Version 1.0

Server-Redundanz und Lastverteilung einfach in eigene Anwendungen integrieren – mit Reliable Server Pooling und RSPLIB

Thomas Dreibholz, dreibh@iem.uni-due.de

Dieses Dokument steht unter der UVM-Lizenz für die freie Nutzung unveränderter Inhalte.

Kurzfassung:

RSPLIB ist die Open-Source-Implementierung von Reliable Server Pooling (RSerPool), dem noch sehr neuen IETF-Standard für Server-Redundanz und Sitzungsverwaltung. Dieser Artikel soll insbesondere zeigen, wie RSerPool mit RSPLIB in eigenen Anwendungen für Hochverfügbarkeit und Lastverteilung eingesetzt werden kann. Im Rahmen des Vortrags soll zudem eine Beispielanwendung in der Praxis demonstriert werden.

Inhaltsverzeichnis

Einführung....2

Reliable Server Pooling....3

Grundaufbau und Komponenten....3

Poolverwaltung....4

Poolnutzung....5

Auswahl von Pool-Elementen....5

Sitzungsverwaltung....5

Protokoll-Stack....7

Automatische Konfiguration....7

Die RSPLIB-Implementierung....8

Überblick....8

Registrar – Der Pool Registrar....9

Die librsplib-Library....9

Die libcpprspserver-Library....13

Installation....13

Ein einfaches Beispiel:
Verteilte Fraktalgrafikberechnung....14

Monitoring....14

Registrars....15

Pool-Element....15

Pool-User....16

Überblick zur Implementierung....17

Der „Scripting Service“:
Lastverteilung per Shellskript....17

Ablauf der Dienstnutzung....17

Konfiguration des Pools....18

Nutzung des Pools am Beispiel POV-Ray....18

Überblick zur Implementierung....20

Zusammenfassung....20

Literaturverzeichnis....21

Einführung

Mit der immer größer werdenden Komplexität von Anwendungen steigt auch der Wunsch, Arbeitsaufgaben in Serverpools zu verteilen. Im Rahmen von Grid-Computing gibt es dafür sehr leistungsfähige und flexible – aber daher auch sehr aufwendig zu installierende und konfigurierende – Softwarelösungen. Die steigende Verbreitung von verfügbarkeitskritischen Diensten in Netzwerken – insbesondere im Internet – führt ebenfalls dazu, dass die gewünschte Verfügbarkeit mittels Komponentenredundanz erreicht werden muss.

Da es keinen Standard für die Verwaltung von Server-Pools sowie von Dienstsitzungen mit solchen Pools gab, musste jede neue Anwendung im Prinzip das Rad neu erfinden, das heißt geeignete Mechanismen selbst implementieren. Die IETF hatte es sich daher zur Aufgabe gesetzt, im Rahmen der Arbeitsgruppe Reliable Server Pooling (RSerPool) einen entsprechenden Standard zu definieren. Die um das Jahr 2000 begonnenen Arbeiten an RSerPool wurden schließlich mit der Verabschiedung der Hauptdokumente als RFCs [1][2][3][4][5][6][7] im Wesentlichen abgeschlossen. Für einige nützliche Erweiterungen existieren weitere Dokumente in Form von Internet Drafts.

RSPLIB ist die Open-Source-Implementierung von RSerPool unter GPLv3-Lizenz, welche am Institut für Experimentelle Mathematik der Universität Duisburg-Essen entwickelt wurde [14]. Das Ziel dieser Implementierung war insbesondere, RSerPool auf verschiedenen Plattformen testen zu können (momentan: Linux, FreeBSD, MacOS X, Solaris) und so Erfahrungen und Ideen in den Standardisierungsprozess von RSerPool einbringen zu können. RSPLIB ist daher die Referenzimplementierung der IETF RSerPool-Arbeitsgruppe geworden.

In diesem Dokument wird zunächst RSerPool im Allgemeinen vorgestellt. Darauf hin folgt ein Überblick zu RSPLIB. Zum Schluss wird eine Einführung zur Nutzung von RSPLIB für eigene Anwendungen mit zwei ausführlichen Beispielen gegeben.

Reliable Server Pooling

Grundaufbau und Komponenten

Das RSerPool-Rahmenwerk erfüllt die folgenden fünf Grundanforderungen [14]:

1. **Ressourcensparsamkeit:** RSerPool-Komponenten sollen sich auch auf Geräten mit eingeschränkten Ressourcen (insbesondere: CPU-Leistung und Speicherkapazität) realisieren lassen.

2. **Echtzeitfähigkeit:** Bei Ausfall einzelner Komponenten soll innerhalb von Zeitvorgaben wieder ein stabiler Systemzustand erreicht werden (insbesondere: Signalisierung von Notrufen über IP-Netzwerke).

3. **Skalierbarkeit:** Die RSerPool-Poolverwaltung soll auch Pools von ggf. mehreren tausend Elementen ermöglichen. Ein (nahezu) unbegrenztes Wachstum wird allerdings nicht angestrebt, um die Komplexität gering zu halten.

4. **Erweiterbarkeit:** RSerPool soll sich leicht an zukünftige Anwendungsanforderungen anpassen lassen. Dazu gehört insbesondere auch die Möglichkeit, einfach neue Regeln zur Serverauswahl zu implementieren.

5. **Einfachheit:** Der Aufwand, um Server in einen Pool hinzuzufügen bzw. daraus zu entfernen sowie den Pool zu nutzen soll möglichst gering sein. Im Regelfall soll es genügen, die Komponenten ein- bzw. auszuschalten – und sämtliche Konfiguration erfolgt vollautomatisch.

Abbildung 1 stellt den Grundaufbau der von der RSerPool-Arbeitsgruppe in der IETF entwickelten RSerPool-Architektur, welche in [1] definiert wird, dar. In der Terminologie von RSerPool werden Server als Pool-Elemente (PE) bezeichnet, innerhalb ihres Pools besitzen sie eine Kennung – ihren sogenannten PE-Identifier (PE-ID) – in Form einer zufällig gewählten 32-Bit-Nummer. Jeder Pool eines Gültigkeitsbereiches, Operation Scope genannt, ist durch einen eindeutigen Pool-Handle (PH) identifiziert. Ein PH wird durch einen beliebigen Bytevektor dargestellt, im Regelfall wird dies jedoch ein ASCII- oder UTF8-String wie z.B. „ComputationPool“ sein. Die Menge aller Pools eines Operation Scopes wird Handlespace genannt. Der Aufbau des Handlespace ist „flach“,

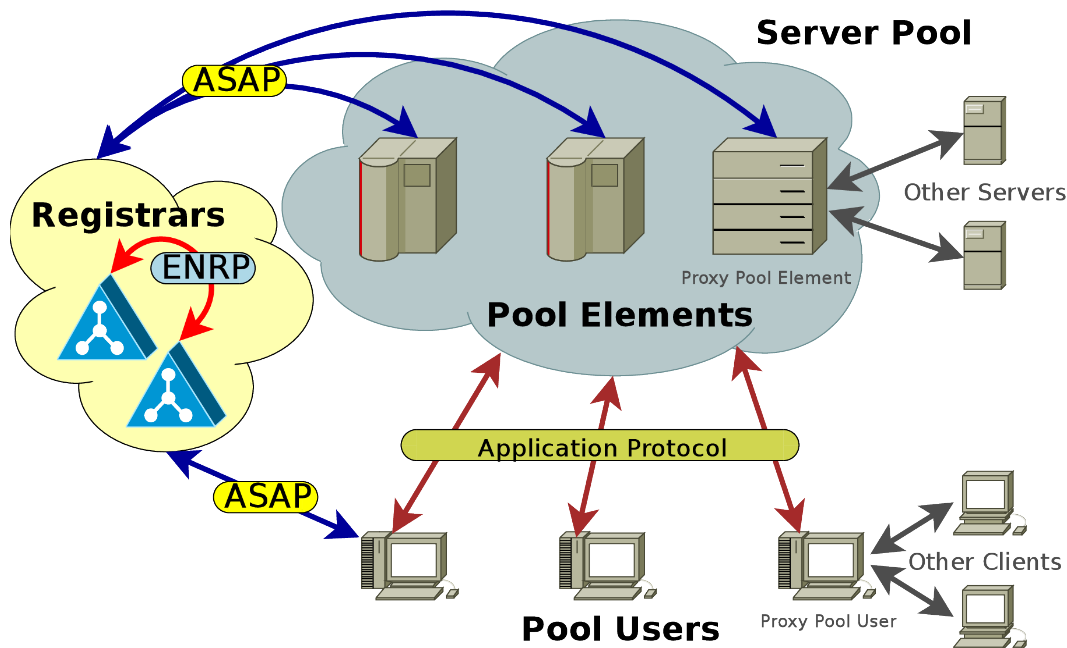

Abbildung 1: Die RSerPool-Architektur

d.h. im Gegensatz zum Domain Name System (DNS) erfolgt keine hierarchische Untergliederung; er besteht einfach nur aus den durch PH identifizierten Pools. Damit wird die Verwaltung erheblich vereinfacht [15].

