded systems. Nevertheless, PEs may be distributed worldwide, for their service to survive localized disasters [19].

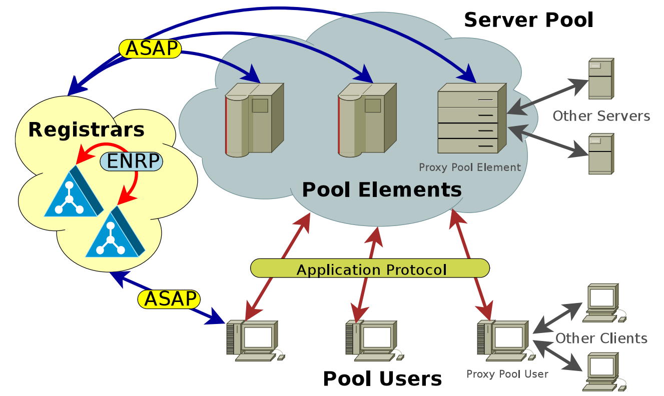

Figure 2. The RSerPool Architecture

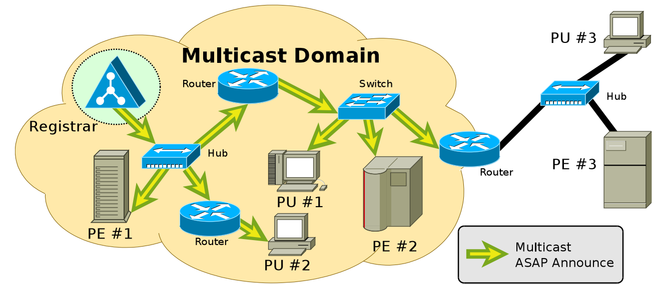

Figure 3. Automatic Configuration by ASAP Announces

A client is called *pool user* (PU) in RSerPool terminology. To use the service of a pool given by its PH, a PU requests a PE selection from an arbitrary PR of the operation scope, using the Aggregate Server Access Protocol (ASAP [20], [21]). The PR selects the requested list of PE identities using a pool-specific selection rule, called *pool policy*. Adaptive and non-adaptive pool policies are defined in [22]. The most important policies are the non-adaptive policies Round Robin and Random and the adaptive policy Least Used. Least Used selects the least-used PE, according to up-to-date load information. The actual definition of *load* is application-specific: for each pool the corresponding application has to specify the actual meaning of *load* (e.g. CPU utilization or storage space usage) and present it to RSerPool in form of a numeric value. Among multiple least-loaded PEs, Least Used applies Round Robin selection (see also [6]).

A PE can register into a pool at an arbitrary PR of the operation scope, again using ASAP transported via SCTP. The chosen PR becomes the *Home PR* (PR-H) of the PE and is also responsible for monitoring the PE's health by *ASAP Endpoint Keep-Alive* messages. If not acknowledged, the PE is assumed to be dead and removed from the handlespace. Furthermore, PUs may report unreachable PEs; if the threshold MAX-BAD-PE-REPORT of such reports is reached, a PR may also remove the corresponding PE. The PE failure detection mechanism of a PU is application-specific.

C. Automatic Configuration

RSerPool components need to know the PRs of their operation scope. While it is of course possible to configure a list

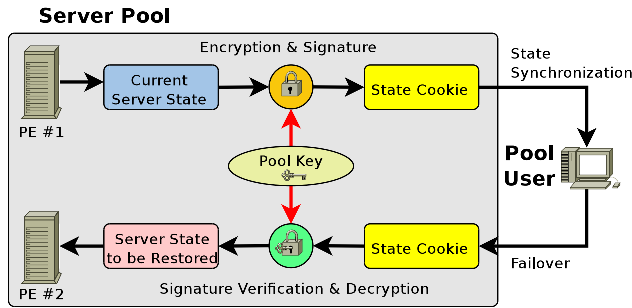

Figure 4. Client-Based State Sharing

of PRs into each component, RSerPool also provides an auto-configuration feature: PRs may send so called ANNOUNCES, i.e. special ASAP and ENRP messages which are regularly sent over UDP via IP multicast. Unlike broadcasts, multicast messages can also be transported over routers (at least, within LANs this is easily possible). The announces of the PRs can be heard by the other components, which can maintain a list of currently available PRs. That is, RSerPool components are usually just turned on and everything works automatically.

An example is provided by Figure 3 for the PU/PE configuration: all PEs and PUs within the multicast domain (e.g. a company LAN) can learn the identity of the PE automatically. Components outside of this domain (e.g. off-site systems in the Internet) need manual configuration.

