Reliable Server Pooling

Evaluation, Optimization and Extension of a Novel IETF Architecture

D I S S E R T A T I O N

to obtain the academic grade
doctor rerum naturalium
(dr. rer. nat.)
in Computer Science

Submitted to the
Faculty of Economics
Institute for Computer Science and Business Information Systems
University of Duisburg-Essen

by
Dipl.-Inform. Thomas Dreibholz
born on 29.09.1976 in Bergneustadt, Germany

Submitted on:	November 28, 2006
Date of Disputation:	March 07, 2007

Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig ohne fremde Hilfe verfaßt und nur die angegebene Literatur und Hilfsmittel verwendet zu haben.

Thomas Dreibholz
November 28, 2006

Abstract

The Reliable Server Pooling (RSerPool) architecture currently under standardization by the IETF RSerPool Working Group is an overlay network framework to provide server replication and session failover capabilities to applications using it. These functionalities as such are not new, but their combination into one generic, application-independent framework is.

Initial goal of this thesis is to gain insight into the complex RSerPool mechanisms by performing experimental and simulative proof-of-concept tests. The further goals are to systematically validate the RSerPool architecture and its protocols, provide improvements and optimizations where necessary and propose extensions if useful. Based on these evaluations, recommendations to implementers and users of RSerPool should be provided, giving guidelines for the tuning of system parameters and the appropriate configuration of application scenarios. In particular, it is also a goal to transfer insights, optimizations and extensions of the RSerPool protocols from simulation to reality and also to bring the achievements from research into application by supporting and contributing relevant results to the IETF's ongoing RSerPool standardization process.

To achieve the described goals, a prototype implementation as well as a simulation model are designed and realized at first. Using a generic application model and appropriate performance metrics, the performance of RSerPool systems in failure-free and server failure scenarios is systematically evaluated in order to identify critical parameter ranges and problematic protocol behaviour. Improvements developed as result of these performance analyses are evaluated and finally contributed into the standardization process of RSerPool.

Keywords:
Reliable Server Pooling, Evaluation, Optimization, Extension

Acknowledgements

This thesis is the result of my work as research associate in the Computer Networking Technology Group of the Institute for Experimental Mathematics at the University of Duisburg-Essen. At this point, I would like to express my acknowledgement to everybody having supported me during my RSerPool research.

In particular, I would like to thank my primary advisor, Erwin P. Rathgeb, for his support of my RSerPool research and this thesis, as well as my secondary advisor Klaus Echtle for his thesis review. Furthermore, I would like to express my special thanks to my former colleague Andreas Jungmaier for his great cooperation in the SCTPLIB/SOCKETAPI project (which is the basis of my RSPLIB prototype), as well as Michael Tüxen from the Münster University of Applied Sciences for his support of the RSPLIB project and our IETF standardization contributions. Furthermore, I would like to thank my student assistants Sebastian Rohde (for writing the initial version of the Demonstration System GUI for the RSPLIB prototype) and Jobin Pulinthanath (for application testing, as well as drawing a lot of my hand-written RSerPool sketches using the DIA drawing program).

In addition, I would also like to thank the Siemens ICN Information and Communication Networks department for supporting my RSerPool project during its first three years, as well as the Deutsche Forschungsgemeinschaft (DFG) for sponsoring the project for another four years.

Finally, I would like to thank my father Ernst Günter Dreibholz and my mother Annelore Dreibholz for the years of encouragement and support.

Contents

Glossary

API	Application Programming Interface
AROS	Amiga Research Operating System
ASAP	Aggregate Server Access Protocol
ASCII	American Standard Code for Information Interchange
ATM	Asynchronous Transfer Mode
BMBF	Bundesministerium für Bildung und Forschung
BSD	Berkeley Software Distribution
CORBA	Common Object Request Broker Architecture
CRC	Cyclic Redundancy Check
CRC − 32	Cyclic Redundancy Check, 32 bits
DFG	Deutsche Forschungsgemeinschaft
DHCP	Dynamic Host Configuration Protocol
DiffServ	Differentiated Services
DNS	Domain Name System
DoD	U.S. Department of Defense
DoS	Denial of Service
DPF	Distance Penalty Factor
ENRP	Endpoint Handlespace Redundancy Protocol
FCS	Future Combat System
FDDI	Fibre Distributed Data Interface
FES	Future Event Set
FSM	Finite State Machine
FTP	File Transfer Protocol
GK	Gatekeeper
GNU	GNU is not Unix
GPL	GNU General Public License
GPS	Global Positioning System
GUI	Graphical User Interface
HTTP	Hyper-Text Transport Protocol
ICMP	Internet Control Message Protocol
ICMPv4	Internet Control Message Protocol, Version 4
ICMPv6	Internet Control Message Protocol, Version 6
ID	Identifier
IEEE	Institute for Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IGMP	Internet Group Management Protocol
IntServ	Integrated Services

IOS	Internet Operating System (Cisco)
IP	Internet Protocol
IPFIX	IP Flow Information Export
IPsec	IP Security
IPv4	Internet Protocol, Version 4
IPv6	Internet Protocol, Version 6
ISDN	Integrated Services Digital Network
ISO	International Standards Organization
ITU	International Telecommunications Union
LAN	Local Area Network
LGPL	GNU Lesser General Public License
LLC	Logical Link Control
LU	Least Used (Pool Policy)
LU – DPF	Least Used with Distance Penalty Factor (Pool Policy)
MAC	Media Access Control
MG	Media Gateway
MGC	Media Gateway Controller
MIB	Management Information Base
MMU	Memory Management Unit
MTBF	Mean Time Between Failure
MTU	Maximum Transmission Unit
NAM	Network Animator (NS-2)
NED	Network Definition Language (OMNeT++)
NGW	Network Gateway
NS	Name Server
NS – 2	Network Simulator 2
OMNeT + +	Objective Modular Network Testbed in C++
OSI	Open Systems Interconnection
PC	Personal Computer
PDA	Personal Digital Assistant
PDF	Portable Document Format
PE	Pool Element
PH	Pool Handle
PLU	Priority Least Used (Pool Policy)
PPE	Proxy Pool Element
PPID	Payload Protocol Identifier
PPP	Point-to-Point Protocol
PPU	Proxy Pool User
PR	Pool Registrar
PSTN	Public Switched Telephone Network
PU	Pool User
QoS	Quality of Service
RAND	Random (Pool Policy)
RFC	Request for Comments
RR	Round Robin (Pool Policy)
RSerPool	Reliable Server Pooling
RSVP	Resource Reservation Protocol

RTT	Round-Trip Time
SASP	Server/Application State Protocol
SCTP	Stream Control Transmission Protocol
SG	Signalling Gateway
SHA	Secure Hash Algorithm
SHA – 1	Secure Hash Algorithm, 192 bits
SHA – 256	Secure Hash Algorithm, 256 bits
SIGTRAN	Signalling Transport Working Group (IETF)
SIP	Session Initiation Protocol
SLA	Service Level Agreement
SLP	Service Location Protocol
SNMP	Simple Network Management Protocol
SOE	SCTP Offload Engine
SPoF	Single Point of Failure
SS7	Signalling System No. 7
SSH	Secure Shell
TCP	Transmission Control Protocol
TLV	Type - Length - Value
TOE	TCP Offload Engine
UDP	User Datagram Protocol
VoIP	Voice over IP
VPN	Virtual Private Network
WAN	Wide Area Network
WG	Working Group (IETF)
WRAND	Weighted Random (Pool Policy)
WRAND – DPF	Weighted Random with Distance Penalty Factor (Pool Policy)
WRR	Weighted Round Robin (Pool Policy)

Chapter 1

Introduction

SERVICES requiring a certain minimum availability are becoming increasingly important. In this chapter, the necessity to ensure service availability is motivated first. After that, the scope of Reliable Server Pooling – the IETF's novel framework to achieve such availability – is presented. This is followed by a short introduction to Reliable Server Pooling itself and an overview of the goals of this thesis.

1.1 Motivation

The Internet is becoming increasingly important for all kinds of applications. In particular, there is also a growing number of availability-critical services being provided in the Internet. Consider an e-commerce scenario offering a virtual shop: as long as the service is working, the provider and its customers can be satisfied. For an e-shop provider, the Internet provides an inexpensive platform to offer products without geographical limitations. Furthermore, the Internet allows a customer to conveniently visit shops online. But in case of a service failure, the provider cannot gain revenue and – even worse – the annoyed customers simply use another service and never come back again. Clearly, the availability of a service like e-commerce is critical and appropriate actions have to be taken to keep such a service running, even in the case of server failures and network problems.

Formally, the *Availability* of services can be classified as in Gray and Siewiorek (1991): so called fault-tolerant systems have an availability of 99.99%, i.e. an average downtime of 50 minutes per year. Systems in the high availability category improve the availability by an order of magnitude to 99.999%, i.e. the average downtime is reduced to 5 minutes per year. To actually achieve a certain minimum availability, mechanisms of multiple research areas are being combined. These basics will be introduced in the following section.

1.2 Scope and Related Work

Although availability has become increasingly important and there are various application-specific solutions to ensure the availability of certain services, there had not been activities to standardize a generic, application-independent framework – until the IETF had started to define such an approach in 2001: Reliable Server Pooling (RSerPool). While the primary motivation of RSerPool has been the availability of SS7-based telephone signalling components, the goal of RSerPool has been set to define an application-independent framework for availability-critical services, being usable for practically any kind of application.

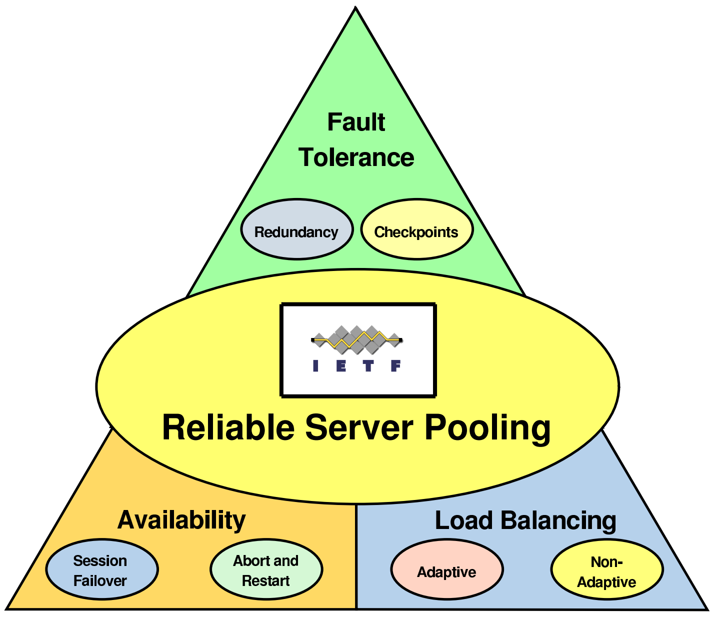


Figure 1.1: Reliable Server Pooling and Related Concepts

While availability solutions are not entirely new, their combination into a single, unified and common framework – the RSerPool architecture – is. RSerPool combines the mechanisms of three different research areas, as illustrated in figure 1.1: availability, load balancing and fault tolerance. These three areas will be described in the following, before RSerPool itself is introduced in more detail.

1.2.1 Availability

A basic method to improve the availability of a service is server replication. Instead of having one server representing a single point of failure, servers are simply duplicated. Most approaches like Linux Virtual Server (see LVS Project (2003)), Cisco™ Distributed Director (see Cisco Systems (2000)) or application-layer anycast (see Bhattacharjee et al. (1997)) simply map a client's session to one of the servers and use the *Abort-and-Restart Principle* in case of a server failure (i.e. the sessions of a failed server are lost and have to be restarted from scratch). While this approach is sufficient for its main application – web server farms – it causes unacceptable delays for long-lasting sessions and is useless for applications like video conferences or real-time transactions (see Uyar et al. (2004)). More sophisticated approaches like FT-TCP (Fault-Tolerant TCP, see Alvisi et al. (2001)), M-TCP (Migratory TCP, see Sultan et al. (2002)) or RSerPool (see Tüxen et al. (2006)) provide a Session Layer to allow a resumption of the interrupted session on a new server. A survey of methods for the necessary server state replication can be found in Wiesmann et al. (2000), a very handy technique is the client-based state sharing described in Dreibholz (2002).

1.2.2 Load Balancing

The existence of multiple servers for redundancy reasons automatically leads to the issues of *Load Distribution* and *Load Balancing*. While load distribution (see Berger and Browne (1999)) only refers to the assignment of work to a processing element, load balancing refines this definition by requiring the assignment to maintain a balance across the processing elements. This balance refers to an application-specific parameter like CPU load or memory usage.

A classification of load distribution algorithms can be found in Gupta and Bepari (1999); the two classes being important for this thesis are adaptive and non-adaptive ones. Adaptive strategies base their assignment decisions on the processing elements' current status (e.g. CPU load) and therefore require up-to-date information. On the other hand, non-adaptive algorithms do not require such status data. An analysis of adaptive load distribution algorithms can be found in Kremien and Kramer (1992); performance evaluations for web server systems using different algorithms are presented in Colajanni and Yu (2002), Dykes et al. (2000), Cardellini et al. (2000).

1.2.3 Fault Tolerance

Fault Tolerance denotes the property of a system enabling it to continue operating properly in the event of the failure of some of its components. While a general introduction into the concepts of this topic can be found in Echtle (1990), two sub-topics have to be emphasized: redundancy concepts and checkpointing.

Important redundancy concepts are explained in Engelmann and Scott (2005): using the simple *Active/Standby* concept, one component is active while one or more backup components are activated if the primary one becomes unavailable. Furthermore, the standby component can be classified into cold-standby (the backup system has to be initialized first), warm-standby (there is a regular state replication between the active and the standby system) and hot-standby (the system state is replicated on every change, i.e. the backup system can losslessly take over the service of the primary system). On the other hand, the *Active/Active* approach provides multiple active systems, allowing for a better utilization of the server resources.

Another important sub-topic of fault tolerance is the checkpointing mechanism. *Checkpointing* is a technique for rollback recovery: an application can set a checkpoint for its current state. In case of problems during service continuation, it can return to the saved state and try again. Checkpointing not only works for local applications (see Plank et al. (1995), where the state is simply saved to disk), but also for distributed networking applications (see León et al. (1993), Seligman and Beguelin (1994)). In this case, the state information is transmitted to other servers or – as proposed by Dreibholz (2002) – transmitted to the client in the form of a state cookie.

1.2.4 Reliable Server Pooling

The scope of RSerPool (see Tüxen et al. (2006)) is to provide an open, application-independent and highly available framework for the management of server pools and the handling of a logical communication (session) between a client and a pool. Essentially, RSerPool constitutes a communications-oriented overlay network, where its Session Layer allows for session migration comparable to Sultan et al. (2002), Alvisi et al. (2001). While server state replication is highly application-dependent and out of the scope of RSerPool, it provides mechanisms to support arbitrary schemes[1]. RSerPool is not restricted to a certain redundancy concept. Its pool management provides sophisticated server selection strategies[2] for load balancing, both adaptive and non-adaptive. Custom algorithms for new applications can be added easily, as described in Dreibholz and Rathgeb (2005b).

[1]See Dreibholz and Rathgeb (2005d), Dreibholz (2002), Conrad and Lei (2005a).

[2]See Tüxen and Dreibholz (2006b), Dreibholz and Rathgeb (2005e,c), Dreibholz, Rathgeb and Tüxen (2005).

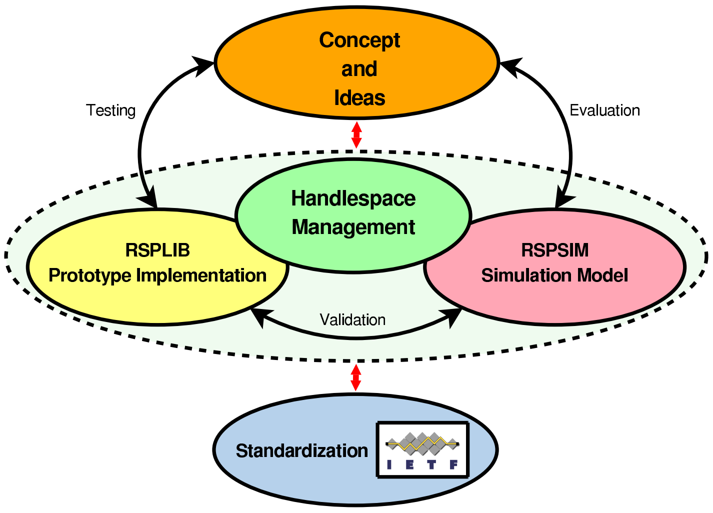


Figure 1.2: The RSerPool Project Concept

1.3 Goals of this Thesis

As already mentioned in section 1.2, the research areas on which RSerPool is based on are not entirely new, but their combination into a single, unified framework is. When our University's RSerPool project – and therefore the work on the contents of this thesis – has been started in 2002, the IETF had already defined the requirements and goals of RSerPool as RFC in Tüxen et al. (2002). Several existing server pooling solutions had already been compared by the IETF (see Loughney et al. (2005)), but since none of the existing technologies had been found capable to satisfyingly fulfil the requirements, it had been necessary to create a new framework. The description of this new framework – RSerPool – as well as basic definitions of the protocols had been published as Internet Drafts (an archive of all historic versions of these drafts can be found at Dreibholz (2006c)). But since nobody had developed an implementation, these documents still included various open issues, flaws and errors. For example, the protocols drafts had only provided the names of a few server selection procedures, but did not actually define them. Some necessary or useful fields in the protocol messages have been missing and the auto-configuration had required users (many, possibly untrustworthy) to send multicast messages (a waste of bandwidth and a security flaw). Furthermore, without possibilities to perform proof-of-concept tests or at least to evaluate the new protocols simulatively, it had not been clear that the RSerPool framework would work as intended.

Initial goal of this thesis therefore has been to gain insight into the complex RSerPool mechanisms by performing proof-of-concept tests using lab experiments and by performing simulations covering a more comprehensive range of parameter values. Further goals have been to systematically validate the RSerPool architecture and its protocols, provide improvements and optimizations where necessary and propose extensions if useful for the applicability of RSerPool. Based on these evaluations, recommendations to implementers and users of RSerPool should be provided, giving guidelines for the tuning of system parameters and the appropriate configuration of application scenarios. In particular, it is also a goal to transfer insights, optimizations and extensions of the RSerPool protocols from simulation to reality and also bring the achievements from research into application by supporting and contributing relevant results to the IETF's ongoing RSerPool standardization process.

In order to fulfil the goals of this thesis, the following work items have been identified:

- Design and implementation of a RSerPool prototype for tests in real networks and to support the standardization;
- Design and implementation of a RSerPool simulation model;
- Development of an efficient and flexible handlespace management component, required for both, the prototype implementation and the simulation model;
- Definition of a generic application model and performance metric, as well as its realization for both, prototype implementation and simulation model;
- Simulative evaluation of the RSerPool and related application parameters in order to identify critical parameter ranges and problematic protocol behaviour;
- Finding improvements and validating their effectiveness, as well as
- Finally contributing the improvements into the standardization process.

An illustration of the work area coherence is provided in figure 1.2.

1.4 Organization of this Thesis

This thesis is structured as follows: at first, a short overview of the necessary networking basics is provided in chapter 2. After that, a detailed description of the RSerPool framework is given in chapter 3, including design decisions, application scenarios, the two RSerPool protocols ASAP and ENRP and a detailed overview of the availability mechanisms used in RSerPool systems.

Since an efficient management of the handlespace (i.e. the set of pools and their servers) is crucial for both, the prototype implementation and the simulation model, chapter 4 presents the developed handlespace management approach first. Afterwards, the actual implementation of the prototype RSPLIB is introduced in chapter 5 and the description of the simulation model RSPSIM is depicted in chapter 6.

Chapter 7 deals with the performance evaluation of the handlespace management approach. The actual evaluation of the RSerPool system performance is presented in chapter 8 (for failure-free scenarios) and in chapter 9 (for server failure scenarios). The final chapter of this thesis (chapter 10) provides a conclusion and outlook on future work.

Chapter 2

Networking Basics

In this chapter, the networking basics important for this thesis are presented. The intention of the following sections is to outline the context of Reliable Server Pooling, as well as determining important definitions and terminology. Since the particular networking basics are well documented, only a short introduction is given here. For a detailed description, see the provided references and computer networks textbooks like Tanenbaum (1996).

2.1 The OSI and TCP/IP Networking Models

The two most important networking models are the *Open Systems Interconnection Model* (OSI Model) by the International Standards Organization (ISO) and the *TCP/IP Reference Model* (TCP/IP Model) by the U.S. Department of Defense (DoD); they are presented in figure 2.1. The idea of both models is to divide up the functionalities of a complex networking system into *Layers*, each one built upon the one below it. The purpose of each layer is to provide a defined *Service* to the upper layers, shielding any details of how this service is actually realized.

The entities on different machines that comprise the corresponding layers are denoted as *Peers*. Two peers communicate using rules and conventions called *Protocol*. A *Protocol Stack* is a set of protocols used by a certain system, defining one protocol per layer. In order to transmit a message from layer n on one machine to layer n on another one, each layer passes its data and control information

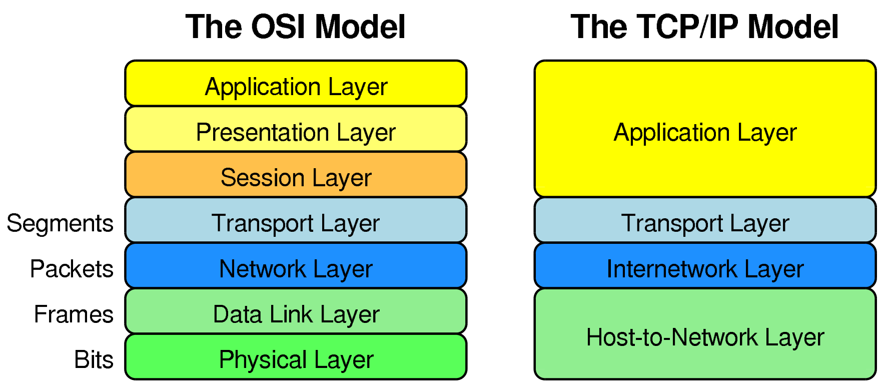


Figure 2.1: The OSI and TCP/IP Networking Models

to its lower layer, until the lowest one is reached. On the other machine, this procedure is repeated in reverse order. The operations and services a layer offers to its upper one define a layer's *Interface*. Due to the defined interface, an implementation for a layer can be simply replaced by another one – without any change to the other layers. This makes it easy to adapt a complex networking stack to changing requirements.

The seven layers of the OSI model have the following functionalities:

Physical Layer: This layer handles the physical transmission of data over a certain medium. In particular, it defines how to send bits e.g. via copper or fibre wires or transmit them via a radio link.

Data Link Layer: The second layer provides the functionalities to transfer data between network entities. This incorporates physical addressing as well as data framing and error correction. A data unit handled by this layer is denoted as *Frame*.
The Data Link Layer is often separated in two sub-layers: the *Medium Access Control* (MAC) sub-layer handles the ordered access to the physical medium; the *Logical Link Control* (LLC) sub-layer copes with upper-layer protocol identification and data framing as well as error detection and recovery.

Network Layer: Functionalities for transferring variable length data sequences from a source to a destination endpoint via one or more networks are provided by this layer. In particular, it provides a logical, hierarchical addressing scheme and network routing. The data entity handled by this layer is denoted as *Packet*.

Transport Layer: The fourth layer provides the transfer of user data and includes the segmentation and reassembly of large data blocks, reliable transport, flow control and congestion handling. A data entity of this layer is called *Segment*.

Session Layer: In the fifth layer, dialogue control between end-user applications is realized. This includes mechanisms for duplex or half-duplex operation and the definition of checkpoints, adjournment and restart of a *Session*.

Presentation Layer: The Presentation layer is responsible for translating data encodings between different end-user systems. For example, this could mean to convert data between ASCII and Unicode representations. In particular, this layer also provides encryption and decryption as well as compression and decompression of data.

Application Layer: This layer provides the actual service of the end-user application.

The seven layer model is sometimes extended by two more layers: the Financial Layer (layer 8) and the Political Layer (layer 9). While this can mainly be seen as humour, it is not too far away from reality.

The TCP/IP model simplifies the OSI model by a reduction from seven to four layers:

Host-to-Network Layer: This layer combines the functionalities of the OSI model's Physical and Data Link layers. That is, it includes the physical transmission as well as the controlled access to a transport medium.

Internetwork Layer: The Internetwork Layer is correlated to the Network Layer of the OSI model. Therefore, each Internetwork Layer protocol can also be seen as a Network Layer protocol. Important protocols of this layer are IPv4 (see subsection 2.3.1) and IPv6 (see subsection 2.3.2).

Transport Layer: The Transport Layer of the TCP/IP model directly maps to the OSI model's corresponding layer. The most important Transport Layer protocols are TCP (see subsection 2.4.2), UDP (see subsection 2.4.1) and SCTP (see subsection 2.4.3).

Application Layer: The Application Layer of the TCP/IP model combines the functionalities of the OSI model's Session, Presentation and Application Layers. Some important protocols of this layer are the Hyper-Text Transfer Protocol (HTTP, see Fielding et al. (1999)), the File Transfer Protocol (FTP, see Postel and Reynolds (1985)) and the Simple Network Management Protocol (SNMP, see Harrington et al. (2002)).
Note, that it is not possible to uniquely map Application Layer protocols of the TCP/IP model to the OSI model. For example, FTP provides the conversion of character encodings – which can be seen as a Presentation Layer functionality. Therefore, mappings in literature may vary.

Since Reliable Server Pooling in particular provides Session Layer functionalities, the OSI model provides a more fine-grained description. Therefore, it is used throughout this thesis. To overcome the ambiguity of the Application Layer mappings described above, the following convention is used: all protocols directly interacting with the service provided for the user are mapped to the OSI Model's Application Layer.

2.2 The Standardization of Network Protocols

Next to their OSI model, the ISO has also defined its own protocols for each of their layers. Due to various reasons, none of their protocols achieved a higher distribution. For a detailed discussion of the reasons, see section 1.4.4 of Tanenbaum (1996). Probably one of the most important lacks of the OSI standards has been that a huge fee has to be paid for each(!) of their standards documents. This clearly did not motivate many people to deal with the ISO standards.

The creation of open, free to the public standards has been the goal of the Internet Engineering Task Force (IETF). Unlike ISO standardization, the activities of the IETF are based on public mailing lists and meetings, open to every person interested in it. Standards are created by discussions and rough consensus, a resulting standards document is published as Request for Comments (RFC). All RFCs can be accessed on the IETF website (see IETF (2006)) by anybody interested in, free of charge. Due to the IETF's conception in contrast to the ISO standardization, the IETF protocols have become highly successful. Today, nearly all significant protocols of layer 3 and above are standardized by the IETF as RFCs, e.g. IPv4 and IPv6, TCP, SCTP and HTTP.

2.3 The Network Layer

In this section, the important Network Layer protocols IPv4 and IPv6 are shortly introduced. The main intention of this section is to show the aspects of the protocols that are relevant for this thesis. In particular, this includes address scoping and multicast handling. For a detailed introduction into both protocols, see the provided references.

2.3.1 IP Version 4

Version 4 of the Internet Protocol (IP), therefore denoted as IPv4, has been defined in September 1981 as RFC in Postel (1981b). Until today (2006), it is still the basis of the current Internet. IPv4 provides 32-bit addresses, therefore the number of hosts is theoretically limited to $2^{32} = 4,294,967,296$. In practice, however, the number of possible hosts is of course significantly lower, due to reserved address spaces and inefficient address assignment.

Unlike global addresses, the scope of the following address spaces is limited: all addresses from 127.0.0.0 to 127.255.255.255 belong to a host itself. That is, these addresses can only be used between endpoints on the same host. 10.0.0.0 to 10.255.255.255, 172.16.0.0 to 172.31.255.255 and 192.168.0.0 to 192.168.255.255 are private address spaces that can be freely used within local networks. Addresses within these private spaces are not routed within the Internet.

Together with IPv4, the Internet Control Message Protocol (ICMP), version 4 – therefore denoted as ICMPv4 – has been defined in Postel (1981a). It provides functionalities for testing as well as for reporting errors like the unreachability of a destination address.

IPv4 also provides the possibility to send multicast messages. For this purpose, the address range from 224.0.0.0 to 239.255.255.255 is reserved. While the first four bits of the address are fixed, the remaining 28 bits specify the destination multicast group. All endpoints having joined the corresponding multicast group should receive a message addressed to the group's multicast address. The functionalities for joining and leaving a multicast group are provided by the Internet Group Management Protocol (IGMP). The latest version of the IGMP protocol is version 3; it is defined in Cain et al. (2002).

2.3.2 IP Version 6

Mainly due to the address space limitations of IPv4, version 6 of the Internet Protocol (IPv6) has been developed and standardized[1] as RFC in Deering and Hinden (1998b). In contrast to IPv4, IPv6 uses a simplified header format and 128-bit addresses. This provides the theoretical possibility for up to $2^{128} \approx 3.402824 * 10^{38}$ addresses. Even in pessimistic mapping scenarios, this provides sufficient space for millions of addresses per square metre of earth surface.

The textual address representation is defined in section 2.2 of Hinden and Deering (2006): an IPv6 address is written as eight colon-separated groups of four hexadecimal digits (i.e. 16 bits). Leading zeros in a group may be omitted. Example: 2001:638:501:4ef1:204:23ff:fe9f:1087. In order to simplify the representation, a double-colon "::" indicates one or more groups of zeros. This compression may occur at most once in an address. It may also be used to compress leading or trailing zeros. Examples: 2001:5c0:0:2::24, 2001:638:501:4ef1::, ::1. Subnet masks are written in prefix notation (i.e. the length of the subnet mask in bits). Examples: 2001:638:501:4ef1::/64, 2001::/16.

Like IPv4, IPv6 also provides different address scopes (defined in Hinden and Deering (2006)):

- ::1/128 is the loopback address and can only be used by an endpoint itself.
- fe80::/10 denotes the *Link-Local* address space. It is only valid for the local physical link, e.g. an Ethernet segment.
- fec0::/10 denotes the *Site-Local* address space. It is equivalent to a private address space in IPv4. The use of site-local addresses is deprecated as defined in the RFC Huitema and Carpenter (2004). That is, it should not be used any more.

[1] An experimental version 5 of the IP protocol using 64-bit addresses never reached RFC status. Nevertheless, the version number had already been assigned. Therefore, the successor of IPv4 is IPv6.

- fc00::/7 denotes *Local IPv6 Unicast Addresses*. This is the replacement for site-local addresses, defined in Hinden and Haberman (2005). Although only routed locally (e.g. in the network of a company), they are at a very high probability globally unique (due to a pseudo-random global ID part). The property of uniqueness solves multiple problems described in Huitema and Carpenter (2004).

Together with IPv6, the Internet Control Message Protocol, version 6 – therefore denoted as ICMPv6 – has been defined in Deering and Hinden (1998a). Like ICMPv4, it provides functionalities for testing as well as for reporting errors like the unreachability of a destination address.

Analogously to IPv4, IPv6 also provides multicast capabilities. The address space ff00::/8 has been reserved for this purpose. To join or leave a multicast group, the IGMP protocol is used again. For details, see Cain et al. (2002).

An important difference to IPv4 is that IPv6 mandatorily requires the support of IPsec. That is, it supports authentication and confidentiality in the Network Layer by default. For details on IPsec, see the RFCs Kent and Atkinson (1998c,a,b). IPsec is optional for IPv4.