Poolverwaltung

Die Verwaltung des Handlespaces eines Operation Scopes obliegt den Pool Registrars (PR), welche auch kurz als Registrars bezeichnet werden. Damit ein einzelner PR keinen Single Point of Failure darstellt, welcher bei Ausfall das gesamte System unbrauchbar macht, sind die PRs ebenfalls redundant ausgelegt: In jedem Operation Scope sollte es mindestens zwei PRs geben. Die im Operation Scope vorhandenen PRs gleichen ihre Sicht auf den Handlespace mittels des Endpoint haNdlespace Redundancy Protocols (ENRP) [3] ab. Dies bedeutet, dass bei Ausfall eines PRs jeder andere PR des Operation Scopes gleichwertig die Aufgaben eines ausgefallenen übernehmen kann.

Mittels des Aggregate Server Access Protocols (ASAP) [2] kann sich ein PE bei einem beliebigen PR des Operation Scopes in einen Pool registrieren. Der vom PE zur Registrierung gewählte PR wird für das PE zum sogenannten Home-PR (PR-H). Er informiert nicht nur die anderen PRs des Operation Scope über Hinzufügung und Entfernung seines PEs (via ENRP), sondern überwacht dieses auch durch regelmäßige Keep-Alive-Nachrichten über die ASAP-Verbindung zwischen PE und PR. Diese Keep-Alive-Nachrichten müssen vom PE innerhalb einer vorgegebenen Zeitspanne beantwortet werden. Bleibt eine Antwort aus, wird dies als Ausfall interpretiert und das PE sofort aus dem Handlespace entfernt. Zudem muss sich ein PE regelmäßig re-registrieren. Bei einer Reregistrierung hat es zudem die Möglichkeit, Registrierungsinformationen wie die Liste der Transportadressen über die es erreichbar ist oder Policyinformationen (dies wird später erklärt werden) zu ändern. Durch die Überwachung der PEs durch ihren jeweiligen PR-H wird erreicht, dass die im Handlespace verzeichneten PEs mit sehr hoher Wahrscheinlichkeit auch tatsächlich erreichbar sind.

Die Erweiterung „Takeover Suggestion“ für ENRP, welche die automatische Verteilung

von PEs auf PRs realisiert, um somit die PR-H-Funktionalitäten im Operation Scope gleichmäßig auf alle PRs zu verteilen, ist in [9] definiert.

Poolnutzung

In der Terminologie von RSerPool werden die Clients, welche den Dienst eines Pools nutzen, als Pool-User (PU) bezeichnet. Zur Nutzung eines Dienstes fragt ein PU zuerst über das ASAP-Protokoll bei einem beliebigen PR des Operation Scopes um Auflösung des PH in eine Liste von PE-Identitäten des durch den PH gegebenen Pools nach. Diese Prozedur wird Handle Resolution genannt. Der PR wird, vorausgesetzt der gewünschte Pool existiert im Handlespace, nach einer für den Pool festgelegten Auswahlregel – der sogenannten Pool Member Selection Policy oder kurz Pool Policy – eine Liste von PE-Identitäten des Pools auswählen und dem PU zurück liefern.

Auswahl von Pool-Elementen

Zur Auswahl der PEs sind zahlreiche Verfahren bereits im RFC [6] definiert, z.B. die zufällige Auswahl (Random), die Auswahl reihum (Round Robin) oder die PEs mit der augenblicklich geringsten Auslastung (Least Used). Während die ersten beiden Verfahren keine Zusatzinformationen benötigten (nicht-adaptive Verfahren), benötigt Least Used als adaptives Verfahren möglichst aktuelle Zustandsinformationen über die augenblickliche Auslastung der PEs. Diese als Policyinformationen bezeichneten Zustände gibt ein PE bei seiner Registrierung an; eine Aktualisierung ist per Reregistrierung möglich. Je nach Häufigkeit von Änderungen erzeugt dies natürlich entsprechenden Overhead; allerdings kann dabei natürlich auch je nach Diensteigenschaften eine deutlich bessere Lastbalancierung erreicht werden. Eine adaptive Policy für weit verteilte Szenarien mit hoher Latenz wird in [10] definiert.

Nachdem der PU nun eine Liste von durch den PR ausgewählten PE-Identitäten erhalten hat, schreibt er diese in seinen lokalen Auswahlcache. Aus diesem wählt er dann – wiederum anhand der Pool Policy – genau ein PE aus, um zu diesem dann eine Verbindung aufzubauen und den Dienst des PEs mit dem Applikationsprotokoll zu nutzen. Ist eine erneute Auswahl notwendig – z.B. weil das gewählte PE nicht erreichbar war oder ausgefallen ist – kann für eine erneute Auswahl ggf. der lokale Cache genutzt werden. Dies erspart die Kommunikation mit einem PR. Damit der PR nicht unnötig Zeit damit verbringt, PE-Identitäten auszuwählen, welche der PU ohnehin nicht nutzt (z.B. wenn kein Cache verwendet wird), ermöglicht die „Handle Resolution Option“-Erweiterung [8] für ASAP dem PU die genaue Angabe der gewünschten Anzahl von PE-Identitäten.

Sitzungsverwaltung

Fällt ein PE während der Dienstnutzung aus, so muss nach Verbindungsaufbau zu einem neu ausgewählten PE der Sitzungszustand wiederhergestellt werden, um die Dienstnutzung fortzusetzen. Die Wiederherstellung einer unterbrochenen Sitzung – als Failoverprozedur bezeichnet – ist natürlich sehr stark von der jeweiligen Anwendung abhängig und daher nicht Teil von RSerPool. Allerdings bietet RSerPool eine umfangreiche Unterstützungsmaßnahme für eine solche Sitzungswiederaufnahme an:

den ASAP-Session-Layer zwischen Transport- und Applikationsschicht. ASAP stellt somit das erste Protokoll auf Schicht 5 des OSI-Modells (Session Layer) dar, welches von der IETF standardisiert wurde!

Mittels des ASAP-Session-Layers baut die Applikationsschicht eines PUs eine logische Verbindung – die Session (deutsch: Sitzung) – zwischen sich selbst und einem Pool auf. Der Session-Layer kümmert sich transparent für die Applikationsschicht um PE-Auswahl, Aufbau einer Verbindung, Überwachung der Verbindung, Neuauswahl eines PEs im Fehlerfall sowie um das Anstoßen einer anwendungsspezifischen Failoverprozedur zur Sitzungswiederherstellung auf einem neuen PE. Über die vom Session-Layer aufgebaute Transportverbindung zwischen PU und PE werden dabei sowohl das Applikationsprotokoll – als Data Channel bezeichnet - als auch eine ASAP-Kontrollverbindung – als Control Channel bezeichnet – gemultiplext.

Die Failoverprozedur ist wie bereits erwähnt anwendungsspezifisch und daher nicht Teil von RSerPool selbst. Bei einem Failover kann daher ggf. die Applikationsschicht informiert werden, um eine eigene Prozedur zum Failover auszuführen. RSerPool bietet jedoch über den Control Channel ein Unterstützungsverfahren für den Failover, das in vielen Fällen schon ausreicht um einen Failover ohne direkte Unterstützung durch die PU-seitige Applikation selbst durchzuführen: Client-basiertes State-Sharing [16].

