High Availability using Reliable Server Pooling

Thomas Dreibholz (dreibh@exp-math.uni-essen.de)
University of Essen, Institute for Experimental Mathematics
Ellernstraße 29, 45326 Essen, Germany
Michael Tüxen (Michael.Tuexen@siemens.com)
Siemens AG, ICN CP SE C 7
81359 Munich, Germany

December 15, 2002

Abstract

Providing fault tolerancy is crucial for a growing number of IP-based applications. There exist a lot of proprietary solutions for this problem, but free alternatives are rare.

Currently, the IETF RSerPool working group is standardizing a protocol suite for Reliable Server Pooling, which copes with the challenge of providing high availability by using redundant servers. Servers for the same service are grouped into a server pool. A server in a pool is called pool element (PE), a user of a pool is called pool user (PU). When a PE fails, its PUs simply select another one from the pool and initiates an application-specific failover procedure. This fail-over is supported by the RSerPool protocol suite. Each PE registers at a name server and is then continously supervised by that specific name server. All name servers of an operational scope provide a redundant system for name resolution from pool handles to transport addresses of pool elements to pool users. RSerPool uses SCTP to provide network fault tolerance and address scoping functionality.

The RSPLIB is a prototype implementation of the RSerPool protocol suite, developed under the GNU Public License in cooperation between Siemens and the Computer Networking Technology Group of the University of Essen. It currently runs under Linux, FreeBSD and Darwin.

Our paper covers aspects of designing and implementing highly available applications using RSerPool with our RSPLIB implementation. First, we will give an introduction to the RSerPool protocol suite and an overview of the RSPLIB components. Then, we describe the RSPLIB API, especially focussing on the implementation of pool element and pool user programs to provide high reliability. Furthermore, we show our current implementation status and future plans. This will be followed by a short look on the problems that can arise when the RSerPool architecture is used. Finally, we give an example how RSerPool can be used to realize highly available services. And last but not least, we show how distributed computing architectures can make use of the RSerPool architecture.

This work is part of KING, a research project of Siemens AG. The work of this project is partially funded by the Bundesministerium für Bildung und Forschung of the Federal Republic of Germany (Förderkennzeichen 01AK045).

1 Introduction to Reliable Server Pooling

The SIGTRAN working group [1] of the Internet Engineering Task Force (IETF) is developing a protocol suite for transporting telephony signaling over IP-based networks. The Signaling System No. 7, the protocol suite used for telephony signaling in the TDM world, provides a very high degree of redundancy and availability. The same service is also desirable for an IP-based solution.

Therefore, it was decided to develop a network fault tolerant transport protocol which is used for all adaptation layers of the SIGTRAN protocol suite. This transport protocol is called the Stream Control Transmission Protocol (SCTP). However, using SCTP does not provide any help if a server fails. Server failures can only be handled by having multiple servers providing the same functionality. All SIGTRAN adaptation layers provide some sort of server pooling using multiple Application Server Processes (ASPs)

in one Application Server (AS). It was decided that having a generic and common solution for the server pooling functionality would be helpful and usable also outside SIGTRAN. Therefore, the Reliable Server Pooling (RSerPool) working group [12] has been founded.

The RSerPool protocol suite focuses on providing server redundancy using server pools. In combination with the network fault tolerant transport protocol SCTP, it is possible to build systems without single points of failure.

The RSerPool architecture uses three classes of elements:

Pool Elements (PEs) These are the servers being part of a pool and providing the same service within a pool.

Pool Users (PUs) These are the clients being served by one PE.

Name Servers (NSs) These nodes provide a translation service and supervise the PEs.

A pool is identified by a pool handle, which is a byte vector of arbitrary length. If a server wants to become a PE for a specific pool, it just registers itself with the pool handle of the pool at one of the name servers. The protocol used between the PEs and the NSs is called the Aggregate Server Accesss Protocol (ASAP), currently being defined in [4]. This name server will supervise this PE to make sure that it is working and informs the other NSs about the new PE. The pool handle is only valid in its operational scope. All NSs within an operational scope have information about all PEs within the operational scope. This means that the namespace used by RSerPool is flat. The protocol used by the NSs to exchange their information is called the Endpoint Name Resolution Protocol (ENRP), currently being defined in [5]. If the IP network provides multicast capabilities, the NSs can send out server announcement messages using IP-multicast. This allows PUs, PEs and other NSs to detect NSs.