D. Session Failover Handling

While RSerPool allows the usage of arbitrary mechanisms to realize the application-specific resumption of an interrupted session on a new server, it contains only one built-in mechanism: Client-Based State Sharing. This mechanism has been proposed by us in our paper [23] and it is now part of the ASAP standard [20]. Using this feature, which is illustrated in Figure 4, a PE can send its current session state to the PU in form of a state cookie. The PU stores the latest state cookie and provides it to a new PE upon failover. Then, the new PE simply restores the state described by the cookie. For RSerPool itself, the cookie is opaque, i.e. only the PE-side application has to know about its structure and contents. The PU can simply handle it as a vector of bytes (However, as described in [7], a more complex handling concept may improve application efficiency). Cryptographic methods can ensure the integrity, authenticity and confidentiality of the state information. In the usual case, this can be realized easily by using a pool key which is known by all PEs (i.e. a "shared secret").

III. The Software Modeling of RSPLIB

A. Overview of the Modeling Scheme

Our implementation of RSerPool – called RSPLIB – has been developed at the Computer Networking Technology Group of the Institute for Experimental Mathematics at the University of Duisburg-Essen, Germany. In order to support research and the ongoing RSerPool standardization process of the IETF, RSPLIB has been publicly available as Open Source under GPL license from the beginning. Meanwhile, it has also become the reference implementation of the IETF RSerPool WG. Source packages can be found on the project homepage [24], binary packages have already been included in Ubuntu Linux since December 2008. RSPLIB consists of three separate parts: a PR implementation, a library for the development of PUs and PEs and a set of example applications.

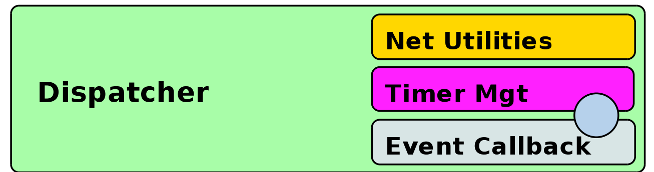

Figure 5. The Dispatcher Groupware Model

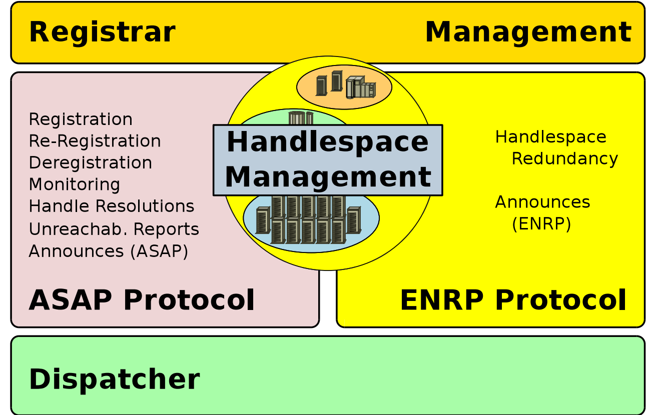

Figure 6. The RSPLIB Registrar Model

In order to support an easy portability of RSPLIB to different platforms, all components are built on top of an abstraction layer denoted as *Dispatcher*, a reusable groupware. The building blocks of the Dispatcher are illustrated in Figure 5. It consists of sub-components responsible for timer management (Timer Mgt), event handling (Event Callback) and an abstraction of the system's transport protocol API. Even though RSPLIB currently already supports Linux, FreeBSD, MacOS X and Solaris, an interface to other systems – particularly Microsoft Windows – should be relatively straightforward.

B. The Registrar Modeling

Clearly, the PR is the key component of an RSerPool system. The building blocks of the PR provided by RSPLIB are shown in Figure 6. The foundation of the PR is the Dispatcher. The central element of a PR is the handlespace management. Efficient handlespace management is crucial for the performance of the system and its scalability to a large number of PEs and PUs. Therefore, a significant research effort has been made on the data structures and algorithms to operate on the handlespace. Realistically, our PR implementation can manage up to many thousands of PEs and PUs. Details on the used data structures and algorithms, which are based on red-black trees, can be found in [6].