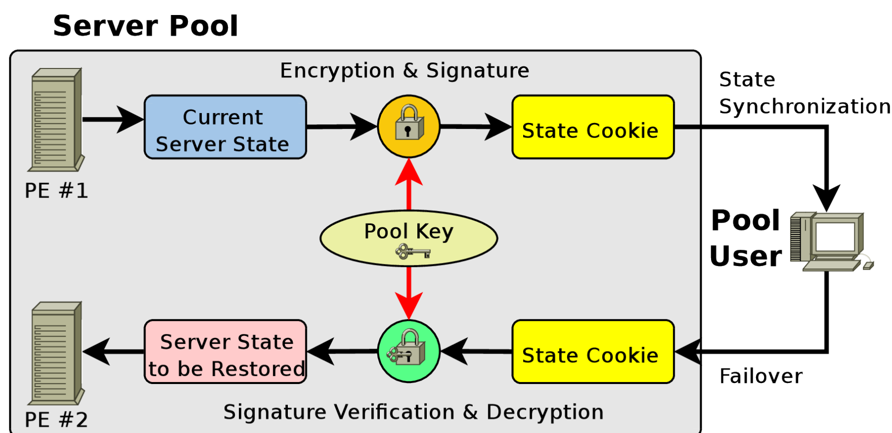

Abbildung 2: Client-basiertes State-Sharing

Die Grundidee dieses Verfahrens ist in Abbildung 2 dargestellt: Der Server verpackt die vollständigen Informationen über seinen aktuellen Sitzungszustand in ein sogenanntes State-Cookie und schickt dieses an den Client. Der Client sichert nun jeweils das zuletzt empfangene State-Cookie. Fällt der Server aus, so wird das gespeicherte State-Cookie an den neuen Server verschickt. Dieser kann nun anhand der enthaltenen Informationen den Sitzungszustand wiederherstellen. Damit der Client das State-Cookie nicht manipulieren kann, wird es vom Server im einfachsten Fall einfach mit einem allen Servern des Pools bekannten Schlüssel signiert. Der empfangende Server kann diese Signatur überprüfen und ggf. die Sitzungswiederherstellung verweigern. Zudem können Informationen im State-Cookie verschlüsselt werden, um ein Auslesen durch den Client zu verhindern.

Da das Verfahren des Client-basierten State-Sharings sehr einfach und für viele Fälle anwendbar ist, wurde es in ASAP integriert. Über den Control Channel kann ein PE Cookie-Nachrichten an den PU schicken; der Session-Layer auf PU-Seite speichert das jeweils letzte State-Cookie und schickt es bei einem Failover in Form einer Cookie-Echo-Nachricht zum neuen PE. Damit ist der Failover auf PU-Seite vollkommen transparent für die Applikationsschicht. Auf der PE-Seite muss die Applikationsschicht lediglich das

State-Cookie auslesen und den Sitzungszustand wiederherstellen.

Protokoll-Stack

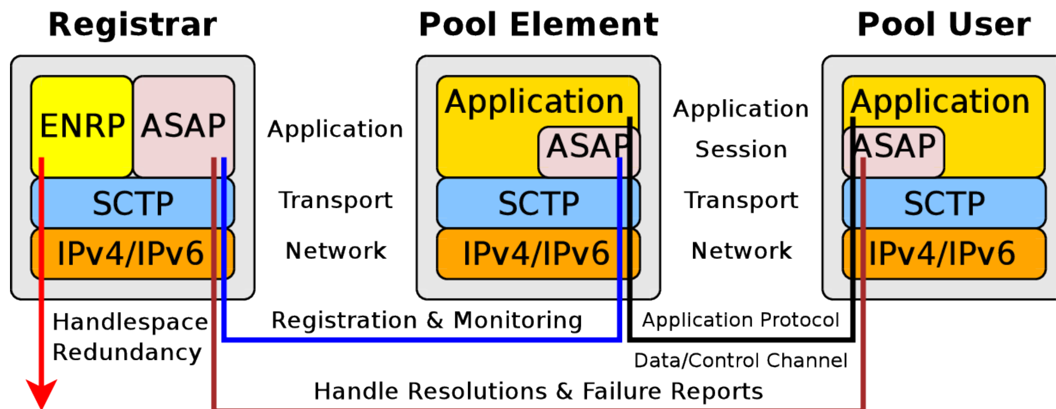

Abbildung 3: Der RSerPool-Protokollstack

Zur Veranschaulichung stellt Abbildung 3 den Protokollstack von RSerPool für PR, PE und PU dar. Aus Sicht des PR sind ASAP und ENRP Applikationsprotokolle: Über ASAP wird für PEs die Poolverwaltungsfunktionalität angeboten, für PUs die Handle Resolution; über ENRP gleichen die PRs ihre Sicht auf den Handlespace ab. Für PE und PU stellt ASAP den Session-Layer dar.

Grundlegendes Transportprotokoll für ASAP und ENRP ist das Stream Control Transmission Protocol (SCTP) [18][19], aufgrund seiner hohen Toleranz gegenüber Netzwerkausfällen und insbesondere auch der Möglichkeit zum Multihoming.

Automatische Konfiguration

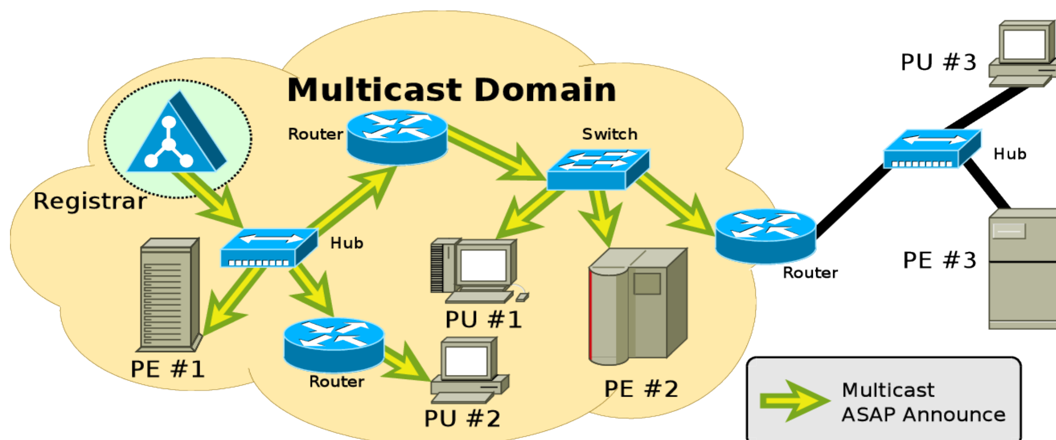

Abbildung 4: Automatische Konfiguration durch Announces

Damit ein PE oder PU die Funktionalitäten von RSerPool nutzen kann, muss im Wesentlichen nur mindestens ein PR des Operation Scope bekannt sein. Möglich ist natürlich, eine Liste von PRs statisch vorzukonfigurieren. Dies macht jedoch eine Neukonfiguration notwendig, sobald Änderungen an der Anzahl oder den Adressen von PRs gemacht werden.

Um den Aufwand einer manuellen Neukonfiguration von PR-Adressen zu vermeiden, besteht die Möglichkeit dass PRs in regelmäßigen Intervallen über ASAP via UDP mittels IP-Multicast sogenannte Announces verschicken[1]. Diese enthalten einfach die

1 Abbildung 3 stellt die UDP-basierten Announces aus Platzgründen nicht dar.

Information unter welchen Adressen der jeweilige PR erreichbar ist. Durch Horchen auf einer festgelegten Multicastadresse können PUs und PEs nun einfach die Liste aktuell verfügbarer PRs lernen. Die Verwendung von Multicast gegenüber Broadcast hat zudem den Vorteil, dass in einem geswitchten Ethernet nur die Stationen, die auch wirklich an den Announces interessiert sind, diese auch erhalten und zudem funktioniert das Verfahren auch über Routergrenzen hinweg.

Abbildung 4 verdeutlicht noch einmal das Prinzip der Announces: Alle in der Multicastdomäne vorhandenen PUs und PEs erfahren über Announces automatisch vom Vorhandensein des PRs; PU #3 und PE #3 befinden sich allerdings außerhalb (z.B. Rechner im Internet), daher müssen diese Elemente statisch konfiguriert werden. Damit PRs eines Operation Scope sich gegenseitig finden, wird hier das gleiche Prinzip – Bekanntmachungen via IP-Multicast – ebenfalls für ENRP verwendet.