If a client wants to be served by a PE belonging to a pool identified by a specific pool handle, it sends a name resolution request to a NS. The NS will respond with a subset of all transport addresses, which can be used to access the PEs. This communication is also using ASAP. The selection of the PE is realized in two steps: In the first step, the NS can select a subset of all PEs and their transport addresses in the pool. This selection can be based on the requested transport capabilities and/or the pool policy. In the second step, the PU has to select one of the PEs in the given subset. This can also be based on the pool policy. Examples for pool policies are round robin or least used. Other policies are also available in RSerPool; it is furthermore easily possible to add new ones.

Whenever a PU detects that a PE cannot be reached, one of the NSs is informed. This information, combined with the ongoing supervision, is used to remove PEs from the pool if they are out of service.

In contrast to the Domain Name System (DNS), RSerPool

1. uses a flat namespace,
2. allows arbitrary pool handles,
3. provides a high probability that only PEs are announced which are in service and
4. allows dynamic registration and deregistration.

In case of a failure of a PE, the PU can fail-over to a different PE of the same pool. The grade of intervention of the RSerPool upper layer depends on the required service. It requires no intervention if the upper layer is always using pool handles for sending and allows some messages to be dropped during the fail-over. If a message loss during fail-over is not acceptable, the upper layer has to use application-level acknowledgements and its own buffering.

The communication between PU and PE consists of two channels:

data channel This channel is used for the service related traffic.

control channel This channel is used for RSerPool related traffic.

The control channel is used for different kinds of RSerPool services. First of all, it is allowed that a PU is also a PE of some pool. To provide also redundancy for that PU, the peer of the PU must know the pool handle of the pool the PU belongs to. This information is sent using ASAP as part of a so called business card from the PU to the PE. Another usage of the control channel is the last will. A PE can send a last will to the PU, containing a list of specific PEs of its pool to use for fail-over in case of its failure. A third usage of the control channel are the cookies. A PE can send a cookie to the PU whenever it wants. The PU stores only the latest received cookie. In case of a fail-over to a different PE, it sends the stored cookie to the new PE first. This very simple method can be used to transfer state from the failed PE to the new PE. See also [9] for more information. However, this does not provide a generic method for state sharing between the PEs, which is out of scope of the RSerPool working group.

The data channel and the control channel have to be tied together. If SCTP is the transport protocol for the data channel, the control channel and the data channel can use the same transport connection, called an SCTP association. SCTP supports the multiplexing of multiple upper-layer protocols by using different payload protocol identifiers (PPID). Messages for the control channel will use the registered PPID for ASAP, the data channel the PPID for the specific service.

If UDP is the transport protocol of the data channel, the control channel cannot be multiplexed with the data channel for at least two reasons: One would have to define a TCP-friendly congestion control algorithm for the ASAP traffic and provide reliable transport within the ASAP layer. Both would make ASAP much more complex. Therefore, it is currently being suggested that the control channel is transported over TCP or SCTP. Since all ASAP messages use a Tag-Length-Value (TLV) structure, both transport protocols can be used. The control channel and the data channel are tied together by sending first a message on the control channel describing the transport layer endpoints of the data channel.

If TCP is the transport protocol of the data channel, there are two possible solutions. One can use a separate TCP connection for the control channel and use the same mechanism as for the UDP-based data channel to tie the control and data channel together. The advantage of this method is that the service related traffic, the data channel, is not modified. The second solution is to multiplex the control and data channel on one TCP connection. In this case, the multiplexing layer as defined in [6] has to be used.

This handling of data channel and control channel is currently being discussed in the RSerPool working group. The support of other transport layers, for example DCP, or protocols not using a transport layer but running directly on top of IP is currently not being discussed. But the RSerPool protocol suite can be easily extended to support them in the future.