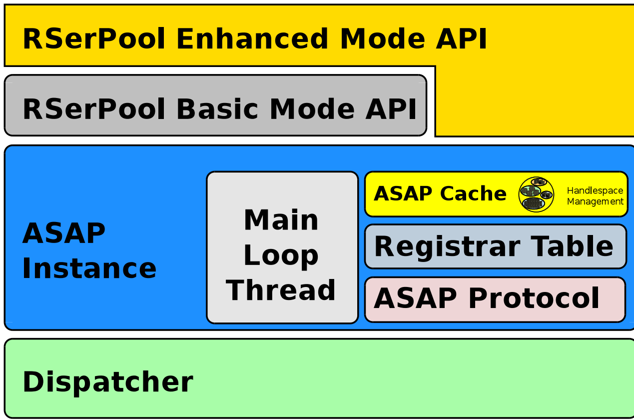

Figure 7. The RSPLIB PU/PE Library Model

In addition to the ENRP and PR-side ASAP protocols, the PR implementation furthermore features some security improvements suggested by us in [25], [26] to reduce the impact of Denial of Service (DoS) attacks on the deployed RSerPool systems. For the support of data integrity, authenticity and confidentiality, the PR currently relies on IPsec [27]. However, adding support for SCTP-aware TLS is an intended feature for a future version.

C. The PU/PE Library Modeling

The implementation of the PE and PU functionalities has been realized as a function library, which can be linked statically or dynamically to corresponding programs. The building blocks of the PU/PE library are presented in Figure 7: again the Dispatcher component is reused to encapsulate the system-dependent functionalities. On top of the Dispatcher, the ASAP Instance component realizes the core ASAP functionalities on the PU and PE sides. It consists of the following sub-components:

1) ASAP Protocol: This sub-component realizes the PU and PE side of the ASAP protocol, including message encapsulation and decapsulation.
2) Registrar Table: PR identities – configured statically by the administrator or learned by listening to the corresponding PRs' multicast ASAP Announces – are stored into the Registrar Table. The Registrar Table sub-component also takes care of flushing expired entries and to pick a randomly selected PR when needed.
3) ASAP Cache: The ASAP Cache reuses the handlespace management implementation of the PR to realize the PU-side cache. Each time a handle resolution is performed, its results are propagated into this cache and may be reused for further handle resolutions. For details on efficient cache usage, see also [5], [28]; an appropriate configuration of the cache may reduce PR query overhead.
4) Main Loop Thread: The Main Loop Thread is an event loop that handles timer events (e.g. flushing out-of-date PR entries in the Registrar Table) and socket events (e.g. answering ASAP Endpoint Keep-Alives). In order to simplify the usage of the RSerPool functionalities, the Main Loop Thread has been realized as a separate thread. That is, it runs in the background so that the application using the PU/PE library does not have to take care of the RSerPool event processing.

The ASAP Instance cannot be directly accessed by the application itself. Instead, the application access is provided by the RSerPool API on top of the ASAP Instance.

IV. The RSerPool API

The RSerPool API is a very important part of RSPLIB, another reusable groupware. It provides a method for existing applications to access RSerPool. The API consists of two API layers: the simple Basic Mode API and the powerful Enhanced Mode API.

A. The Basic Mode API

In order to let RSerPool support existing applications which do not require the support of the ASAP Session Layer (i.e. the Control Channel between PU and PE), the Basic Mode API offers the core RSerPool functionalities: registration, reregistration and deregistration for PEs, as well as handle resolution and failure reporting for PUs.

A non-RSerPool client application usually connects to a server by first resolving its hostname into a transport address using DNS and then creating and connecting a socket. The Unix function to resolve a hostname into a transport address is called *getaddrinfo()*[2]. Therefore, the Basic Mode PU-side API mimics the DNS resolution API of Unix for the handle resolution call: the RSPLIB function call *rsp_getaddrinfo()* resolves a PH into the transport address of a policy-selected PE. The structures including the transport addresses are compatible to the standard *getaddrinfo()* call. In case of a PE failure, the PU can report this failure to its PR using the *rsp_pe_failure()* function call.

On the PE side, the function call *rsp_pe_register* registers a PE into a pool. RSPLIB automatically takes care of answering ASAP Endpoint Keep-Alives by the PR. For deregistration, the function call *rsp_pe_deregister()* is provided.

Due to space limitations, the basic mode API is not explained in detail here; more information can be found in [29]. Instead, we focus on the more interesting Enhanced Mode API.

B. The Enhanced Mode API

The Enhanced Mode API offers the full RSerPool Session Layer functionalities, i.e. it takes care for connection establishment and maintenance as well as for triggering an application-specific failover procedure in case of PE failure.