Die RSPLIB-Implementierung

RSPLIB ist eine Open-Source-Implementierung von RSerPool unter GPLv3-Lizenz. Sie wurde an der Universität Duisburg-Essen im Rahmen einer Dissertation [14] entwickelt und steht – neben umfangreichem weiteren Material zu RSerPool – unter [20] zum Download zur Verfügung.

Überblick

Die zur Zeit aktuelle, stabile Version von RSPLIB ist 2.7.11. Sie besteht aus folgenden Teilen:

1. dem PR *Registrar,*
2. der Library *librsplib* zur Erstellung von PUs und PEs mit Unterstützung für Basic-Mode-API und Enhanced-Mode-API (Erklärung folgt unten),
3. der Library *libcpprspserver* als C++-Wrapper um *librsplib*, um noch einfacher PEs in C++ zu realisieren sowie
4. einer Reihe von Test- und Demoprogrammen.

Registrar – Der Pool Registrar

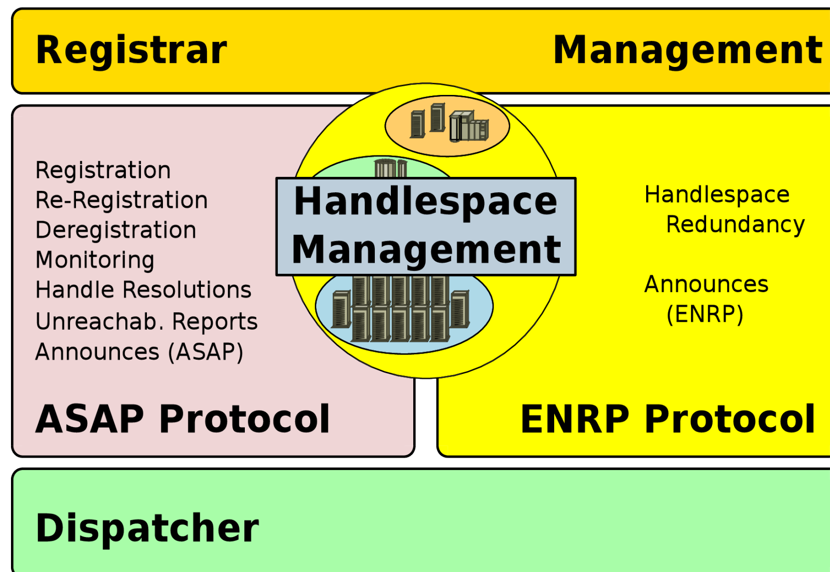

Abbildung 5: Der Aufbau des Registrars

Der grundsätzliche Aufbau des PRs ist in Abbildung 5 dargestellt. Wesentlicher Bestandteil ist natürlich die Implementierung der Protokolle ENRP [3] und ASAP [2]. Des weiteren werden noch die „Takeover Suggestion“-Erweiterung [9] für ENRP und die „Handle Resolution Option“-Erweiterung [8] für ASAP unterstützt. Zudem werden alle Pool Policys aus [6] sowie [10] implementiert.

Der in der Abbildung als Dispatcher dargestellte Block stellt die systemabhängige Kapselung von Timer-, Thread- und Socketfunktionalitäten dar. Da ein Ziel von RSPLIB die leichte Portierbarkeit ist, wurden daher sämtliche betriebssystemabhängigen Teile im Dispatcher gekapselt. Für eine Portierung auf ein neues System müssen daher nur dessen Bestandteile entsprechend modifiziert werden müssen.

Die *librsplib*-Library

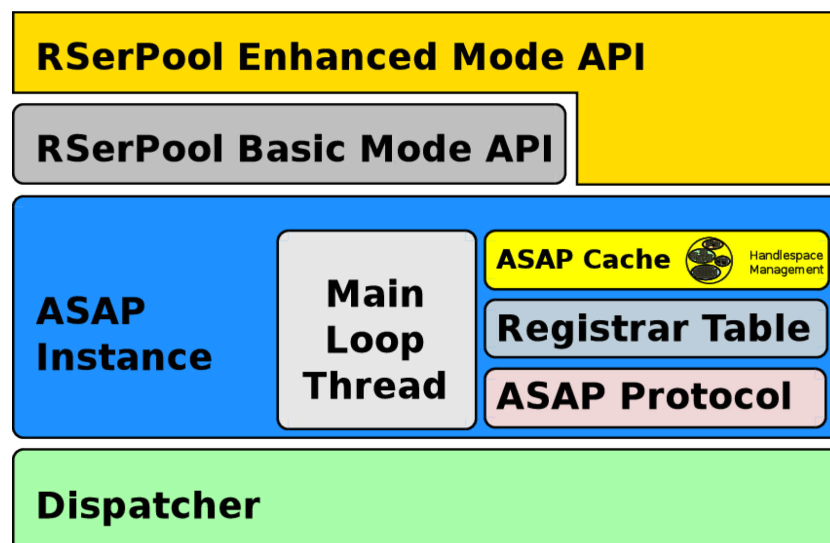

Abbildung 6: Der Aufbau der librsplib-Library

Die *librsplib*-Library stellt für Server und Clients die Schnittstelle zu RSerPool dar, d.h. sie enthält die Funktionen zur Implementierung von PEs und PUs. Im Wesentlichen besteht diese Library aus einer auf dem Dispatcher zur Kapselung plattformabhängigen Codes aufbauenden Implementierung einer ASAP-Instanz. Deren Aufbau wird in Abbildung 6 dargestellt. Die ASAP-Instanz besteht aus drei Blöcken:

ASAP Protocol: Dies ist die Implementierung des ASAP-Protokolls [2] einschließlich der „Handle Resolution Option“-Erweiterung [8].

Registrar Table: In der Registrar Table wird eine aktuelle Liste von verfügbaren PRs vorgehalten. Die Einträge dieser Liste werden dabei dynamisch durch Multicast-Announces der entsprechenden PRs gelernt (automatische Konfiguration) und/oder statisch vorgegeben.

ASAP Cache: Dieser Cache sorgt beim PU für eine Zwischenspeicherung der vom PR nach Handle Resolution erhaltenen PE-Liste. Vom Prinzip her ist der Cache ein Ausschnitt des kompletten Handlespaces; die Algorithmen und Datenstrukturen zu dessen Verwaltung [15] lassen sich hier einfach wiederverwenden.

Aufbauend auf der ASAP-Instanz ist das eigentliche API der *librsplib*-Library realisiert. Dieses API besteht aus zwei wesentlichen Teilen: dem Basic-Mode-API und dem darauf aufbauenden Enhanced-Mode-API. Mit dem Basic Mode stellt die *librsplib*-Library nur die wesentlichen ASAP-Funktionen zur Poolverwaltung und -nutzung zur Verfügung. Für PEs wären das die Registrierung, Reregistrierung und Deregistrierung; für PUs gibt es die Möglichkeit zur Handle Resolution. Die komplette Verwaltung des Data Channels obliegt dabei dann der Anwendung selbst.

Erst mit Verwendung des Enhanced-Mode-APIs wird RSerPool zu einem wirklichen Session-Layer; in diesem Fall übernimmt die *librsplib*-Library PE-Auswahl, Verbindungsaufbau, Multiplexing von Control und Data Channel, Ausfallerkennung und Failoverunterstützung. Für die Implementierung des Enhanced-Mode-APIs wird dabei im Wesentlichen das Basic-Mode-API verwendet um den Session-Layer zu realisieren. Im Folgenden wird das Enhanced-Mode-API kurz mittels Pseudocode-Beispielen beschrieben.