It should be noted that the ASAP communication between the PE and the NS (i.e. registration, deregistration, supervision) must be transported over SCTP. The communication between the PU and the NS (i.e. name resolution) can be transported over TCP or SCTP. This does not force PUs, which only want to access TCP-based services, to implement SCTP as an additional transport protocol just for simple name queries. It is required, that the PU side implementation of the RSerPool suite can be very light-weighted. The communication between the name servers can be SCTP-based or IP multicast based, if this is supported by the IP layer. The IP layer can be based on IP version 4 and/or IP version 6.

Using SCTP for handling network failures and the RSerPool protocol suite for handling server or host failures, one can provide a system without a single point of failure. However, to provide a highly reliable service it is also required that no one can modify the name system without being authorized. That is, a PE communicating to a NS should be authenticated. Security is a very important part of RSerPool. Threats have been analyzed and the protocol design of the RSerPool suite ensures that existing protocols, e.g. Transport Layer Security (TLS) and IP-Security (IPSec), are applicable. Defining own solutions is out of the scope of RSerPool.

2 The Implementation

Our implementation of the RSerPool protocol suite has been released under the GNU Public License (GPL) and can be downloaded under [13] and [11]. A key requirement for our implementation has been the portability to different platforms, especially also non-Unix ones. While currently only Linux, FreeBSD and Darwin are supported, there are also plans to use it for devices like mobile phones or PDAs. Applications for such devices especially include highly available e-commerce and mobile commerce. The portability

requirement has leaded to our decision not to assume the availability of threads (even single-task operating systems like FreeDOS or M$-DOS should be supported). Furthermore, C instead of C++ has been chosen as implementation language for the reason that ANSI-C compilers are more widespread than C++ ones; although an object-oriented language would have advantages for implementation effort and maintainability of the code. However, our implementation's core has been realized in an object-oriented scheme. IPv6 has been fully supported from the beginning of the project.

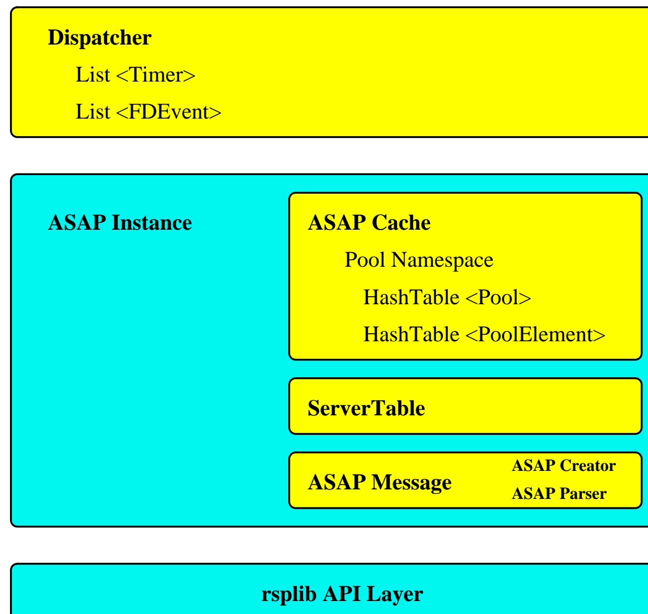

Figure 1: The RSPLIB implementation overview

An overview of our implementation can be found in figure 1. We describe the parts shown there in the following.

2.1 The Dispatcher Class

The *Dispatcher* class manages timers and events on sockets. That is, file descriptors (here, especially for sockets) can be registered at a *Dispatcher* object for the events read, write and/or exception. Furthermore, *Timer* objects can be registered. Both, sockets and timers, have to provide a callback function that is invoked when a registered event occurs or a timer has expired. Waiting for events and invoking the callback function have been implemented as two separate parts of the *Dispatcher* class: First, the method *dispatcherGetSelectParameters()* obtains the parameters for the Unix-standard function *select()*; that is, *fdset*s for read, write and exception on file descriptors and a *timeval* structure containing the maximum timeout. The user of the *Dispatcher* object may now add additional file descriptors to the *fdset*s or decrease the timeout. Then, *select()* can be called, followed by invoking the *Dispatcher* method *dispatcherHandleSelectResult()*, which finally handles possible socket or timer events.