1) Pool User Side: In a non-RSerPool client application, a connection to a server is usually established by resolving the server's hostname into a transport address by using DNS, creating a socket (using the Unix *socket()* call) and connecting this socket to the resolved transport address (using the Unix *connect()* call). After that, the calls *send()* and *recv()* can be used to send and receive data via the socket. Finally, the socket is removed by a call to *close()*. For more details, see [30].

[2] The deprecated *gethostbyname()* call is similar.

Algorithm 1 A Pool User using the Enhanced Mode API

```
// Create RSerPool session
session = rsp_socket(0, SOCK_STREAM, IPPROTO_SCTP);
rsp_connect(session, "StreamingMediaPool", ...);

// Run application: request radio stream
rsp_send(session, "GET /streams/music.ogg"
                  "&rate=44100&bps=16&ch=2 HTTP/1.0\r\n\r\n");
while((length = rsp_recv(session, buffer, ...)) > 0) {
   doSomething(buffer, length, ...);
}

// Close RSerPool session
rsp_close(session);
```

In order to make it easy for a programmer to write an RSerPool-based application, the approach for the Enhanced Mode API is to mimic the Unix sockets API. Listing 1 shows an example: First, a session is created by using the *rsp_socket()* call in line 2 (a session is also denoted as *RSerPool Socket* for compatibility reasons). After that, the session is connected to a pool by calling *rsp_connect()* in line 3. The pool is given by its PH (here: "StreamingMediaPool").

After establishment of the session, it can be used for the application protocol. In lines 6 to 10, the example application downloads a file by sending a request (by using *rsp_send()*) and receiving the file (by using a sequence of *rsp_recv()* calls). Note, that as soon as a PE has received the download request and sent a cookie including file name and current position to the PU, the RSPLIB can transparently handle failovers. That is, no additional application code is necessary. After completion of the file download, the session is finally removed by using the *rsp_close()* call (line 13).

2) Pool Element Side: A non-RSerPool server application usually creates a socket (by again using the *socket()* call), binds it to a specific port number (by using the *bind()* call, e.g. TCP port 80 for a web server), sets the socket into the "listen" mode (by using the *listen()* call) and accepts incoming connections by using the *accept()* call. For serving the newly connected client, a new thread may be created. It handles the application protocol on the new connection by using *send()* and *recv()* calls. The connection is closed by a call to *close()*. For more details, see [30].

As for the PU, the PE-side Enhanced Mode API also mimics the Unix sockets API. An example is given in Listing 2: first, a RSerPool socket is created (line 22) and the PE is registered to a pool given by its PH (here: "StreamingMediaPool"). The RSPLIB library will automatically take care of re-registrations. After registering the PE, a loop waits for incoming sessions (by using *rsp_poll()* in line 28), accepts them (using *rsp_accept()* in line 32) and creates new threads to serve them.

The thread function handling the session is shown in lines 2 to 17: first, it checks whether the first message received by *rsp_recv()* (line 4) is a state cookie. In this case, a saved session state is restored and the first command is read (line 7 to 8). After that, the loop from lines 10 to 15 handles commands and saves the current session states as state cookies (using *rsp_send_cookie()* in line 13; the RSPLIB library sends the cookies via ASAP Cookie messages over the Control Channel to the PU's Session Layer). Finally, the session is shut down by using *rsp_close()*.

The Enhanced Mode API also provides a UDP-like programming model (i.e. one socket handles multiple connections) and supports *poll()*/*select()*-based implementations (i.e. single-threaded programs). Due to space limitations, these programming schemes are not explained here. Some more details can be found at [24].

V. THE SCRIPTING SERVICE

The *Scripting Service* (SS) is one of the *rsplib* services which is based on the Enhanced Mode API. SS is a simple, useful and strong tool to apply RSerPool for distributing workload in a computation pool, based on shell scripts. This service is not "just another demo", it can be applied for real-world applications – as we will show in our proof-of-concept in Section VI. The Scripting Service works as follows: a PU can establish a session with a pool and upload a TAR/GZIP-archived workload package to a selected computation PE. The PE unpacks the archive into a temporary directory. Next, it runs a script (named `ssrun`) which is provided by the archive. This script can do arbitrary work and finally write an output archive. The output archive (i.e. the results of the work performed by the PE) is finally downloaded by the PU.

By using the Scripting Service, it becomes very easy to realize workload distribution of an application – e.g. to perform ray-tracing computations (as we will show in Section VI) or to process simulation runs. A user of this service just has to create appropriate workload packages (e.g. by using a small shell script to assemble them) and to call the Scripting Service PU – named `scriptingclient` – which is provided by RSPLIB. In order to avoid installing e.g. the computation program on the PEs themselves – which may be difficult in a heterogeneous pool of different operating system versions, installed software, etc. – the workload packages could also contain all needed binary executables and their shared libraries. Although this would mean an increased bandwidth requirement (which can be neglected in LAN setups), this could lead to a significantly reduced deployment effort.