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

// Run application: request radio stream
rsp_send(session, "GET /streams/MyMovie.ogg HTTP/1.0\r\n\r\n");
while((length = rsp_recv(session, buffer, ...)) > 0) {
   doSomething(buffer, length, ...);
}

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

Listing 1: Pseudocode-Beispiel für einen Pool User mit dem Enhanced-Mode-API

Grundidee des Enhanced-Mode-APIs ist es, Sessions ähnlich wie SCTP- bzw. TCP-Sockets zu realisieren, so dass die Portierung bestehender Anwendungen so einfach wie möglich

ist. Ein Beispiel für die PU-Programmierung ist dazu in Listing 1 dargestellt: Der PU erstellt zunächst eine Sitzung (*rsp_socket()*-Aufruf), welche auch als RSerPool-Socket bezeichnet wird. Danach stellt er eine logische Verbindung mit einem durch PH gegebenen Pool her (*rsp_connect()*-Aufruf mit PH "VideoStreamingPool"). Die eigentliche Verbindungsaufnahme mit einem PE des Pools erfolgt dabei für die Applikation vollkommen transparent im Session-Layer. In der Applikation selbst kann nun auf der logischen Verbindung das eigentliche Applikationsprotokoll genutzt werden. Im Beispiel ist dies HTTP; es wird die Datei "/streams/MyMovie.ogg" per GET-Befehl angefordert und anschließend übertragen. Der ASAP Session-Layer sorgt dabei für die Zuordnung der Kommunikation zu einem PE des Pools.

Einen Failover mit State-Cookie kann der Session-Layer vollautomatisch handhaben. Senden die PEs aus dem Beispiel ihren Zustand (Dateiname und aktuelle Übertragungsposition) laufend per ASAP-Cookie, so kann die Übertragung durch ein neues PE automatisch vom Session-Layer wiederaufgenommen werden. Im Applikationscode des PUs selbst ist dafür nichts[1] weiter zu tun.

1 Es wird hier davon ausgegangen, dass *doSomething()* Teile, die vom alten PE zwischen Cookie und Ausfall gesendet wurden und damit vom neuen PE erneut gesendet werden, als Duplikate einfach ignoriert.

```
// 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, "VideoStreamingPool", ...);

   // 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);
}
```

Listing 2: Pseudocode-Beispiel für ein Pool Element mit dem Enhanced-Mode-API

Die Struktur der Realisierung eines Thread-basierten PEs mit dem Enhanced-Mode-API ist in Listing 2 dargestellt: In der *main()*-Funktion wird zunächst wieder ein RSerPool-Socket erstellt (*rsp_socket()*-Aufruf) und dann damit ein PE mittels *rsp_register()* erzeugt; diese Funktion sorgt auch für die Registrierung. In einem Hintergrundthread der *librsplib*-Library wird danach zudem automatisch für Reregistrierung und Beantwortung der ASAP Keep-Alives gesorgt. Analog zu Sockets wird nach Registrierung des PEs auf eingehende Sessions gewartet (*rsp_poll()*, analog zum *poll()*-Aufruf für Sockets). Wurde eine neue Session aufgebaut, nimmt der Server diese mit *rsp_accept()* (analog zum *accept()*-Aufruf für Sockets) an und erzeugt einen neuen Thread für ihre

Bedienung (in *serviceThread()*).

Erste Aufgabe des Service-Threads ist es festzustellen, ob eine Sitzungswiederherstellung durchgeführt werden muss; in diesem Fall basierend auf dem in RSerPool eingebauten Mechanismus des Client-basierten State-Sharings mit State-Cookies. Wurde ein State-Cookie vom PU geschickt (in Form eines ASAP Cookie-Echo über den Control Channel), wird der in ihm gespeicherte Zustand wiederhergestellt (sofern dieser gültig ist!).

In der „while"-Schleife werden nun die Kommandos des PUs abgearbeitet. Nach Ausführung jedes Kommandos wird ein Cookie mit dem aktuellen Zustand der Sitzung geschickt. Das heißt, die Sitzung könnte dann an dem durch das State-Cookie beschriebenen Punkt fortgesetzt werden.

Neben der Realisierung mittels Threads – was auf Multi-Core-Maschinen sehr praktisch ist – lässt sich ein PE natürlich auch mit einer einzelnen *rsp_poll()*-Schleife realisieren, welche alle aktuell vorhandenen Sitzungen bedient. Da das Vorgehen – bei Kenntnis von Socket-Programmierung mit TCP oder SCTP – relativ offensichtlich ist, wurde hier auf ein Beispiel verzichtet.

Die *libcpprspserver*-Library

Die *libcpprspserver*-Library stellt zwei C++-Klassen zum Erstellen von PEs zur Verfügung: *UDPLikeServer* und *TCPLikeServer*. Mittels *UDPLikeServer* lassen sich Dienste realisieren, welche einzelne Nachrichten – von beliebigen Sessions – entgegen nehmen und darauf hin eine entsprechende Antwort generieren. Das Verhalten ist also sehr ähnlich zu einem UDP-Server, daher der Name *UDPLikeServer*. Im Gegensatz dazu lassen sich mittels *TCPLikeServer* Dienste realisieren, welche für jede Session einen eigenen Thread zu dessen Bedienung erstellen (also wie im Beispiel zu *librsplib* oben). *TCPLikeServer* kümmert sich dann – neben der eigentlichen PE-Verwaltung – auch um das Erzeugen und geordnete Beenden dieser Threads.

Installation

RSPLIB lässt sich auf den unterstützten Plattformen – also momentan Linux, FreeBSD, MacOS X und Solaris – einfach aus den Sources von der RSPLIB-Homepage [20] installieren:

```
./configure --enable-qt
make
make install
```

Die Option --enable-qt sorgt dafür, dass auch das Qt-basierte Beispiel zur verteilten Fraktalgrafikberechnung, welches im folgenden Abschnitt kurz beschrieben wird, übersetzt wird. Ist Qt auf dem System nicht vorhanden, so kann diese Option einfach weggelassen werden. Die Fraktal-Beispielanwendung wird dann natürlich nicht erstellt.

Unter Ubuntu Linux 11.04 („Natty Narwhal") steht die aktuelle Version von RSPLIB als Paket im Standard-Repository zur Verfügung. Hier genügt zur Installation von Registrar,

Tools, Beispielen, Entwicklerdateien und Dokumentation folgender Befehl:

```
sudo apt-get install \
   rsplib-registrar rsplib-tools rsplib-services \
   librsplib-dev rsplib-docs
```

Der Registrar wird als Dienst „rsplib-registrar“ installiert. Bei Installation auf vielen Rechnern sollten zwischen 2 und max. 5 Rechner ausgewählt werden, auf denen der Registrar-Dienst laufen soll. Auf allen anderen Rechnern sollte das Paket „rsplib-registrar“ weggelassen werden. Der ENRP-Overhead steigt mit der Anzahl der PRs; mehr als maximal 5 PRs bringen kaum noch verbesserte Ausfallsicherheit, benötigen jedoch zunehmend Bandbreite und Rechenzeit.

[20] beschreibt zudem, wie das RSPLIB-PPA für ältere Ubuntu-Distributionen eingebunden werden kann, um auch darunter die aktuellsten RSPLIB-Pakete direkt zu installieren.

Für FreeBSD steht die aktuelle RSPLIB-Version als Port zur Verfügung. Zur Installation reichen folgende Befehle aus:

```
cd /usr/ports/net/rsplib
make
make install
```

Ein einfaches Beispiel: Verteilte Fraktalgrafikberechnung

Im Folgenden wird nun beschrieben, wie sich ein einfaches Beispielszenario mit RSPLIB aufbauen lässt. Dazu wird ein Dienst verwendet, welcher die Berechnung von Mandelbrot'schen Fraktalgrafiken anbietet. Neben einem entsprechenden PE enthält das RSPLIB-Paket ebenso einen PU mit graphischer Benutzeroberfläche. Das Beispielszenario kann mit einem oder mehreren Rechnern aufgebaut werden.

Alle im Folgenden beschriebenen Programme besitzen Manual Pages, welche weitere Informationen zu den möglichen Kommandozeilenparametern enthalten!

Monitoring

Wird ein RSerPool-Szenario mit mehreren Rechnern aufgebaut – was in der Praxis natürlich der Normalfall ist – so wird der Aufbau relativ schnell unübersichtlich: wo läuft was, wie ist der aktuelle Zustand der einzelnen Komponenten? RSPLIB bietet mit dem sogenannten Component Status Protocol (CSP) eine sehr praktische Möglichkeit, die einzelnen Komponenten regelmäßig Statusmeldungen an eine Monitoringstation senden zu lassen. Auf der Monitoringstation muss dazu einfach das Programm *cspmonitor* gestartet werden:

```
cspmonitor
```

Bei einer großen Anzahl von Komponenten bietet es sich an, den „Kompakt-Modus“ mit

reduzierter und hierdurch platzsparenderer Darstellung zu verwenden:

```
cspmonitor -compact
```

Statusinformationen werden ausgegeben, sobald diese – per UDP auf Port 2960 – empfangen wurden. Da momentan noch keine RSPLIB-Komponenten laufen gibt es noch nichts interessantes zu sehen.