Another important property of IPv6 is the availability of a so called *Flow Label* in the IPv6 header. A flow label is a 20-bit number that is – together with its IPv6 source address – network-unique for a flow to a certain destination. If the upper-layer data is encrypted using IPsec and therefore being unreadable for network-nodes other than the receiver, this can be used to identify certain flows, e.g. to ensure Quality of Service (QoS, see Dreibholz (2001) for details) properties like an assured bandwidth. An idea to add a flow label option to IPv4 is proposed in Dreibholz (2005a).

2.4 The Transport Layer

In this section, the important Transport Layer protocols UDP, TCP and SCTP are introduced. Since UDP and TCP are well-known standards and ample documentation is available, the introduction here is quite short and only outlines the important facts necessary to understand their differences to SCTP. The SCTP protocol, since very relevant for Reliable Server Pooling, is explained in a more detailed form.

2.4.1 UDP

The User Datagram Protocol (UDP), defined as RFC in Postel (1980), provides a *connection-less*, *message-oriented* transmission of data. A message is also denoted as *Datagram*, hence the protocol's name. To provide the possibility of multiple UDP instances per endpoint, UDP uses a 16-bit *Port* number to identify source and destination instance in the Transport Layer.

The message framing is preserved by UDP. That is, if the upper layer transmits a message of size n, the destination endpoint will provide the full message to its upper layer. The 16-bit Internet Checksum (defined in Braden et al. (1988)) is used to detect transmission errors. In case of an error, the whole message is dropped. UDP does not ensure the sequence of the datagrams. That is, they may be delivered in any order at the receiver side. This is denoted as *unordered delivery*. Furthermore, the transport of UDP is *unreliable*. There is no mechanism in UDP to detect the loss of messages. If lossless transmission is required, this has to be ensured by the upper layer. Furthermore, UDP does not provide any flow and congestion control.

2.4.2 TCP

Unlike UDP, the Transmission Control Protocol (TCP) defined as RFC in Postel (1981c) provides a *connection-oriented*, *stream-oriented* transmission of byte streams. Before any user data can be transferred, a connection has to be established using a 3-way handshake. TCP does not preserve message framing, e.g. a sender may send messages of n bytes while the receiver side receives them byte-wise. For the receiver, it is not possible to determine the original blocks given by the sender's upper layers to the Transport Layer. If the preservation of message framing is required, this has to be realized by the upper layers.

TCP provides a *reliable* service: all data sent by the sender will be received on the receiver side in exactly the same order. Lost segments will be detected and retransmitted. Furthermore, TCP provides window-based flow and congestion control. That is, the transmission speed is controlled based on the receiver's and network's current capabilities. Like UDP, the TCP protocol uses 16-bit port numbers to identify instances on the same endpoint. Furthermore, the data correctness is ensured using the 16-bit Internet Checksum (see Braden et al. (1988)).

2.4.3 SCTP

The Stream Control Transmission Protocol (SCTP) was originally intended to transport SS7 (Signalling System No. 7) telephone signalling over IP networks. Since SS7 networks have strict availability requirements[2], these demands also have to be fulfilled by the transport over IP. Nevertheless, various other application scenarios have been found. This section gives an introduction to the SCTP protocol.

2.4.3.1 Introduction

The SCTP protocol has been defined in 2000 by the IETF Signalling Transport Working Group (SIGTRAN) as RFC in Stewart et al. (2000), with some updates in Stone et al. (2002), Stewart, Arias-Rodriguez, Poon, Caro and Tüxen (2006). An overview of SCTP is provided in Jungmaier (2005), Jungmaier et al. (2001, 2000a), Ong and Yoakum (2002), application scenarios are described in Coene (2002).

SCTP provides a connection-oriented, message-oriented transport of data. To provide the identification of Transport Layer instances, a 16-bit port number as for TCP and UDP is used. All SCTP messages are protected against transmission errors using the 32-bit CRC-32 checksum[3] as defined in Stone et al. (2002). The originally used Adler-32 checksum has been replaced due to error detection weaknesses. In the following, the important features of SCTP are shortly explained.

2.4.3.2 Packet Format

The format of a SCTP packet is presented in figure 2.2. It always includes source and destination port numbers, the checksum and a *Verification Tag*. The verification tag is a 32-bit random number, negotiated at connection setup for each transmission direction. It has to be equal for all following packets in the same direction. This is used as a security mechanism against blind attacks: an attacker being unable to read the packet stream not only has to guess the port numbers (which are in many cases fixed), but also has to guess the verification tag.

[2]See Gradischnig and Tüxen (2001), Gradischnig et al. (2000), Jungmaier et al. (2000a), Jungmaier (2005).

[3]A CRC-32 checksum does not denote a sum in mathematical sense. Nevertheless, this is the terminology used by the standards documents.

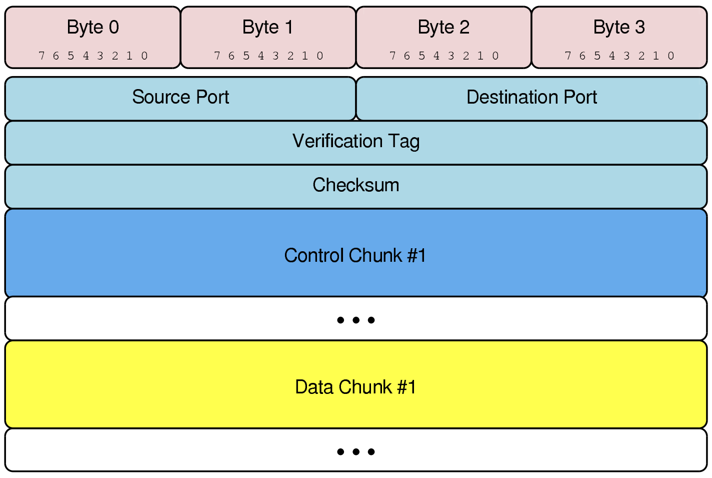


Figure 2.2: The SCTP Packet Format

Each SCTP packet includes one or more *Chunks*, which may include user data (Data Chunk) or control data (Control Chunk). Extensions to the SCTP protocol (see subsubsection 2.4.3.7) can be easily realized by defining new types of control chunks.

2.4.3.3 Association Establishment

Before any user data can be sent, it is first necessary to establish a connection, called *Association* in SCTP terminology. This is realized by a 4-way handshake as shown in figure 2.3. To set up an association, an INIT control chunk is sent to the peer instance B first. If B accepts the setup request, it stores all necessary information for association setup into a so called *Cookie*, which is then cryptographically signed by a secret key. The cookie is sent back to A using an INIT ACK chunk. After that, B deallocates all resources related to the new association. A handles the cookie as arbitrary byte vector and must return it without any changes to B using a COOKIE ECHO chunk. B can verify its authenticity using its secret key and can subsequently establish the association using the information from the cookie. After that, it confirms the successful association setup using a COOKIE ACK chunk.

The 4-way handshake solves an important problem of TCP's 3-way handshake: for TCP, an attacker could use a spoofed source address to send SYN requests (see Postel (1981c)) to a TCP server. Upon reception of the SYN, the TCP instance will reserve resources for the new connection. Even if the sender is not existing, the resources remain reserved for several minutes (depending on the TCP implementation). If an attacker sends large numbers of SYNs (known as *SYN Flooding* attack), this will lead to a Denial of Service (DoS), preventing legitimate users of the server to connect to it due to a lack of resources.

On the other hand, using the 4-way handshake of SCTP, the sender of the INIT must receive the INIT ACK to be able to reply with a COOKIE ECHO. That is, its endpoint address must be valid. Otherwise, the SCTP server will not reserve any resources for the association.

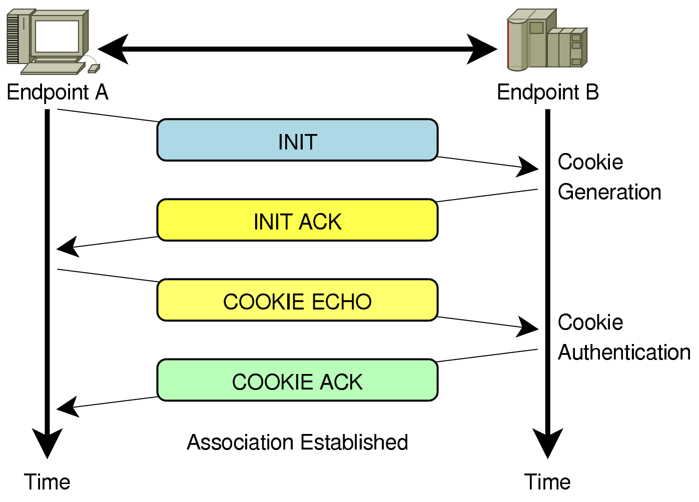


Figure 2.3: The SCTP Association Setup

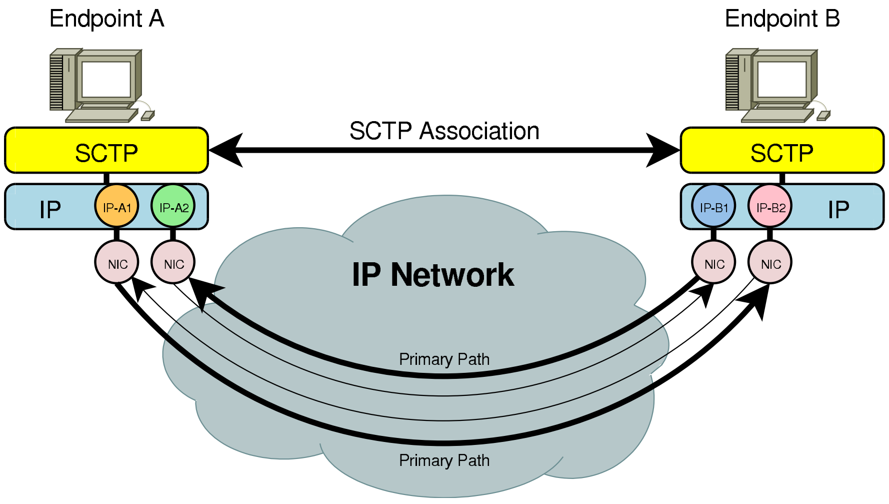


Figure 2.4: The Multi-Homing Feature of SCTP

2.4.3.4 Multi-Homing

Presumably the most important feature of SCTP is the *Multi-Homing*. That is, endpoints of an association can utilize more than one Network Layer address, e.g. due to multiple network cards. Figure 2.4 illustrates the principle: SCTP supports this fact by viewing each Network Layer address of an association's peer endpoint as possible *Path* to this endpoint. A Network Layer address can in principle belong to an arbitrary Network Layer protocol. In particular, an endpoint may also use multiple Network Layer protocols, e.g. IPv4 and IPv6. That is, a peer endpoint can be reached under the address of each path. If there are n disjoint paths within the network, any n-1 paths may fail without interrupting the association.

SCTP monitors each of an association's paths using HEARTBEAT chunks. An endpoint receiving such a chunk has to reply to it using a HEARTBEAT ACK chunk. By receiving the HEARTBEAT ACK upon its HEARTBEAT chunk, an endpoint has verified that a path is actually usable. For the transport of user data, one of the possible paths is chosen as so called *Primary Path*. If the primary path fails, another one is selected and a *Path Failover* is performed – transparently for the upper layers.

Using the Add-IP extension described in subsubsection 2.4.3.7, it is furthermore possible to add new Network Layer addresses (and therefore paths) or remove them during association runtime.

2.4.3.5 Congestion Control

Using exactly one primary path – instead of balancing the traffic among multiple paths and utilizing multiple networks – has been realized due to congestion control reasons: the connection-less paradigm of IP networks provides no way to ensure that multiple paths are disjoint. In particular, it is not possible to decide whether congestion on two paths is caused by a single common link. SCTP realizes a per-path TCP-friendly congestion control. That is, utilizing n paths over the same bottleneck link would appear like n TCP-controlled flows. Clearly, since all traffic is caused by the same association, this would be quite unfair to other TCP-based flows. However, work is in progress to cope with this problem. See Jungmaier (2005) for details.

2.4.3.6 User Data Transport

Another interesting feature of SCTP is its support for multiple streams over one association, the so called *Multi-Streaming*. A *Stream* is an uni-directional channel between the two endpoints; the number of streams in each direction is negotiated at association setup. Figure 2.5 illustrates the principle of multi-streaming: Endpoint A sends data packets on different streams. The SCTP layer segments the data blocks into data chunks and transmits them over the association; the peer endpoint B combines the data chunks back into the full messages and provides them to the upper layer (i.e. SCTP preserves the message framing).

There is no *Head-of-Line Blocking* for data messages on different streams. That is, if a message of one stream is fully recombined at endpoint B, it is given to the upper layer – regardless of messages in other streams. This means that messages of different streams may overtake each other. However, by default, the order of messages within the same stream is preserved (so called *Ordered Delivery*). If ordered delivery is not necessary, it can be turned off on a per-message basis. In this case, unordered delivery similar to UDP is provided.

To support the communication of different upper-layer protocols over the same stream, a per-message *Payload Protocol Identifier* (PPID) is provided. This identifier denotes a 32-bit number specifying the upper-layer protocol.

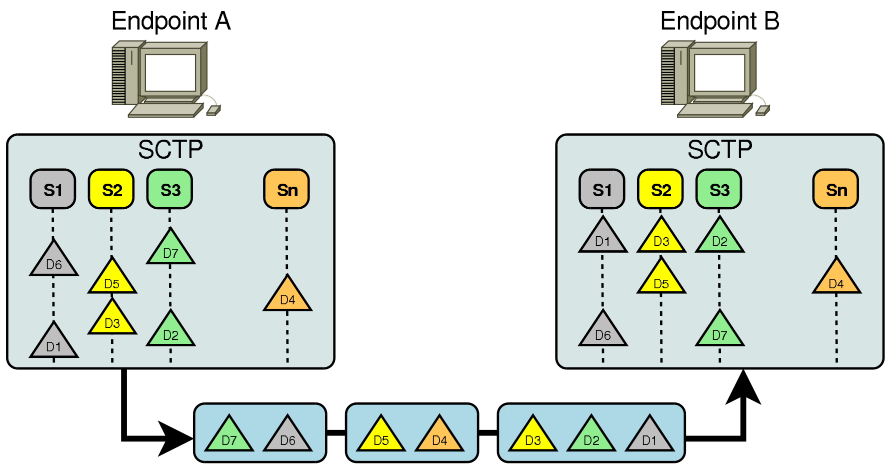


Figure 2.5: The Multi-Streaming Feature of SCTP

2.4.3.7 Extensions

As explained in subsubsection 2.4.3.2, the protocol design of SCTP provides the possibility to easily add extensions by defining new chunk types. This subsubsection describes the currently defined protocol extensions.

Partial Reliability (Pr-SCTP) The Partial Reliability extension for SCTP, defined in Stewart et al. (2004), allows to specify a timeout for the transmission of a message on a per-message basis. Retransmissions of a message are only triggered until its given timeout expires. After expiration of this timeout, the message is simply dropped.

By using a timeout of 0ms, SCTP behaves like UDP, in the way that it only tries to send the message once. On the other hand, SCTP still keeps providing congestion and flow control for every message. That is, unlike UDP, a sender will not overload the network or the receiver with its data.

Pr-SCTP is supported by all major SCTP implementations; both endpoints tell their peer instance on association setup whether or not Pr-SCTP is available. If one of the endpoints is not capable of Pr-SCTP, a message timeout will simply be ignored, i.e. the transport will become reliable.

Authentication Chunk The authentication of chunks is provided by the Authentication Chunk extension defined in Tüxen et al. (2005). Using a pre-shared key plus a random key negotiated at association setup, it is possible to ensure that an endpoint keeps communicating with the peer endpoint it has established the association to. That is, an attacker will be unable to hijack an association.

Dynamic Address Reconfiguration (Add-IP) The Dynamic Address Reconfiguration extension (called Add-IP in short) is defined in Ramalho et al. (2006) and provides the possibility to add Network Layer addresses to and remove them from an association during its runtime without interrupting the association or bothering the upper layer. The primary reason for this extension has been to support seamless IPv6 renumbering in telecommunications signalling scenarios. That is, it allows for replacing the IPv6 prefix on a provider change – without interrupting any established signalling relation. In particular, it would also be possible to seamlessly upgrade an IPv4 network to IPv6 – as long as the endpoints already have support for both protocols. Other interesting application scenarios include the support of mobility (Mobile SCTP, see Riegel and Tüxen (2006)).

To ensure that an attacker may not easily change an association's endpoint to himself, Add-IP mandatorily requires the usage of the Authentication Chunk extension.

Packet Drop The Packet Drop extension defined in Stewart, Lei and Tüxen (2006a) can be used to inform a sender of packet drops not related to congestion. In particular, it can be used if error-prone transmission channels like satellite links are used for the transmission and the receiver detects a checksum difference in a received SCTP packet. Then, a packet drop report can tell the sender about the reception error and hereby force an immediate retransmission. Otherwise, the sender would assume a packet loss due to congestion and reduce its data rate.

Stream Reset The Stream Reset extension defined in Stewart, Lei and Tüxen (2006b) is used to reset a stream of an association, in order to allow its usage for a different application. In particular, it resets the message sequence number for a certain stream in order to allow the new application of this stream to use it for keeping track of the message sequence.

Secure-SCTP (S-SCTP) While SCTP can be used with IPsec or TLS[4] (see Jungmaier, Rescorla and Tüxen (2002)) to ensure authenticity, integrity and confidentiality of the transmissions, none of both solutions is optimal. While IPsec lacks of efficiency, TLS does not support the security of control chunks as well as unordered delivery and Pr-SCTP (for a detailed discussion of the problems, see Unurkhaan (2005)). The Secure-SCTP extension (S-SCTP) provides a security extension for SCTP, supporting both, authentication and encryption, on a per-message basis and providing full support for the ordered and unordered delivery features as well as for the Pr-SCTP extension. This proposed extension is described as Internet Draft in Hohendorf et al. (2006).

2.5 Summary

In this chapter, a short summary of the important networking basics have been given: the OSI and TCP/IP models, the Network Layer protocols IPv4 and IPv6 – including ICMPv4, ICMPv6 and IGMP, as well as the Transport Layer protocols TCP, UDP and SCTP. The SCTP protocol provides some important features making it superior to TCP and UDP: its security against attacks due to verification tag and 4-way handshake, its resilience due to multi-homing, the preservation of message framing and its support for multiple streams.

[4]Transport Layer Security, see Dierks and Allen (1999), Blake-Wilson et al. (2003).

Chapter 3

Reliable Server Pooling

THIS chapter explains the Reliable Server Pooling concept, its functionalities and its two protocols: ASAP (Aggregate Server Access Protocol) and ENRP (Endpoint Handlespace Redundancy Protocol). The first part of this chapter describes the motivation for RSerPool and its architecture (section 3.1 to section 3.5), including application scenarios for RSerPool in section 3.6. After that, an overview of the RSerPool component types is given in section 3.7. In particular, this section illustrates the interaction among its different classes of components. This is followed by the description of the two RSerPool protocols themselves in section 3.9 and section 3.10, as well as the currently defined server selection procedures in section 3.11. Based on the functionalities and protocols descriptions of RSerPool, the failure model as well as a summary of the mechanisms for service reliability and availability are presented in section 3.12. Finally, a short introduction to the security concept of RSerPool is given in section 3.13.

3.1 Introduction

According to Echtle (1990), *Reliability* denotes the ability of a system to operate correctly for a given time under valid operating conditions. An approach to ensure the reliability of a system is to ensure the availability of its components. As defined in Grams (1999), the term *Availability* denotes the probability of a correct function within the meaning of reliability. In classical telecommunications-oriented networks, component availability is ensured by redundant links and devices. An example is presented in Rathgeb (1999), where several mechanisms to increase the availability of a commercial ATM switch are described.

The focus of the Reliable Server Pooling architecture, which will be presented and evaluated in this thesis, is – despite its name – the availability of services. Nevertheless, the overall goal of its users is to achieve the required reliability of a service being built on top of this architecture. The description of the failure model as well as details on the provided mechanisms to ensure service reliability and availability will be presented in section 3.12; but first, it is necessary to introduce the concept of Reliable Server Pooling and its protocols.

The term *Server Pooling* denotes the management of redundant server resources for the same service in order to ensure its availability and/or to realize load distribution among different servers. Various solutions for the realization of server pooling are available, both proprietary (like Cisco™ Distributed Director, see Cisco Systems (2000)) and Open Source (like Linux Virtual Server, see LVS Project (2003)). Most of the available solutions focus on very specific services; the service most frequently addressed by server pooling systems is HTTP (see Fielding et al. (1999)). But with the growing demand and number of provided services, it became more and more difficult to develop and maintain a specific server pooling solution for each service – instead of "re-inventing the wheel" again and again, a common, unified and standardized framework is required.

The development and standardization of such an architecture for server pooling has been set as the goal of the IETF Reliable Server Pooling Working Group (IETF RSerPool WG, see IETF RSerPool WG (2005)). As a result, the working group has created their concept *Reliable Server Pooling*, abbreviated as RSerPool, which at the moment consists of one RFC, several Internet Drafts and our prototype implementation RSPLIB (see chapter 5) as reference implementation.

3.2 The Requirements for RSerPool

As key requirements for the Reliable Server Pooling architecture to be defined, the IETF RSerPool WG had identified the following points:

Lightweight: The RSerPool solution may not require a significant amount of resources (e.g. CPU power or memory). In particular, it should be possible to realize RSerPool-based systems also on low-power devices like mobile phones, PDAs and embedded devices.

Real-Time: Services like telephone signalling (see subsection 3.6.1) have very strict limitations on the duration of failovers. In the case of component failures, it may be necessary that a "normal" system state is re-established within a time frame of only a few hundreds of milliseconds. In telephone signalling, such a feature is in particular crucial if dealing with emergency calls.

Scalability: For providing services like Distributed Computing (see subsection 3.6.5), it is necessary to manage pools of many hundreds or even thousands of servers (e.g. animation rendering pools). The RSerPool architecture must be able to efficiently handle such pools. But number and size of pools are limited to a company or organization. In particular, it is not a goal of RSerPool to manage the complete Internet in one set of pools.

Extensibility: It must be possible to easily adapt the RSerPool architecture to future applications. In particular, this means to have the possibility to add new server selection procedures. That is, new applications can define special rules on which server of the pool is the most appropriate to use for the processing of a request (e.g. the least-used server).

Simplicity: The effort necessary to configure RSerPool components (i.e. to add or remove servers) should be as small as possible. In the ideal case, the configuration should happen automatically, i.e. it should only be necessary to turn on a new server.

A more detailed description of the requirements for the server pooling architecture can be found in Tüxen et al. (2002). The draft Loughney et al. (2005) discusses the applicability of existing technologies like the Domain Name System (DNS), the Common Object Request Broker Architecture (CORBA), the Service Location Protocol (SLP) and Layer 4/Layer 7 switching to achieve the requirements described above. In summary, no existing technology has been found capable to sufficiently fulfil all of the requirements. Therefore, the definition of a new framework – Reliable Server Pooling (RSerPool) – is justified.

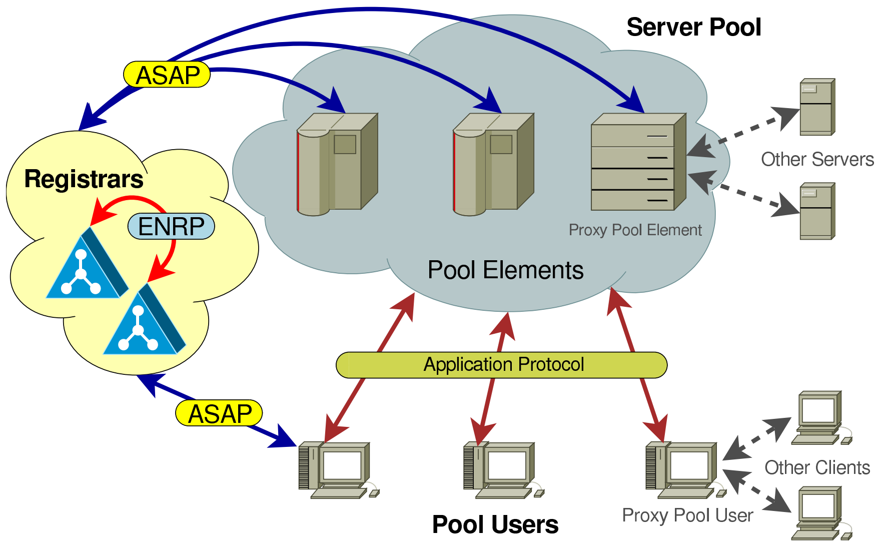


Figure 3.1: The RSerPool Concept

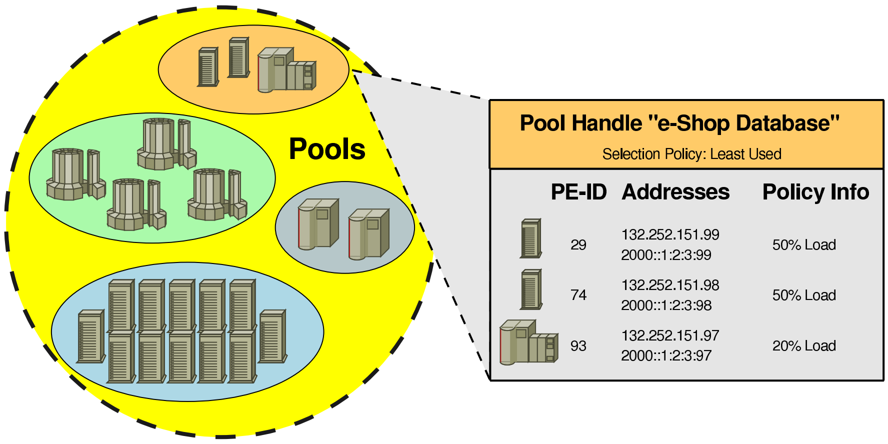
3.3 The RSerPool Architecture

Figure 3.1 shows the building blocks of the RSerPool architecture, which has been defined by the IETF RSerPool WG in Tüxen et al. (2006). In the terminology of RSerPool, a server is denoted as a *Pool Element* (PE). In its *Pool*, which groups servers providing the same service[1], it is identified by its *Pool Element Identifier* (PE ID). The PE ID is a 32-bit random number which is chosen upon a PE's registration into its pool. The set of all pools is denoted as the *Handlespace*. In older literature, it may be denoted as *Namespace*. This denomination has been dropped in order to avoid confusion with the DNS. Each pool in a handlespace is identified by a unique Pool Handle (PH), which is represented by an arbitrary byte vector. Usually, this is an ASCII or Unicode name of the pool, e.g. "DownloadPool" or "WebServerPool".

Each handlespace has a certain scope (e.g. an organization or company), denoted as *Operation Scope*. As explained in section 3.2, it is explicitly not a goal of RSerPool to manage the global Internet's pools within a single handlespace. Due to the limitation of operation scopes, it is possible to keep the handlespace "flat". That is, PHs do not have any hierarchy in contrast to the DNS with its top-level and sub-domains. This constraint results in a significant simplification of the handlespace management (see also chapter 4 for details on handlespace management). Figure 3.2 presents an example for the information stored in a handlespace. In this example, a pool denoted by its PH "e-Shop Database" consists of 3 PEs with their IPv4 and IPv6 transport addresses as well as their policy information (to be explained in detail later; here: information on server load).

Within an operation scope, the handlespace is managed by redundant *Registrars* (Pool Registrar, PR). In literature, this component is sometimes also denoted as *ENRP Server* or *Name Server* (NS). Since "registrar" is the most expressive term, this denotation is used in the whole document. PRs have to be redundant in order to avoid a PR to become a single point of failure (SPoF). Each PR of an operation scope is identified by its *Registrar ID* (PR ID), which is a 32-bit random number. It is not

[1]Therefore, a server providing multiple services is registered in multiple pools.



PE-ID	Addresses	Policy Info
29	132.252.151.99 2000::1:2:3:99	50% Load
74	132.252.151.98 2000::1:2:3:98	50% Load
93	132.252.151.97 2000::1:2:3:97	20% Load

Figure 3.2: The Handlespace

necessary to ensure uniqueness of PR IDs[2]. A PR maintains a complete copy of the operation scope's handlespace. PRs of an operation scope synchronize their view of the handlespace using the Endpoint HaNdlespace Redundancy Protocol (ENRP, defined in Xie et al. (2006), Stewart, Xie, Stillman and Tüxen (2006b)). Older versions of this protocol use the term Endpoint Namespace Redundancy Protocol. This naming has been replaced to avoid confusion with DNS, but the abbreviation has remained. Due to handlespace synchronization by ENRP, PRs of an operation scope are functionally equal. That is, if any of the PRs fails, each other PR is able to seamlessly replace it.

Using the Aggregate Server Access Protocol (ASAP, defined in Stewart, Xie, Stillman and Tüxen (2006a,b)), a PE can add itself to or remove itself from a pool by requesting a registration or deregistration at an arbitrary PR of the operation scope. In case of successful registration, the PR chosen for registration becomes the PE's *Home-PR* (PR-H). A PR-H not only informs the other PRs of the operation scope about the registration or deregistration of its PEs, it also monitors the availability of its PEs by ASAP Endpoint Keep-Alive messages. Such a monitoring message has to be acknowledged by the PE within a certain time interval. If the PE fails to answer within a certain timeout, it is assumed to be out of service and immediately removed from the handlespace. Furthermore, a PE is expected to re-register regularly. At a re-registration, it is also possible for the PE to change its list of transport addresses or its policy information (to be explained later).

To use the service of a pool, a client – called *Pool User* (PU) in RSerPool terminology – first has to request the resolution of the pool's PH to a list of PE identities at an arbitrary PR of the operation scope. This selection procedure is denoted as *Handle Resolution*. If the requested pool is existing, the PR will select a list of PE identities according to the pool's *Pool Member Selection Policy*, also simply denoted as *Pool Policy*.

Possible pool policies are e.g. a random selection (Random) or the least-loaded PE (Least Used). While in the first case it is not necessary to have any selection information (PEs are selected randomly), it is required to maintain up-to-date load information in the second case of selecting the least-loaded PE. This case is also illustrated in figure 3.2. Using an appropriate selection policy, it is e.g. possible to equally distribute the request load onto the pool's PEs. Details will be presented in section 3.11.

[2]As discussed in section 3.2.1 of Xie et al. (2006), the chance of identical PR IDs is very low when good random numbers are used. Furthermore, no severe consequence of such a conflict has been identified. Therefore, the complexity of a PR ID negotiation in ENRP has been omitted.

After reception of a list of PE identities from a PR, a PU will write the PE information into its local cache. This cache is denoted as *PU-side Cache*. Out of its cache, the PU will select exactly one PE – again using the pool's selection policy – and establish a connection to it using the protocol of the application, e.g. HTTP over SCTP or TCP in case of a web server. Using this connection, the service provided by the server is used. If the establishment of the connection fails or the connection is aborted during service usage, a new PE can be selected by repeating the described selection procedure. If the information in the PU-side cache is not outdated, a PE identity may be directly selected from the cache, skipping the effort of asking a PR for handle resolution. After re-establishing a connection with a new PE, the state of the application session has to be re-instantiated on the new PE. The procedure necessary for session resumption is denoted as *Failover Procedure* and is of course application-specific. For an FTP download for example, the failover procedure could mean to tell the new FTP server the file name and the last received data position. By that, the FTP server will be able to resume the download session. Since the failover procedure is highly application-dependent, it is not part of RSerPool itself, though RSerPool provides far reaching support for the implementation of arbitrary failover schemes by its Session Layer mechanisms explained in subsection 3.9.5.