The scripting pool can e.g. use the Least Used policy to achieve a balanced workload among its nodes. Each PE can handle up to SSMaxThreads sessions simultaneously [31];

Algorithm 2 A Pool Element using the Enhanced Mode API

```
// Service thread loop function
void serviceThread(session)
{
   rsp_recv(session, command, ...);
   if(command is a cookie) {
      // Got a cookie -> restore session state
      Restore state;
      rsp_recv(session, command, ...);
   }
   do {
      // Handle commands from pool user
      Handle command;
      rsp_send_cookie(session, current state);
      rsp_recv(session, command, ...);
   } while(session is active);
   rsp_close(session);
}

int main(...)
{
   // Create and register pool element
   poolElement = rsp_socket(0, SOCK_STREAM, IPPROTO_SCTP);
   rsp_register(poolElement, "StreamingMediaPool", ...);

   // Handle incoming session requests
   while(server is active) {
      // Wait for events
      rsp_poll(poolElement, ...);

      if(incoming session) {
         // Accept new session
         session = rsp_accept(poolElement, ...);
         Create service thread to handle session;
      }
   }

   // Deregister pool element
   rsp_deregister(poolElement);
   rsp_close(poolElement);
}
```

a PE's load value is set according to its actual number of sessions. For example, on a PE having a 4-core CPU, it is useful to set SSMaxThreads=4 to get all cores utilized. If a PE rejects a session (e.g. when it is already serving SSMaxThreads sessions), or if it goes out of service (e.g. when the PC is turned off), the session is simply restarted from scratch (i.e. the so-called "abort and restart" principle [7]) after a short delay (e.g. 5s). This delay avoids overloading the network with reject-and-retry floods [31] when there are too few PEs available.

Note that the Scripting Service not only provides simple load balancing: the RSerPool-based service is also highly available without additional effort (if there are at least 2 PRs and 2 PEs, of course).

VI. A Proof of Concept

A. Introduction

The "Persistence of Vision Ray-Tracer" [32] (POV-RAY) is a well-known Open Source ray-tracing application, i.e. it computes 3-D images with realistic lighting. Such image computations are very time-consuming: for example the image

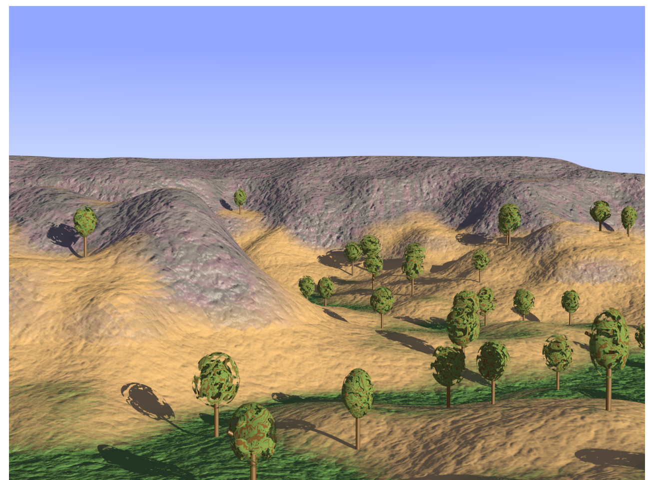

Figure 8. The Computed POV-RAY Example "`landscape.pov`"

shown in Figure 8, which is a computation result of the `landscape.pov` example provided by the POVRAY package, needs about 12 minutes to compute in a screen resolution of 1024×768 on a recent 64-bit, 2.3 GHz AMD Athlon 4400+ system. Clearly, such computations (e.g. as the images of a movie) are a strong candidate for workload distribution and a very illustrative example for utilizing the Scripting Service.

B. *Applying the Scripting Service for* POV-RAY

In order to use the Scripting Service for the POV-RAY computations, it is first necessary to write an appropriate `ssrun` shell script which will be executed on the PEs. Our working example is presented in Listing 3: it takes the name of the results archive as its first argument (line 2). In line 4 to line 5, POV-RAY is executed with the input file named `input.pov` (this file is a description of the image to be computed) and the output file named `output.png` (this is the completed image, in PNG format). The log output of the computation is written into a file named `output.txt`. In particular, this log output may contain error messages in case of problems. Therefore, it is useful to also archive this file – together with the resulting image (line 6) – for debugging purposes. Finally, the script returns with its status value. For the case where either the POV-RAY computation (line 4) or the archiving (line 6) has failed, it will be set to 1. A value of 0 (line 3) means that the computation has been successfully completed.