Note, that the *Dispatcher* class itself does not require that the operating system has some form of *select()* function. Since waiting for events has to be realized outside of the *Dispatcher* class, a wrapper function may be realized which takes the select parameters (*fdset*s and timeout) and implements the desired functionality using an OS-specific system call.

The *Dispatcher* class provides two methods, *dispatcherLock()* and *dispatcherUnlock()*. Their only functionality is to invoke the callback functions *lock()* and *unlock()*, which have to be specified to the *Dispatcher* constructor. These callbacks can realize obtaining and releasing a recursive mutex in the case that the used operating system (e.g. Linux) supports threading (e.g. using *libpthread*). The current default is to

use *libpthread* when it is available (Linux, FreeBSD, Darwin). The *Dispatcher* class uses the *dispatcherLock()* and *dispatcherUnlock()* functions everywhere where exclusive access to its structures is necessary. Of course, classes derived from *Dispatcher* may use this functionality, too. Using these two functions, system-independent thread-safety is realized in our implementation.

2.2 The ASAPInstance Class

The *ASAPInstance* class takes care for locating a name server and maintaining a connection to it. Furthermore, it provides registration and deregistration of pool elements, name resolution and policy-based pool element selection. Finally, it maintains the name resolution cache. It consists of three classes: *ServerTable*, *ASAPCache* and *ASAPMessage*.

2.2.1 ServerTable

A *ServerTable* object is responsible for creating multicast sockets to listen for server announces. Received announces are added to the server table. Furthermore, static entries may be added when the underlying network does not support or prohibits multicasts. The dynamic entries in the table are flushed when they are not refreshed within a certain configured interval.

The *ServerTable* class furthermore provides the establishment of a connection to one of the announced name servers. To make this process as fast as possible, especially for the case that some servers of the list have just become unreachable, it is tried to connect to several name server addresses from the server table simultaneously. The first successfully established connection is taken as new name server connection. To increase speed, the connection trial timeout does not use the system's default. Instead, it uses a configured, lower value (e.g. 5 to 10 seconds).

2.2.2 ASAPCache

An *ASAPCache* object contains a *PoolNamespace* object, which contains the part of the name server's namespace cached by name resolutions. A timer realizes purging pool elements and pools that have not been refreshed within a certain configured interval. The *PoolNamespace* object contains hash tables for *Pool* objects and *PoolElement* objects. A *Pool* object contains a list of owned *PoolElement*s. A pool element selection by a pool policy (e.g. least used or round robin) specified in a *PoolPolicy* object is also realized within the *Pool* class.

2.2.3 ASAPMessage

The *ASAPMessage* class is responsible for the creation of outgoing ASAP messages (*ASAPCreator* module) and parsing incoming ones (*ASAPParser* module).

2.3 The RSPLIB API layer

Due to portability reasons, no core class uses global variables. Therefore, all of these classes are reentrant. Using an operating system with MMU-based memory management, a shared library can be loaded once and then mirrored to the programs' memory spaces. On writes, the affected memory pages can simply be duplicated, creating an exclusive copy for the program. Having no MMU, which is usually the case for mobile phones, PDAs and also routers, this is impossible. But having a reentrant library, it may be loaded once into the global memory and accessed by all programs. An excellent example for such a realization is the good old AmigaOS operating system.

The RSPLIB API layer provides a simple and small wrapper for the core classes. While the core classes have been designed for portability and reusability, the RSPLIB wrapper should make the usage of the RSerPool functionality as simple as possible. This especially includes providing a programming interface as similar as possible to the current non-RSerPool socket and name resolution API.