To make it possible for RSerPool components to configure automatically as requested in section 3.2, PRs can announce themselves via UDP over IP multicast (details can be found in subsection 3.9.4 and subsection 3.10.2). These announces can be received by PEs, PUs and other PRs, allowing them to learn the list of PRs currently available in the operation scope. The advantage of using IP multicast instead of broadcast is that this mechanism will also work across routers (e.g. LANs connected via a VPN) and the announces will – in case of e.g. a switched Ethernet – only be heard and processed by stations actually interested in this information. If IP multicast is not available, it is of course possible to statically configure PR addresses.

3.4 A Migration Path for Legacy Applications

RSerPool is a completely new protocol framework. To make it possible for existing specialized or proprietary server pooling solutions to iteratively migrate to a RSerPool-based solution, it is mandatory to provide a migration path. For clients without support for RSerPool, the RSerPool concept (see figure 3.1) provides the possibility of a *Proxy Pool User* (PPU). A PPU handles requests of non-RSerPool clients and provides an intermediation instance between them and the RSerPool-based server pool. In subsection 3.6.4, a detailed example will be provided. From a PE's perspective, PPUs behave like regular PUs.

Similar to a PPU allowing the usage of a non-RSerPool client, it is possible to install a *Proxy Pool Element* (PPE) to continue using a non-RSerPool server in a RSerPool environment.

3.5 The Protocol Stack

Figure 3.3 shows the protocol stack of PR, PE and PU. The ENRP protocol is only used for the handlespace synchronization among PRs, all communications between PE and PR (registration, reregistration, deregistration, monitoring) as well as between PU and PR (handle resolution, failure reporting) is based on the ASAP protocol. The failover support, based on an optional Session Layer between PU and PE (to be explained in detail in subsection 3.9.5), is also using ASAP. In this case, the ASAP protocol data (Control Channel) is multiplexed with the application protocol's data (Data Channel) over the same connection. Using the Session Layer functionality of ASAP, a pool can be viewed as a single, highly available server from the PU's Application Layer perspective. Failure

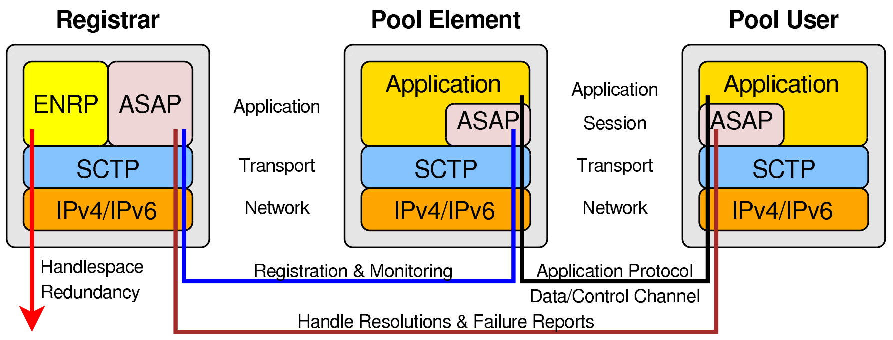


Figure 3.3: The RSerPool Protocol Stack

detection and handling are mainly performed automatically by the Session Layer, transparently for the Application Layer.

The transport protocol used for RSerPool is in most cases SCTP (see subsection 2.4.3). The important properties of SCTP requiring its usage instead of TCP are the following:

- Multi-homing and path monitoring by heartbeat messages for improved availability (see subsubsection 2.4.3.4) and verification of transport addresses (see subsection 3.9.2 for details);
- Address reconfiguration (Add-IP, see subsubsection 2.4.3.7 and Ramalho et al. (2006)) to enable mobility and interruption-free address changes (e.g. adding a new network interface for enhanced redundancy);
- Message framing (see subsubsection 2.4.3.6) for simplified message handling (especially for the Session Layer, to be explained in subsection 3.9.5);
- Security against blind flooding attacks, due to the 4-way handshake and the verification tag (see subsubsection 2.4.3.3) and
- Protocol identification by PPID (see subsubsection 2.4.3.6) for protocol multiplexing (required for the ASAP Session Layer functionality, to be explained in subsection 3.9.5).

For the transport of PR announces by ASAP (explained in subsection 3.9.4) and ENRP (explained in subsection 3.10.2) via IP multicast, UDP is used as transport protocol. The usage of SCTP is mandatory for all ENRP communication among PRs and for the ASAP communication between PEs and PRs. For the ASAP communication between PU and PR as well as for the Session Layer communication between PE and PU, it is recommended to use SCTP – but the usage of TCP together with an adaptation layer defined in Conrad and Lei (2005b) is possible. This adaptation layer adds functionalities like heartbeats, message framing and protocol identification on top of a TCP connection. Nevertheless, some important advantages of SCTP are missing – especially the high immunity against flooding attacks (see subsubsection 2.4.3.3) and the multi-homing property (see subsubsection 2.4.3.4). The only meaningful reason to use TCP is if the PU implementation cannot be equipped with a SCTP stack, e.g. if using a proprietary embedded system providing a TCP stack only.

3.6 The Application Scenarios

In this section, the application scenarios of RSerPool are explained.

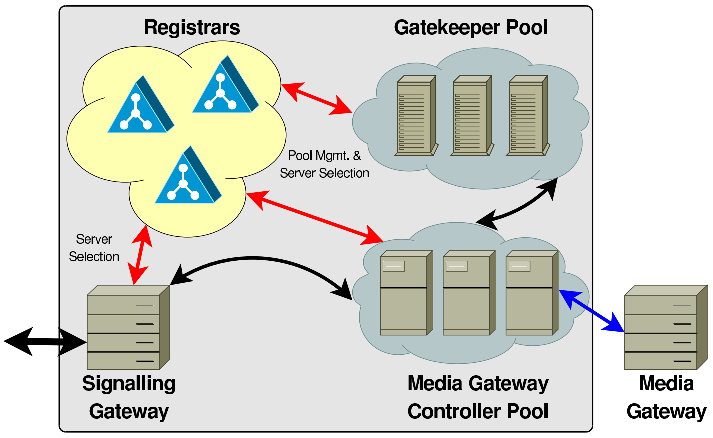


Figure 3.4: Telephone Signalling with Separated GK and MGC

3.6.1 Telephone Signalling

Telephone signalling over IP networks using the SCTP protocol is the application scenario that originally motivated the development of RSerPool. A Public Switched Telephone Network (PSTN) defines strict requirements for the availability of the used components. If there is a failure of signalling components, a stable system state must be reached again within only a few hundreds of milliseconds (see also Gradischnig and Tüxen (2001)). To cope with these requirements for the transport of telephone signalling (SS7 protocol) over IP networks, the SCTP protocol (see subsection 2.4.3) has been developed. Its features multi-homing and path monitoring protect a signalling network against network problems, but it cannot improve the availability of the signalling endpoints themselves. The protection against component failures by redundancy has been the motivation for RSerPool. In the following, a short introduction to the application of RSerPool for telephone signalling is given. Details can be found in Tüxen et al. (2006).

In telephone networks, signalling and media transport (e.g. a telephone call) are handled separately. A *Network Gateway* (NGW) in a telephone network usually consists of *Signalling Gateways* (SG), *Media Gateway Controller* (MGC), *Media Gateways* (MG) and *Gatekeeper* (GK). The SG receives signalling information (e.g. a call setup) and transfers it to the MGC, which is responsible for the connection signalling. It can ask the GK for access control (e.g. “May this user establish a call to that destination?”) and address resolution (e.g. a 0800 number to the actual endpoint). After that, the MGC can send the signalling messages to the next NGW. Furthermore, the MGC ensures that a MG is configured to handle the data stream (e.g. to convert ISDN voice data to a VoIP format for a NGW between ISDN and VoIP networks).

Obviously, MGC and GK have very high availability requirements and a protection against failures is mandatory. Using RSerPool, these components are realized by pools as illustrated in figure 3.4. For a SG, it is therefore first necessary to perform a handle resolution and server selection as explained in section 3.3. That is, a SG becomes a PU and the MGCs of the MGC pool are the PEs.

But GK and MGC are both realized as pools. That is, from the perspective of a MGC PE, it is itself a PU of the GK pool. On the other hand, a GK PE is itself a PU of the MGC pool. In this symmetric case, a GK can replace a failed MGC just like a MGC can replace a failed GK by another PE of the opposite pool. For the RSerPool architecture, the symmetric case only has importance for the communication between PE and PU. This will be explained in detail in subsubsection 3.9.5.3.

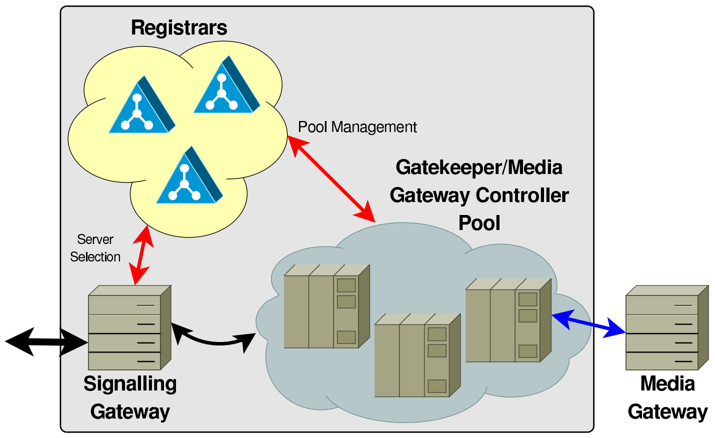


Figure 3.5: Telephone Signalling with Combined GK and MGC

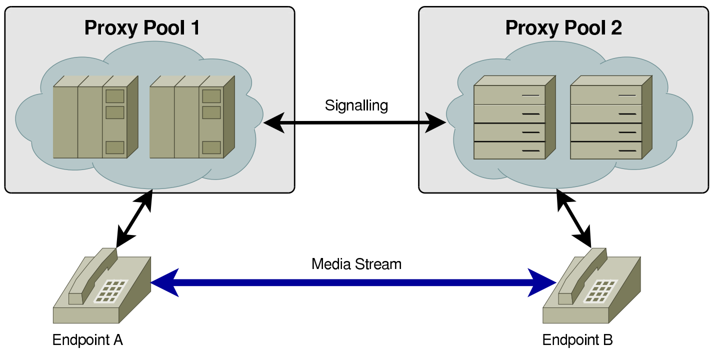


Figure 3.6: RSerPool-based SIP Proxies

A special case of the described scenario is shown in figure 3.5: here, MGC and GK functionalities are combined into a single component – again made redundant by RSerPool. This results in a simplified management of the redundancy.

3.6.2 Session Initiation Protocol (SIP)

Just like the usage of RSerPool for classical telephone signalling via SS7 over IP, it is also possible to apply RSerPool mechanisms to SIP-based communications (see Rosenberg et al. (2002)) like Voice over IP (VoIP). Services like connection signalling, user localization, accounting and forwarding are realized by SIP proxies. Clearly, these proxies are crucial for the operation of the system. In order to achieve an availability comparable to classical telephone systems (e.g. to forward emergency calls), it is mandatory to keep these components redundant. The application scenario for RSerPool-based SIP communications is presented in figure 3.6: SIP proxies consist of a pool of redundant SIP proxy PEs. For more details, see Conrad et al. (2002), Renier et al. (2005), Bozinovski, Gavrilovska, Prasad and Schwefel (2004), Bozinovski (2004), Bozinovski, Schwefel and Prasad (2004), Bozinovski, Gavrilovska and Prasad (2003), Bozinovski, Renier, Schwefel and Prasad (2003).

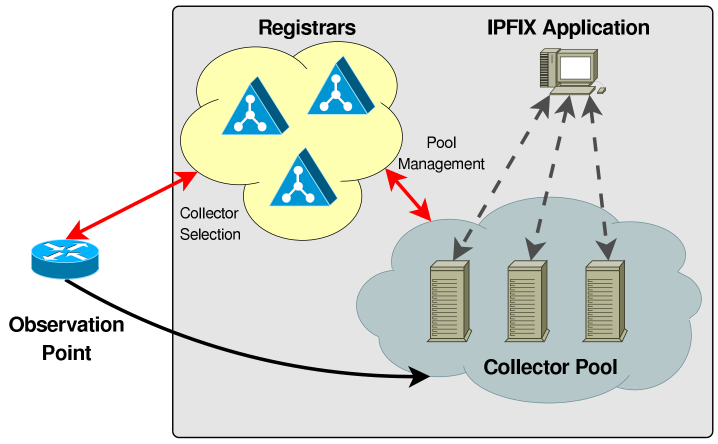


Figure 3.7: IPFIX with RSerPool

3.6.3 IP Flow Information Export (IPFIX)

In the IP Flow Information Export (IPFIX) framework defined in Sadasivan et al. (2006), so called *Observation Points* in a network gather information about currently running flows and forward them to so called *Collector Points*. The collector points provide the stored flow information data to applications, e.g. for the analysis of the network utilization.

Observation points may e.g. be routers, which usually possess only a limited amount of memory. Therefore, if a collector point becomes unreachable, it is mandatory to find another one quickly. Otherwise, the collector will not be able to store any more flow information and data loss will occur. Furthermore, it should be ensured that observation points distribute equally over the set of collector points, in order to avoid overloading a single collector while others remain idle.

Both requirements – a quick failover to a new collector point and balancing of the observation point load – can be fulfilled by the RSerPool architecture. In Dreibholz, Coene and Conrad (2006), the concept of using RSerPool for IPFIX is described. This concept is also illustrated in figure 3.7: collector points are PEs of a collector pool, observation points are in the role of PUs. An observation point PU can choose a collector by handle resolution and server selection as described in section 3.3. An appropriate pool policy, e.g. Round Robin or Least Used, can ensure that the observation point load is distributed equally over the set of collector PEs.

3.6.4 Load Balancing

Load balancing is a RSerPool application scenario currently very actively discussed by the IETF RSerPool WG. It is described in Tüxen et al. (2006), Coene et al. (2004). Figure 3.8 illustrates the concept of load balancing using RSerPool for the example of a web server pool: the load caused by the clients (web browsers) has to be balanced equally among the web servers of the pool. A *Load Balancer* component realizes this distribution using RSerPool methods. This distribution is completely transparent for the web browser (client). The users can continue using their existing web browser software, while the load balancer component adopts the role of a PPU (Proxy PU, see also section 3.4).

Currently, it is under discussion by the IETF RSerPool WG to combine the load balancer protocol SASP (Server/Application State Protocol) with the RSerPool architecture. The SASP protocol has

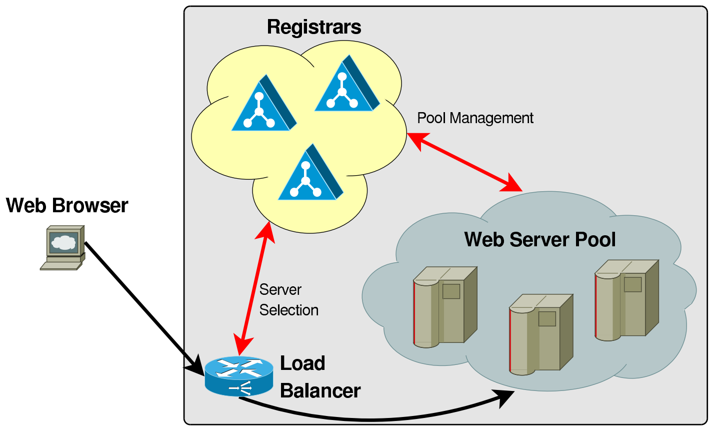


Figure 3.8: Load Balancing with RSerPool

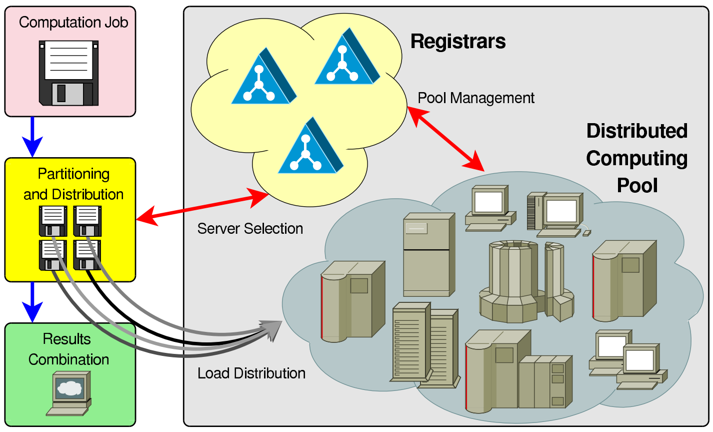


Figure 3.9: Distributed Computing with RSerPool

been developed by IBM and described as Internet Draft in Bivens (2006).

3.6.5 Real-Time Distributed Computing

Another application scenario having some overlap with load balancing is Distributed Computing. Ideas for this application scenario have been described in Dreibholz and Tüxen (2003), Zhang (2004), Dreibholz, Rathgeb and Tüxen (2005) and refined in Dreibholz and Rathgeb (2005e,c,d). The concept of using RSerPool for real-time distributed computing also has been described as Internet Draft in Dreibholz (2006a). Figure 3.9 illustrates this concept: at the user side, a large computation request is generated. For example, this could be a simulation (being composed of multiple independent runs) or the rendering of an animation sequence. The generated computation request is split up into smaller parts. These parts are distributed by parallel PU instances among the computation PEs of the pool using RSerPool mechanisms. The processed partial results are finally combined to the result of the complete computation.

Unlike already existing applications like SETI@HOME (see SETI Project (2003)) – where computation jobs are downloaded by clients from a central server and their completed result may be uploaded to the server possibly some days later – the idea of real-time distributed computing is proposed. That is, if a computation request cannot be handled in a propper manner (e.g. timely), an immediate failover to another PE is performed.

3.6.5.1 Requirements

The basic idea of distributed computing (see also Davies (2004)) is to utilize currently unused computation capacity for useful things. For example, this could mean to use the capacity of office PCs during night-time and if their users do not utilize them. But when a PC's capacity is required back by its user, it has to be immediately revoked from the computation pool. That is, pools may be very dynamic.

Furthermore, computation requests for distributed computing usually have a long runtime of many minutes or hours. If a computation server becomes unavailable, an appropriate failover mechanism has to ensure that already computed partial results do not have to be re-computed again. Instead, a computation session should be resumed by another PE. This is particularly relevant if the distributed computing capacity is provided by user PCs: unlike servers in a computing centre, they have a significantly lower availability. For example, a user could simply turn the PC off or the network connection could break.

Finally, distributed computing pools can be very large, consisting of many hundreds or even thousands of computation servers. For example, a large company could decide to add all of its office PCs to a simulation pool. In such pools, the probability of highly differing capacities is very likely. That is, there are very heterogeneous computation servers and the server selection procedure has to incorporate this aspect to effectively utilize the pool.

3.6.5.2 Applicability of RSerPool

RSerPool is equipped with all functionalities being necessary to fulfil the described requirements: By providing the possibilities for registration and deregistration, highly dynamic pools can be realized. The features for the automatic configuration also make setting up computation server processes easy (see subsubsection 3.7.1.1). RSerPool allows the implementation of arbitrary failover procedures (see subsection 3.9.5) to cope with the necessity of session resumption. Furthermore, it provides monitoring of registered PEs (see subsubsection 3.7.1.3), so that failed elements are quickly removed from the handlespace (e.g. when a PC is turned off without deregistering from the pool). Due to its simple handlespace management (flat, i.e. without hierarchy) and the possibility for the definition of special server selection policies, RSerPool is also able to handle large pools of heterogeneous resources. For further details on this subject, see Dreibholz and Rathgeb (2005b), Dreibholz (2004d).

3.6.6 Mobility Support for SCTP

As already explained in subsection 2.4.3, the SCTP protocol in combination with its Dynamic Address Reconfiguration extension (Add-IP, see Ramalho et al. (2006)) provides the possibility to add transport addresses to and remove them from a running association. Using this feature, it is possible for a mobile endpoint to perform a handover into a new network (Mobile SCTP, see Riegel and Tüxen (2006)). During handover, the following two cases can occur:

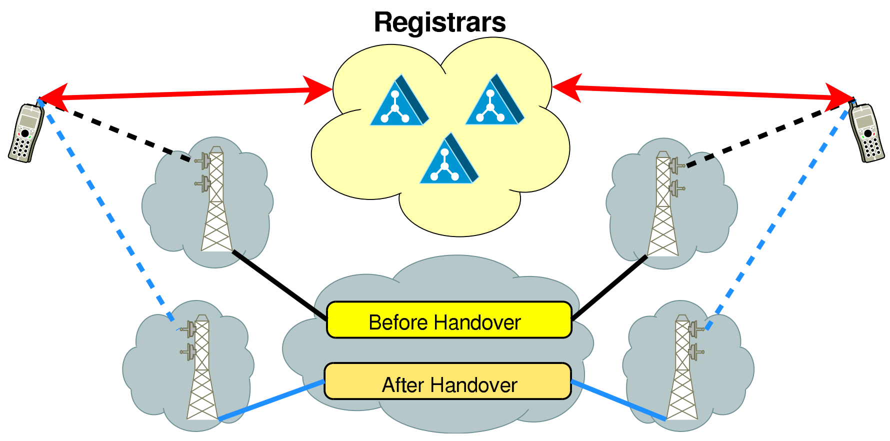


Figure 3.10: SCTP Mobility with RSerPool

Break before Make: The connection to the old network has been broken before a new network is reached. Using a radio connection, this would mean that the current network gets out of reach before a new network can be contacted.

Make before Break: A new network can already be contacted while the connection to the current network is still working. For a radio connection, this would mean that a new network is already usable while the old one is still in reach.

As long as there is always at most one endpoint in a handover procedure using "break before make", Add-IP will work as expected: the endpoint being disconnected from all networks can tell the other side its new address just after being connected to the new network. But if both endpoints perform a "break before make" handover simultaneously, neither one can inform its peer about its address change – no endpoint knows under which new address its peer is reachable.

In Dreibholz, Jungmaier and Tüxen (2003), we have proposed to utilize RSerPool for the storage of transport addresses and validated its applicability using our prototype implementation. The result of this work has been the Internet Draft Dreibholz and Pulinthanath (2006). An illustration of the proposed mobility approach is shown in figure 3.10: at least one endpoint has to register as PE under a PH known to its peer. As soon as the PE's transport addresses change (i.e. in case of a handover), it has to perform a re-registration that propagates the address change into the handlespace. Then, the peer endpoint can – in the role of a PU – ask a PR to resolve the PH into the new transport address of the PE endpoint and use the mechanisms of Add-IP to continue the association.

3.6.7 Other Application Scenarios

Finally, there are some further application scenarios for RSerPool, which will only be summarized shortly: Dreibholz (2002) describes an e-commerce scenario, where an e-shop service is provided to users. If one of the PEs providing the shop service fails, RSerPool ensures a failover to another one – transparently for the customer. The failover mechanism described in subsubsection 3.9.5.2 furthermore makes the failover handling transparent for the Application Layer on the user side.

In Uyar, Zheng, Fecko, Samtani and Conrad (2003), Uyar et al. (2004), Uyar, Zheng, Fecko and Samtani (2003a,b), the usage of RSerPool for Future Combat Systems (FCS) in battlefield networks is

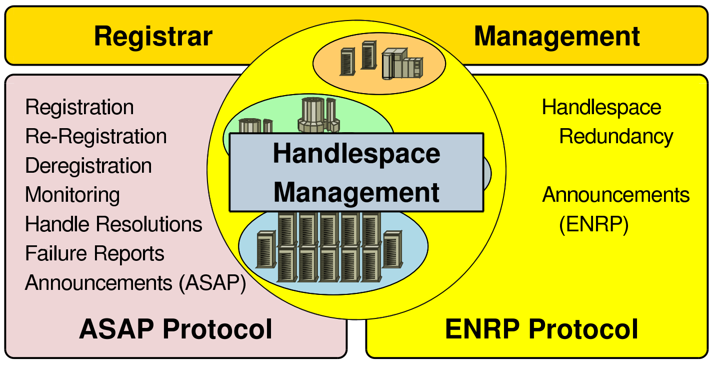


Figure 3.11: The Building Blocks of a Registrar

suggested. That is, mobile and non-mobile PEs provide military services. The RSerPool mechanisms ensure a continuous service, despite of servers being targets of destruction.

3.7 The RSerPool Components

In this section, the functionalities and behaviours of the RSerPool components PR, PE and PU are described. In particular, the interaction among ASAP, ENRP and the component functionalities is in the focus of this section. ASAP and ENRP messages and the detailed functionalities of both protocols are explained in the subsequent sections 3.9 (ASAP protocol) and 3.10 (ENRP protocol).

3.7.1 Registrar

The functionalities of a PR are

1. The transmission of announces for automatic configuration,
2. Registration, re-registration and deregistration of PEs,
3. Monitoring the availability of PEs by keep-alive messages,
4. Handle resolution (i.e. PE selection by pool policy) for PUs,
5. Handlespace consistency audit and synchronization among the PRs of the operation scope and
6. Takeover of the PR-H functionality for PEs of a failed PR.

The building blocks of a PR are illustrated in figure 3.11. Clearly, the local handlespace copy is the central element of a PR. Using the ENRP protocol (described in section 3.10), this handlespace copy is kept in consistence with the other PRs of the operation scope. The ASAP protocol (described in section 3.9) is used to communicate with PEs and PUs. A PR management layer above both protocols coordinates the access to the handlespace copy. The necessary PR management functionality of this layer is described in the following.

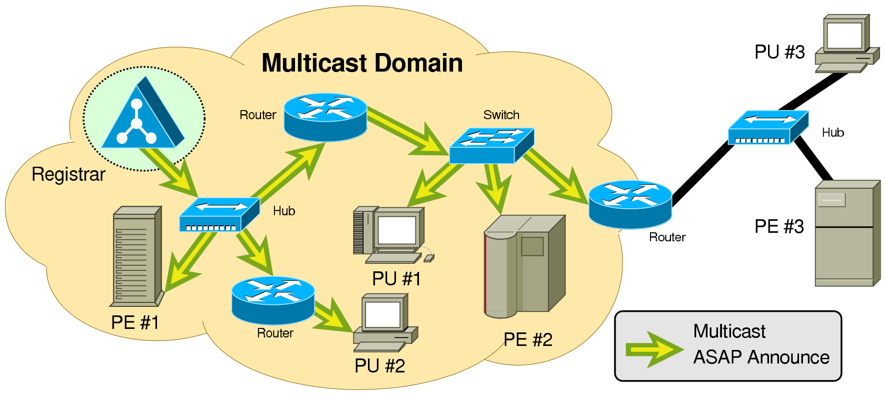


Figure 3.12: Automatic Configuration of ASAP

3.7.1.1 Announces

For a PR to be automatically found by other PRs as well as PEs and PUs, it can send out ENRP and ASAP announce messages via UDP over IP multicast. Figure 3.12 illustrates the concept by using the example of ASAP for the configuration of PEs and PUs; for detailed information about the ENRP configuration behaviour for PRs, see subsection 3.10.2. IP multicast is used to allow announces across routers. Furthermore, a router or switch only forwards the announces into segments where stations are really interested in their reception (by having joined into a certain multicast group). This saves both, bandwidth and processing power. In the example shown, the stations within the multicast domain (PE #1 and PE #2 as well as PU #1 and PU #2) are automatically informed about the existence of the PR.

If multicast is not possible in the network or in certain parts of it (e.g. due to security policies that do not allow multicast), there is always the possibility to statically configure PR addresses into other PRs as well as PEs and PUs. In this case, of course, it is not any longer possible to simply turn on a component without any further configuration effort. In the example scenario, this is the case for PE #3 and PU #3.

3.7.1.2 Pool Management

A main task of the PR is to register PEs into the handlespace, re-register them (i.e. update their registration information) and finally deregister them from the handlespace on request via ASAP. In order to register a PE into the handlespace, the following information is required:

Pool Handle: The PH of the pool into which the PE desires to register.

PE ID: A 32-bit random number chosen by the PE as its identification number within the pool.

Transport Addresses: The list of transport addresses under which the PE is reachable for its application. This includes the Network Layer protocol addresses (usually IPv4 and/or IPv6) as well as the used transport protocol (e.g. SCTP, TCP or UDP) and the Transport Layer protocol's port number (e.g. SCTP port).

Policy Information: A description of the pool's policy (e.g. Least Used or Round Robin) along with the PE state information necessary for the specified policy. If the policy is Least Used for example, the PE's current load state has to be given. For details on pool policies, see section 3.11.

Before the PE can be registered into the pool, multiple verification steps are necessary:

1. It is necessary to check whether the PE is permitted to register (authentication and authorization).

2. It has to be verified which Network Layer addresses of the PE's given transport addresses are actually part of the SCTP association between PR and PE. Due to the multi-homing property of SCTP (see subsubsection 2.4.3.4), there should be a SCTP path to each of the PE's provided Network Layer addresses if it is really reachable under the corresponding address. Due to wrong configuration settings, a PE could e.g. specify its localhost address (e.g. 127.0.0.1 for IPv4 or ::1 for IPv6) or a private address (e.g. 192.168.x.y), which is unreachable for the PR. All Network Layer addresses not being part of the SCTP association are useless and therefore dropped from the PE information. If there are no remaining addresses from the intersection of the sets of given transport addresses and addresses being part of the SCTP association, the PE's registration request is rejected.

3. After address verification, a PR has to check if the PE's requested pool is already existing. If this is the case, it has to be checked whether the PE's requested policy settings (policy type and the information required by the policy) are consistent with the pool's settings. If an incompatibility is detected, the registration request is denied. For example, it is not possible to register a PE requesting Round Robin selection into a pool using the Least Used policy.

4. All PEs of a pool must use the same Transport Layer protocol for the transport of application data (e.g. SCTP or TCP). If the PE's settings are incompatible to the existing pool, the PR rejects the registration. For example, a TCP-based PE cannot be registered into a pool of SCTP-based PEs.

5. If the requested pool is not existing, it is created using the policy requested by the PE.

If all steps have been passed, the registration is granted and the PE is finally added to the pool.

Upon successful registration into the handlespace, the registering PR becomes the PR-H of the newly registered PE. By using an update message via ENRP (to be explained in detail in subsection 3.10.4), it tells all other PRs of the operation scope about the new registration. These PRs will also add the new PE into their local handlespace copies.

A re-registration is equivalent to a registration. The PE information in the handlespace is updated analogously to a registration. This especially means that again a verification of the addresses, pool policy consistency and protocol compatibility has to be made. The update is – analogously to a registration – propagated by the PR-H to the other PRs of the operation scope by using an update message via ENRP.

To perform a deregistration, it is sufficient for the PE to specify its PH and PE ID. The PR-H distributes the information about the deregistration to the other PRs, again by using an ENRP update message.