The 6-lined script shown in Listing 3 has already everything necessary to perform the remote POV-RAY runs using RSerPool/RSPLIB! On the PU side, the only task necessary is to pack an input file `input.pov` together with the `ssrun` script above into a Tar/GZip archive, provide it to RSPLIB's `scriptingclient` application and finally unpack the results (The PU-side script is not shown here due to space limitations; it can be found in our RSerPool tutorial [33]). That is, the programming effort for such a task – consisting of just a few lines of shell code – is really small.

C. *Experimental Proof-of-Concept Evaluation*

In order to show the effectiveness of our script to distribute POV-RAY runs into a pool, we have set up 5 PCs with a recent 64 bit, 2.3 GHz AMD Athlon 4400+ CPU running under FreeBSD 7.1 with RSPLIB version 2.6.4. An old 32-bit AMD Athlon 1.3 GHz system running under Ubuntu Linux version 9.04 ("Jaunty Jackalope") hosts a PR and also executes the PU-side script to distribute the runs. Note, that we have a heterogeneous setup, with different operating systems (Linux and FreeBSD) and CPU architectures (32-bit and 64-bit). Also, the PC handling pool management (by the PR) and workload distribution does not need a high CPU power – the old AMD Athlon system assembled in early 2001 (i.e. almost 9 years ago) is already sufficient for the lightweight RSerPool architecture! Clearly, the computation PCs must be powerful to run POV-RAY (while the actual RSerPool PE handling also requires minimal overhead).

In our pool, we have computed the POV-RAY example images in resolution 1024×768 for increasing the number of PEs in our compute pool. Each run has been repeated 5 times. The results plot – shown in Figure 9 – shows the average runtime in seconds as well as the 95% confidence intervals. The used pool policy has been Least Used, since it has the best load balancing quality (see [28], [31] for details on policies).

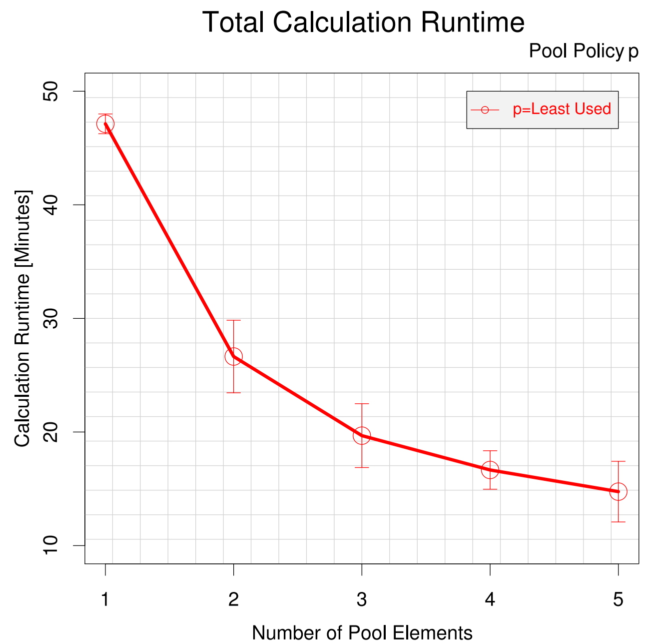

Figure 9. Resulting Runtime for Increasing Number of Compute PEs

Having only 1 PE, the computation takes about 47 minutes. With 2 PEs, the computation duration almost halves (27 minutes) and with 5 PEs it is reduced to around 15 minutes. The reason why it does not reduce to about $\frac{1}{5}$ of the runtime for 1 PE is that the computation work packages are quite heterogeneous. Some of the example images render in a few seconds while more complex images (like Figure 8) takes about 12 minutes. That is, for 5 PEs most of the PEs will become idle after a few minutes while some PEs still process the long-running requests. Clearly, this is a property of the application workload and not an issue of the load distribution architecture.

In summary, we have shown that the distribution of our workload to a pool of compute nodes works efficiently. Furthermore, the workload distribution can be realized easily, with an effort of only a few lines of shell code as introduced in Subsection VI-B.