While the RSerPool protocol suite is currently still under standardization at the IETF and functionality may be heavily changed, added or removed, the programming API should be as stable as possible and only

changed when it is absolutely necessary. A changed API requires recompiling the applications while a binary-compatible one simply allows to update a shared library. To allow introducing new parameters and skipping obsolete ones, the RSPLIB API has been inspired by a simple but effective idea from the good old AmigaOS 2.04, the so called tag items. Such tag items are simple arrays containing (*Tag*, *Data*) tuples. *Tag* is an at least function-unique number denoting a certain parameter; the *Data* field contains the parameter's value (e.g. an integer value or a pointer to a structure). The special tag *TAG_DONE* denotes the end of the tag item array.

3 The RSPLIB API

3.1 Initialization and Clean-up

Before any other function of the RSPLIB library can be used, it has to be initialized using the *rspInitialize()* function. This will create global *Dispatcher* and *ASAPInstance* objects. The tag items for *rspInitialize()* allow specifying custom *lock()* and *unlock()* functions for the *Dispatcher* object to ensure thread-safety by a specific operating system's functionality (e.g. *libpthread* or GNU *pth* for Unix-clones or Semaphore functions for AmigaOS). After the initialization, the RSPLIB is ready for usage. Note, that at this point no connection to a name server has been established. The RSPLIB only listens for server announces. A connection will be established when it is required for the first time. The reason is simple: Depending on the name servers' announce interval, it may take a few seconds until the first announce is received. Until this happens, the program can probably do more useful things than waiting.

To remove all objects created during initialization and runtime of the RSPLIB, the function *rsp-CleanUp()* has to be called. It will free all resources allocated by the RSPLIB. Such a functionality is mandatory under all operating systems without resource tracking (e.g. DOS or AmigaOS). That is, only the program itself remembers which memory blocks, file descriptors, etc. have been allocated or opened; the program itself is responsible to free or close them before exiting. Any non-freed memory block or non-closed file descriptor would be allocated or opened until the next reboot.

But the clean-up functionality is also useful for debugging purposes under Linux. The Valgrind [2] memory debugger completely interprets the x86 assembler code of programs and is therefore able to track the usage of every bit. If there are any remaining memory blocks allocated by the RSPLIB after *rsp-CleanUp()*, there must be a memory leak. Due to the tracking functionality of Valgrind, the location of the lost allocation is displayed and can be corrected easily. During development of the RSPLIB, Valgrind has shown to be an excellent tool for identifying and correcting all kinds of memory problems.

3.2 The Event Loop

The RSPLIB functions register timer events (e.g. pool element cache and server table maintenance) and socket events (e.g. the name server connection or the multicast sockets for server announces) at the global *Dispatcher* object. To wait for such events using *select()*, the *dispatcherGetSelectParameters()* and *dis-patcherHandleSelectResult()* have to be used. This is encapsulated in the *rspSelect()* function. This function could replace the *select()* function in the program's main loop or be called within an own thread, depending on the program structure and the operating system's capabilities.

3.3 The Pool User API

As stated before, the RSPLIB's API should be as compatible to the "normal" programming interface for network applications as possible. Before we explain the pool user functionality of the API, let us first have a short look at the usual program structure of a client application shown in algorithm 1: A given hostname or IP address is resolved or converted into a *sockaddr* structure by the function *getaddrinfo()*. Alternatively, the older function *gethostbyname()* can be used; but here, thread-safety and IPv6 support are not ensured. The next step is to create a socket and connect to the peer. Finally, the *addrinfo* structure created by *getaddrinfo()* has to be freed using *freeaddrinfo()*.