3.7.1.3 Pool Monitoring

Another important task of a PR is the monitoring of all PEs for which it has the role of a PR-H. This functionality is illustrated in form of a message sequence diagram in figure 3.13: A PE has been

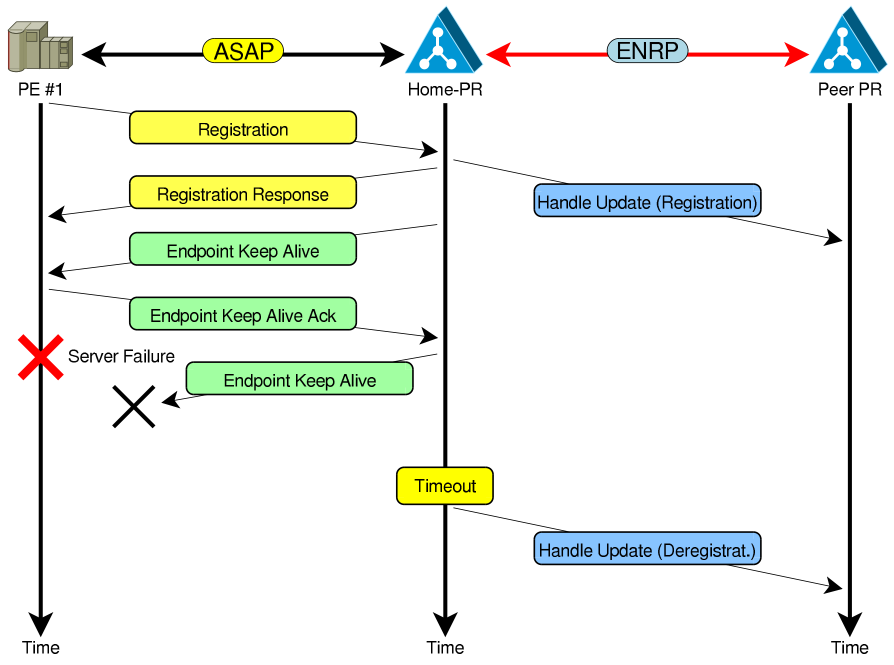


Figure 3.13: The Registration and Monitoring of a Pool Element

registered at a PR (using an ASAP Registration message, the PR confirmed the granted request by an ASAP Registration Response message; to be explained in detail in subsubsection 3.9.2.1). Due to successful registration, the PR became the PE's PR-H. It propagates the registration to the other PRs of the operation scope using an ENRP Update message.

Furthermore, the PR-H starts to send ASAP Endpoint Keep-Alive messages to its "owned" PEs in regular intervals (see subsubsection 3.9.2.2). A PE is required to answer such a message by an immediate ASAP Endpoint Keep-Alive Ack, which has to be received by the PR within a given time interval. Upon reception, the PR assumes that the PE is still reachable and working properly. If the timeout for the reception of the keep-alive acknowledgement expires, the PR removes the PE from its handlespace copy. Furthermore, it tells the other PRs of the operation scope that the PE has been removed (using an ENRP Update message).

Since the connection between PR and PE is using SCTP, and SCTP itself also monitors the reachability of its peer endpoints by SCTP heartbeat messages (see subsubsection 2.4.3.4), a PE failure may be detected already by the Transport Layer itself. In this case of course, the PE can be immediately removed from the handlespace without using the ASAP Endpoint Keep-Alive mechanism. This double safeguarding – on Transport Layer by SCTP and on Application Layer by ASAP – has the following reason: The SCTP protocol is usually implemented in the kernel of the PE's operating system, but ASAP is assumed to be realized as part of the application (e.g. as a shared library) in userspace. While the application could become unusable due to a Denial of Service attack or simply a programming bug or configuration mistakes, the kernel's SCTP implementation may still correctly answer all SCTP heartbeat messages – although the application itself has stopped working. Due to the Application Layer keep-alives of ASAP, the application itself is forced to respond. This leads to a maximum operational safety for the PEs listed in the handlespace.

3.7.1.4 Server Selection and Failure Reporting

For PUs, a PR provides the resolution of a PH into a list of PE identities selected from the handlespace according to the selection policy of the pool. A detailed description of certain pool policies is given in section 3.11.

The monitoring of PEs by their PR-H and the verification of the PE addresses by their SCTP association to the PR-H ensure a high degree of reliability for the provided handlespace information. Nevertheless, it may be possible that a PU cannot reach a PE any more. In this case, the PU should inform its PR about the unreachability of this PE (using an ASAP Endpoint Unreachable message, to be explained in detail in subsection 3.9.3). The PR keeps the number of unreachability reports for each PE in its handlespace management. If the threshold MAX-BAD-PE-REPORT is reached, the PE is removed from the handlespace. The default setting for this threshold is 3 (see section 4.2 of Xie et al. (2006)). If the PU's PR is also PR-H of the concerning PE, it could immediately send an ASAP Endpoint Keep-Alive message to verify its reachability.

3.7.1.5 Handlespace Audit and Handling of PR Failures

A further task of a PR is the audit of the handlespace consistency among the PRs of the operation scope, as well as handlespace synchronization to resolve inconsistencies. Furthermore, in case of a PR failure, the remaining PRs have to negotiate which PR takes over the PR-H functionality for the PEs of the failed PR. Both functionalities are strongly related to ENRP; therefore, they are explained in the description of ENRP in subsection 3.10.5 (handlespace audit and synchronization) and subsection 3.10.6 (takeover procedure).

3.7.2 Pool Element

Next to its main task, which is clearly the provision of its application, a PE has to perform registration, re-registration and finally deregistration. Furthermore, it has to answer ASAP Endpoint Keep-Alives from its PR-H (see subsubsection 3.7.1.3) to confirm its availability.

Before a PE can register at a PR, it has to choose a PE ID. That is, a good, non-zero, pseudo-random number of 32 bits has to be computed according to the guidelines defined in Eastlake et al. (1994) and section 4.2.1 of Xie et al. (2006). Furthermore, a connection to a PR has to be established. The addresses of possible PRs may be learned dynamically by the PRs' announce messages via IP multicast (see subsubsection 3.7.1.1) or can also be statically configured. After that, the PE can register into its desired pool (given by its PH) under its PE ID, with a list of transport addresses (including Network Layer addresses, transport protocol and port) and policy information.

In case of successful registration, the PE has to renew its registration regularly by performing a re-registration. Using a re-registration, it is also possible to renew the PE's list of transport addresses or policy information (e.g. to update the PE's load state in case of the Least Used policy). Furthermore, the PE is required to answer ASAP Keep-Alive messages by ASAP Keep-Alive Acks.

To protect a RSerPool system against PR failures, a double safeguarding is applied:

1. If the PR-H fails, the PE can simply use another PR for re-registration. Upon re-registration at another PR, this new PR automatically becomes the PE's new PR-H. Since the PE ID does not change on re-registration, the new PR knows that the registration request has to be handled as an update (i.e. the PR-H has changed).

2. If a PR failure is not immediately detected by the PE, the ENRP protocol provides features to negotiate the takeover of the PR-H functionality by other PRs of the operation scope (details of

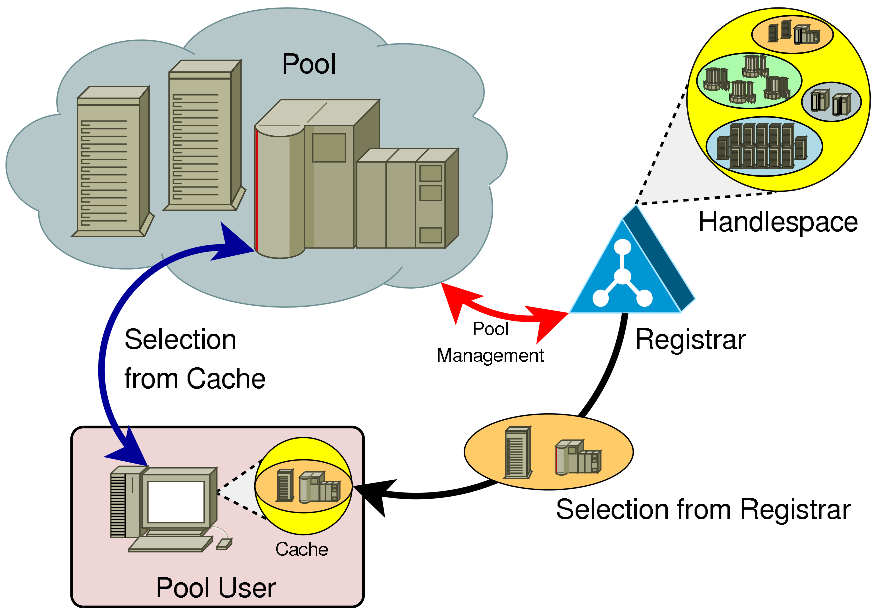


Figure 3.14: The Server Selection by Registrar and Pool User

the takeover procedure will be explained in subsection 3.10.6). In this case, the new PR-H tells the PE that it has become its PR-H using a special ASAP Endpoint Keep-Alive message (see subsubsection 3.9.2.2 for details). The PE then has to adopt the PR as its PR-H.

Another important task of a PE is to support the resumption of sessions started by another PE, as part of the failover procedure of the PU. That is, the new PE has to find out the session state of the old PE and seamlessly continue the application session. The realization of such mechanisms and their support by RSerPool are strongly related to the ASAP protocol. Therefore, they are introduced in detail as part of the ASAP description in subsection 3.9.5.

3.7.3 Pool User

In order to use the service of a pool, a PU first has to request the resolution of the pool's PH into a list of policy-selected PE identities from a PR. Analogously to a PE, it is necessary to connect to an arbitrary PR of the operation scope. The PR addresses are either dynamically learned by multicast announces (see subsubsection 3.7.1.1) or statically configured. After handle resolution at the PR, the PU writes the received list of selected PE identities into its local cache (the PU-side cache). An illustration of this process is given in figure 3.14. The cache is a local, temporary, partial copy of the handlespace. Its entries expire, if not updated by the results of a subsequent handle resolution at a PR, after a configured timeout. This timeout is called *Stale Cache Value*. Depending on the accuracy requirements of application and pool policy for the cached handlespace data (e.g. having up-to-date load information for the Least Used policy), the stale cache value can be configured smaller (more accurate, but more handle resolutions at a PR) or higher (less accurate, fewer handle resolutions at a PR). A stale cache value of zero specifies that the cache's content may only be used for the current server selection and is deleted afterwards. That is, each server selection mandatorily requires to query a PR. To finally connect to a PE, a single PE identity is selected out of the cache, again according to

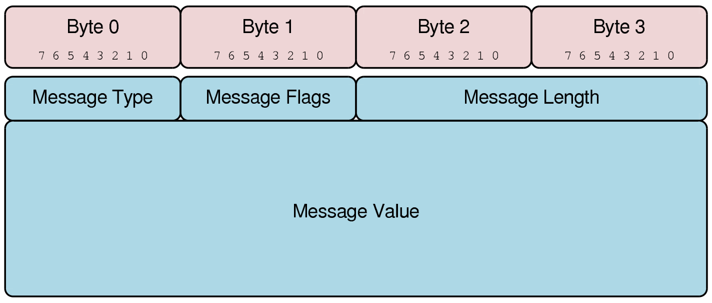


Figure 3.15: The Message Structure

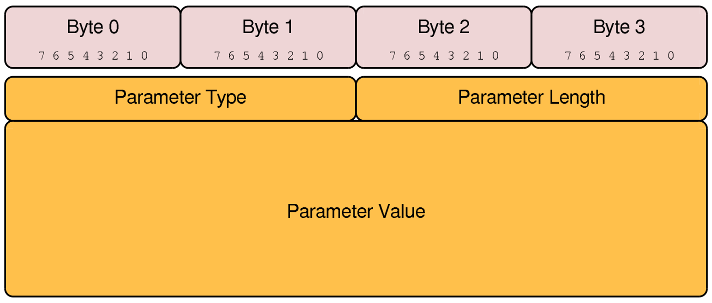


Figure 3.16: The Parameter Structure

the pool's policy. A transport connection to this selected PE can then be established, over which the PE's service can be used with the application protocol.

If the selected PE is unreachable or it fails during the usage of its service, the PU should notify its PR about the failure using an ASAP Endpoint Unreachable message (details will be explained in subsection 3.9.3). After that, a new PE has to be selected. Depending on the setting of the stale cache value, a new PE may be immediately selected from the cache. In this case, the overhead (bandwidth, time) of querying a PR is skipped. Otherwise, the complete procedure of at first querying a PR is repeated as explained above. After successfully establishing a connection to the new PE, this PE has to resume the interrupted session. Since the failover support mechanisms of RSerPool are strongly related to the ASAP protocol, a detailed depiction is provided as part of the ASAP description in subsection 3.9.5. Upon successful execution of the application-specific failover procedure, the application session is resumed.

3.8 The Protocol Design

Each ASAP and ENRP *Message* is structured as shown in figure 3.15. This structure is defined in Stewart, Xie, Stillman and Tüxen (2006b) and includes the following fields:

Message Type: The type number of the message.

Message Flags: Type-specific message flags.

Message Length: The length of the complete message, including its header.

Message Value: The type-specific content of the message. If necessary, the message is padded with zeros to the next larger size being a multiple of 4 bytes.

Since the common header structure (type, flags and length) is identical for every message type of the ASAP and ENRP protocols, the description of the protocols in the following sections 3.9 and 3.10 will not explicitly mention these fields.

The message content consists of a type-specific field set having a constant length. After that, *Parameters* of variable length may follow. Parameters are structured as shown in figure 3.16:

Parameter Type: The type of the parameter.

Parameter Length: The total length of the parameter, including its header.

Parameter Value: The type-specific content of the parameter. If necessary, the parameter is padded with zeros to the next larger size being a multiple of 4 bytes.

Due to the three fields which are common for all parameters, they are also denoted as TLV (type - length - value) blocks. This common structure allows an easy extension of the protocol: it is possible to add new parameters without changing every existing implementation. A protocol implementation confronted with unknown parameter types can simply skip them. The exact behaviour on the processing of unknown parameters is defined by the first two bits of the parameter type:

00: The message including the unknown parameter is ignored, the sender will not be notified.

01: The message is ignored, but the sender will get an error message.

10: The unknown parameter is silently skipped, i.e. the sender will not be informed.

11: The unknown parameter is skipped, but the sender will be notified.

For the notification on skipped messages, error messages (ASAP Error for ASAP and ENRP Error for ENRP) are defined. These messages also have to be used if no answer is generated in response to the received message. Otherwise, the response message can be extended by an Error Parameter including the skipped parameter(s). See Stewart, Xie, Stillman and Tüxen (2006b) for details on this parameter type.

3.9 The Aggregate Server Access Protocol

In this section, the Aggregate Server Access Protocol (ASAP), defined in Stewart, Xie, Stillman and Tüxen (2006a,b), is explained. First, an overview of ASAP's functionalities is given. The communication to fulfil these tasks is explained in detail afterwards.

3.9.1 Overview

The ASAP protocol is used for the communication between PR and PE or PU, as well as for the communication between PU and PE (Session Layer). For a classification into the protocol stack of PR, PE and PU see section 3.5. ASAP has the following tasks:

- Registration, re-registration and deregistration as well as monitoring of PEs by their PR-H for PEs,

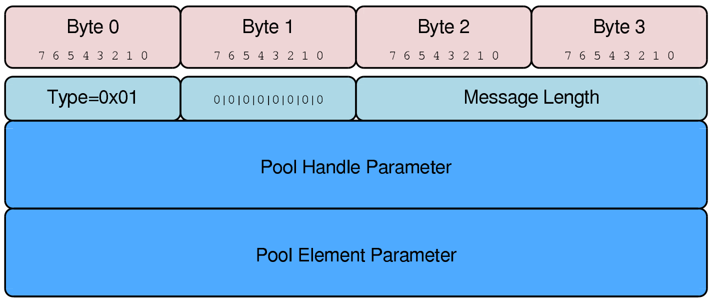


Figure 3.17: The ASAP Registration Message

- Handle resolution (i.e. server selection by pool policy) for PUs,
- Transmission of PR announces for the automatic configuration of PEs as well as of PUs and
- The support of failover procedures (Session Layer) between PUs and PEs.

The ASAP communication for these tasks is described in the following.

3.9.2 Pool Element Functionality

In this section, the PE functionalities of the ASAP protocol – registration, re-registration, monitoring and deregistration – are explained.

3.9.2.1 Registration and Reregistration

In order to register into a pool or to renew its registration by performing a re-registration, a PE sends an ASAP Registration message to an arbitrary PR of the operation scope as described in subsubsection 3.7.1.2. Figure 3.17 shows the structure of this message type. It includes the PH of the pool to register into in form of a Pool Handle Parameter and all information about the PE in form of a Pool Element Parameter.

The structure of the Pool Handle Parameter is depicted in figure 3.18, it simply includes the PH as byte vector. Figure 3.19 shows the structure of the Pool Element Parameter. It includes the following entries:

Pool Element Identifier: This field represents the PE ID of the PE to be registered.

Home Registrar Identifier: Here, the PR ID of the PE's PR-H is given. If it is unknown, 0 is specified.

Registration Life: The Registration Life parameter is a specification of the registration lifetime in milliseconds. For example, a value of 300,000ms means that the registration will be valid for at least 5 minutes. This value is also a measure of the PE's reliability and in particular used to calculate the frequency of re-registrations (to be explained below).

User Transport Parameter: This is the definition of the application's transport endpoint. Depending on the application's transport protocol, this is a SCTP Transport Parameter for SCTP, TCP Transport Parameter for TCP or UDP Transport Parameter for UDP. Depending on the protocol,

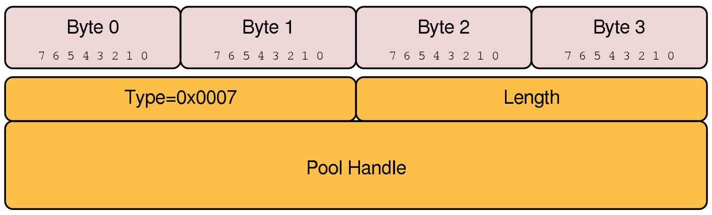


Figure 3.18: The Pool Handle Parameter

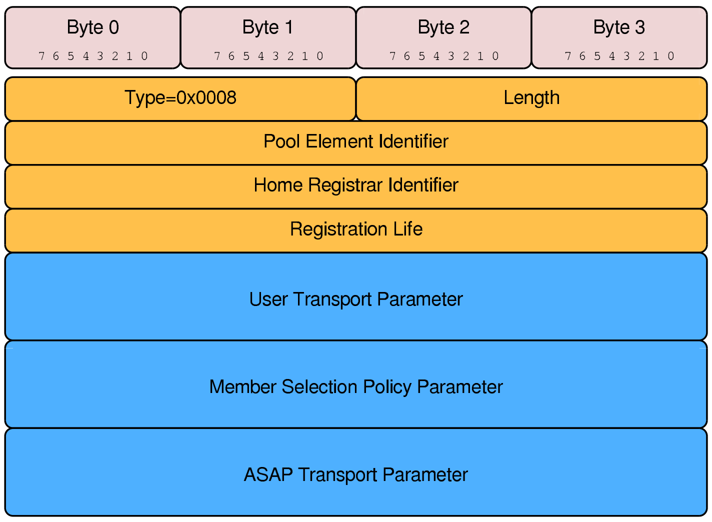


Figure 3.19: The Pool Element Parameter

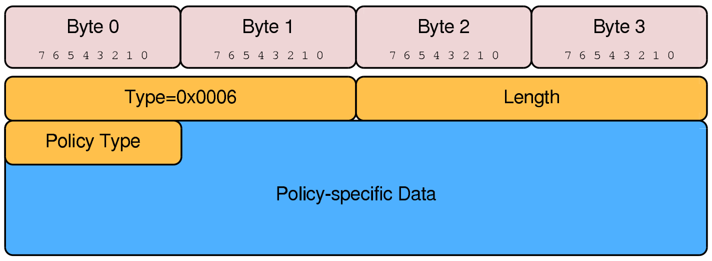


Figure 3.20: The Policy Parameter

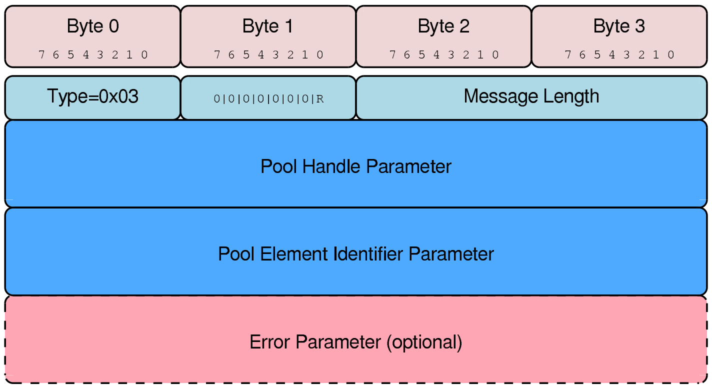


Figure 3.21: The ASAP Registration Response Message

this parameter includes one (TCP, UDP) or at least one (SCTP) Network Layer address (e.g. a set of IPv4 and IPv6 Address Parameters) as well as the Transport Layer protocol's port number. See Stewart, Xie, Stillman and Tüxen (2006b) for the definitions of these parameters. Furthermore, the SCTP and TCP Transport Parameters allow to specify a *Transport Use*. The connection between PU and PE may either consist of a Data Channel for the application protocol only (i.e. without RSerPool support for failover; Transport Use is 0x0000) or of a multiplexed Data and Control Channel (i.e. with an ASAP-based Session Layer to support a failover, to be explained in subsection 3.9.5; Transport Use is 0x0001).

ASAP Transport Parameter: The PE's ASAP endpoint transport address is specified here in form of a SCTP Transport Parameter. The specification of this address is necessary to allow a PR after a takeover procedure (see subsection 3.7.2) to tell the PE that it has become its new PR-H. The takeover procedure will be explained in detail in subsection 3.10.6.

Member Selection Policy Parameter: In this entry, the requested pool policy as well as its policy-specific parameters are provided in form of a Policy Parameter. The structure of the Policy Parameter is illustrated in figure 3.20; it includes at least the type of the policy (e.g. Round Robin or Least Used). The rest of the parameter is policy-specific and e.g. includes the PE's load state for the Least Used policy. Policies and their parameters will be explained in detail in section 3.11.

After sending a Registration message to a PR, a PE waits at most the period of time specified by the timer *T2-Registration*. The default value for this timer is 30s (see section 5.1 in Stewart, Xie, Stillman and Tüxen (2006a)). If no response from the PR is received within the timeout interval, a new PR has to be selected and the registration to be retried.

The PR's answer to a Registration message is a Registration Response message. Figure 3.21 presents the structure of this message type: it includes the PE's PH in form of a Pool Handle Parameter (see figure 3.18) and the PE's PE ID in form of a Pool Element Identifier Parameter (which consists of the PE ID only). Bit 0 of the message flags is denoted as the *Reject* flag. If it is set to 0, the registration request has been granted and successful; otherwise, it has been rejected. In case of a rejected registration, an optional Error Parameter may include the reasons for the rejection (e.g. no authorization, inconsistent policy settings or invalid transport addresses). See Stewart, Xie, Stillman and Tüxen (2006b) for details on this parameter type.

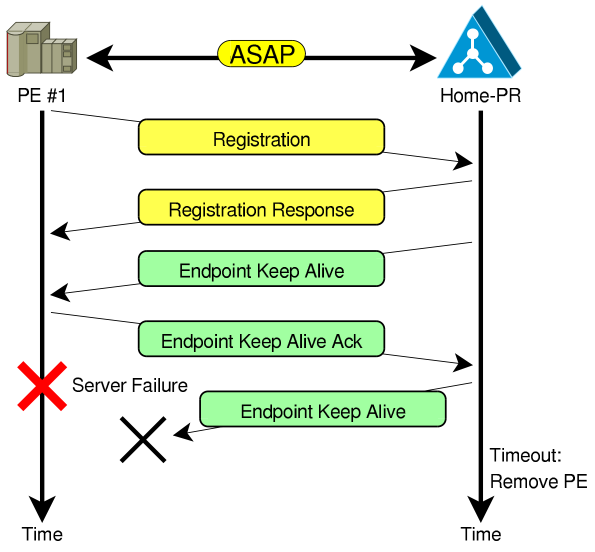


Figure 3.22: The Pool Element Monitoring by its Home Registrar

After successful registration, the timer *T4-Reregistration* has to be started. Its interval is defined in section 5.1 of Stewart, Xie, Stillman and Tüxen (2006a) and set to 10 minutes or the Registration Life minus 20 seconds, whichever value is smaller. Each time this timer expires, the registration is repeated as re-registration and the timer restarted.

3.9.2.2 Monitoring

After a PE has been registered into a pool, its availability is monitored by its PR-H using ASAP Keep-Alive messages, as illustrated in figure 3.22: in regular intervals, an Endpoint Keep-Alive is sent to the PE. The PE has to respond using an ASAP Endpoint Keep-Alive Ack message. If the PE fails to answer within a certain timeout, the PR-H removes it from the handlespace.

The Endpoint Keep-Alive message (see figure A.1 for an illustration) includes the PR ID of the sending PR and the PH of the PE's pool in form of a Pool Handle Parameter (see figure 3.18). Bit 0 of the flags field is denoted as *Home* flag. If it is set and the sending PR differs from the PE's current PR-H, the PE should adopt the sending PR as its new PR-H. This flag is used for the takeover procedure of failed PRs (see subsection 3.7.2; the takeover procedure itself is described in subsection 3.10.6).

The Endpoint Keep-Alive Ack message (see figure A.2 for an illustration) includes the PE's PE ID and the PH of the PE's pool in form of a Pool Handle Parameter (see figure 3.18).

3.9.2.3 Deregistration

To deregister from its pool, a PE sends an ASAP Deregistration message to its PR-H (see figure A.4 for an illustration of this message type). It includes the PH of the PE's pool in form of a Pool Handle Parameter (see figure 3.18) and the PE's PE ID in form of a Pool Element Identifier Parameter.

After transmission of a Deregistration message, a PE waits for a time interval defined by the timer *T3-Deregistration* for the reception of an ASAP Deregistration Response message. The default value for this timer is 30s (see section 5.1 in Stewart, Xie, Stillman and Tüxen (2006a)). If no response of

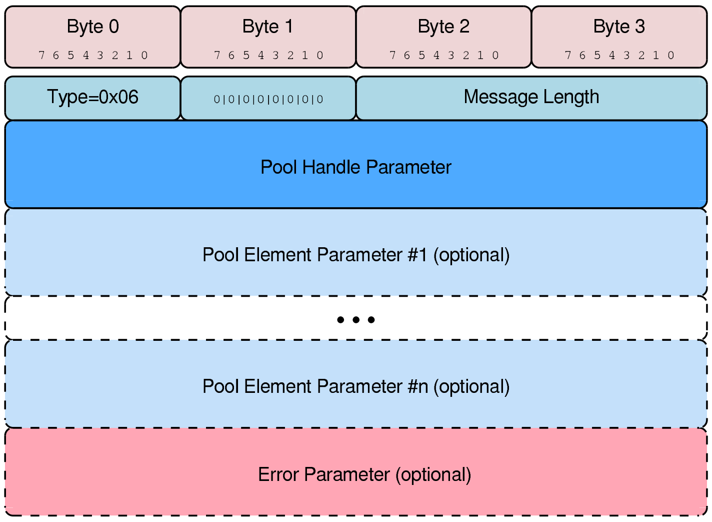


Figure 3.23: The ASAP Handle Resolution Response Message

the PR is received within the timeout interval, a new PR has to be selected and the deregistration to be retried.

The Deregistration Response message (see figure A.5 for an illustration of this message type) includes the PH of the PE's pool in form of a Pool Handle Parameter (see figure 3.18) and the PE's PE ID in form of a Pool Element Identifier Parameter. Furthermore, an optional Error Parameter is included if the deregistration has been rejected. In this case, the contents of the Error Parameter give the reasons for the rejection. See Stewart, Xie, Stillman and Tüxen (2006b) for details on this parameter type.

3.9.3 Pool User Functionality

In this section, the PU functionalities of the ASAP protocol – handle resolution and failure reports – are explained.

3.9.3.1 Handle Resolution

The main functionality of the PR provided to PUs is handle resolution including the PE selection by policy as explained in subsection 3.7.3. In order to request a handle resolution, a PU sends an ASAP Handle Resolution message to an arbitrary PR of the operation scope. This message type (see figure A.9 for an illustration of the structure) simply includes includes the PH of the requested pool in form of a Pool Handle Parameter (see figure 3.18).

After sending a Handle Resolution message to a PR, a PU waits at most the period of time specified by the timer *T1-ENRPRequest*. The default value for this timer is 15s (see section 5.1 in Stewart, Xie, Stillman and Tüxen (2006a)). If no response of the PR is received within the timeout interval, a new PR has to be selected and the handle resolution to be retried.

The PR's response to a handle resolution request is an ASAP Handle Resolution Response message (see figure 3.23 for an illustration of this message type). It includes the PH of the requested pool in form of a Pool Handle Parameter (see figure 3.18). Furthermore, if the handle resolution has been successfully processed, the response includes a list of selected PE identities in form of Pool Element

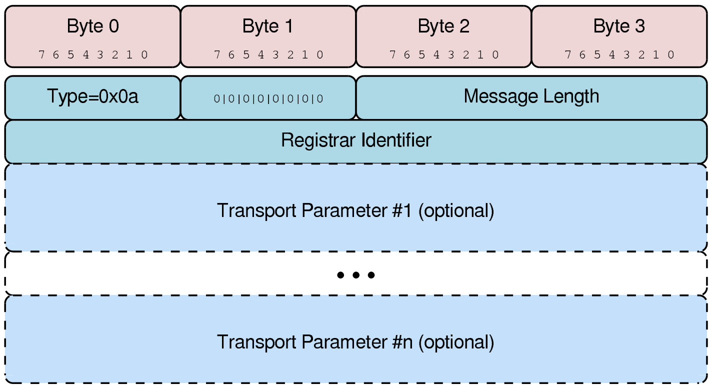


Figure 3.24: The ASAP Server Announce Message

Parameters (see figure 3.19). In case of an error, the message includes an Error Parameter describing the problem (e.g. missing authorization or the requested pool is not existing). For details on the Error Parameter, see Stewart, Xie, Stillman and Tüxen (2006b).

3.9.3.2 Failure Report

If a selected PE is not reachable or if it fails during the usage of its service (of course, it depends on the application how a "failure" is defined, see also subsection 3.12.1 for details), the PU should report this failure to the PR. The message type for reporting a PE failure is the ASAP Endpoint Unreachable (see figure A.10 for an illustration of this message type). It includes the PH of the PE's pool in form of a Pool Handle Parameter (see figure 3.18) and the PE's PE ID in form of a Pool Element Identifier Parameter. The PR does not send any response to a failure report.

3.9.4 Automatic Configuration Functionality

As explained in subsubsection 3.7.1.1, a PR can announce its existence using ASAP Server Announce messages sent over UDP via IP multicast. The transmission interval of the Server Announce messages is defined by the timer *T6-Serverannounce*. Its default value is one second (see section 5.1 in Stewart, Xie, Stillman and Tüxen (2006a)). Figure 3.24 shows the structure of the Server Announce message type: it includes the PR ID of the sending PR. Optionally, the Server Announce can include a set of Transport Parameters defining under which transport addresses the PR is reachable. Allowed are SCTP and TCP Transport Parameters. If no Transport Parameter is specified, the PR is reachable under the source Network Layer address of the announce message, using SCTP as transport protocol and the SCTP port number equal to the UDP source port number of the Server Announce message.

By listening to Server Announce messages and noting the obtained PR information into its *Server Table*, a PE or PU can dynamically learn the identities of the currently available PRs. If no new Server Announce is received for an entry of the Server Table within an interval given by the timer *T7-ENRPoutdate*, the PR entry is removed. The default value for this timer is 5s (see section 5.1 in Stewart, Xie, Stillman and Tüxen (2006a)).

3.9.5 Session Layer Functionality

In this section the ASAP Session Layer is explained. First, the concept of the Data and Control Channel is introduced. After that, the two Session Layer mechanisms to support session failovers are described: cookies and business cards.