VII. CONCLUSIONS

In this paper, we have provided an overview of Reliable Server Pooling (RSerPool), a common and lightweight framework for server pool and session management. We have introduced how to build up our reusable Dispatcher, RSPLIB Registrar and RSPLIB PU/PE Library of RSerPool by applying software engineering modeling theory.

The RSPLIB groupware is the reference implementation of RSerPool. Its two API layers provide the access to the RSerPool functionalities for applications. From a programmer's perspective, these APIs mimic the Unix socket API. These APIs make it rather easy and convenient to apply RSerPool in existing and new applications.

Algorithm 3 The `ssrun` Script for POV-RAY Image Computation Runs

```
1 #!/bin/sh
2 OUTPUT_ARCHIVE=$1
3 SUCCESS=1
4 povray -w -h +a0.3 -D +FN8 +Ooutput.png +Iinput.pov \
5    >output.txt 2>&1 || SUCCESS=0
6 tar czvf $OUTPUT_ARCHIVE output.png output.txt || SUCCESS=0
7 exit $SUCCESS
```

An application example of RSerPool is the Scripting Service. By using an example for the distribution of POV-RAY ray-tracing computation workload in a compute pool, we have shown that the application of RSerPool and the Scripting Service is simple, strong and the effort to realize our application only needs a few lines of shell script code. Finally, we have shown the effectiveness of our example approach – the distribution of image computations – by a proof-of-concept evaluation.

REFERENCES

[1] E. P. Rathgeb, "The MainStreetXpress 36190: a scalable and highly reliable ATM core services switch," *International Journal of Computer and Telecommunications Networking*, vol. 31, no. 6, pp. 583–601, Mar. 1999.

[2] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov, "Wrapping Server-Side TCP to Mask Connection Failures," in *Proceedings of the IEEE Infocom 2001*, vol. 1, Anchorage, Alaska/U.S.A., Apr. 2001, pp. 329–337, ISBN 0-7803-7016-3.

[3] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode, "Migratory TCP: Highly available Internet services using connection migration," in *Proceedings of the ICDCS 2002*, Vienna/Austria, July 2002, pp. 17–26.

[4] D. Gupta and P. Bepari, "Load Sharing in Distributed Systems," in *Proceedings of the National Workshop on Distributed Computing*, Jan. 1999.

[5] T. Dreibholz, "Reliable Server Pooling – Evaluation, Optimization and Extension of a Novel IETF Architecture," Ph.D. dissertation, University of Duisburg-Essen, Faculty of Economics, Institute for Computer Science and Business Information Systems, Mar. 2007.

[6] T. Dreibholz and E. P. Rathgeb, "An Evaluation of the Pool Maintenance Overhead in Reliable Server Pooling Systems," *SERSC International Journal on Hybrid Information Technology (IJHIT)*, vol. 1, no. 2, pp. 17–32, Apr. 2008.

[7] ——, "Overview and Evaluation of the Server Redundancy and Session Failover Mechanisms in the Reliable Server Pooling Framework," *International Journal on Advances in Internet Technology (IJAIT)*, vol. 2, no. 1, pp. 1–14, June 2009.

[8] I. Foster, "What is the Grid? A Three Point Checklist," *GRID Today*, July 2002.

[9] X. Zhou, T. Dreibholz, F. Fa, W. Du, and E. P. Rathgeb, "Evaluation and Optimization of the Registrar Redundancy Handling in Reliable Server Pooling Systems," in *Proceedings of the IEEE 23rd International Conference on Advanced Information Networking and Applications (AINA)*, Bradford/United Kingdom, May 2009, pp. 256–262, ISBN 978-0-7695-3638-5.

[10] R. Stewart, "Stream Control Transmission Protocol," IETF, Standards Track RFC 4960, Sept. 2007.

[11] R. Stewart, Q. Xie, M. Tüxen, S. Maruyama, and M. Kozuka, "Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration," IETF, Standards Track RFC 5061, Sept. 2007.

[12] P. Conrad, A. Jungmaier, C. Ross, W.-C. Sim, and M. Tüxen, "Reliable IP Telephony Applications with SIP using RSerPool," in *Proceedings of the State Coverage Initiatives, Mobile/Wireless Computing and Communication Systems II*, vol. X, Orlando, Florida/U.S.A., July 2002, ISBN 980-07-8150-1.

[13] J. R. Iyengar, P. D. Amer, and R. Stewart, "Concurrent Multipath Transfer Using SCTP Multihoming Over Independent End-to-End Paths," *IEEE/ACM Transactions on Networking*, vol. 14, no. 5, pp. 951–964, Oct. 2006.