Im Folgenden empfiehlt es sich, auf allen am Szenario beteiligten Rechnern zwei Umgebungsvariablen zu setzen, um die Statusausgabe an den gewünschten *cspmonitor* zu schicken:

```
CSP_SERVER="<Adresse>:2960"
CSP_INTERVAL="333“
```

CSP_SERVER gibt Adresse und UDP-Portnummer (Default: 2960) des *cspmonitor* an. Hier ist natürlich die passende Adresse (IPv4, IPv6 oder Hostname) einzusetzen. CSP_INTERVAL ist das Sendeintervall in Millisekunden. Für die Demoanwendung ist ein kleines Intervall (hier: 333ms) sinnvoll; in der Praxis lässt sich der Overhead durch ein größeres Intervall (z.B. 5000ms) reduzieren.

Registrars

Da jedes RSerPool-Szenario mindestens einen PR benötigt, muss ein solcher natürlich gestartet werden. Im Normalfall – mit mehreren Rechnern – genügt folgender Aufruf:

```
registrar
```

Läuft das komplette Szenario zum Test auf dem gleichen Rechner, so gibt es eine wichtige Bedingung: der Rechner muss zumindest über eine IP-Adresse mit Site-Local-Scope verfügen, also z.B. eine private IPv4-Adresse. Zudem muss auf einem Netzwerkinterface das Multicast-Flag gesetzt sein (für die Announces). Ist das nicht der Fall, kann beides über das Dummy-Interface erreicht werden, z.B.:

```
ip link set dummy0 up multicast on
ip addr add 10.255.255.1/24 dev dummy0
```

Nach Start des PR sollte die Ausgabe am *cspmonitor* den neuen PR anzeigen. Werden weitere PRs gestartet, so werden diese ebenfalls angezeigt. Zudem sind die ENRP-Verbindungen zwischen den PRs – sofern *cspmonitor* nicht im Kompakt-Modus läuft – zu sehen.

Pool-Element

Jetzt können PEs des Fraktalberechnungsdienstes gestartet werden. Verschiedene Dienste sind in das gleiche Programm – *server* – integriert. Sie werden per Parameter ausgewählt:

```
server -fractal
```

Die Ausgabe von *cspmonitor* sollte nun auch die PE(s) anzeigen.

Für Tests gibt es einen weiteren interessanten Parameter: Mit -fgpfailureafter lässt sich angeben, dass das PE eine Verbindung nach der gegebenen Anzahl von Datenpaketen an den PU trennt. Damit lässt sich ein Verbindungsausfall simulieren und der PU wird zu einem Failover veranlasst. Für einen Abbruch nach 20 Paketen wäre der Aufruf also folgender:

```
server -fractal -fgpfailureafter=20
```

Pool-User

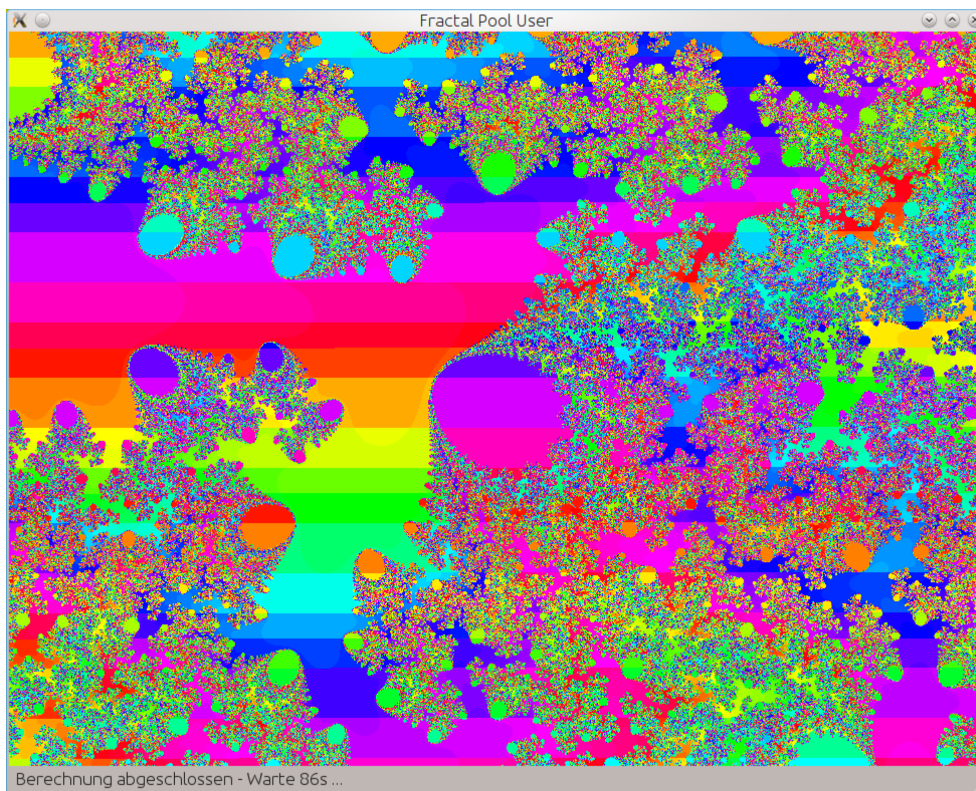

Abbildung 7: Ausgabefenster des Fraktal-Pool-Users

Das RSerPool-Szenario ist nun bereit zum Starten des Fraktal-PUs:

```
fractalpooluser
```

Der Fraktal-PU fragt nun zufällige Bildberechnungen nach. Abbildung 7 zeigt eine Beispielausgabe, mit aktivierter Anzeige von Failovern (Erklärung folgt unten).

Die Eingabeparameter befinden sich in XML-Dateien im Verzeichnis */usr/local/share/fgpconfig* oder */usr/share/fgpconfig* (je nach Installation). Wird der Fraktal-PU aus dem Source-Verzeichnis direkt – also ohne Installation – gestartet, kann das Verzeichnis (*fgpconfig* unterhalb des Sourceverzeichnisses *rsplib*) mit den Eingabedateien als Parameter angegeben werden:

```
./fractalpooluser -configdir=fgpconfig
```

Die Angabe des Eingabeparameterverzeichnisses ist optional. Kann der Fraktal-PU keine Eingabedateien laden, so verwendet er immer die eingebauten Parameter für ein

Standardbild. Das reicht natürlich zum Test, die Ausgabe ist dann aber relativ langweilig.

Sobald der Fraktal-PU gestartet ist, lässt sich mit der rechten Maustaste das Kontextmenü aufrufen. Interessant sind hier insbesondere folgende Einstellungen:

Show Failovers bzw. Zeige Failover: Das Setzen der Option führt dazu, dass bei jedem Failover die Farbe etwas verändert wird. Jeder Failover wird so in der Ausgabe leicht erkennbar (siehe das Beispiel in Abbildung 7).

Sessions bzw. Sitzungen: Statt das komplette Bild in einer einzelnen Session berechnen zu lassen, ist es natürlich auch möglich, mehrere Sessions gleichzeitig zu verwenden. Das Gesamtbild wird dann entsprechend in Rechtecke – für jede Session eines – aufgeteilt. Stehen entsprechend viele PEs zur Verfügung, so wird natürlich die Bildberechnung deutlich beschleunigt.

Show Sessions bzw. Zeige Sessions: Ist diese Option gesetzt, so erhält jede Session eine etwas andere Farbe. Die einzelnen Sessions des Bildes werden so in der Ausgabe erkennbar.