3.9.5.1 Data Channel and Control Channel

If a PU has detected a failure of its currently used PE and therefore has selected a new PE and successfully established a new connection, it is necessary that the session state between the PU and the failed PE is re-instantiated on the new server. The failover mechanism to actually resume the session is application-specific and therefore cannot be a direct part of RSerPool. Nevertheless, RSerPool provides a powerful support for realizing arbitrary failover schemes. This support functionality is defined in Conrad and Lei (2005a): the ASAP Session Layer based on the Data Channel/Control Channel concept.

As already explained in the introduction to the RSerPool protocol stack in section 3.5, the protocol used for the connection between PU and PE is the application's protocol (e.g. HTTP for access to a web server PE). This communication channel is denoted as the *Data Channel*; all messages transmitted over this channel completely belong to the application itself and are out of RSerPool's scope. Clearly, to provide any useful service, a Data Channel is mandatory.

Optionally, ASAP offers the possibility for an ASAP communication channel between PU and PE. This communication channel is completely controlled by the ASAP Session Layer and denoted as the *Control Channel*. It provides the following functionalities:

1. Cookies for the failover support by client-based state sharing (to be described in the subsequent subsubsection 3.9.5.2), as well as
2. Business Cards for the signalling of specific PEs for failover and the signalling of symmetric PE/PE communications (to be described in the subsequent subsubsection 3.9.5.3).

If a Control Channel is used, both Data and Control Channel are multiplexed over a single connection between PU and PE. The management of this connection behooves the ASAP Session Layer, which is responsible for the establishment and teardown of the connection and the detection of failures and breaks. The Application Layer of the PU then establishes a *Session* to a certain pool (given by its PH) instead of performing a handle resolution and connecting to a selected PE. ASAP's Session Layer takes care of all related tasks:

- Handle resolution and selection of a PE,
- Establishment of a connection to the selected PE,
- Monitoring the connection to detect failures and breaks,
- Selecting a new PE and establishing a new connection if the current PE fails and
- Invoking the application-specific failover procedure by notifying the Application Layer if necessary.

The connection between PU and PE is usually realized by an SCTP association (see section 3.5). The multiplexing of Data and Control channel is realized using different PPIDs (i.e. 11 for ASAP and a different one for the application protocol). The reason for multiplexing by different PPIDs instead of using multiple SCTP stream numbers is explained in the subsequent subsubsection 3.9.5.2.

3.9.5.2 ASAP Cookies

In many application scenarios, a failover mechanism described in Dreibholz (2002) can be used: the *Client-based State Sharing*. The principle of this mechanism is depicted in figure 3.25. During usage of the PE's service, the PE regularly sends its current session state in form of a *State Cookie* to the PU. This may be realized on a regular basis, on important changes or on every state update. For the PU, a state cookie can be viewed as an arbitrary byte vector. However, it is also suggested in Dreibholz (2002) that for efficiency reasons the PU could be allowed to read certain parts of the cookie or even be allowed to modify them. Cryptographic mechanisms at the PE side can ensure that confidential information in parts of the state cookie cannot be read by the PU (by using encryption) or be altered (by using a digital signature). In the simplest case, all PEs of a pool can use a shared secret key for authentication and encryption.

The PU is only required to store the latest state cookie (i.e. the most up-to-date server state). In case of a session failover, this stored cookie will be sent to the new PE. Then, the new PE can verify the state cookie by checking its signature, decrypt any encrypted parts and finally re-instantiate the session state. After that, the session can be continued.

Since the PU's only tasks are to store the latest state cookie and to send this cookie to a new PE upon failover, the realization of the described failover mechanism is quite simple. It is furthermore easily scalable with the number of PUs. The main limitation of its applicability is that the PU may have the possibility to restore older states by using an out-of-date cookie. This may be a security threat for certain applications (see Dreibholz (2002) for details).

Due to the simplicity and universal applicability of client-based state sharing, it has been possible to contribute it into the ASAP protocol standard documents Stewart, Xie, Stillman and Tüxen (2006a), Conrad and Lei (2005a), in the way that ASAP supports the transmission of state cookies over the Control Channel. The message sequence is illustrated in figure 3.26: the PE sends its current state via the Control Channel to the PU using an ASAP Cookie message (see figure A.12 for an illustration of this message type). It includes the state cookie itself in form of a Cookie Parameter. The Cookie Parameter simply consists of the cookie in form of a byte vector (see Stewart, Xie, Stillman and Tüxen (2006b) for the complete definition and figure A.14 for an illustration of this parameter type).

The ASAP Session Layer on the PU side memorizes the latest cookie. If PE #1 fails, the currently stored cookie is sent to the newly selected PE #2 using an ASAP Cookie Echo message (see figure A.13 for an illustration of this message type) as first message of the communication. The format of this message type is equal to the Cookie message. After the PE #2 has re-instantiated the stored session state, the application is continued and the new PE again starts sending Cookie messages including the up-to-date session states.

The PE-side application decides when there are appropriate points to send a Cookie message. That is, the Cookie messages are tightly linked to the communication over the Data Channel. For a distributed computing application as described in subsection 3.6.5, a partial result could be returned which is followed by a cookie to define a checkpoint for a possible session resumption. Clearly, in this case it is expected that the PU receives the partial result before the cookie, since the obtained partial result will not be re-generated by a new PE if using the sent cookie for failover. This implies the necessity to ensure the message sequence of the Data and Control Channel communication. It is therefore not possible to use an own SCTP stream for each channel. In this case, overtaking of messages would be possible (as this is actually the feature of "no head-of-line blocking"). But since two different protocols are multiplexed over the same association, it is simply possible to set different PPIDs for the messages of each protocol (the standard PPID for ASAP is 11). In this case, the message order is ensured.

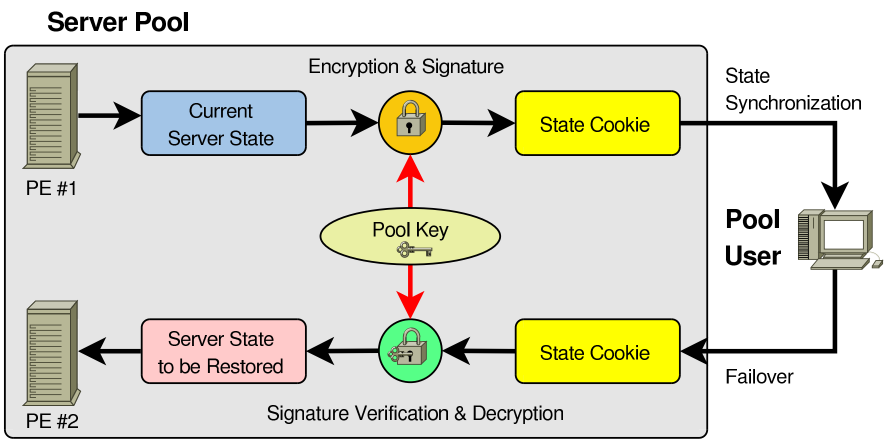


Figure 3.25: The Concept of Client-Based State Sharing

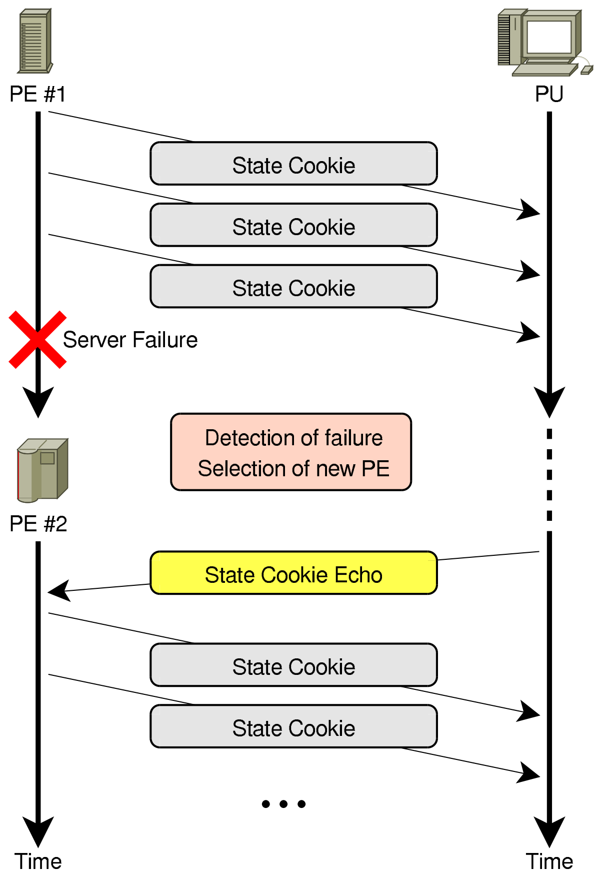


Figure 3.26: A Session Failover using Client-Based State Sharing

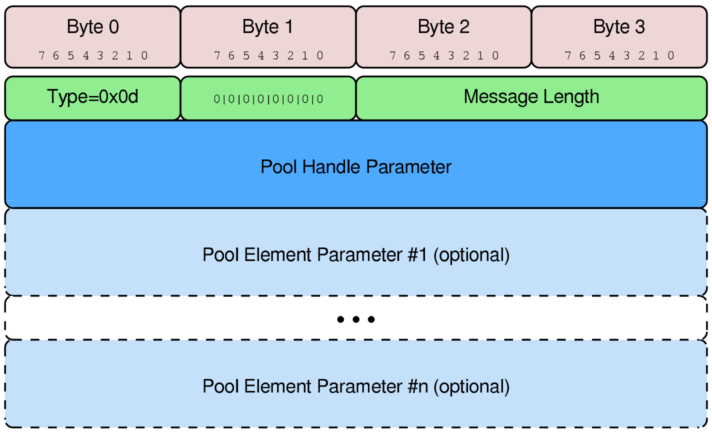


Figure 3.27: The ASAP Business Card Message

Using client-based state sharing, a completely automatic failover by the ASAP Session Layer is possible. For the PU's Application Layer, it is only necessary to establish a session to a server pool. The mapping to a PE and failovers using state cookies may be completely transparent[3] to the Application Layer.

3.9.5.3 ASAP Business Cards

Depending on the application, there may be constraints restricting the set of PEs usable for failover. The ASAP Business Card message sent via the Control Channel is used to inform peer components about such constraints.

Restricted PE Sets for Failover The first case to use a Business Card is if only a restricted set of PEs in the pool may be used for failover. For example, in a large pool, each PE can share its complete set of session states with a few other PEs only. This keeps the system scalable, i.e. a PE in a pool of n servers does not have to synchronize all session states with the other n-1 PEs. In this case, a PE has to tell its PU the set of PE identities being candidates for a failover using an ASAP Business Card message over the Control Channel.

Figure 3.27 presents the structure of this message type: it includes the PH of the PE's pool in form of a Pool Handle Parameter (see figure 3.18) and the list of failover candidates as a sequence of PE identities in the form of Pool Element Parameters (see figure 3.19).

A PE may update the list of possible failover candidates at any time by sending another Business Card. The PU has to store the latest list of failover candidates. Of course, if a failover becomes necessary, the PU has to select from this list using the appropriate pool policy – instead of performing the regular PE selection by handle resolution at a PR. Therefore, some literature also denotes the Business Card by the more expressive term *Last Will*.

Symmetric Communication In particular for the telecommunications signalling example presented in subsection 3.6.1 using separate MGC and GK, a PU using the service of a pool B may itself be a PE of pool A. In this case, a PE of pool B may perform – in the role of a PU using the service of pool

[3]Of course, the Application Layer can request to be notified on failovers.

A – a failover to another PE of pool A. In the described telecommunications example, this scheme is used between the MGC pool and the GK pool.

In such symmetric scenarios, a PU has to tell its PE that it is itself a PE of another pool. This is realized by sending an ASAP Business Card message to the PE over the Control Channel, providing the PH of its pool. Optionally, also specific PE identities for failover may be provided. The format remains the same as explained in the previous paragraph. If the PE detects a failure of its PU, the PE may – now in the role of a PU – use the provided PH for a handle resolution to find a new PE or use the provided PE identities to select one. After that, it can perform a failover to that PE.

3.10 The Endpoint Handlespace Redundancy Protocol

In this section, the Endpoint Handlespace Redundancy Protocol (ENRP) defined in Xie et al. (2006), Stewart, Xie, Stillman and Tüxen (2006b) is described. First, an overview of its functionalities is given. After that, these functionalities are explained in detail.

3.10.1 Overview

The ENRP protocol is used to synchronize the handlespace among PRs of an operation scope. For a classification into the protocol stack of a PR see section 3.5. The following items describe ENRP's tasks:

- Automatic configuration of a PR,
- Initialization of a PR,
- Propagating PE registration changes (registration, re-registration, deregistration) to other PRs by the PR-H of the PE,
- Auditing the handlespace consistency among PRs and re-synchronizing their views if necessary, and
- Orderly taking over the PEs of failed PRs.

The ENRP communication for these tasks is described in the following.

3.10.2 Automatic Configuration

PRs need to know the list of all PRs currently active in the operation scope. To obtain this list, which is denoted as *Peer Table*, ENRP provides three kinds of configuration mechanisms: static configuration by the administrator, announces via multicast and learning from peer PRs. Since all mechanisms are based on the ENRP Presence message, this message type is explained first. After that, the configuration mechanisms themselves are introduced.

3.10.2.1 The ENRP Presence Message

The automatic configuration of PRs is strongly based on the transmission of ENRP Presence messages. Figure 3.28 shows the structure of this message type. It includes the following information:

Sender Registrar ID: This is the PR ID of the sending PR.

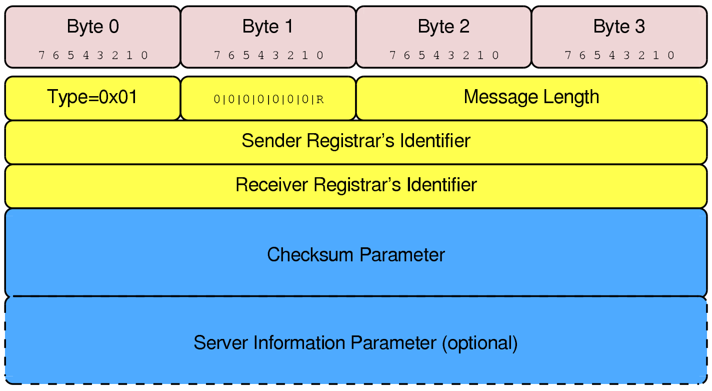


Figure 3.28: The ENRP Peer Presence Message

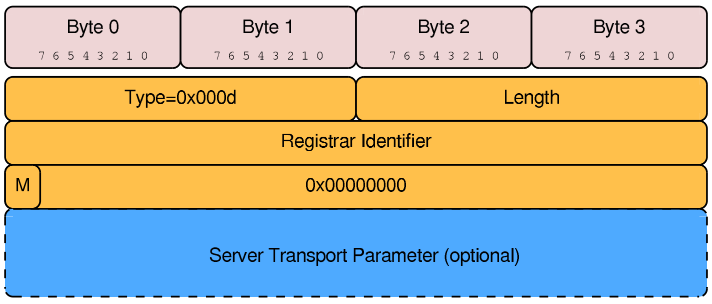
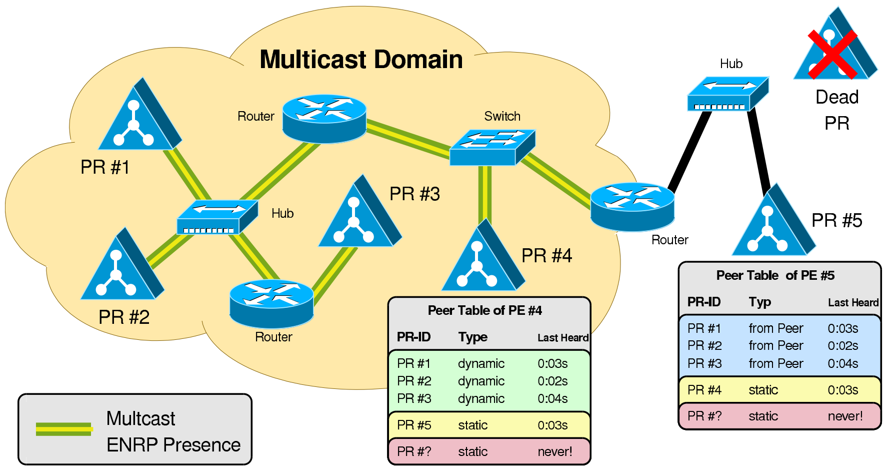


Figure 3.29: The Server Information Parameter



Peer Table of PE #4

PR-ID	Type	Last Heard
PR #1	dynamic	0:03s
PR #2	dynamic	0:02s
PR #3	dynamic	0:04s
PR #5	static	0:03s
PR #?	static	never!

Peer Table of PE #5

PR-ID	Typ	Last Heard
PR #1	from Peer	0:03s
PR #2	from Peer	0:02s
PR #3	from Peer	0:04s
PR #4	static	0:03s
PR #?	static	never!

Figure 3.30: The Peer Tables of Registrars

Receiver Registrar ID: The PR ID of the message's destination PR is given in this field. If the message is used for multicast announces, this ID is set to 0 (i.e. the receiver is not specified).

Checksum Parameter: The Checksum Parameter (see Stewart, Xie, Stillman and Tüxen (2006b) for its definition and figure A.11 for an illustration of this parameter type) includes a checksum of the PE entries for which the PR is the PR-H. It is used for the handlespace consistency audit being explained in subsection 3.10.5.

Server Information Parameter: This optional parameter includes a Server Information Parameter describing the ENRP transport endpoint of the particular PR. The structure of the Server Information Parameter is illustrated in figure 3.29. It includes the PR's ID, as well as a description of the transport endpoint using an SCTP Transport Parameter (see Stewart, Xie, Stillman and Tüxen (2006b) for its definition). That is, it specifies under which Network-Layer addresses and SCTP port number the PR is reachable for ENRP communication. The M-bit ("Multicast" bit) denotes that the PR uses ENRP over UDP via IP multicast for its ENRP communication. Since this possibility is under discussion for removal by the IETF RSerPool WG, it is not further explained here. See Xie et al. (2006) for details.

The R-bit in the flags field ("Response Required" flag) signalizes that the sender of the Presence message requires the receiver's immediate response in form of an own Presence message. Using a message with this flag set, a PR can verify that another PR is still alive.

3.10.2.2 Dynamic and Static Peer Table Configuration

Analogously to the ASAP Server Announce messages sent via IP multicast (see subsection 3.9.4), a PR can announce its availability by sending Presence messages over UDP via IP multicast. In this case, the "Reply Required" flag is never set, and the received PR ID is unspecified (i.e. set to 0).

As for PEs and PUs, it is of course also possible to statically configure PR addresses into the Peer Table. Figure 3.30 illustrates the content of a PR's Peer Table: PR #1 to PR #4 are located within the multicast domain and are able to automatically detect the existence of each other PR in the domain (these entries in the Peer Table are denoted as *dynamic*). Since PR #5 is located outside of the multicast domain, it has a statically configured entry for PR #4 (the entry is denoted as *static*). On the other hand, PR #5 also has a static entry for PR #4. Furthermore, both Peer Tables include a static entry for a dead peer PR #?. This PR has never been reached, therefore only its transport addresses are known, but not its PR ID.

PR #5 only has a static entry for PR #4 of the multicast domain. In order to avoid statically configuring all PR identities into every PR outside of a multicast domain, a third method for configuration is provided by ENRP. This method will be explained after introducing the ENRP connection setup and maintenance.

3.10.2.3 Maintaining Connections to Peer Registrars

In a regular interval given by the constant PEER-HEARTBEAT-CYCLE[4], a PR tries to send a Presence message to every other PR in its Peer Table, over an SCTP association to the corresponding PR. If it is not already existing, it is tried to establish an association. The "Reply Required" flag of the Presence message is not set. On the other hand, the peer PR will also send Presence messages over the SCTP connection. By that, it is ensured on the Application Layer that each peer PR is active. The elapsed time since the last Presence message received from a peer is kept as part of the corresponding peer PR's entry in the Peer Table and denoted as *Last Heard*[5].

If no Presence message is received from a peer within the timeout MAX-TIME-LAST-HEARD (see section 3.9.3 of Xie et al. (2006); the default value is 61s as defined in section 4.1 of Xie et al. (2006)), a Presence message having the "Reply Required" flag set is sent. This forces the peer to reply immediately. If again no response is received within the timeout MAX-TIME-NO-RESPONSE (see section 3.9.3 of Xie et al. (2006); the default value is 5s as defined in section 4.1 of Xie et al. (2006)), the peer must be considered as dead and therefore be removed. Furthermore, the takeover procedure (see subsection 3.10.6) is started, in order to negotiate which other PR takes over the PR-H functionality for the PEs owned by the failed PR.

3.10.2.4 Obtaining the Peer Table from Peer Registrars

Beside the possibilities to dynamically configure the Peer Table by multicast announces and statically by manually inserting entries, ENRP provides a third way for its configuration: the acquisition of Peer Table entries from peer PRs. After a SCTP association to a peer PR has been successfully established as described in the previous subsubsection 3.10.2.3, the peer PR's Peer Table can be requested using an ENRP Peer List Request message. Figure 3.31 shows the structure of this message type. It only includes the PR IDs of the sending and the receiving PRs.

Upon reception of the Peer List Request, the peer PR responds using an ENRP Peer List Response message. The format of this message type is shown in figure 3.32; it includes the PR IDs of sending and receiving PRs, as well as the requested Peer Table in form of a Server Information Parameter (see figure 3.29; the contents of this parameter have been explained in subsubsection 3.10.2.1). Entries of the response that do not already exist in the receiver's Peer Table have to be added. After that, connections to the new entries can be established as described in subsubsection 3.10.2.3.

In the example provided in figure 3.30, PR #5 has learned the entries for PR #1 to PR #3 by asking PR #4 (its identity is known due to static configuration). The entries received by the Peer List Response of PR #4 are marked with *from Peer*.

[4]See section 3.9.2 of Xie et al. (2006); the default value is 30s as defined in section 4.1 of Xie et al. (2006).

[5]Of course, from an implementer's perspective, only the time stamp of the last Presence message is stored into the table. The Last Heard value can be calculated as the difference between the current time stamp and the stored one.

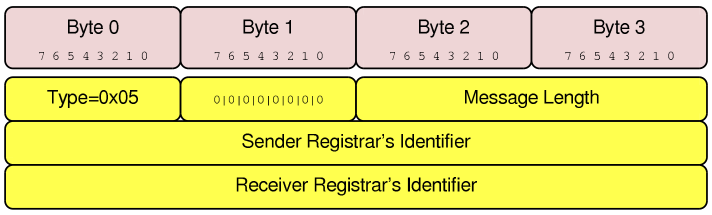


Figure 3.31: The ENRP Peer List Request Message

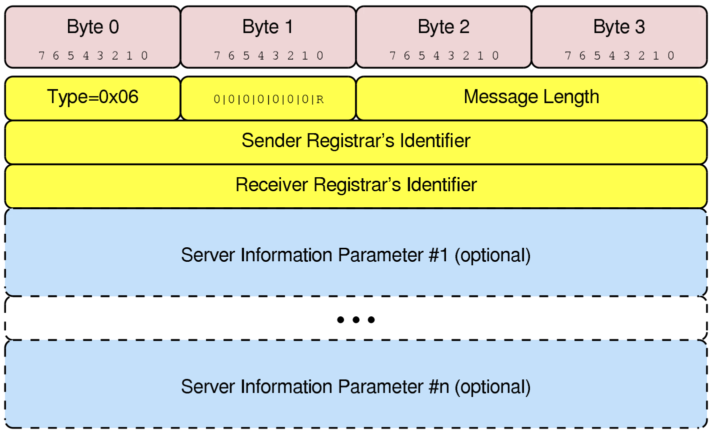

Byte 0
7 6 5 4 3 2 1 0
Byte 1
7 6 5 4 3 2 1 0
Byte 2
7 6 5 4 3 2 1 0
Byte 3
7 6 5 4 3 2 1 0
Type=0x06
0 0 0 0 0 0 0 R
Message Length
Sender Registrar's Identifier
Receiver Registrar's Identifier
Server Information Parameter #1 (optional)
• • •
Server Information Parameter #n (optional)

Figure 3.32: The ENRP Peer List Response Message

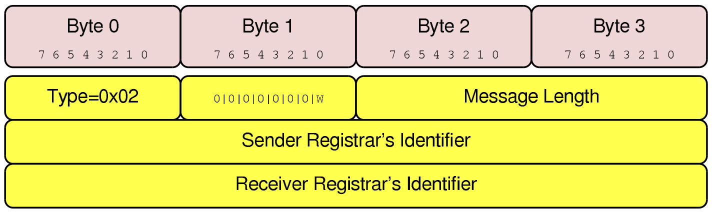


Figure 3.33: The ENRP Handle Table Request Message

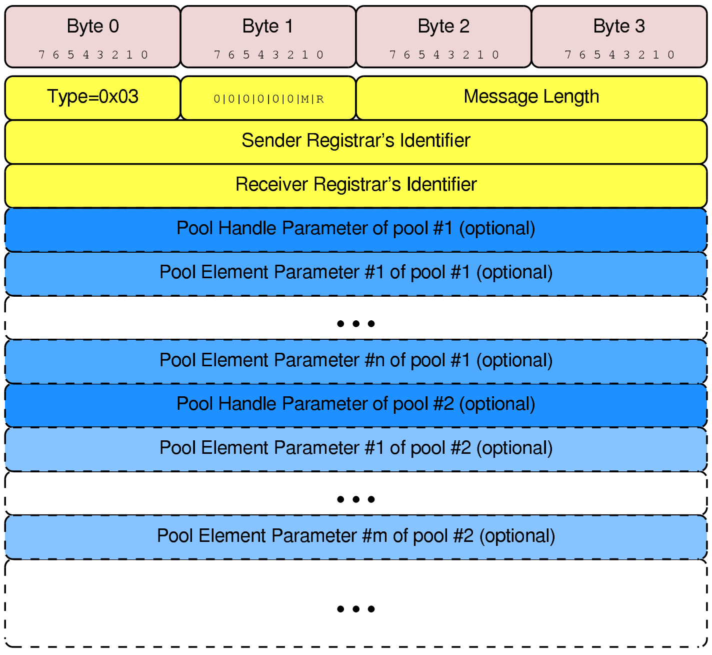

Byte 0
7 6 5 4 3 2 1 0
Byte 1
7 6 5 4 3 2 1 0
Byte 2
7 6 5 4 3 2 1 0
Byte 3
7 6 5 4 3 2 1 0
Type=0x03
0|0|0|0|0|0|M|R
Message Length
Sender Registrar's Identifier
Receiver Registrar's Identifier
Pool Handle Parameter of pool #1 (optional)
Pool Element Parameter #1 of pool #1 (optional)
• • •
Pool Element Parameter #n of pool #1 (optional)
Pool Handle Parameter of pool #2 (optional)
Pool Element Parameter #1 of pool #2 (optional)
• • •
Pool Element Parameter #m of pool #2 (optional)
• • •

Figure 3.34: The ENRP Handle Table Response Message

3.10.3 Registrar Initialization

When a PR is started up, it first has to select its PR ID. This is a non-zero, 32-bit random number. Uniqueness in the operation scope has not to be enforced (see section 3.2.1 of Xie et al. (2006)). After that, the PR has to obtain the list of other PRs in the operation scope using the three methods explained in subsection 3.10.2 (i.e. statically configured, learned via multicast announces and obtained from peers).

If at least one other PR has been found and a SCTP association to it has been successfully established, a so called *Mentor PR* is chosen from the list of available PRs (e.g. by randomly selecting an entry). The Mentor PR is used for obtaining the complete handlespace. For this purpose, the handlespace is requested using an ENRP Handle Table Request message. The structure of this message type is presented in figure 3.33, it includes the PR IDs of sending and receiving PRs. The W-bit in the flags field denotes the “Own Children Only” flag. If this flag is set, only the PE entries for which the

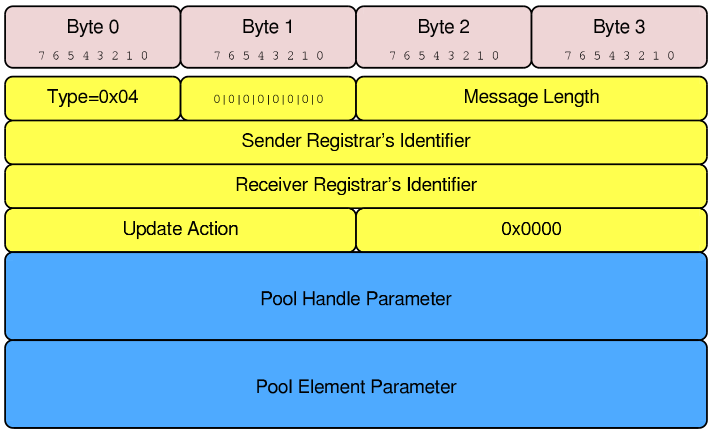


Figure 3.35: The ENRP Handle Update Message

receiving PR is the PR-H are requested. This flag is used for the handlespace synchronization in case of an inconsistency (to be explained in subsection 3.10.5); since the complete handlespace is required from the Mentor PR, it is therefore not set.

Upon reception of the Handle Table Request, the Mentor PR responds using an ENRP Handle Table Response message. The format of this message type is presented in figure 3.32, it includes the PR IDs of sending and receiving PRs. Optionally, the message includes handlespace data as sequence of PHs (in form of Pool Handle Parameters, see figure 3.18) and PE identities – belonging to the corresponding pool denoted by the PH – in form of Pool Element Parameters (see figure 3.19). See subsubsection 3.9.2.1 for a detailed description of these two parameter types. If the R-bit in the flags field – denoted as "Reject" flag – is set, the Handle Table Request has been rejected. This could be the case if the chosen Mentor PR is also in an initialization phase. That is, another PR has to be selected as Mentor PR and the handlespace acquisition to be repeated. The M-bit in the flags field is denoted as "More" flag. If it is set, the response does not include the complete handlespace data but only a partial result (e.g. due to the limited size of a message). A subsequent Handle Table Request will return the next block (i.e. the Mentor PR has to hold a state, remembering where to continue). On the other hand, if the More flag is not set, obtaining the handlespace has been accomplished.

As soon as the complete handlespace has been transmitted from the Mentor PR, the PR is ready and can start announcing its availability to PUs and PEs by sending ASAP Server Announce messages (see subsection 3.9.4). If it has not been possible to obtain the handlespace within a configured timeout, the PR assumes to currently be alone within the operation scope and also goes from the initialization phase into normal operation.

3.10.4 Handle Update

When a PR-H performs the registration, re-registration or deregistration of a PE, it is responsible for also informing the other PRs of the operation scope about the changed handlespace content. The message type for this purpose is the ENRP Handle Update message. As shown in figure 3.35, it includes the PR IDs of sending and receiving PRs, as well as the PE's PH in form of a Pool Handle Parameter (see figure 3.18) and the PE information in form of a Pool Element Parameter (see figure 3.19). A detailed description of the contents of these two parameters can be found in subsubsection 3.9.2.1. Finally, the Handle Update includes the field *Update Action*, defining whether the given PE should be

Algorithm 1 The 16-Bit Internet Checksum Algorithm

```
unsigned short calculateInternetChecksum16(void* data, size_t count)
{
   unsigned short* addr = (unsigned short*)data;
   unsigned int    sum  = 0;

   while(size >= sizeof(*addr)) {   /* Main calculation loop */
      sum += *addr++;
      size -= sizeof(*addr);
   }
   if(size > 0) {   /* Add left-over byte, if any */
      sum += * (unsigned char*)addr;
   }
   while(sum >> 16) {   /* Fold 32-bit sum to 16 bits */
      sum = (sum & 0xffff) + (sum >> 16);
   }
   return(~sum);
}
```

(re-)registered (PE_ADD, 0x0000) or deregistered (PE_DEL, 0x0001).