Algorithm 1 Network client program flow

```
struct addrinfo* ai = NULL;
...
getaddrinfo("linux.conf.au", ..., &ai);
...
sd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
if(sd >= 0) {
   if(connect(sd, ai->ai_addr, ai->ai_addrlen) {
      ...
   }
   freeaddrinfo(ai);
   ...
}
```

Algorithm 2 Pool user program flow

```
struct EndpointAddressInfo* eai;
...
rspInitialize();
...
poolHandle = "DownloadPool";
rspNameResolution(poolHandle, strlen(poolHandle), &eai);
...
sd = socket(eai->ai_family, eai->ai_socktype, eai ->ai_protocol);
if(sd >= 0) {
   if(connect(sd, eai->ai_addr, eai->ai_addrlen) {
      ...
      if(failure) {
         rspFailure(poolHandle, strlen(poolHandle), eai->ai_identifier);
         ...
      }
      ...
   }
   ...
}
...
rspFreeEndpointAddressArray(eai);
...
rspCleanUp();
```

To adapt the application for the usage of RSerPool, it is first necessary to replace the resolution of a hostname by an RSerPool name resolution. It has to resolve a given pool handle to a list of one or more pool elements and then select one of them by the given pool policy (e.g. the least used). Then, it can try to connect to the pool element's address (or one of the addresses). In case of failure, pool element selection or even the name resolution can be repeated. To keep the changes to the application as small as possible, the RSPLIB's *rspNameResolution()* takes arguments similar to *getaddrinfo()*. But instead of a hostname, the pool handle and its length are specified here. The structure *addrinfo* has been extended by a field with the pool element identifier of the returned pool element (*ai_identifier*). This structure is called *EndpointAddressInfo* and has to be freed after usage by *rspFreeEndpointAddressArray()*. The program structure of the pool user, that is how the client is now denoted in RSerPool terminology, looks as shown in algorithm 2. Note, that repetition loops for the case of a connection failure during establishment or usage are not shown for simplicity.

When the connection to a pool element fails during establishment or usage, the pool user may inform the name server about this failure using the *rspFailure()* call. The name server may then decide to remove the pool element from its namespace.

3.4 The Pool Element API

The ASAP functionality of a pool element consists of registering itself at a name server, renewing this registration regularly and finally deregistering itself. Furthermore, a pool element has to answer keep-alive messages from the name server (supervision functionality).

To register a pool element, the function *rspRegister()* is used. It takes the pool handle, pool handle size and the pool element's addresses in form of an *EndpointAddressInfo* structure and additional parameters as tag items. The policy type (e.g. least used or weighted round robin) and policy parameters (e.g. load for least used and weight for weighted round robin) are defined as tags. This allows adding more types and parameters without affecting the API. To finally deregister a pool element, the function *rspDeregister()* has to be called.

The *rspRegister()* function also has to be called to re-register a pool element, that is renewing its registration. During re-registration, the pool element's addresses and policy settings may be changed. If the name server fails, that is the connection breaks or there is no answer within a certain configured interval, a new name server will be searched and the pool element will be registered there.

The reason for not implementing an automatic re-registration functionality e.g. using a timer in the ASAPInstance class, is that such a function may block for a certain amount of time when the name server connection fails: The failure has to be detected, a new name server has to be searched, a new connection established and all pool element have to be registered there. If the *rspSelect()* function is invoked from the main loop of a single-threaded server application, this would block the service! A recommendation for pool elements is therefore to use an own thread or process for the RSerPool functionality.

3.5 Implementation Status and Future Plans

The current implementation status of the RSPLIB is as follows: ASAP has been implemented conforming to version 05 of the ASAP draft [4]. Currently, control channels are not supported, because the methods for tying together the control and data channel are currently still under discussion at the IETF RSerPool working group [12]. Therefore, business cards, last wills and cookies have not been implemented yet. But it is expected that this functionality, based on SCTP as the transport protocol, can be realized during the next few weeks. Our name server currently does not support sharing its data with others. Since the ENRP draft [5], which describes the name server communication, has been heavily changed and is still under discussion, no implementation effort has been invested here, yet. Since the draft seems to stabilize now, a full-featured name server is our project's next major step.

As explained in the introduction, the RSerPool architecture is strongly related to the SCTP protocol. Currently, our project uses our SCTP userland implementation SCTPLIB [13] for Linux, FreeBSD and Darwin with its standard-compliant socket API [14]. This SCTP implementation is another successful cooperation project between the University of Essen and Siemens. Since our socket API conforms to the standard, almost no changes should be necessary to make the RSPLIB run with kernel SCTP, e.g. the Linux-SCTP implementation [15] of the upcoming kernel 2.6 or the SCTP implementation being part of the KAME stack [16] for FreeBSD. Tests with kernel SCTP implementations have been planned for the next few weeks.