[14] T. Dreibholz, M. Becke, J. Pulinthanath, and E. P. Rathgeb, "Implementation and Evaluation of Concurrent Multipath Transfer for SCTP in the INET Framework," in *Proceedings of the 3nd ACM/ICST OMNeT++ Workshop*, Málaga/Spain, Mar. 2010, ISBN 78-963-9799-87-5.

[15] ——, "Applying TCP-Friendly Congestion Control to Concurrent Multipath Transfer," in *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, Perth/Australia, Apr. 2010.

[16] E. Unurkhaan, "Secure End-to-End Transport - A new security extension for SCTP," Ph.D. dissertation, University of Duisburg-Essen, Institute for Experimental Mathematics, July 2005.

[17] P. Lei, L. Ong, M. Tüxen, and T. Dreibholz, "An Overview of Reliable Server Pooling Protocols," IETF, Informational RFC 5351, Sept. 2008.

[18] Q. Xie, R. Stewart, M. Stillman, M. Tüxen, and A. Silverton, "Endpoint Handlespace Redundancy Protocol (ENRP)," IETF, RFC 5353, Sept. 2008.

[19] T. Dreibholz and E. P. Rathgeb, "On Improving the Performance of Reliable Server Pooling Systems for Distance-Sensitive Distributed Applications," in *Proceedings of the 15. ITG/GI Fachtagung Kommunikation in Verteilten Systemen (KiVS)*, Bern/Switzerland, Feb. 2007, pp. 39–50, ISBN 978-3-540-69962-0.

[20] R. Stewart, Q. Xie, M. Stillman, and M. Tüxen, "Aggregate Server Access Protcol (ASAP)," IETF, RFC 5352, Sept. 2008.

[21] T. Dreibholz, "Handle Resolution Option for ASAP," IETF, Individual Submission, Internet-Draft Version 06, Jan. 2010, draft-dreibholz-rserpool-asap-hropt-06.txt, work in progress.

[22] T. Dreibholz and M. Tüxen, "Reliable Server Pooling Policies," IETF, RFC 5356, Sept. 2008.

[23] T. Dreibholz, "An Efficient Approach for State Sharing in Server Pools," in *Proceedings of the 27th IEEE Local Computer Networks Conference (LCN)*, Tampa, Florida/U.S.A., Oct. 2002, pp. 348–352, ISBN 0-7695-1591-6.

[24] ——, "Thomas Dreibholz's RSerPool Page," 2010.

[25] T. Dreibholz, E. P. Rathgeb, and X. Zhou, "On Robustness and Countermeasures of Reliable Server Pooling Systems against Denial of Service Attacks," in *Proceedings of the IFIP Networking*, Singapore, May 2008, pp. 586–598, ISBN 978-3-540-79548-3.

[26] T. Dreibholz, X. Zhou, E. P. Rathgeb, and W. Du, "A PlanetLab-Based Performance Analysis of RSerPool Security Mechanisms," in *Proceedings of the 10th IEEE International Conference on Telecommunications (ConTEL)*, Zagreb/Croatia, June 2009, ISBN 978-953-184-131-3.

[27] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol," IETF, Standards Track RFC 2401, Nov. 1998.

[28] T. Dreibholz and E. P. Rathgeb, "On the Performance of Reliable Server Pooling Systems," in *Proceedings of the IEEE Conference on Local Computer Networks (LCN) 30th Anniversary*, Sydney/Australia, Nov. 2005, pp. 200–208, ISBN 0-7695-2421-4.

[29] T. Dreibholz and M. Tüxen, "High Availability using Reliable Server Pooling," in *Proceedings of the Linux Conference Australia (LCA)*, Perth/Australia, Jan. 2003.

[30] W. Stevens, B. Fenner, and A. Rudoff, *Unix Network Programming*. Addison-Wesley Professional, 2003, ISBN 0-131-41155-1.

[31] X. Zhou, T. Dreibholz, and E. P. Rathgeb, "A New Approach of Performance Improvement for Server Selection in Reliable Server Pooling Systems," in *Proceedings of the 15th IEEE International Conference on Advanced Computing and Communication (ADCOM)*, Guwahati/India, Dec. 2007, pp. 117–121, ISBN 0-7695-3059-1.

[32] POV-Team, "POV-Ray – The Persistence of Vision Ray Tracer," 2010.

[33] T. Dreibholz, "SCTP and Reliable Server Pooling – A Practical Exercise," University of Duisburg-Essen, Institute for Experimental Mathematics," NGComNet Practical Exercise, July 2009.