3.10.5 Handlespace Audit and Synchronization

If a PR has temporarily lost the connectivity to other PRs of the operation scope, its view of the handlespace may be inconsistent to the views of the other PRs. During interruption of the ENRP connections, updates applying registrations, re-registrations or deregistrations may have occurred, which have been missed by the disconnected PR. Therefore, it is necessary to regularly check the consistency of the handlespace – this check is denoted as *Handlespace Audit* – and triggering a re-synchronization if necessary.

The approach used for the handlespace audit is to calculate a checksum over the PE identities "owned" by a PR, that is the set of PEs for which a PR is the PR-H. For this purpose, a 16-bit Internet checksum (also used e.g. for the IPv4 and TCP header checksums) is calculated over the sequence of each PE's PH and PE ID as defined in section 3.1.1 of Xie et al. (2006). The algorithm for the checksum calculation is defined in Braden et al. (1988), Rijsinghani (1994), a pseudo-code representation can be found in algorithm 1. Important properties of this algorithm are that it is simple and efficient to implement. Furthermore, it is possible to calculate it incrementally. That is, it allows to subtract the checksum of a PE on deregistration and the addition of a checksum on PE registration without having to re-compute the checksums of all other PEs. See subsection 4.4.4 as well as Rijsinghani (1994) for details on the computation of the incremental checksum.

The choice of the 16-bit Internet checksum is the result of some lengthy discussions on the IETF RSerPool WG's mailing list and at the 63rd IETF meeting: Motorola might have a software patent on checksums longer than 16 bits. Taking the 16-bit Internet checksum that is already used for the IPv4 header, TCP header and many other applications has been considered as the safest solution to avoid running into unforeseeable trouble with the U.S. software patent system.

Having computed the checksum for its PE entries, a PR-H can publish it as part of its Presence message (see subsection 3.10.2). This allows the receiver of a Presence message to compare the included checksum to the checksum it expects for the sending PR (by referring to its own view of

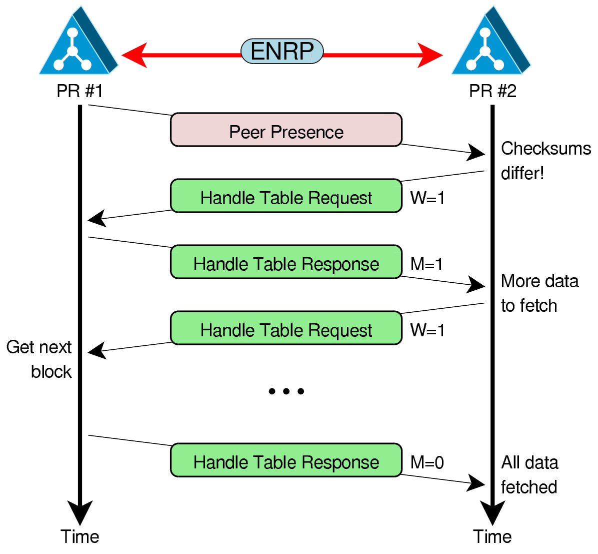


Figure 3.36: An Example for Handlespace Audit and Resynchronization

the handlespace). Note, that a PR can efficiently maintain a checksum for each other known PR of the operation scope by using incremental checksum updates. More details on efficient handlespace management can be found in chapter 4 and Dreibholz and Rathgeb (2005b), Dreibholz (2004d). If the checksum from a Presence message is equal to the expected one, the handlespace referring to the peer PR's PE entries is assumed to be consistent. The probability of an undetected inconsistency is 1:65,536 (i.e. about 0.0015%), due to the 16-bit checksum.

If the two checksums differ, a re-synchronization of the handlespace is necessary. First, all PE entries owned by the remote PR (i.e. the PEs for which the PR is the PR-H) are marked in the local handlespace copy. After that, the PE information for all PEs owned by the remote PR is requested from it by using the ENRP Handle Table Request message already introduced in subsection 3.10.3. In the current case, the W-flag ("oWn children only") is set, in order to request the PR's own PE entries only. The entries received in the response of the remote PR are used to update the local handlespace copy. Finally, all marked entries not being updated are removed (these entries are obsolete). After that, the handlespace view of the PEs owned by the remote PR is consistent again.

To make the handlespace audit and re-synchronization procedure clearer, figure 3.36 presents an example for the message sequence. PR #2 receives a Presence message from PR #1 and detects a handlespace inconsistency due to differing checksums. Therefore, PR #2 requests the handlespace data for the PEs owned by PR #1 using a Handle Table Request (the "Own Children Only" flag is set). PR #1 answers with the first part of its handlespace data. The Handle Table Response does not include the full data, since the message capacity would be to small. Therefore, the M-flag ("More") is set. PR #1 requests the subsequent data packet by another Handle Table Request. Finally, the last Handle Table Response is received (M-flag is not set). After removing the obsolete entries, the handlespace information of PR #1's PEs is again consistent in PR #2's view of the handlespace.

3.10.6 Takeover Procedure

If a failure of a PR is detected, the remaining PRs have to negotiate which PR takes over the ownership of the PEs currently owned by the out-of-order PR. This negotiation procedure, together with the

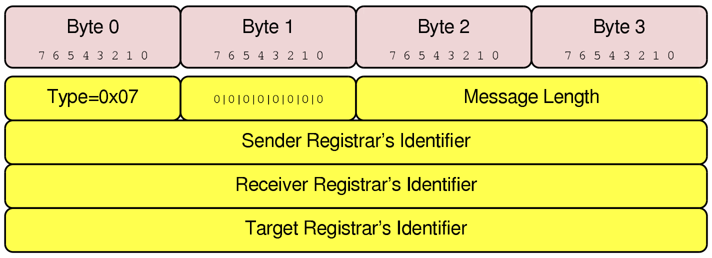


Figure 3.37: The ENRP Init Takeover Message

actual ownership change, is denoted as *Takeover Procedure* or simply Takeover.

To initiate a takeover, the PR which has detected a failure of another PR first sends an ENRP Init Takeover message to all of its peers, including the putatively failed PR itself. Figure 3.37 shows the structure of this message type. It includes the PR IDs of sending and receiving PRs, as well as the PR ID of the putatively failed PR – the target of the takeover procedure, therefore denoted as *Target*. After transmission of the Init Takeover messages, a consent to the takeover is expected from all other PRs, except the target itself (which is presumably dead, of course).

Upon reception of an Init Takeover message, there are three possible cases for a PR to react:

1. The PR itself is the target of the takeover procedure. In this case, it immediately sends a Presence message (see subsection 3.10.2) to all other PRs in order to avoid being taken over.

2. The PR has already started its own takeover procedure for the target. In this case, it checks whether its own PR ID is smaller than the PR ID of the PR sending the Init Takeover message. In this case, it aborts its own takeover procedure and confirms the takeover by the other PR using an ENRP Init Takeover Ack message (see below).

3. If the other two other cases do not apply, the PR gives its consent to the takeover, using an ENRP Init Takeover Ack message (see below).

The consent to a takeover is given by an ENRP Init Takeover Ack message. Its structure is equal to the Init Takeover, except for a different message type. If the takeover is confirmed by all other PRs except the takeover target, a PR has won the takeover negotiation and can finally complete the takeover. For this purpose, it sends a Takeover Server message to all remaining PRs of the operation scope. The format of this message type is equal to Init Takeover and Init Takeover Ack, except for the message type. Upon reception of the Takeover Server message, a PR changes the ownership of all PEs owned by the takeover target PR to the takeover sender PR.

Finally, the new PR-H of the PEs taken over has to tell the PEs themselves about being taken over. For this purpose, an ASAP Endpoint Keep-Alive message (see subsubsection 3.9.2.2) is sent to each of the PEs. The "Home" flag of these messages is set, therefore the PEs adopt the new PR as their PR-H. After that, the takeover procedure is accomplished.

To make the order of events in a takeover procedure clear, an example is illustrated in figure 3.38. Here, PR #2 detects the failure of a fourth PR (not shown in the figure). It therefore starts a takeover by sending Init Takeover messages to PR #1 and PR #3. PR #1 on his part has also detected the failure and started a takeover. Since PR #1 has a smaller PR ID than PR #2, it aborts its own takeover and gives PR #2 its consent for the requested takeover in form of an Init Takeover Ack message. PR #3 also confirms PR #2's takeover request. After receiving the consents of PR #1 and PR #3, PR #2 has

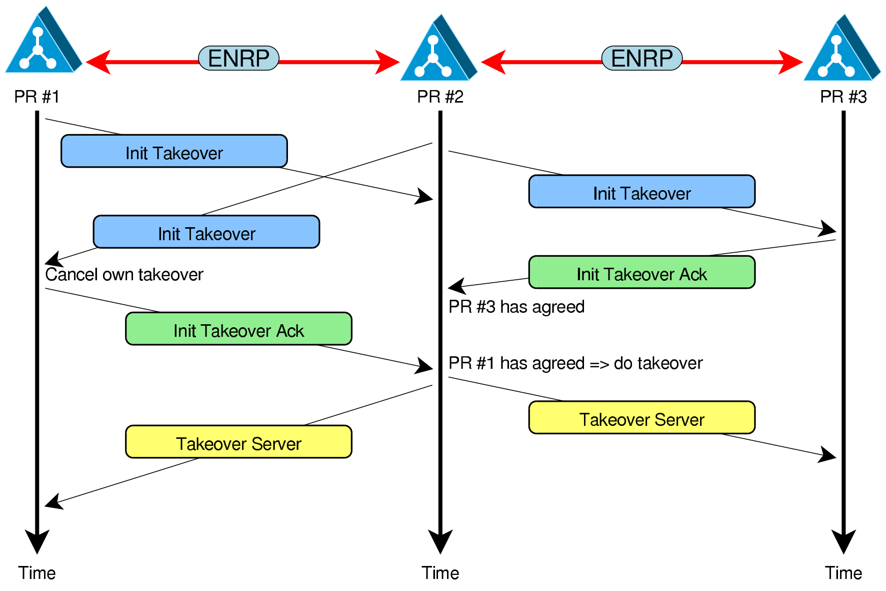


Figure 3.38: A Takeover Example

won the takeover and can actually complete it by sending Takeover Server messages to PR #1 and PR #3, as well as sending ASAP Endpoint Keep-Alives with the “Home” flag set to the PEs taken over (not shown in the figure). Upon reception of the Takeover Server messages, PR #1 and PR #3 change the ownership of all PEs currently belonging to the failed PR to their new PR-H (i.e. to PR #2).

It has to be noted that if a takeover fails for whatever reason, the PEs of the failed PR themselves detect the unreachability of their PR-H upon their next re-registration. That is, after at most the re-registration interval (T4-Reregistration) plus the registration timeout (T2-Registration), a PE is induced to look for a new PR-H. See also subsection 3.7.2 for details on this double safeguarding mechanism.

3.11 The Pool Member Selection Policies

The pool member selection policies – also simply denoted as pool policies – define the load distribution and balancing mechanisms of RSerPool systems. In this section, their basics are explained at first. After that, the actual policies defined in the Working Group Draft Tüxen and Dreibholz (2006b) are explained.

The Working Group document is the result of our contributions to the standardization of RSerPool: the initial RSerPool documents had specified a few policy names only, but an exact specification had been missing. Therefore, as part of the policy performance evaluations described in chapter 8, a complete specification of a set of policies has been created. These policies have been evaluated in Dreibholz, Rathgeb and Tüxen (2005), the resulting guidelines for their implementation as well as their specifications have been submitted to the IETF as Individual Submission Internet Draft Tüxen and Dreibholz (2005). This document has then become the Working Group Draft Tüxen and Dreibholz (2006b) of the IETF RSerPool WG (see also Dreibholz (2004b,c)).

3.11.1 Basics

The existence of multiple PEs for redundancy automatically leads to *Load Distribution* and *Load Balancing*. While load distribution according to Berger and Browne (1999) only refers to the assignment of work to a processing element, load balancing refines this definition by requiring the assignment to maintain a balance across the PEs. The balance refers to an application-specific parameter like CPU load or memory usage. A pool policy determines the distribution of load created by PUs among PEs of a pool, also providing the possibility to balance it.

Before it is possible to define rules for load distribution and balancing, the meaning of the term *Load* in the context of RSerPool has to be defined exactly: here, load denotes a value in the interval from 0% (unloaded) to 100% (fully loaded). For storage reasons, the load value is encoded as 32-bit unsigned integer and therefore 0% maps to 0x00000000 and 100% maps to 0xffffffff. For each application, it is therefore necessary to map the application-specific meaning of *Utilization* to the view of RSerPool. Formally, this mapping is defined as follows:

$$m : U_{\text{Application}} \rightarrow [0,\, 1] \subset \mathbb{R}$$

$$m(u) = \frac{u}{\max(U_{\text{Application}}) - \min(U_{\text{Application}})} \tag{3.1}$$

In this formula, $U_{\text{Application}}$ denotes the application's set of utilization values. For an FTP server, this could be the number of users from 0 to 10. That is, $U_{\text{FTP}} = \{0,\, 1,\, \ldots,\, 10\}$ and at a utilization of 7 users, its load can be calculated as $m(7) = \frac{7}{10-0} = 70\%$. A gatekeeper for telephone signalling (see subsection 3.6.1) could define its utilization as the number of MGC requests in its queue. If there are currently 10 requests for a maximum number of 50, $U_{\text{GK}} = \{0,\, 1,\, \ldots,\, 50\}$ and the load is $m(10) = 20\%$.

In the Working Group Draft Tüxen and Dreibholz (2006b), pool policies are classified into two categories: adaptive and non-adaptive ones. For an *Adaptive Policy*, a regular update of the policy information is necessary. An example for such a policy is selecting the PE currently having the least load. Since a PE's load is variable, it has to be propagated into the handlespace regularly or on every change. On the other hand, a *Non-Adaptive Policy* does not require such information. An example for such a policy is random selection.

In the following two subsections, the relevant policies defined in the Working Group Draft Tüxen and Dreibholz (2006b)[6] are explained.

3.11.2 Non-Adaptive Policies

The relevant non-adaptive policies from Tüxen and Dreibholz (2006b) are introduced in this subsection.

3.11.2.1 Round Robin and Weighted Round Robin

Using the *Round Robin* (RR) policy, the PEs are selected in turn. Round Robin is the so called *Default Policy* of RSerPool. That is, every RSerPool implementation must support it. All other policies are optional. The choice of Round Robin as default policy is based on the recommendations of Dreibholz, Rathgeb and Tüxen (2005).

A generalization of Round Robin is Weighted Round Robin (WRR). For this policy, each PE provides a positive integer weight constant, describing its processing power (depending on the application

[6]This document also defines the corresponding Policy Parameters.

of course, e.g. CPU power, memory, disk space etc.) proportional to the other PEs. Analogously to Round Robin, the PEs are again selected in turn. But for each round, a PE is selected as many times as its weight constant specifies. That is, a PE of weight 5 is selected 5 times, while a PE of weight 1 is chosen only once.

It is important to note that Round Robin as well as Weighted Round Robin are "stateful". That is, the selection of a certain PE depends on the previous selections. Since for RSerPool multiple components (PRs and PU-side caches) may perform selections independently of each other, the resulting overall selection behaviour could show a significant difference from the desired round robin scheme. Fallacies and pitfalls of these policies are described in section 8.8.

3.11.2.2 Random and Weighted Random

The Random (RAND) policy selects PEs – as expected – by random. The probability for choosing a PE is equal for all elements of the pool. A generalization of Random is Weighted Random (WRAND). In this case, each PE provides a positive weight constant which specifies its desired selection probability proportional to the other PEs. That is, a powerful PE of weight 9.75 is on average selected 9.75 times more frequently than a PE of weight 1.

In contrast to the Round Robin and Weighted Round Robin selection, the Random policies are "stateless". That is, the choice of the next PE does not depend on the current selection. Depending on the application scenario, this could lead to a significantly improved load distribution. For more details, see section 8.8.

3.11.3 Adaptive Policies

This subsection describes the relevant adaptive policies defined in Tüxen and Dreibholz (2006b).

3.11.3.1 Least Used

The Least Used (LU) pool policy bases its selection decision on the current load states of the PEs, i.e. it selects the least-loaded server. Load in this case is defined as explained in subsection 3.11.1. If multiple PEs are on the same lowest load level, round robin or random selection should be performed among these least-loaded elements.

3.11.3.2 Priority Least Used

Even if a pool consists of PEs providing very heterogeneous capacities, the selection performed by Least Used is always based on the PEs' actual load. But for such scenarios, selecting a low-loaded but low-capacity PE may result in a worse performance than using a higher-loaded but also more powerful PE. For example, PE #1 is 10% loaded and a new request would increase its load by another 2%. On the other hand, PE #2 is 8% loaded, but an additional request to handle would increase its load by additional 8%. Clearly, it would be beneficial to choose PE #1, since its load would increase to 12% instead of 18% for PE #2.

The Priority Least Used (PLU) policy, introduced and evaluated in Dreibholz, Rathgeb and Tüxen (2005), lets a PE specify a *Load Increment* constant. This value, given in the units of load, specifies how much an additional request will increase the PE's load. After that, the PE having the currently lowest sum of load plus load increment is chosen. That is, while LU selects PEs based on the *current* load, Priority Least Used bases its choice on the load level that a PE will have *after* a new request is assigned to it. In case of multiple PEs on the same lowest sum level, round robin or random selection can be applied among them. In the example above, PU #1 would be chosen – its sum of load and load increment is 12%, instead of the 18% for PE #2.

3.12 The Mechanisms for Service Reliability and Availability

A RSerPool system – based on the ENRP and ASAP protocols described in the previous sections – includes various mechanisms to detect and handle component failures, in order to support applications in providing a reliable service. In this section, these mechanisms are summarized to make their functionalities clear.

3.12.1 Failure Model

For the analysis of RSerPool's component failure handling performance, as well as for the correct understanding of the following description of RSerPool's mechanisms for service reliability and availability, it is first necessary to define a *Failure Model*. In particular, the failure model specifies what the term "failure" actually means.

For the context of this thesis, a *Failure* of a PR, PE or PU component occurs in form of a so called *Silent Failure*: a faulty component simply disappears, i.e. it does not any longer respond to requests according to its protocol specifications (e.g. ASAP, ENRP or an application protocol). For the context of this thesis, the reason for a failure – denoted as *Fault* – is irrelevant: for the user of a service, it does not make a difference whether the interruption of its service is e.g. caused by a crash of the component, a broken network connection or the failure of an intermediate network device like a router.

It is important to note that the used failure model does not cover the following aspects:

- The output of wrong calculation results (i.e. due to memory or CPU faults) is out of the scope of RSerPool. From the perspective of RSerPool, a faulty component could simply halt itself if it detects its own malfunction by appropriate mechanisms (see Echtle (1990) for details on this subject). In this case such a shutdown becomes equal to a silent failure.
- Intentional distribution of misinformation by an attacker is out of this thesis's scope. However, some security considerations are presented in section 3.13.

The number of tolerable component failures depends on the application providing its service by a pool: while the RSerPool functionality itself can be provided with at least one PR, the minimum number of necessary PEs is application-dependent (but also at least one, of course).

After having defined the failure model, it is possible to explain RSerPool's mechanisms for service reliability and availability in detail. Since both RSerPool protocols strongly rely on the features of the SCTP protocol, the role of the SCTP mechanisms is described at first. This is followed by the explanation of the actual RSerPool mechanisms.

3.12.2 Mechanisms of the Transport Layer

RSerPool systems are based on the SCTP Transport Layer protocol (see subsection 2.4.3). Therefore, problems concerning the underlying layers may be solved in the Transport Layer already. Table 3.1 summarizes the problems which can occur on these layers and the mechanisms provided to detect them:

Congestion: Probably, the most frequent problem in a network is the overload of bottleneck links. Clearly, an overloaded link leads to packet loss. In case of SCTP (as well as for TCP), lost packets are detected by sequence number differences. The sender retransmits packets not being acknowledged within a certain timeout (in case of reliable transport). Furthermore, the congestion control algorithm ensures that the available bandwidth is fairly shared among the associations utilizing a bottleneck link.

Transmission Errors: Bit errors due to unreliable transmission are detected by the CRC-32 checksum[7] within each SCTP packet. In case of a wrong checksum, the packet is simply dropped and retransmitted by the sender (due to missing acknowledgement; in case of reliable transport). Furthermore, using the Packet Drop extension, a receiver can explicitly notify its sender about packet rejection due to transmission errors (see subsubsection 2.4.3.6 and Stewart, Lei and Tüxen (2006a)). Therefore, a transmission error can be differentiated from a packet loss due to congestion.

Link and Router Problems: Links – as well as routers – may fail, e.g. due to damages or simply power loss. The multi-homing feature of SCTP (see subsubsection 2.4.3.4) allows to connect an endpoint to different – and for logical reasons independent – networks. In this case, a problem within one network – detected by path monitoring using heartbeat chunks – can simply be handled by using another path for the transport of data. That is, the multi-homing feature of SCTP provides link redundancy. For the upper layers, the switching of the primary path is transparent (see also Jungmaier, Rathgeb and Tüxen (2002), Jungmaier (2005) for details).

3.12.3 Mechanisms of the Session Layer

The handling of server failures is the main purpose RSerPool has been created for. But a PE is not the only component type which can fail – a failure is possible for a PU and a PR as well. Table 3.2 summarizes the mechanisms by which each component type detects the failures of a peer component type:

Pool User Failure: If a PU fails, the association to its PR is broken. This is detected by the transport protocol (i.e. usually by SCTP). Except for cleaning up the resources allocated for the association maintenance, nothing else has to be done by the PR.
The same happens for an association with a PE. Only in case of a symmetric scenario, i.e. the PE is also a PU (see subsubsection 3.9.5.3), the PE has to perform a failover – now in the role of a PU.

Pool Element Failure: A PU has to detect the failure of its PE by application-specific mechanisms. In the usual case, this is realized by application timeouts: if the PE does not answer within a given timeout, the PU assumes that the PE is dead and performs a failover. For the application model being introduced later in this thesis, the failure detection functionality is realized by the CalcAppKeepAlive and CalcAppKeepAliveAck messages (for details see section 8.3).
For the PR, there are two mechanisms to detect a PE failure:

1. First, if the PR is the PR-H of the PE, a failure is detected by an ASAP Endpoint Keep-Alive message (see subsubsection 3.7.1.3): if the PE fails to answer using an ASAP End-

[7] Note again that a CRC-32 checksum does not denote a sum in mathematical sense. Nevertheless, this is the terminology used by the standards documents.

Problem	Detection Mechanism
Congestion	Sequence Numbers and Acknowledgements
Transmission Error	Checksum, Packet Drop Extension
Link and Router Problems	Path Monitoring, Multi-Homing

Table 3.1: The Network and Component Failure Detection Mechanisms of SCTP

Failed Component	Detection by	Detection Mechanism
Pool User	Pool Element	Application-Specific (Timeout or Broken Connection)
Pool User	Registrar	Broken Connection
Pool Element	Pool User	Application-Specific (Timeout or Broken Connection)
Pool Element	Registrar	ASAP Endpoint Keep-Alive Timeout
Pool Element	Registrar	ASAP Endpoint Unreachable(s) by Pool User(s)
Registrar	Pool User	ASAP Request Timeout
Registrar	Pool Element	ASAP Request Timeout
Registrar	Registrar	ENRP Presence Timeout
Registrar	Registrar	ENRP Request Timeout

Table 3.2: The Component Failure Detection Mechanisms of RSerPool

point Keep-Alive Ack message, it is assumed to be dead. The speed of this failure detection mechanism depends on the settings of the Keep-Alive Transmission and Timeout intervals (see subsubsection 4.3.2.2).

2. The second detection mechanism for PE failures is the reporting of a PE unreachability by PUs, using ASAP Endpoint Unreachable messages (see subsubsection 3.9.3.2). A PR counts the number of unreachability reports for each PE; if the number reaches the configured limit MAX-BAD-PE-REPORT (see subsubsection 3.7.1.4), the PE is assumed to be dead.

Clearly, upon detection of a PE failure, the PR's action is to remove the corresponding PE identity from the handlespace.

Registrar Failure: PEs and PUs detect a failure of their PR by request timeouts; i.e. a request for registration, deregistration or handle resolution is not answered within the specific timeout of the corresponding request type. In this case, another PR has to be contacted and used. For the PE, this also means to re-register at the new PR.
The remaining PRs also perform a takeover procedure for the PEs of the failed PR (see subsection 3.10.6). The PR winning the takeover sends an ASAP Endpoint Keep-Alive message with the "Home" flag set to each PE whose ownership has been taken, in order to notify them about their new owner.

For the RSerPool failure handling evaluations of this thesis, only failures of PEs and PRs are relevant; PU restarts are application-specific and out of scope. The usual procedure of a PU failure handling is to restart the whole session. For example, if the user has rebooted the PU host after a system crash, he simply starts the PU application again. Another approach is to regularly save the application state and allow a PU restart from the latest stored state; see Plank et al. (1995) for more details on this subject.

3.12.4 Support for Redundancy Models

Clearly, the provision of component redundancy concepts leads to the usage of different redundancy models (see also Engelmann and Scott (2005)). For the RSerPool application scenarios described in section 3.6, it is assumed that the majority of the services uses the active/active model (see also section 1.2.3). That is, all servers of the pool should be utilized to make best use of the available resources. Configuring an appropriate pool policy for the PE selection (see section 3.11), load balancing (see section 1.2.2) can be used to reasonably distribute the workload among the servers of the pool.

But RSerPool is not restricted to the active/active model only. Using an appropriate policy, e.g. the redundancy model policy defined in Xie and Yarrol (2004), it is easily possible to also apply the active/standby model as well. That is, some PEs are in standby mode and only used in case of a failure of the active ones. The actual behaviour of the standby component, i.e. whether it provides cold-standby, warm-standby or hot-standby (see section 1.2.3 for the definitions) is in the responsibility of the application only and out of the scope of RSerPool.

3.13 Security Considerations

Considerations for the security of RSerPool are discussed in detail in Stillman et al. (2005). Therefore, only a short summary of the important points is given here. In general, RSerPool endpoints are protected quite well against blind flooding attacks, due to the usage of SCTP as transport protocol (see subsection 2.4.3 for details on the security mechanisms of SCTP). The main threat for RSerPool systems is that an attacker may bring misinformation into the handlespace, either by ASAP or ENRP.

If an attacker is able to masquerade as an authorized PE, it can create faked registrations and by artful choice of policy parameters (e.g. the lowest load or the highest weight) imply the choice of unsuitable or non-existing PEs. This may lead to a denial of service, at least for the pool for which the compromised PE authorization is valid for. If an attacker is even able to masquerade as PR, it can fill arbitrary misinformation into the handlespace and cause a denial of service for the whole RSerPool network. Furthermore, PEs may choose the attacker's PR as PR-H.

To cope with the described threats, RSerPool mandatorily requires bilateral authenticity and integrity protection – among PRs as well as between PRs and PEs or PUs. Both requirements can be achieved by applying existing standard security technologies: either IPsec[8] on the Network Layer or TLS[9] on the Transport Layer. Alternatively, the Secure-SCTP extension for SCTP (see subsubsection 2.4.3.7) also provides a suitable solution.

Optionally, all three techniques also offer the possibility for encryption to ensure confidentiality. That is, depending on the users' requirements, ASAP and ENRP traffic can also be encrypted.

3.14 Summary

In this chapter, the Reliable Server Pooling (RSerPool) framework for the management of and access to server pools has been presented. The provision of component redundancy for SS7 telephone sig-

[8]See Kent and Atkinson (1998c,a,b).

[9]See Dierks and Allen (1999), Blake-Wilson et al. (2003).

nalling over IP networks had been the initial motivation of RSerPool. Due to its generic applicability, various new application scenarios have shown up, e.g. the usage for real-time distributed computing and load balancing. In the first part of the introduction to RSerPool, its components – registrars (PR), servers (pool elements, PE) and clients (pool user, PU) – and their interaction have been presented. After that, the two RSerPool protocols and their behaviour have been introduced in detail: the Aggregate Server Access Protocol (ASAP) for the communication of pool elements and pool users with registrars and the Endpoint Handlespace Redundancy Protocol (ENRP) for the synchronization of the handlespace among registrars. Finally, server selection procedures (pool policies) and security considerations have been explained.

Chapter 4

The Handlespace Management

THIS chapter describes the design and implementation for an efficient handlespace management. At first, a motivation for the effort on optimizing the handlespace management is provided. This is followed by the requirements for the handlespace management. After that, the design and implementation of the handlespace management approach used for both, the RSerPool prototype implementation RSPLIB and the simulation model RSPSIM, are described. This chapter is concluded by remarks on the validation of the implementation.

4.1 Introduction

As explained in section 3.6, RSerPool is applicable for a wide variety of applications. Each application may have its own requirements with respect to server selection, that is it may specify its own pool policy. Furthermore, pools of certain applications like real-time distributed computing (see subsection 3.6.5) and load balancing (see subsection 3.6.4) may become very large. For example, a big international company could decide to add all of its office PCs to a computation pool for simulation processing. A pool of e.g. 10,000 to 100,000 PCs can provide an enormous computation capacity: as of 2006, a single usual office PC provides a 2.5+ GHz CPU including FPU and vector unit, 512+ MBytes of memory and 80+ GBytes of harddisk space. Scaled by 4 to 5 orders of magnitude, the overall computation capacity of such a pool can not only compete with a supercomputer, it furthermore comes extraordinarily inexpensive: usually, an office PC is only utilized during working hours and even then it waits most of the time for user input.

In summary, pools may become large and there are many possible pool policies which have to be supported by PRs and PUs. It is therefore necessary to think about how to efficiently manage such handlespaces.

4.2 Implementation History and Lessons Learned

For the first, fast-track version of the RSPLIB prototype implementation (see also chapter 5), the naïve way of realizing the handlespace has been taken by using linear lists. That is, the handlespace has been realized as a list of pools, where each pool included a list of PEs. As list implementation, the functions provided by the GLIB library (see GNOME Project (2001)) have been used. This naïve approach has been rather simple, but it worked reasonably well for small pools of less than 10 PEs using the Round Robin, Least Used or Random policy (the standards documents at this time only defined the Least Used and Round Robin policies). However, tests have shown that more pool policies

Handlespace				
Pool PH h_1	Policy π_1	Pool Element ID-#e_{11}	Policy Info $\hat{\pi}_{11}$	Addresses a_{11}
		...	...	...
		Pool Element ID-#e_{1m_1}	Policy Info $\hat{\pi}_{1m_1}$	Addresses a_{1m_1}
...	...	...		
Pool PH h_n	Policy π_n	Pool Element ID-#e_{n1}	Policy Info $\hat{\pi}_{n1}$	Addresses a_{n1}
		...	...	...
		Pool Element ID-#e_{nm_n}	Policy Info $\hat{\pi}_{nm_n}$	Addresses a_{nm_n}

Table 4.1: The Handlespace Datatype Structure

may be useful and there have been ideas for application scenarios requiring much larger pools (e.g. real-time distributed computing, see subsection 3.6.5).