To keep up-to-date with our development, see the news section of [11] and/or subscribe to our mailing list under [10] or [11]. Our mailing list archive can be found under [11].

4 Problems of Application

There are many solutions available that provide high reliability. Some of them try to hide the fail-over and take over the IP-addresses of failed servers to another server. This always results in problems with state sharing, because you would need to share the transport layer state if a connection-oriented protocol like TCP or SCTP is used. But this transport layer state changes quite rapidly. In summary, hiding a fail-over introduces much more complexity. A comparison and discussion of other techniques can be found in [7].

The RSerPool protocol suite also allows enhancing existing protocols and building new ones. To enhance existing ones, it is important that a node using the enhanced protocol can still interoperate with nodes

only implementing the base protocol. This requires, that the packet format of the base protocol on the wire does not change.

For SCTP-like services, the only point is that all base protocols are specified in a way that all messages with PPIDs not matching the base protocol are silently discarded. This means, that if a node receives an ASAP message it does not understand, it simply discards it. For UDP- and TCP-like services this means that a separate SCTP association or TCP connection is used for the control channel or the control channel is not used at all. To make it easy to port an application which does not use the control channel to the RSerPool suite, we decided to mimic the DNS calls. This allows for very easy transition.

Now the application can make use of the RSerPool based name resolution service. On the one hand, this results in faster lookups and the client only gets transport layer addresses of servers which are in service. On the other hand, pool handles are more flexible than DNS names. This is a result of the fact that pool handles are byte vectors. However, it is out of scope of the RSerPool working group to define how the pool handles are derived at the PU and/or PE side.

5 Usage Example

Using the RSerPool protocol suite to privide a highly reliable service, multiple servers providing this service are running in the network and SCTP should be the transport protocol for handling network failures. The servers providing the same service register for the same pool handle. Possibly, some of them are able to do state sharing, some are not. This only has to be known by the servers themselves.

If a client wants to use a specific service, it has to know the pool handle of the pool. This pool handle can be a result of administration in the network or even a result of a computation. Of course, the PEs and the PUs then have to use the same algorithm.

After using one NS for name resolution, the PU connects to the PE. The PE can now send a last will, informing the PU that the PE does state sharing with some other PEs and the PU should fail-over to them in case of an error. If the PU is also a PE, it can inform the PE about this by sending its pool handle as part of a business card. It can also send a last will, too. This allows for symmetric communication between pools.

The handling of the control channel can mainly be realized by the RSerPool implementation, with some triggers by the upper layer.

With RSerPool the transport layer state and the security state (if you use TLS or IPSec) are not shared. When a PE fails, a new transport layer connection is used with a new security relation. To shorten the fail-over time, these connections and security relations can be set up in advance.

It should be noted that the NSs are autonomous systems which normally do not need to be configured when they are running. They might need some configuration before they are started, but then they get all the information through ENRP. This is similar to what is needed for routers. Multiple NSs must be used in an operational scope to avoid single points of failure.

In the basic RSerPool architecture, the PEs are providing the service and are not addressed by IP addresses anymore, but by pool handles which are most of the time resolved to transport addresses of servers being up and running. Because the name registration and deregistration is dynamic and very fast, RSerPool can also be used for server addressing in environments where transport layer mobility is used to support mobility. See [8] for more information.

6 Distributed Computing

The basic idea behind RSerPool has been high availability. But the simple and flexible RSerPool architecture can also be used for another interesting application: distributed computing. To explain such an application scenario, let us first have a look at a well known distributed computing application: *SETI@home* [3]. The goal of this project is to find extraterrestrial life by analysing radio data received from a radio telescope. To efficiently analyse the gigantic amount of data, computer users can contribute processing power by installing a client software on their computers. This software downloads small fractions of the data from the *SETI@home* server, processes the calculations on the data when the system is otherwise

idle, and finally uploads the result back to the server. Having a large number of clients results in sufficient computation power to process the radio data. But when the server is down, the clients' trial to provide their computation service to the project fails.