Überblick zur Implementierung

Das Fraktal-PE ist in *fractalgeneratorservice.cc* implementiert, wobei der Dienst selbst in *server.cc* initialisiert wird. Es verwendet die Klasse *TCPLikeServer* von *libcpprspserver*, um die Aufrufe zur *librsplib* zu kapseln und für jede zu bedienende Session einen eigenen Thread zu erzeugen. Der jeweilige Thread führt dann natürlich auch die eigentliche Berechnung aus. Failover wird mittels ASAP-Cookies realisiert: Die Bildparameter einschließlich der aktuellen Position werden regelmäßig als Cookies verschickt. Ein solches Cookie kann dann bei einem Failover verwendet werden, um die Bildberechnung an der angegebenen Stelle fortzusetzen.

Der Fraktal-PU ist in *fractalpooluser.cc* implementiert und setzt direkt auf der *librsplib* auf. Die graphische Benutzeroberfläche wird mittels Qt realisiert. Für jede Session wird ein eigener Thread erzeugt (*FractalCalculationThread*), welcher per *librsplib* die Berechnung mittels eines RSerPool-Sockets verteilt.

Der „Scripting Service“: Lastverteilung per Shellskript

Der Scripting Service ist eine weitere Beispielanwendung von RSPLIB. Diese Anwendung ermöglicht Lastverteilung per Shellskript; es ist nicht nur eine weitere Demo sondern kann – wie im Folgenden an einem Beispiel gezeigt wird – auch für nützlichere Dinge verwendet werden.

Ablauf der Dienstnutzung

Der PU des Scripting Services nimmt als Eingabe eine Tar/GZip-gepackte Datei mit

einem Arbeitsauftrag entgegen. Zudem kann – optional – eine Tar/BZip2-gepackte Arbeitsumgebung (Environment) angegeben werden. Der PU erzeugt nun eine Session; ein PE nimmt nun den Arbeitsauftrag entgegen. Sofern das PE noch nicht die angegebene Arbeitsumgebung besitzt, so wird auch diese an das PE übergeben. Mittels eines Caches wird auf dem PE die Arbeitsumgebung gespeichert; weitere Arbeitsaufträge mit der gleichen Arbeitsumgebung (die Unterscheidung erfolgt per SHA1-Hash) benötigen dann keine weitere Übertragung mehr.

Sobald Arbeitsauftrag und -umgebung beim PE vorliegen, wird beides in ein temporäres Verzeichnis entpackt. Danach wird das im Arbeitsauftrag enthaltene Skript *ssrun* aufgerufen, welches die eigentliche Arbeit ausführt. Das Ergebnis dieser Ausführung wird per Tar/GZip archiviert und zurück zum PU geschickt.

Konfiguration des Pools

Mittels des Scripting Services lässt sich nun sehr einfach eine Lastverteilung für eigene Anwendungen in einem Rechnerpool realisieren. Zunächst muss – neben PRs – das Scripting-Service-PE auf einem oder besser noch auf mehreren Rechnern gestartet werden:

```
server -scripting
```

Das PE erkennt automatisch die Anzahl von CPUs und Cores; entsprechend viele Sessions nimmt ein Scripting-Service-PE dann ggf. gleichzeitig an. Optional kann die Anzahl auch manuell angegeben werden, z.B. für maximal 4 gleichzeitige Sessions wäre folgender Aufruf notwendig:

```
server -scripting -ssmaxthreads=4
```

Zur Sicherheit des Dienstes sollte kurz angemerkt werden, dass in der jetzigen Version des Scripting Services keinerlei Authentifizierung der PUs bzw. der Arbeitsaufträge und -umgebungen durchgeführt wird. Der Pool sollte daher in einer sinnvoll abgesicherten Umgebung aufgesetzt werden. Alternativ ließe sich natürlich auch die Authentifizierung nachrüsten, z.B. basierend auf Signaturen mit GnuPG. In unsicheren Umgebungen wäre es zudem sinnvoll, die Arbeitsaufträge in einer virtuellen Maschine – z.B. mittels XEN, QEMU oder ähnlichem – auszuführen. Hier ist Raum für Verbesserungen und Erweiterungen, was natürlich – da RSPLIB Open Source ist – auch einfach möglich ist.

Nutzung des Pools am Beispiel POV-Ray

Nachdem nun der Scripting-Service-Pool aufgesetzt ist, soll dieser natürlich auch eine sinnvolle Arbeit erledigen. Als Anwendung wird der Open-Source-Raytracer „Persistence of Vision“ (POV-Ray) [22] verwendet. Für das Beispiel muss POV-Ray auf allen PE-Rechnern installiert sein (dass heißt insbesondere, dass das Programm *povray* im Standardsuchpfad liegt).

Dem PU des Scripting Services – *scriptingclient* – muss nun einfach ein Arbeitsauftrag als Tar/GZip-Datei übergeben werden, welcher eine zu berechnende Szene (.pov-Datei und ggf. weitere Dateien wie Includes) und ein *ssrun*-Skript enthält. Das in Listing 3

dargestellte Skript *povray-distribute*[1] erledigt dies:

```
#!/bin/bash
# ====== Get arguments ==================================================
if [ $# -lt 3 ] ; then
   echo >&2 "ERROR: Usage $0 [Width] [Height] [Input POV]"
   exit 1
fi
WIDTH=$1
HEIGHT=$2
INPUT=$3
OUTPUT="`echo $INPUT | sed -e "s/.pov/-$WIDTH-$HEIGHT.png"/g`"
if [ -e $OUTPUT ] ; then
   echo >&2 "ERROR: Output file $OUTPUT already exists!"
   exit 1
fi


# ====== Create temporary directory =====================================
TEMPDIR="temp-$INPUT-$WIDTH-$HEIGHT"
umask 077
rm -rf $TEMPDIR
mkdir $TEMPDIR
cp $INPUT $TEMPDIR/input.pov
find -name "*.inc" | xargs --no-run-if-empty -n1 -i§ cp § $TEMPDIR


# ====== Write ssrun script =============================================
(
   echo "#!/bin/sh"
   echo "OUTPUT_ARCHIVE=\$1"
   echo "SUCCESS=1"
   echo -n "povray -w$WIDTH -h$HEIGHT +a0.3 -D +FN8 +Ooutput.png "
   echo    "+Iinput.pov >output.txt 2>&1 || SUCCESS=0"
   echo "tar czvf \$OUTPUT_ARCHIVE output.png output.txt || SUCCESS=0"
   echo "exit $SUCCESS"
) >"$TEMPDIR/ssrun"
chmod +x "$TEMPDIR/ssrun"


# ====== Create and distribute work package =============================
cd "$TEMPDIR"
find . -name "ssrun" -or -name "input.pov" -or -name "*.inc" | \
   xargs tar czf input.tar.gz
cd ..
scriptingclient -quiet -input=$TEMPDIR/input.tar.gz \
                -output=$TEMPDIR/output.tar.gz
if [ -e "$TEMPDIR/output.tar.gz" ] ; then
   cd "$TEMPDIR"
   tar xzf output.tar.gz
   cd ..
   if [ -e "$TEMPDIR/output.png" ] ; then
     mv $TEMPDIR/output.png $OUTPUT
     rm -rf $TEMPDIR
   else
      echo >&2 "ERROR: No image has been created. Check log:"
      echo "------ LOG ----------------------------------------------"
      cat "$TEMPDIR/output.txt"
      echo "---------------------------------------------------------"
   fi
fi
```

1 Download: http://www.tdr.wiwi.uni-due.de/fileadmin/fileupload/I-TDR/ReliableServer/Downloads/povray-distribute.

Listing 3: povray-distribute – Shellskript zum Durchführen einer POV-Ray-Berechnung im Pool

Um z.B. die POV-Ray-Beispielszene *landscape.pov* in der Auflösung 1600x1200 zu berechnen und das Ergebnis als *landscape.png* zu speichern, ist folgender Aufruf notwendig:

```
./povray-distribute 1600 1200 landscape.pov
```

Um nun mehrere Szenen gleichzeitig zu berechnen genügt das das mehrfache Aufrufen des Skripts *povray-distribute* (mit verschiedenen Szenendateien).