The necessity to evaluate functionalities of RSerPool for research purposes has led to the creation of the RSPSIM simulation model (see also chapter 6). Due to the limitations of the prototype's handlespace management, it has been decided to redesign it: pools have been realized as objects of an abstract class; the actual implementation of a pool including its storage functionality has been provided by a derived class which has also implemented the pool's selection policy. After that, it has been for example possible to realize the Round Robin policy pools using circular lists, Random pools using an array of PEs and Least Used pools using trees sorted by the PEs' load values. For the actual storage of lists or arrays, functions provided by the OMNeT++ library (see Varga (2005b,a)) – on which the simulation is based – have been used. While the new approach addressed scalability as well as extensibility, it became increasingly difficult to maintain and verify the variety of different policy and storage implementations. Furthermore, all new policies also had to be supported by the RSPLIB prototype, which has still used the old handlespace management implementation. Since the new handlespace management approach has been based on the OMNeT++ framework written in C++, it also has not been possible to simply port it back to the C-based prototype.

As a result of the handlespace management experiences obtained from prototype and simulation model, it has been decided to redesign it again – and finally do it the right way! The key requirements for the new handlespace management have been the necessity for only one storage mechanism, which is common for all policies, and the usability of the new system for the simulation model as well as for the prototype implementation. In the following section 4.3, the handlespace management will be defined in form of an abstract datatype. The implementation design will be described in section 4.4.

4.3 An Abstract Handlespace Management Datatype

In this section, the abstract datatype of the new handlespace management approach is described.

4.3.1 Handlespace Structure

Table 4.1 presents the structure of the abstract handlespace management datatype: the handlespace consists of a set of n pools, identified by their unique PHs h_1 to h_n ($h_i \in P$ for $i \in \{1, \ldots, n\}$ and P the set of all possible PHs). Each pool i uses a pool policy $\pi_i \in \Pi$, where Π denotes the set of all pool policies (e.g. $\Pi = \{\text{LU, RR, RAND}\}$). Furthermore, each pool i includes a non-empty set of m_i PE entries, denoted by their PE IDs e_{i1} to e_{im_i} ($e_{ij} \in \{1, \ldots, 2^{32} - 1\} \subset \mathbb{N}$ for all

$i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m_i\}$). Every PE also includes its policy information $\hat{\pi}_{ij} \in \hat{\Pi}_{\pi_i}$, where $\hat{\Pi}_{\pi_i}$ denotes the set of all valid policy information settings for policy π_i. Finally, each PE entry includes its transport address $a_{ij} \in T_{P_i} \times T_A \times \mathcal{P}(N_A)$, where N_A denotes the set of network-layer addresses (i.e. usually the union of IPv4 and IPv6 addresses) and T_A denotes the set of transport-layer addresses (i.e. port numbers) of the transport protocol T_{P_i} chosen for pool i. $T_{P_i} \in T_P$ for $i \in \{1, \ldots, n\}$, where T_P denotes the set of all supported transport protocols and transport uses (e.g. $T_P = \{$SCTP with Control Channel, SCTP with Data Channel only$\}$).

4.3.2 Operations for the Pool Element Functionalities

For the PE functionalities of a PR, the handlespace management has to provide registration handling and timer-based reachability monitoring.

4.3.2.1 Registration Handling

Upon registration, re-registration and deregistration, it is necessary to find an existing PE structure in the handlespace by its PH and PE ID: it is first necessary to find the pool structure the PE belongs to; after that, a lookup for the PE in the given pool can be performed. Furthermore, it is necessary to insert a new PE into the pool upon registration and to remove it from the pool upon deregistration.

Therefore, the handlespace management to be defined has to provide the two methods *hsMgtRegisterPE(PH, PE)* and *hsMgtDeregisterPE(PH, PE)* to provide registration and deregistration functionalities. Since a re-registration is simply a registration of an existing PE – and therefore the place where it is possible to decide whether a registration is actually a re-registration is the registration method itself – no separate method is necessary here.

4.3.2.2 Reachability Timers

A PR-H has to monitor the availability of its PEs by sending keep-alive messages (see subsubsection 3.7.1.3). That is, it must be possible to schedule a *Keep-Alive Transmission Timer* for each PE entry. Clearly, it must also be possible to cancel such a scheduled timer if the PE is being removed.

Furthermore, after an ASAP Keep-Alive message is sent, it is necessary to schedule a *Keep-Alive Timeout Timer*. If such a timer expires, no ASAP Keep-Alive Ack message has been received and the PE is assumed to be dead. Clearly, a Keep-Alive Timeout timer has to be cancelled if it either answers the Keep-Alive message or the PE is being removed.

A PR not being a PR-H for a certain PE has to pay attention to the PE's Registration Lifetime parameter (see subsubsection 3.9.2.1): the PE entry has to expire if it is not updated by a subsequent ENRP Update message within the given time span. Therefore, it is necessary to maintain a *Lifetime Expiry Timer*. This timer is scheduled when a PE entry is created (or updated) upon an ENRP Update message and cancelled (and usually re-scheduled) upon the following Update.

4.3.3 Operations for the Pool User Functionalities

To provide the PU functionalities, the handlespace management first has to perform the PE selection: one or more PE identities have to be selected from a pool given by its PH using the pool's selection policy. That is, after a successful lookup of the pool, a list of PE identities has to be obtained by applying the selection procedure specified by the pool's policy.

Furthermore, the PU functionality of the handlespace management includes the maintenance of the PU-side cache. That is, a *Cache Expiry Timer* has to be scheduled for each PE entry in the cache. When the timer expires, the corresponding PE identity has to be flushed from the cache.

For the pool user functionality of the PE selection, the handlespace management has to provide a function *hsMgtHandleResolution(PH)* which selects a certain number of PE identities from the handlespace.

4.3.4 Operations for the Registrar Functionalities

As explained in subsection 3.10.5, PRs synchronize their views of the handlespace by sequences of ENRP Handle Table Request and Handle Table Response messages. A Handle Table Response message only includes a limited fraction of handlespace data, due to message size limitation (65,535 bytes) and to avoid overloading endpoints and network. That is, if a PR has requested a handlespace copy from one of its peers, the serving PR remembers the place in the handlespace where to continue from when the requesting PR asks for the subsequent part using another Handle Table Request message.

That is, the handlespace management to be defined has to provide a function *hsMgtTraverse-(PR_ID, lastPH&, lastPE_ID&)* which returns a certain number of PE identities owned by the PR given by *PR_ID* (or all, if no PR is specified) and stores PH and PE ID of the last returned PE identity into the provided reference variables *lastPH* and *lastPE_ID*. A subsequent call to this function will resume at the *nearest* PE *after* the provided one.

Finally, for the PR synchronization functionality as described in subsection 3.10.5, it must be possible to obtain the handlespace checksum for

1. All PE identities handled by the PR itself (i.e. the PEs for which it is the PR-H) and

2. For all PE identities belonging to a certain peer PR of the operation scope.

4.4 The Handlespace Management Design

In this section, the implementation design for the abstract handlespace management datatype introduced in section 4.3 is described. While first ideas for this implementation design have already been suggested in Dreibholz (2004d), a more formal presentation has been provided in Dreibholz and Rathgeb (2005b).

The presentation of the handlespace management design in this section is structured as follows: First, the data structure for the handlespace and the realization of pool policies are explained. After that, the management of timer events and checksums is described, followed by the handlespace synchronization handling. Finally, possible algorithms to manage the presented structures and design decisions for the implementation are introduced.

4.4.1 Handlespace Data Structure

The main task of the handlespace management is clearly the storage of pool and PE information. Therefore, it is first necessary to define how these data entities are stored. The handlespace data structure design is shown in figure 4.1 and quite straightforward: the handlespace consists of a *Set* of pools – denoted as *Pool Set* – sorted by the pools' PHs; each pool consists of a Set of PE references sorted by the PE ID (this set is denoted as *Index Set*) and a second Set of PE references sorted by a policy-specific *Sorting Order* (this set is denoted as *Selection Set*). It is intentionally not defined how

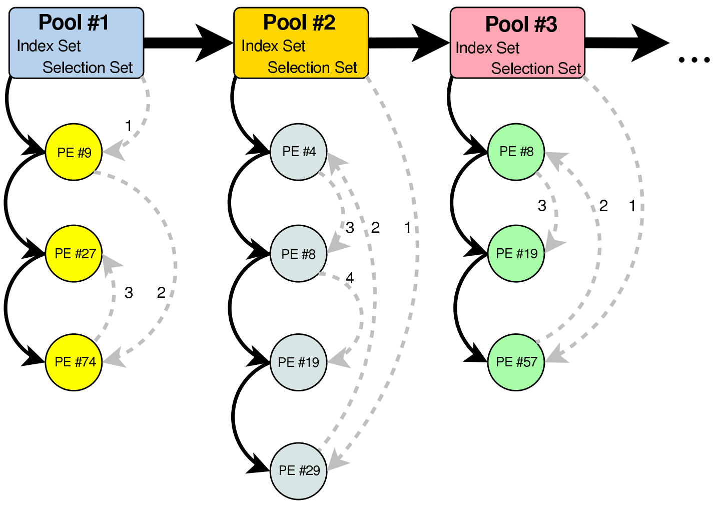


Figure 4.1: The Handlespace Data Structure Design

Pool PH h_i	Policy π_i	Pool Element ID-#e_{i1}	s_{i1}
	$S_i = 1 + \max\{s_{ij} \mid j \in \{1, \ldots, m_i\}\}$	$\ldots$	$\ldots$
		Pool Element ID-#e_{im_i}	s_{im_i}

Table 4.2: Sequence Numbers for Pools and Pool Elements

a *Set* is actually implemented (e.g. by using a linear list); possible choices for actually realizing a Set datatype are presented later in subsection 4.4.7.

Clearly, the effort for implementing insertion, removal and lookup of PEs is reduced to the corresponding operations of the Set datatype. A policy-based selection of PEs is reduced to take specific elements of the Selection Set (sorted by the policy-specific sorting order) using a policy-specific *Selection Procedure*. The next step therefore is to explain how sorting order and selection procedure are defined for certain pool policies. These definitions are the subject of the following subsection.

The *Default Selection Procedure* is simply to take PEs from the beginning of the Selection Set. The reason and a detailed example for this choice will be presented in subsubsection 4.4.2.2.

4.4.2 Policy Realizations

In this subsection, sorting orders and selection procedures for pool policies are presented. To simplify this task, some important helper constructs are defined first. After that, it is shown how to realize some important policies – accompanied by detailed examples.

4.4.2.1 Helper Constructs

Sequence Numbers The first helper construct to be introduced is the definition of sequence numbers for PEs and pools as shown in table 4.2: each pool element $e_{ij} \in S$ of pool i gets a sequence number s_{ij} which is unique in pool i. $S \subset \mathbb{N}_0$ denotes the set of all possible sequence numbers. The global

Pool PH h_i	Policy π_i	Pool Element ID-#e_{i1}	w_{i1}
	$W_i = \sum_{1 \leq j \leq m_i} w_{ij}$	...	...
		Pool Element ID-#e_{im_i}	w_{im_i}

Table 4.3: Pool Element Weights and Weight Sum

sequence number of pool i is defined as

$$S_i = 1 + \max\{s_{ij} | j \in \{1, \ldots, m_i\}\}, \tag{4.1}$$

i.e. the largest PE sequence number of the pool plus one. The uniqueness property of the PEs' sequence numbers within their pool will become very handy for the policy definitions later in this section.

Each time a new PE j is inserted into the pool i upon a registration, when its registration is updated by a re-registration or it is selected by a handle resolution, PE j's sequence number s_{ij} is set to S_i and therefore S_i is increased by one according to its definition in equation 4.1. This guarantees that each sequence number s_{ik} for $1 \leq k \leq m_i$ is unique within pool i. It is obvious that the operations necessary to maintain the sequence number are realizable in $O(1)$[1] time.

While in theory $S = \mathbb{N}_0$, S must be of finite size for any practical implementation. That is, S is usually a 32-bit or 64-bit unsigned integer datatype. Therefore, care has to be taken to avoid a sequence number overflow when S_i is incremented. In this case, a re-sequencing is necessary: The PEs have to be traversed in their current order and their sequence numbers have to be set starting from 0. Then, $S_i = m_i$. Obviously, such a re-sequencing of pool i can be performed in $O(m_i)$ time. In practice however, using a 64-bit unsigned integer datatype for the sequence number, a re-sequencing will never happen in realistic runtime scenarios[2].

Weights and Weight Sum As another helper construct for the definition of pool policies, a weight constant $w_{ik} \in \mathbb{N}$ is introduced for each PE of pool i and $1 \leq k \leq m_i$, as illustrated in table 4.3. Furthermore, the weight sum W_i of pool i is defined as follows:

$$W_i = \sum_{1 \leq j \leq m_i} w_{ij}. \tag{4.2}$$

It is obvious that the operations necessary to maintain the weight sum are realizable in $O(1)$ time.

Now, for any number $r \in [1, \ldots, W_i] \subset \mathbb{N}$, exactly one PE k of pool i fulfils the condition

$$\sum_{1 \leq j \leq k-1} w_{ij} < r \leq w_{ik} + \sum_{1 \leq j \leq k-1} w_{ij}. \tag{4.3}$$

This property will be utilized for the easy selection of random elements as being presented later in subsubsection 4.4.2.5.

4.4.2.2 Round Robin

The first policy to be defined is the default policy of Tüxen and Dreibholz (2006b): Round Robin. Using the sequence numbers helper construct defined in subsubsection 4.4.2.1, its definition becomes

[1] $O(f) := \{g : \mathbb{N} \rightarrow \mathbb{N} | \exists c > 0, n_0 \forall n \geq n_0 \; : \; g(n) \leq cf(n)\}$

[2] Nevertheless, the implementation introduced in this chapter correctly handles the re-sequencing.

Pool “Example”	Policy RR $S_1 = 4$	Pool Element ID-#1	$s_{11} = 1$
		Pool Element ID-#2	$s_{12} = 2$
		Pool Element ID-#3	$s_{13} = 3$

Pool “Example”	Policy RR $S_1 = 5$	Pool Element ID-#2	$s_{12} = 2$
		Pool Element ID-#3	$s_{13} = 3$
		Pool Element ID-#1	$s_{11} = 4$

Table 4.4: A Round Robin Policy Example

rather simple: the sorting order is simply to sort the PEs by sequence number in ascending order. The selection procedure to select a PE is to simply take the first PE of the Selection Set.

An example for using the Round Robin selection is provided in table 4.4. The upper part shows the pool before the selection: PEs #1, #2 and #3 have the sequence numbers 1, 2 and 3. Therefore, the pool's global sequence number S_1 is 3+1=4, according to equation 4.1. Selecting a PE means taking the first PE of the Selection Set, i.e. PE #1 is selected. After that, as defined in subsubsection 4.4.2.1, the sequence number of PE #1 is set to the pool's global sequence number (4) and the global sequence number is incremented by one (to 5). Since PE #1's new sequence number now is the highest one in the pool, this PE goes to the end of the Selection Set. A subsequent selection will chose PE #2, then PE #3, then PE #1 again, etc.. That is, the desired round robin behaviour is provided.

4.4.2.3 Weighted Round Robin

Implementing the Weighted Round Robin policy is slightly more complicated than a simple Round Robin selection: since PEs may be selected multiple times per round, it is first necessary to introduce a counter $v_{ij} \in \mathbb{N}$ for each PE j ($1 \leq j \leq m_i$) of pool i. The introduced variable has been called *Virtual Counter* (since PEs may have multiple virtual occurrences in the round robin list). Furthermore, it must be possible to decide whether a PE has already left the current round. Therefore, a *Round Counter* $r_{ij} \in \mathbb{N}_0$ is introduced for each PE j ($1 \leq j \leq m_i$) of pool i. This round counter r_{ij} denotes the number of the round robin round the PE awaits its next selection in.

Each time a PE entry j of pool i is selected, its virtual counter v_{ij} is updated as follows:

$$v_{ij} = \begin{cases} v_{ij} - 1 & (v_{ij} > 1) \\ w_{ij} & \text{(else)} \end{cases} .$$

In the second case (v_{ij}=1) the PE has left the current round robin round. Therefore, its round counter r_{ij} must be incremented by one and its virtual counter v_{ij} be reset to the PE's weight constant w_{ij}. Care has to be taken of an overflow of the round counter datatype when actually implementing it: In this case, the PEs' round counters in pool i can be renumbered in $O(m_i)$ time. However, using a sufficiently large datatype, this practically never happens[3].

The sorting order of the Selection Set is defined as sorting by the compound key of

- The round counter in ascending order,
- The virtual counter in ascending order and

[3]Nevertheless, the implementation introduced in this chapter correctly handles the round counter renumbering.

Pool h_1	$\frac{\text{Policy WRR}}{S_1 = 8}$	Pool Element ID-#5	$w_{11} = 2$	$r_{11} = 20$	$v_{11} = 2$	$s_{11} = 5$
		Pool Element ID-#1	$w_{12} = 1$	$r_{12} = 20$	$v_{12} = 1$	$s_{12} = 6$
		Pool Element ID-#9	$w_{13} = 1$	$r_{13} = 20$	$w_{13} = 1$	$s_{13} = 7$

Pool h_1	$\frac{\text{Policy WRR}}{S_1 = 9}$	Pool Element ID-#1	$w_{12} = 1$	$r_{12} = 20$	$v_{12} = 1$	$s_{12} = 6$
		Pool Element ID-#9	$w_{13} = 1$	$r_{13} = 20$	$v_{13} = 1$	$s_{13} = 7$
		Pool Element ID-#5	$w_{11} = 2$	$r_{11} = 20$	$v_{11} = 1$	$s_{11} = 8$

Pool h_1	$\frac{\text{Policy WRR}}{S_1 = 10}$	Pool Element ID-#9	$w_{13} = 1$	$r_{13} = 20$	$v_{13} = 1$	$s_{13} = 7$
		Pool Element ID-#5	$w_{11} = 2$	$r_{11} = 20$	$v_{11} = 1$	$s_{11} = 8$
		Pool Element ID-#1	$w_{12} = 1$	$r_{12} = \mathbf{21}$	$v_{12} = 1$	$s_{12} = 9$

Table 4.5: A Weighted Round Robin Policy Example

Pool “Example”	$\frac{\text{Policy LU}}{S_1 = 8}$	Pool Element ID-#7	$l_{11} = 10$	$s_{11} = 6$
		Pool Element ID-#2	$l_{12} = 10$	$s_{12} = 7$
		Pool Element ID-#11	$l_{13} = 40$	$s_{13} = 3$

Pool “Example”	$\frac{\text{Policy LU}}{S_1 = 9}$	Pool Element ID-#2	$l_{12} = 10$	$s_{12} = 7$
		Pool Element ID-#7	$l_{11} = 10$	$s_{11} = 8$
		Pool Element ID-#11	$l_{13} = 40$	$s_{13} = 4$

Table 4.6: A Least Used Policy Example

- The sequence number in ascending order.

As for the Round Robin policy, the default selection procedure (i.e. taking the PE identities from the top of the list) is applied.

An example for the Weighted Round Robin policy is provided in table 4.5. The upper part of the table shows the initial state of the pool: all three PEs are in round 20 ($r_{11} = 20$; $r_{12} = 20$; $r_{13} = 20$), PE #5 is weighted by 2 ($w_{11} = 2$) and still has to be selected 2 times ($v_{11} = 2$) in its current round (20); the other PEs are weighted by 1 ($w_{12} = 1$; $w_{13} = 1$) and still have to be selected once ($v_{12} = 1$; $v_{13} = 1$).

The first selection results in taking PE #5; the pool state after this selection is shown in the middle part of table 4.5. The virtual counter v_{11} of PE #5 has been decreased by 1, i.e. this PE still has to be selected once in the current round robin round (20). A subsequent selection – taking PE #1 – results in the pool status shown in the lower part of table 4.5: PE #1 has left the current round 20 (hence $r_{12} = 21$); the next time this PE will be selected again is within the following round 21.

4.4.2.4 Least Used and Priority Least Used

Using the Least Used policy, a load state $l_{ij} \in [0,\, 1] \subset \mathbb{R}$ is provided for each PE j ($1 \leq j \leq m_i$) of pool i. Clearly, the sorting order of the Selection Set is defined by a sorting key composed of

Pool “Example”	Policy WRAND $S_1 = 5$ $W_1 = 7$	Pool Element ID-#7	$s_{11} = 1,\ w_{11} = 1$
		Pool Element ID-#2	$s_{12} = 2,\ w_{12} = 3$
		Pool Element ID-#8	$s_{13} = 3,\ w_{13} = 2$
		Pool Element ID-#6	$s_{14} = 4,\ w_{14} = 1$

Table 4.7: A Weighted Random Policy Example

- The load state is ascending order and
- The sequence number in ascending order.

While the load state obviously ensures that a least utilized PE is selected, the sequence number not only ensures uniqueness of the composed sorting key, but also provides a round robin selection among multiple least-loaded PEs. This property will also be demonstrated by the example below. The selection procedure is the default one, i.e. the PEs are taken from the top of the Selection Set.

Table 4.6 provides an example for the Least Used policy; the upper part shows the pool state before the selection. In this initial state, PE #7 and PE #2 are loaded by 10%, while PE #11 is loaded by 40%. When PE #7 is selected (since it is the first PE of the Selection Set), its sequence number is updated according to subsubsection 4.4.2.1. The lower part of table 4.6 shows that the selected PE #7 is re-inserted into the Selection Set as last element of the “10% load” section, due to its updated sequence number. That is, using the sequence number as second part of the composed sorting key ensures round robin selection among multiple least-used PEs.

The definition of Priority Least Used is quite similar to Least Used. Next to the definition of the load state, a load increment value $\hat{l}_{ij} \in [0,\, 1] \subset \mathbb{R}$ is defined for each PE j $(1 \leq j \leq m_i)$ of pool i. The load increment $\hat{l}_{ij}$ describes the additional load of the PE caused by the handling of a further session. Then, the composed load can be defined as $l^{*}_{ij} = \min(1.0,\, l_{ij} + \hat{l}_{ij})$, i.e. the sum of both, the load state and the load increment, but not exceeding 100%. Finally, the sorting order of the Selection Set is defined using a sorting key composed of

- The composed load in ascending order and
- The sequence number in ascending order.

4.4.2.5 Random and Weighted Random

Random selections cannot simply take PEs from the top of the Selection Set. Therefore, the weight sum helper construct defined in subsubsection 4.4.2.1 is utilized for the selection procedure: the weight constant w_{ij} of each PE j $(1 \leq j \leq m_i)$ of pool i corresponds to PE j’s proportional selection probability. Then, the selection procedure is simply to choose a random number $r \in_R [1, \ldots, W_i]$ (see also equation 4.2) and take the element k that uniquely fulfils equation 4.3. Using a uniform distribution for the choice of r, the Weighted Random policy provides the desired property of choosing elements with probabilities being proportional to the PEs’ provided weight constants. Random selection is only a special case of Weighted Random selection, where $w_{ij} = 1$ for each PE j $(1 \leq j \leq m_i)$ of pool i. Since it is necessary to define a unique sorting order on the Selection Set for storage purposes, simply the PE ID is taken as sorting key.

An example for the selection procedure of the Weighted Random policy is provided in table 4.7: the PEs #7, #2, #8 and #6 are weighted by 1, 3, 2 and 1; therefore, $W_1 = 1 + 3 + 2 + 1 = 7$.

	Timer	Used where?
1	Keep-Alive Transmission	PR-H
2	Keep-Alive Timeout	PR-H
3	Lifetime Expiry	PR (but not PR-H)
4	Cache Expiry	PU-side Cache

Table 4.8: The Timers of the Handlespace Management

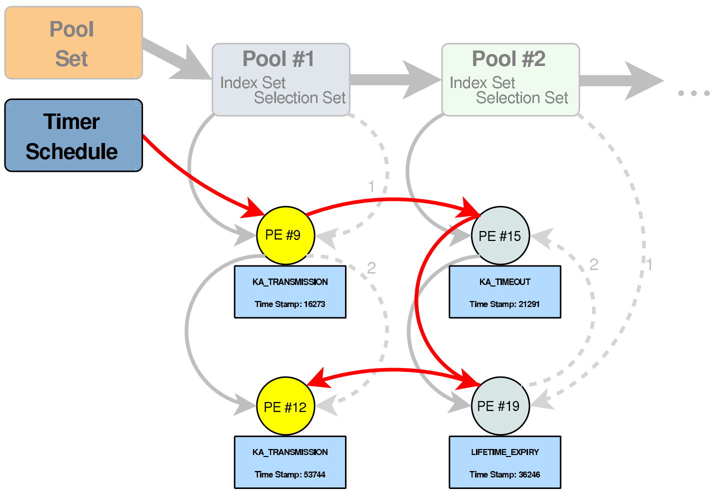


Figure 4.2: The Timer Schedule Structure

To select a PE, a random number $r \in_R [1, \ldots, 7]$ is chosen. Let $r = 5$. In this case, only $k = 3$ satisfies the condition of equation 4.3: $\sum_{1 \leq j \leq k-1} w_{1j} < 5 \leq w_{1k} + (\sum_{1 \leq j \leq k-1} w_{1j})$; that is, $1 + 3 < 5 \leq 2 + (1 + 3)$ and therefore the selected PE is the third (k-th) of the list: PE #8.

4.4.3 Timer Management

Table 4.8 lists the set of handlespace management timers for PE structures and places where the specific timers are used. As shown, the two Keep-Alive timers are only used by PR-Hs, while the Lifetime Expiry timer is used by non-PR-Hs only and the Cache Expiry Timer is used by the PU-side cache only. Furthermore, a PR-H never uses both Keep-Alive timers simultaneously: if the Keep-Alive Transmission timer is scheduled, the PR-H does not wait for a timeout of a sent ASAP Endpoint Keep-Alive message (using the Keep-Alive Timeout timer) and if it has sent such a message, it does not schedule another transmission until the managed PE answers using an ASAP Endpoint Keep-Alive Ack.

The fact that only one of the timers can be scheduled at any certain point of time simplifies the timer management: for any PE structure, it is only necessary to store the type of timer and its schedule time stamp. It is then possible to realize the handlespace timer management as shown in figure 4.2: the *Timer Schedule* can be designed as handlespace-global set of PE references, obviously sorted by the

Algorithm 2 The Two-Part 16-Bit Internet Checksum Algorithm

```
 1 unsigned int beginCalculateInternetChecksum16(void* data, size_t count)
 2 {
 3    unsigned short* addr = (unsigned short*)data;
 4    unsigned int    sum  = 0;
 5
 6    while(size >= sizeof(*addr)) {   /* Main calculation loop */
 7       sum += *addr++;
 8       size -= sizeof(*addr);
 9    }
10    if(size > 0) {   /* Add left-over byte, if any */
11       sum += * (unsigned char*)addr;
12    }
13    return(sum)
14 }
15
16 unsigned short finishCalculateInternetChecksum16(unsigned int sum)
17 {
18    while(sum >> 16) {   /* Fold 32-bit sum to 16 bits */
19       sum = (sum & 0xffff) + (sum >> 16);
20    }
21    return(~sum);
22 }
```

timer schedule time stamp in ascending order in the first order. To enforce uniqueness of the sorting order, the PE references are stored with a sorting order defined by the sorting key composed of time stamp, PE ID and PH. Upon expiry of a PE's scheduled timer, the type-specific handling procedure has to be called.

4.4.4 Checksum Handling

As explained in subsection 3.10.5, the 16-bit Internet Checksum algorithm defined in Braden et al. (1988), Rijsinghani (1994) is used to calculate the handlespace checksum. This checksum algorithm provides an important property: it allows incremental updates. That is, it is not necessary to re-calculate the checksum of the complete handlespace upon a change of the handlespace data. If a new PE is added, its checksum can be added to the current handlespace checksum. On removal, the checksum of the PE can simply be subtracted. If a PE's registration information is updated with information affecting its checksum, the PE checksum before the change can be subtracted and the new PE checksum can be added. In summary, the checksum maintenance can be realized in $O(1)$ time.

However, to actually implement an incremental checksum update, it is insufficient to only store the 16-bit checksum as computed from the function presented in algorithm 1 (see subsection 3.10.5). The reason requires a slightly further analysis of the checksum algorithm: at first, the memory block over which the checksum has to be calculated is viewed as an array of 16-bit unsigned integer values. These 16-bit values, plus a possible left-over byte, are summed up using a 32-bit accumulator (line 6 to 12). In the final step (line 13 to 16), the number of 16-bit overflows (the carry part in the upper 16 bits) is added to the accumulator, which now fits into 16 bits. After that, the accumulator is inverted and returned as 16-bit checksum value.

In order to perform incremental updates, it is necessary to keep the full accumulator value as

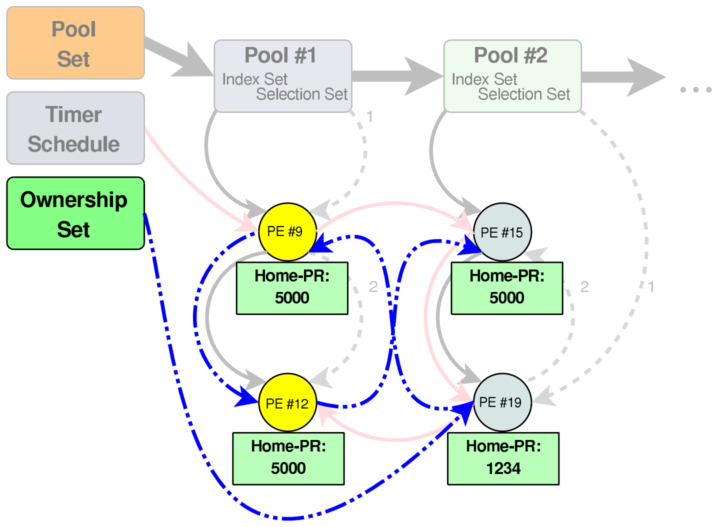


Figure 4.3: The Ownership Set Structure

provided by the first part of the algorithm. Using this value, checksum additions and removals are simply performed by regular 32-bit additions and subtractions. The actual checksum value can then be obtained by applying the carry addition and inversion.

The resulting two-part calculation algorithm for the 16-bit Internet Checksum is shown in algorithm 2. Its function *beginCalculateInternetChecksum16()* calculates the 32-bit accumulator value. This value is used in all places of the handlespace storage structures. Therefore, the addition and subtraction of checksums become trivial. The actual 16-bit Internet Checksum value is obtained from the accumulator value by calling *finishCalculateInternetChecksum16()*. This 16-bit value is only used if the checksum has to be exported from the handlespace management.

4.4.5 Synchronization

For the handlespace synchronization procedure as defined in subsection 3.10.5, it is required to be able to resume the traversal of the handlespace at a certain remembered point. The naïve way to remember the handlespace position could be a pointer to the last PE structure being delivered to the requesting PR in an ENRP Handle Table Response message. The difficulty of this approach is that the handlespace content may change between two successive calls of the *hsMgtTraverse()* function: the remembered PE may already be gone. Therefore, such a pointer requires active maintenance – in particular processing time – on every handlespace change. Furthermore, multiple synchronization operations may be in progress simultaneously (with different peer PRs, of course). This means that pointer maintenance actually means keeping a set of such pointers up-to-date. In summary, the pointer approach would imply to possibly put significant costs on frequent operations to keep the infrequent synchronization operation simple. Therefore, a superior approach seemed to be appropriate.