The RSerPool way of realizing distributed computing like *SETI@home* is somewhat different. Here, the distributed computation clients do not request work. Instead, they register at a name server as pool elements of a certain pool to advertise their computation service. An appropriate pool policy here would be e.g. weighted round robin, where the weight is a composite metric of the pool element's computation power and current system load. Now, when somebody requests distributed computation power, he becomes a pool user of this pool and distributes workload between the pool's elements according to the pool's policy.

The advantage of the RSerPool approach is that the computation clients do not have to contact a central instance and ask for work. Instead, they simply advertise their computation power and are contacted when somebody has work for them to do. Note, that authentication by public key or certificate may be used to verify the identity of the pool user. For example, a user may want to provide computation power for something like *SETI@home*, but not for nuclear weapon simulations.

7 Summany and Conclusions

Our paper has coverd aspects of designing and implementing highly available applications using RSerPool with our RSPLIB implementation. First, we have given an introduction to the RSerPool protocol suite and an overview of the RSPLIB components. Then, we have described the RSPLIB API, especially focussing on the implementation of pool element and pool user programs to provide high reliability. Furthermore, we have shown our current implementation status and future plans. This has been followed by a short look on the problems that can arise when the RSerPool architecture is used. Finally, we have given an example how RSerPool can be used to realize highly available services. And last but not least, we have shown how distributed computing architectures can make use of the RSerPool architecture.

For further information about RSerPool, news about our implementation, download, a mailing list for discussions and our mailing list archive, have a look at our website `http://tdrwww.exp-math.uni-essen.de/dreibholz/rserpool/`.

References

[1] IETF Signaling Transport (SIGTRAN) WG
`http://www.ietf.org/html.charters/sigtran-charter.html`

[2] Valgrind memory debugger
`http://developer.kde.org/~sewardj/`

[3] SETI@home: Search for Extraterrestrial Intelligence at home
`http://setiathome.ssl.berkeley.edu/`

[4] draft-ietf-rserpool-asap-05.txt
`http://www.ietf.org/internet-drafts/draft-ietf-rserpool-asap-05.txt`

[5] draft-ietf-rserpool-enrp-04.txt
`http://www.ietf.org/internet-drafts/draft-ietf-rserpool-enrp-04.txt`

[6] draft-conrad-rserpool-tcpmapping-01.txt
`http://www.ietf.org/internet-drafts/draft-conrad-rserpool-tcpmapping-01.txt`

[7] draft-ietf-rserpool-comp-05.txt
`http://www.ietf.org/internet-drafts/draft-ietf-rserpool-comp-05.txt`

[8] draft-riegel-tuexen-mobile-sctp-01.txt
`http://www.ietf.org/internet-drafts/draft-riegel-tuexen-mobile-sctp-01.txt`

[9] Thomas Dreibholz
An Efficient Approach for State Sharing in Server Pools
IEEE Local Computer Networks Conference 2002, Tampa/Florida, U.S.A.
`http://www.exp-math.uni-essen.de/~dreibh/publications/StateSharing-Paper-ShortVersion.ps.gz`

[10] RSerPool and a prototype implementation
`http://www.sctp.de/rserpool.html`

[11] Thomas Dreibholz's Reliable Server Pooling Page
`http://tdrwww.exp-math.uni-essen.de/dreibholz/rserpool/`

[12] IETF Reliable Server Pooling (RSerPool) WG
`http://www.ietf.org/html.charters/rserpool-charter.html`

[13] SCTP and a prototype implementation
`http://www.sctp.de/sctp.html`

[14] draft-ietf-tsvwg-sctpsocket-05.txt
`http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-sctpsocket-05.txt`

[15] Linux Kernel SCTP
`http://sourceforge.net/projects/lksctp/`

[16] SCTP on KAME
`http://www.kame.net/`