Im POV-Ray-Beispiel wurde keine Arbeitsumgebung verwendet. Das Beispiel ließe sich jedoch so erweitern, dass das Programm *povray* inklusive aller benötigten Dateien in eine Arbeitsumgebung gepackt wird. Dann wäre es nicht mehr notwendig, POV-Ray auf allen PE-Rechnern zu installieren. In heterogenen Pools (z.B. verschiedene Linux-Distributionen und Versionen, FreeBSD; i386, amd64) ist dies jedoch nicht so trivial wie es scheint. Da dies den Rahmen dieses Artikels sprengen würde, sei als Beispiel für eine Scripting-Service-Anwendung mit Arbeitsumgebung auf die Simulation Processing Tool-Chain (SimProcTC) [21] verwiesen, welche Simulationsläufe im Pool verteilt.

Überblick zur Implementierung

Das Scripting-Service-PE ist in *scriptingservice.cc* implementiert, wobei der Dienst selbst in *server.cc* initialisiert wird. Es verwendet die Klasse *TCPLikeServer* von *libcpprspserver*, um die Aufrufe zur *librsplib* zu kapseln und für jede zu bedienende Session einen eigenen Thread zu erzeugen. Sobald Arbeitsauftrag und -umgebung vorliegen, wird das Skript *scriptingcontrol* aufgerufen, welches sich um das Entpacken von Arbeitsauftrag und -umgebung, das Aufrufen von *ssrun* (aus dem Arbeitsauftrag) und das Aufräumen nach Beendigung von *ssrun* kümmert. Der Cache für die Arbeitsumgebung ist in *environmentcache.cc* implementiert.

Der Scripting-Service-PU ist in *scriptingclient.c* implementiert und setzt direkt auf der *librsplib* auf.

Zusammenfassung

In diesem Artikel wurden Reliable Server Pooling (RSerPool) – der noch sehr neue IETF-Standard für Server-Redundanz und Sitzungsverwaltung – sowie dessen Open-Source-Implementierung RSPLIB vorgestellt. Insbesondere wurde dabei anhand von Beispielen gezeigt, wie RSPLIB im eigenen Netzwerk eingesetzt werden kann, um Hochverfügbarkeit und Lastverteilung für eigene Anwendungen zu realisieren.

Literaturverzeichnis

[1] Lei, P.; Ong, L.; Tüxen, M.; Dreibholz, T.: "An Overview of Reliable Server Pooling Protocols" (TXT, 32 KiB), IETF, Informational RFC 5351, September 2008.

URL: http://www.ietf.org/rfc/rfc5351.txt.

[2] Stewart, R.; Xie, Q.; Stillman, M.; Tüxen, M.: "Aggregate Server Access Protcol (ASAP)" (TXT, 115 KiB), IETF, RFC 5352, September 2008.

URL: http://www.ietf.org/rfc/rfc5352.txt.

[3] Xie, Q.; Stewart, R.; Stillman, M.; Tüxen, M.; Silverton, A.: "Endpoint Handlespace Redundancy Protocol (ENRP)" (TXT, 81 KiB), IETF, RFC 5353, September 2008.

URL: http://www.ietf.org/rfc/rfc5353.txt.

[4] Stewart, R.; Xie, Q.; Stillman, M.; Tüxen, M.: "Aggregate Server Access Protocol (ASAP) and Endpoint Handlespace Redundancy Protocol (ENRP) Parameters" (TXT, 49 KiB), IETF, RFC 5354, September 2008.

URL: http://www.ietf.org/rfc/rfc5354.txt.

[5] Stillman, M.; Gopal, R.; Guttman, E.; Holdrege, M.; Sengodan, S.: "Threats Introduced by RSerPool and Requirements for Security" (TXT, 37 KiB), IETF, RFC 5355, September 2008.

URL: http://www.ietf.org/rfc/rfc5355.txt.

[6] Dreibholz, T.; Tüxen, M.: "Reliable Server Pooling Policies" (TXT, 32 KiB), IETF, RFC 5356, September 2008.

URL: http://www.ietf.org/rfc/rfc5356.txt.

[7] Dreibholz, T.; Mulik, J.: "Reliable Server Pooling MIB Module Definition" (TXT, 83 KiB), IETF, RSerPool Working Group, RFC 5525, April 2009.

URL: http://www.ietf.org/rfc/rfc5525.txt.

[8] Dreibholz, T.: "Handle Resolution Option for ASAP" (TXT, 11 KiB), IETF, Individual Submission, Internet Draft Version 08, draft-dreibholz-rserpool-asap-hropt-08.txt, work in progress, January 1, 2011.

URL: http://www.watersprings.org/pub/id/draft-dreibholz-rserpool-asap-hropt-08.txt.

[9] Dreibholz, T.; Zhou, X.: "Takeover Suggestion Flag for the ENRP Handle Update Message" (TXT, 11 KiB), IETF, Individual Submission, Internet Draft Version 05, draft-dreibholz-rserpool-enrp-takeover-05.txt, work in progress, January 1, 2011.

URL: http://www.watersprings.org/pub/id/draft-dreibholz-rserpool-enrp-takeover-05.txt.

[10] Dreibholz, T.; Zhou, X.: "Definition of a Delay Measurement Infrastructure and Delay-Sensitive Least-Used Policy for Reliable Server Pooling" (TXT, 16 KiB), IETF, Individual Submission, Internet Draft Version 07, draft-dreibholz-rserpool-delay-07.txt, work in progress, January 1, 2011.

URL: http://www.watersprings.org/pub/id/draft-dreibholz-rserpool-delay-07.txt.

[14] Dreibholz, T.: "Reliable Server Pooling – Evaluation, Optimization and Extension of a Novel IETF Architecture" (PDF, 9079 KiB), University of Duisburg-Essen, Faculty of Economics, Institute for Computer Science and Business Information Systems, March 2007.

URL: http://duepublico.uni-duisburg-essen.de/servlets/DerivateServlet/Derivate-16326/Dre2006-final.pdf.

[15] Dreibholz, T.; Rathgeb, E. P.: "An Evaluation of the Pool Maintenance Overhead in Reliable Server Pooling Systems" (PDF, 3288 KiB), SERSC International Journal on Hybrid Information Technology (IJHIT), vol. 1, no. 2, pp. 17–32, ISSN 1738-9968, April 2008.

URL: http://www.tdr.wiwi.uni-due.de/fileadmin/fileupload/I-TDR/ReliableServer/Publications/IJHIT2008.pdf.

[16] Dreibholz, T.; Rathgeb, E. P.: "Overview and Evaluation of the Server Redundancy and Session Failover Mechanisms in the Reliable Server Pooling Framework" (PDF, 1156 KiB), International Journal on Advances in Internet Technology (IJAIT), vol. 2, no. 1, pp. 1–14, ISSN 1942-2652, June 2009.

URL: http://www.tdr.wiwi.uni-due.de/fileadmin/fileupload/I-TDR/ReliableServer/Publications/IJAIT2009.pdf.

[17] Dreibholz, T.; Zhou, X.; Becke, M.; Pulinthanath, J.; Rathgeb, E. P.; Du, W.: "On the Security of Reliable Server Pooling Systems" (PDF, 539 KiB), International Journal on Intelligent Information and Database Systems (IJIIDS), vol. 4, no. 6, pp. 552–578, ISSN 1751-5858, December 2010.

URL: http://www.tdr.wiwi.uni-due.de/fileadmin/fileupload/I-TDR/ReliableServer/Publications/IJIIDS2010.pdf.

[18] Stewart, R.: "Stream Control Transmission Protocol" (TXT, 337 KiB), IETF, Standards Track RFC 4960, September 2007.

URL: http://www.ietf.org/rfc/rfc4960.txt.

[19] Dreibholz, T.; Rüngeler, I.; Seggelmann, R.; Tüxen, M.; Rathgeb, E. P.; Stewart, R.: "Stream Control Transmission Protocol: Past, Current, and Future Standardization Activities", IEEE Communications Magazine, no. 4, ISSN 0163-6804, April 2011.

[20] Dreibholz, T.: "Thomas Dreibholz's RSerPool Page", 2011.

URL: http://tdrwww.iem.uni-due.de/dreibholz/rserpool/.

[21] Dreibholz, T.: „SimProcTC – Simulation Processing Tool-Chain", 2011.

URL: http://www.iem.uni-due.de/~dreibh/omnetpp.

[22] POV-Team: "POV-Ray – The Persistence of Vision Ray Tracer", 2011.

URL: http://www.povray.org.