The resulting synchronization implementation approach simply remembers PH and PE ID of the last PE obtained by the traversal function. A subsequent call can then find the PE in the handlespace which has the next-nearest identity after the remembered one, according to the sorting order of pools (by PH) and PEs (by PE ID). After the lookup has been performed, the traversal can be continued from the obtained PE. That is, only the synchronization operation itself carries the burden of making

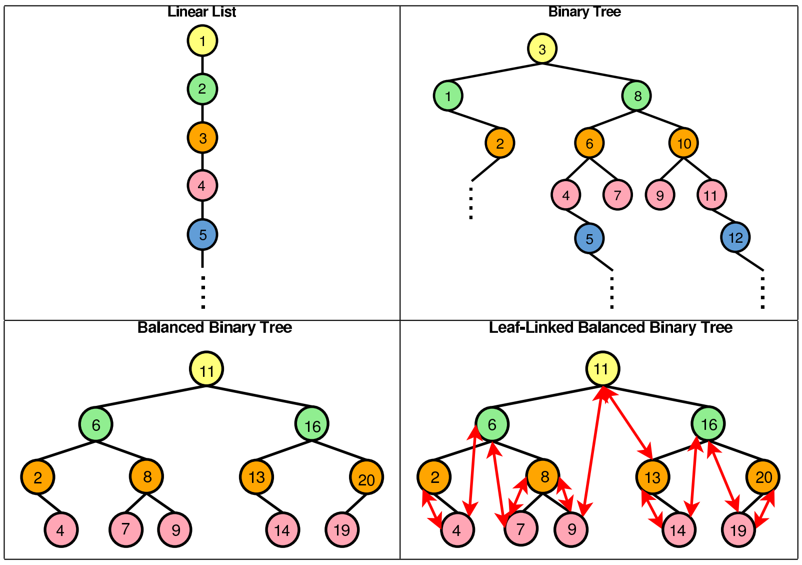


Figure 4.4: An Overview of Storage Structures

its own resumption possible – all other handlespace operations remain unaffected.

To simplify traversing the PEs owned by a specific PR, a further set denoted as *Ownership Set* is introduced. In this set, the PE references are stored with a sorting order defined by the sorting key composed of Home-PR ID, PE ID and PH. Figure 4.3 presents an example Ownership Set structure.

4.4.6 Pool Handle Management

The RSerPool standards documents do not set a limit for the size of a pool handle. While in fact the length of a pool handle is only limited by the message size (therefore, it could be as long as about 65,000 bytes; see section 3.8), it is impractical to support overly long handles. In this case, it would be necessary to dynamically allocate memory for a PH in addition to the pool structure itself. Therefore, is has been decided to limit the PH size to 32 bytes and reserve a fixed size of 32 bytes for the PH, as part of the pool structure itself. This results in a simplification of the handlespace management.

32 bytes seem to be sufficient for any reasonable ASCII or Unicode representation of a pool and are also sufficient for storing a 256-bit SHA-256 hash value[4] or a random value of the same size. Such hash values could be used to create PHs in the scenario of RSerPool-based Mobility Support for SCTP (see subsection 3.6.6).

Last but not least, a reasonable limit for the PH size also enhances security: the PH is the only field of an ASAP Registration message (see subsubsection 3.9.2.1) which can be freely set by the registering PE. That is, an attacker having gained the permission to register PEs would be able to quickly let PRs allocate large portions of memory in order to store PHs.

4.4.7 Storage Structures and Algorithms

The previous subsections frequently used the term Set to denote a storage class being able to keep objects in a certain order. However, it has not been explained how a Set is actually implemented. This

[4]See NIST (2002), Eastlake and Jones (2001), Rivest (1992).

Algorithm	Operation	Average Runtime	Worst Case Runtime
Linear List	Insertion	$O(n)$	$O(n)$
	Removal	$O(n)$	$O(n)$
	Lookup	$O(n)$	$O(n)$
Binary Tree	Insertion	$O(\log n)$	$O(n)$
	Removal	$O(\log n)$	$O(n)$
	Lookup	$O(\log n)$	$O(n)$
Treap	Insertion	$O(\log n)$	$O(n)$
	Removal	$O(\log n)$	$O(n)$
	Lookup	$O(\log n)$	$O(n)$
Red-Black Tree	Insertion	$O(\log n)$	$O(\log n)$
	Removal	$O(\log n)$	$O(\log n)$
	Lookup	$O(\log n)$	$O(\log n)$

Table 4.9: Storage Structures and their Computational Complexity

task is the final step to complete the handlespace management. Clearly, since the efficiency of the handlespace management heavily relies on the performance of the underlying storage structures and algorithms, it is crucial to carefully consider their choice and implementation.

The naïve solution to actually realize a Set is obviously a linear list, a more appropriate storage structure is based on binary trees. To explain and compare various possible structures would exceed the scope of this chapter; such descriptions can be found in computer science literature like Cormen et al. (1998). In short, possible solutions for the Set datatype are linear lists and binary trees. The average and worst case access times of both structures are presented in table 4.9, an illustration of the structures is provided in figure 4.4. While in the average case operations on a binary tree have a runtime of $O(\log n)$, their worst case runtime is still in $O(n)$, as for the linear list. However, balanced trees like the AVL tree (see Adelson-Velskii and Landis (1962)) ensure that this worst case (almost) never occurs.

The state-of-the-art techniques for balanced trees are the following structures:

Treap: A treap is a binary tree, where each node n includes a priority n_{Priority}. This priority is a random number, which is chosen when the node is inserted. The insertion and removal operations on a treap are similar to a regular binary tree, except for enforcing the following condition by applying appropriate rotations of the tree's nodes: if node c is a child of node p, then $c_{\text{Priority}} \geq p_{\text{Priority}}$. The operations runtime of the treap is $O(\log n)$ on average, but still $O(n)$ in the worst case. However, the randomization makes this case very unlikely. For a detailed introduction to treaps, see Seidel and Aragon (1996), Aragon and Seidel (1989).

Red-Black Tree: Unlike the treap, the red-black tree uses a deterministic strategy to keep it balanced. This results in a guaranteed operations runtime of $O(\log n)$. A red-black tree is defined as follows: each node n includes a colour n_{Colour}, which may be either red or black (hence the name "red-black tree"). By appropriately rotating the nodes of the tree, the following constraints are enforced upon insertion and removal of nodes:

- The root node is black,
- All leaves are black,

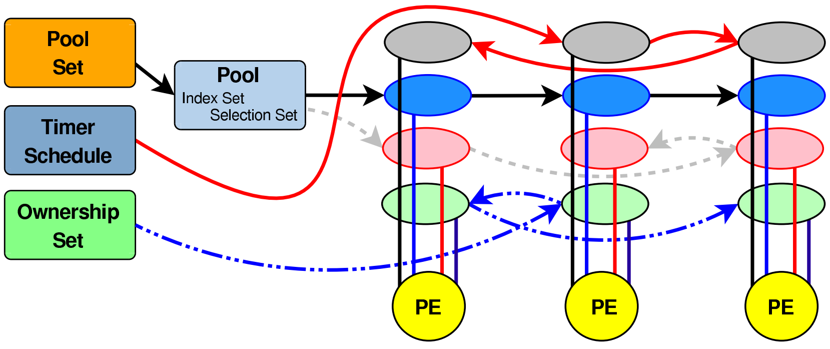


Figure 4.5: A Linkage Implementation using Separate Node Structures

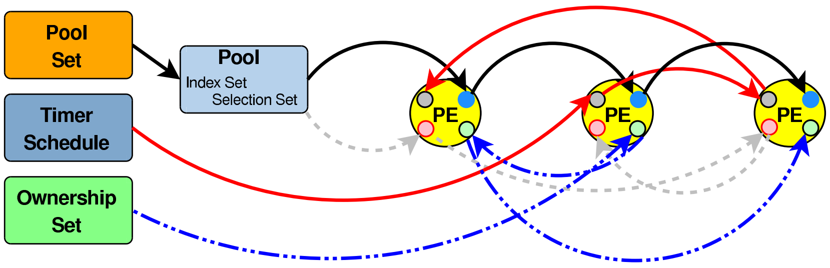


Figure 4.6: A Linkage Implementation using Integrated Node Structures

- Both children of a red node are black and
- All paths from any given node to its leaf nodes include the same number of black nodes.

Based on these constraints, it can be proven that the longest possible path from the root to a leaf is at most twice as long as the shortest possible path. That is, a logarithmic height is assured. Details on red-black trees can be found in Guibas and Sedgewick (1978), Cormen et al. (1998).

An important observation of the handlespace management's usage of the Set datatype is that accesses to the next or previous element of a given element are frequent. A useful optimization if using a tree-based implementation might therefore be to further link the elements using a doubly-linked linear list. This strategy is denoted as *Leaf-Linking*. Assuming an access runtime of $O(\log n)$, this additional linking does not increase the overall complexity.

Since the implementation of the Set datatype is crucial for the handlespace management performance and it is not obviously clear which implementation option actually provides superior performance for realistic handlespace size and access scenarios, performance tests have been made. Their results will be presented in chapter 7.

4.4.8 Node-Linkage Implementation

Independent from the storage algorithm used to implement the Set datatype to keep the handlespace structures and timers, it is worth to think about how objects are stored in such sets. Well-known libraries such as GLIB (see GNOME Project (2001)) or the classes provided by OMNET++ (see Varga (2005b,a)) work as follows: whenever an object is added to a storage class like a list or tree, a node structure is created first. This node structure is then inserted into the list itself and references to the actual object. Figure 4.5 illustrates this design for the handlespace management structures. The advantage of this approach is that the actual object which is managed by the list does not need to know any details of the list it is stored in; in particular, an arbitrary object may be kept in a virtually unlimited number of lists.

However, the approach of using separate nodes has some severe disadvantages: First, the insertion of objects may fail due to insufficient resources to allocate the node structure. While capturing this exception is no significant problem if e.g. a new PE is inserted (in this case, the PE registration could be rejected with an appropriate error code), failing e.g. to schedule a timer becomes a severe problem: should a PE be removed if it is impossible to schedule its Keep-Alive Timeout Timer? What should the RSerPool protocols do? ASAP does not have any possibilities to meaningfully signal such an event to the PE.

Another problem of using separate node structures is memory fragmentation[5]: a modern operating system utilizes the memory management unit (MMU) of a processor to map virtual address space to physical memory and therefore a continuous memory block in virtual address space may actually map to physical memory blocks being widely distributed in the physical address space (see Eisele (2002) for more details). However, lightweight operating systems like AmigaOS/AROS (Amiga Research Operating System, see AROS Development Team (2005a)) or in particular router operating systems like the Cisco™ Internet Operating System (IOS, see Cisco Systems (2004)) do not support a MMU – the hardware for which these systems are designed for simply does not provide it. Over the time, the memory of such operating systems gets fragmented: due to consecutive allocations and deallocations of small memory blocks, the number of large continuous memory blocks decreases, making it more and more difficult to find such a block. After some time, allocations of larger memory blocks may fail, due to the lack of continuous memory blocks – although ample free memory (but in small, scattered pieces) may be free. The approach to cope with such problems is to keep per-application pools of pre-allocated memory and take the application memory out of such pools (see AROS Development Team (2005b) for how this is realized for AmigaOS/AROS).

Finally, allocations and deallocations require runtime – regardless of whether the memory is taken from a dedicated memory pool or from a global memory management. The more node structures are created and destroyed, the more runtime is wasted for memory management.

In summary, the approach of separate node structures seems to be unsuitable and inefficient for the handlespace management. Therefore, a superior approach has been chosen: the objects to be stored are directly equipped with the node structures necessary to keep them in their sets. For example, PE references are usually kept in the Index Set, Selection Set, Ownership Set and the Timer Schedule. Therefore, the PE structure includes one integrated node structure for each of the four sets as shown in figure 4.6. As part of the PE structure, the memory is already available with the PE structure itself. No further memory management operations for this PE are necessary until it is finally removed. In particular, no Set-related operation for this PE can fail due to a lack of resources. And finally, the approach saves some[6] amount of memory, since it is more efficient to manage one block of allocated memory than to manage five (the PE structure itself plus four separate node structures).

[5]See Neely (1996), Berger et al. (2001), Berger (2002), Berger et al. (2002) for details on this subject.

[6]The amount of saved memory is highly dependent on the CPU architecture and the used operating system.

4.5 The Handlespace Management Validation

The code actually implementing the handlespace management design introduced in section 4.4 includes more than 11,500 lines of C code, of which about 5,500 lines actually implement the handlespace management and about 3,500 lines realize four classes of storage implementations: linear list, binary tree, treap and red-black tree. The remaining lines provide helper functions for various tasks – like random number generation, string handling, timestamp management and conversion as well as the handling of transport addresses. Although carefully created, it is obvious that the code has some errors and it therefore has been necessary to take adequate care of its validation before using it for simulations and tests.

The validation strategy is based on four building blocks: assertions, functions for consistency checking, regression tests and the usage of debugging software. These four items are described in the following.

4.5.1 Assertions

An important experience from the implementation of the RSPLIB and former projects has been that wrong parameters passed to functions may often result in problems at completely different parts of the code. Since such errors are difficult to track down, the handlespace management implementation checks assertions at all crucial parts of the code. Especially, this means to check the validity of important parameters passed to internal functions in the handlespace management. For public functions, such checks are mandatory in order to provide a robust implementation. Further important assertions to check are e.g. whether or not a timer is already scheduled when it is tried to schedule or cancel it. Together with the following part of the validation strategy – the consistency checking functions – assertion checking has been proven to be a very useful technique in the development of the handlespace management.

4.5.2 Consistency Checking Functions

The handlespace management consists of a complex structure of pools, PEs and timers as well as a connection to the peer list management. A function which can verify the consistency of the handlespace management therefore has been realized. This function includes checks like the following ones:

- Are the storage structures (e.g. red-black tree) valid?
- Is each PE referenced in the Index Set also referenced by the Selection Set and vice versa?
- Does each PE in a pool use the same policy, transport protocol and options?
- Are the policy parameters of each PE valid?
- Has each PR entry in the peer list the correct checksum referring to the managed handlespace?

The handlespace management validation function is invoked in form of an assertion check after each operation on the handlespace. Since this operation is rather expensive, this assertion check is turned on by a compile-time option for testing purposes. Production versions of the implementation do not apply the consistency checks.

4.5.3 Regression Tests

To verify the correct handling of different input by the handlespace management operations, regression tests have been applied. A set of different input data together with the expected results has been collected in form of test routines which are invoked by the regression test program with the consistency checking functions enabled. For example, there is a test routine checking that the registration function rejects a registration trying to register a PE using the Least Used policy into an already existing pool using Weighted Random. The regression test program has been used after every change of the implementation to verify that it is still working correctly for the collected test cases. During development the test cases have been continuously extended each time new but uncovered errors have been detected, so that the current version – including about 1,900 lines of test code – can be expected to be quite reliably cover the cases of different inputs.

4.5.4 Validation Software

To further enhance the correctness of the handlespace management implementation, the memory debugging software VALGRIND[7] has been intensively used, in particular in combination with the regression test program. In short, VALGRIND is a x86 binary code interpreter that actually executes a program and keeps track of all memory accesses. In particular, it does not only remember which bit belongs to an allocated chunk of memory but also keeps track which bit is still uninitialized.

That is, VALGRIND does not only detect accesses to invalid or already deallocated memory blocks but also warns if an uninitialized bit is used e.g. in a conditional branch. Especially the second category of errors is otherwise very difficult to detect: for example, a 16-bit variable is uninitialized, i.e. it contains a random value. While the probability is 65,535:1 that it contains a non-zero value, it may just contain 0 in an inappropriate moment and lead to a severe and almost untraceable (since difficult to reproduce) malfunction of the system.

4.6 Summary

In this chapter, the design and implementation of the handlespace management – used for both, the RSerPool prototype implementation RSPLIB as well as the simulation model RSPSIM – have been introduced. The main requirements for the handlespace management approach have been the support and extensibility for various pool policies as well as the efficient handling of large pools. These requirements are achieved by reducing the effort of maintaining the handlespace to the management of sorted sets, while pool policies are defined by a certain sorting order as well as a selection procedure. Based on these ideas, a set of policy realizations has been presented, together with detailed examples. After that, the handling of timers, the checksum handling, the synchronization procedure and the pool handle management have been explained. Finally, some general optimizations of the handlespace data storage have been presented. The last open issue is how to actually realize the Set datatype on which the handlespace management is based. This question will be answered as part of the handlespace management performance evaluations in chapter 7.

[7]See Seward and Nethercote (2005), Valgrind Developers (2005).

Chapter 5

The RSPLIB Prototype Implementation

THIS chapter describes the design and implementation of the RSPLIB prototype. First, a short overview of the prototype's history is given. After that, the requirements for the prototype and its important design decisions are explained. This is followed by a description of the prototype parts: the PR, the PU/PE library and the demonstration system. A survey of other RSerPool implementations concludes this chapter.

5.1 Introduction

The beginning of our RSerPool activities has been a three-year cooperation project between our group (i.e. the Computer Networking Technology Group of the Institute for Experimental Mathematics at the University of Duisburg-Essen) and Siemens AG, Munich, which has also been supported[1] by the Bundesministerium für Bildung und Forschung (BMBF, i.e. the German Ministry for Education and Research). Goal of this cooperation, started October 01, 2001, has been to realize the world's first Open Source prototype implementation of the upcoming RSerPool standard in order to

- Verify that the protocols defined by the IETF drafts are actually useful and working,
- To perform research on the capabilities of RSerPool and suggest improvements of the standard, as well as
- Bringing changes and improvements into the IETF standardization process.

The RSerPool project – called the RSPLIB prototype – has been the continuation of another successful cooperation with Siemens AG: the implementation of the Open Source, userland SCTP prototype SCTPLIB together with its API library SOCKETAPI (see Jungmaier (2005) and Tüxen (2001)). Since October 01, 2004, our RSerPool activities including the RSPLIB prototype are supported by the Deutsche Forschungsgemeinschaft (DFG), after the cooperation project with Siemens AG had been successfully finished the day before.

Since the beginning of the project, the RSPLIB prototype is publicly available under the GNU General Public License[2] (GPL) and can be downloaded at Dreibholz (2006c). It is now used by the IETF RSerPool WG as reference implementation. The prototype implementation consists of three separate parts – a PR, a library for PUs and PEs, as well as a demonstration system. These parts

[1]Förderkennzeichen 01AK045.

[2]See Free Software Foundation (1991).

are described after first defining the requirements for the prototype and providing a survey of the important design decisions.

5.2 The Requirements for the Prototype

An important requirement for the RSerPool prototype implementation to be designed and implemented has been to make it Open Source. That is, everybody interested in the deployment of RSerPool is able to test the prototype and to possibly contribute an extension or improvement. Therefore, we have chosen the GPL license, which on one hand allows modifications, but on the other hand also forces that modified versions have to be made publicly available again.

The most fundamental requirement for the prototype has been the independence of the underlying hardware architecture and operating system. In particular, the prototype should have been designed with applications on embedded devices in mind. That is, it should have been possible to use the prototype in networking and telecommunications equipment (e.g. routers and telephone signalling devices) being built by our project partner Siemens, as well as on mobile phones or PDAs.

5.3 The Design Decisions

Since platform and operating system independence is one of the main requirements for the RSPLIB prototype, it has been necessary to choose a common programming language first. Although an implementation in C++ would have somewhat simplified the implementation and maintainability of the code, it has been decided to use ANSI-C instead: ANSI-C compilers are available for almost any kind of exotic platform, while C++ ones are scarce.

The applicability of the RSPLIB prototype on exotic systems – in particular on systems without MMU – also has had a significant impact on the design and implementation of the handlespace management. See section 4.4 for further details on this subject.

In 2002, IPv6 support has already been available by all major operating systems, therefore providing the required IPv6 support has been easy. However, the SCTP protocol has been a novelty. While kernel implementations of SCTP for operating systems like Linux (LKSCTP, see LKSCTP (2006)), FreeBSD (KAME, see KAME (2006)), MacOS X and Solaris have already been under development, none of these SCTP stacks had by far reached a kind of maturity necessary to base a large software project like a RSerPool implementation on them. The only SCTP implementation powerful and stable enough to base the RSPLIB development on has been our own SCTP prototype SCTPLIB (see Tüxen (2001)). Together with its API library SOCKETAPI, it includes an API compatible to the upcoming standard defined in Stewart, Xie, Yarroll, Wood, Poon and Tüxen (2006). That is, basing the RSPLIB prototype on this standard SCTP API should ensure independence from the used SCTP implementation. However, not all functionalities defined in the API draft are currently supported by all implementations. That is, some kind of wrapper functionality is required for these implementations as part of the RSPLIB.

The RSPLIB prototype consists of a PR (described in section 5.5), a PU/PE library (described in section 5.6) and a demonstration system (described in section 5.7). Since all three parts require a common abstraction layer for the underlying operating system functionalities, this layer is described before the actual parts in the following section.

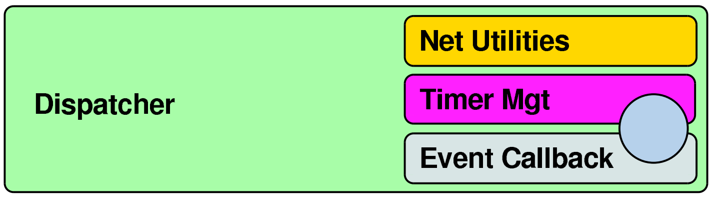


Figure 5.1: The Dispatcher Component

5.4 The Foundation Components

To cope with the requirement that the RSPLIB prototype should be easily portable to new operating systems, all system-dependent functionalities have been encapsulated into an abstraction layer. This abstraction layer has been called *Dispatcher*, its building blocks are presented in figure 5.1. These sub-components – Net Utilities, Timer Mgt and Event Callback – are described in the following.

5.4.1 Network Utilities

The duties of the *Net Utilities* sub-component of the Dispatcher include functionalities like transport address handling and conversion, byte order conversion and socket management. Mainly, these functionalities are simply provided by small wrapper functions for the underlying operating system's appropriate operations.

But one of the most important task of the Net Utilities sub-component is to build wrappers for missing SCTP functions of different SCTP implementations as explained in section 5.3: As of 2006, only the SCTPLIB implementation provides full support of the complete SCTP API defined in Stewart, Xie, Yarroll, Wood, Poon and Tüxen (2006); for all other implementations, the Net Utilities component has to provide some wrapper functionalities. However, it is only a question of time until the functionalities of these implementations catch up and the wrapper functions become obsolete.

5.4.2 Timer Management and Event Callback Handling

The handling of timers and events on network sockets (e.g. input on a SCTP association) are handled by the two sub-components *Timer Mgt* and *Event Callback*. The Event Callback component is responsible for managing event notifications on network sockets (e.g. notifying about incoming data). Furthermore, its duties also include timer event notifications in cooperation with the Timer Mgt sub-component. The Timer Mgt sub-component itself is responsible for managing, setting and cancelling timers.

Using the Dispatcher component as base, the three RSPLIB parts have been realized. These parts are described in the following three sections.

5.5 The Registrar

A PR is a basic requirement for a RSerPool system, therefore the first task of creating the RSPLIB prototype has been to design and implement a PR. Figure 5.2 shows the building blocks of the PR implementation. Clearly, the Dispatcher component provides its foundation and encapsulates the system-dependent functionalities (see section 5.4). The central element of a PR is the handlespace

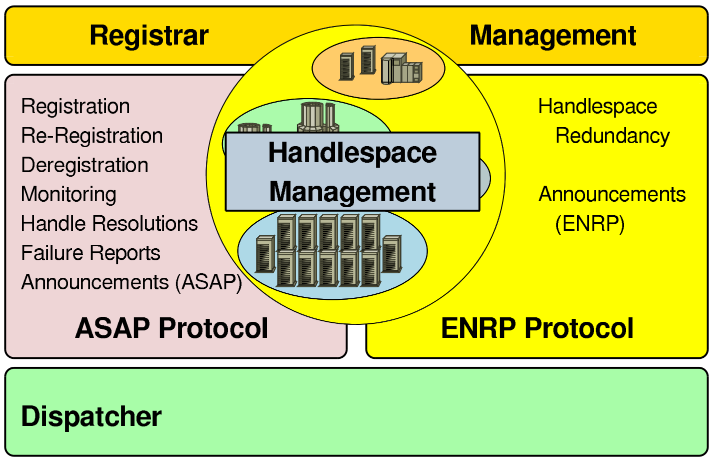


Figure 5.2: The RSPLIB Registrar

management. The handlespace management design and implementation are described in detail in chapter 4. As expected, the two components ASAP Protocol and ENRP Protocol provide the PR-side implementation of the two RSerPool protocols. This obviously includes message encapsulation and decapsulation but also the handling of keep-alive and expiry timers in cooperation with the handlespace management (see also subsection 4.4.3).

The *Registrar Management* component is the mediation layer between the two protocols and the handlespace management. It verifies that all operations on the handlespace requested via one of the protocols are allowed and rejects requests if necessary. For example, the Registrar Management checks whether the transport addresses under which a PE desires to register are matching the addresses it uses for the ASAP association with the PR (see also subsubsection 3.7.1.2). Another responsibility of the Registrar Management is the authentication and authorization of requests.

Currently, the PR prototype does not implement its own security mechanisms and relies on IPsec[3] instead. A future version may also support TLS[4] or our own approach Secure-SCTP (see subsubsection 2.4.3.7).

5.6 The PU/PE Library

The implementation of the PE and PU functionalities has been realized as a function library called RSPLIB. This library has also given our project its name. The RSPLIB library is described in this section. A short overview of its API can be found in subsection 5.6.2.

5.6.1 Building Blocks

The building blocks of the PU/PE library are presented in figure 5.3. As for the PR implementation, the Dispatcher component (see section 5.4) 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:

[3]See Kent and Atkinson (1998c,a), Deering and Hinden (1998b).
[4]See Dierks and Allen (1999), Blake-Wilson et al. (2003).

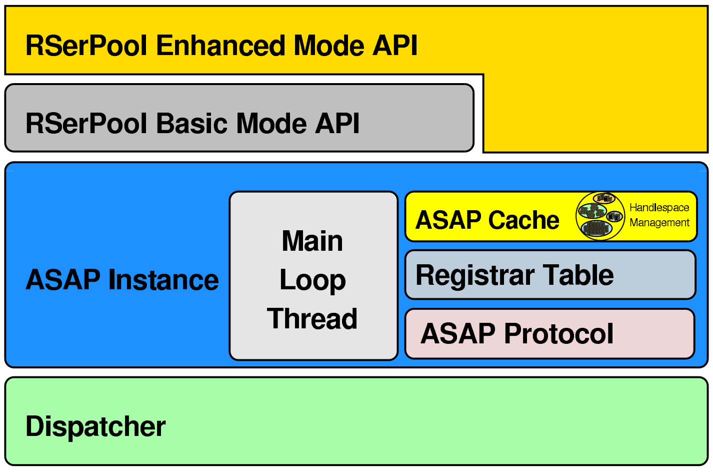


Figure 5.3: The RSPLIB Library

ASAP Protocol: Clearly, this sub-component realizes the PU and PE side of the ASAP protocol and provides ASAP message encapsulation and decapsulation.

ASAP Cache: The ASAP Cache reuses the handlespace management implementation described in chapter 4 to realize the PU-side cache. That is, each time a handle resolution is performed, its results are propagated into this cache and may be reused for further handle resolutions (see also subsubsection 3.7.1.1).

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 (see also subsection 3.9.4). It is also in the responsibility of the Registrar Table to flush expired entries and to establish an association to a randomly selected PR on request.

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 as described in subsubsection 3.9.2.2). 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 background so that the application using the library does not have to take care of the RSerPool event processing.

The ASAP Instance cannot be directly accessed by the application itself. Instead, two levels of API layers are built on top of the ASAP Instance: the Basic Mode API and the Enhanced Mode API. They are shortly described in the following subsection.

5.6.2 The RSerPool API

As mentioned in the previous subsection, the PU/PE Library provides two levels of API: the Basic Mode and the Enhanced Mode. While the Basic Mode API only provides the basic functionalities for registration, re-registration and deregistration for PEs, as well as for handle resolution and failure reporting for PUs, the Enhanced Mode API provides the complete ASAP Session Layer functionalities.

The Basic Mode and Enhanced Mode APIs have been suggested as part of our research and experience with the RSPLIB prototype. They are now under standardization by the IETF as results of our

Algorithm 3 An Example for a Pool User using the Basic Mode API

```
 1 /* Select a pool element from pool "MyPool" */
 2 rsp_getaddrinfo("MyPool", &eai);
 3 ...
 4
 5 /* Create a socket */
 6 sd = socket(eai->ai_family, eai->ai_socktype, eai->ai_protocol);
 7 if(sd >= 0) {
 8    /* Connect to pool element */
 9    if(connect(sd, eai->ai_addr, eai->ai_addrlen) {
10       ...
11       if(failure) {
12          /* Failure occurred -> report it */
13          rsp_pe_failure("MyPool", eai->ai_identifier);
14          ...
15       }
16       ...
17    }
18 }
```

discussions[5] at IETF meetings and are described in detail in the Working Group Draft Silverton et al. (2005).

5.6.2.1 The Basic Mode API

As mentioned in the introduction, the Basic Mode API only provides the basic functionalities for registration, reregistration and deregistration for PEs, as well as handle resolution and failure reporting for PUs. The main reason for using this API is to provide RSerPool support in existing applications which do not require the support of the Session Layer (i.e. the Control Channel between PU and PE, see subsection 3.9.5). In the following two paragraphs, the PU and PE sides of the Basic Mode API are shortly explained; more details can be found in Dreibholz (2005b), Dreibholz and Tüxen (2003).

Pool User Side 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()*[6].

Since the main intention of the Basic Mode API is to add RSerPool support to existing applications, it has been decided to mimic the DNS resolution API of Unix for the handle resolution call. Algorithm 3 presents the principle: instead of resolving a hostname via *getaddrinfo()* into a transport address, the RSPLIB function *rsp_getaddrinfo()* (line 1) resolves a PH into the transport address of a policy-selected PE. The structures including the transport address are compatible to the standard *getaddrinfo()* call.

In case of a PE failure, it is the duty of the PU to report this failure (using *rsp_pe_failure()* in line 13) as well as to perform a failover after selecting a new PE and connecting to it (not shown here).

[5]See Tüxen et al. (2004) and Silverton et al. (2004).

[6]The old *gethostbyname()* call is similar.

Algorithm 4 An Example for a Pool Element using the Basic Mode API

```
1 void registrationLoopThread()
2 {
3    while(!shuttingDown) {
4       rsp_pe_register("MyPool", ...);
5       usleep(reregistrationInterval);
6    }
7    rsp_pe_deregister(poolHandle, ...);
8 }
```

Algorithm 5 An Example for a Pool User using the Enhanced Mode API

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

/* Run application: file download */
rsp_send(session, "GET_Linux-CD.iso_HTTP/1.0\r\n\r\n");
while((length = rsp_recv(session, buffer, ...)) > 0) {
   doSomething(buffer, length, ...);
}

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

Pool Element Side A PE-side code example for the Basic Mode API is presented in algorithm 4: in line 4, the PE is registered by calling *rsp_pe_register*(). This function is called as part of a loop (line 3 to 6) in the interval given by the variable *reregistrationInterval*. The function *usleep*() waits the corresponding time span. Finally, the PE is deregistered by a call to *rsp_pe_deregister*().

Since a PE should not only take care of its registration but in particular provide its actual application service, it is assumed that the registration loop function shown in algorithm 4 is executed by a separate thread. This ensures that the PE's application service will never be interrupted by any RSerPool task.

5.6.2.2 The Enhanced Mode API

While the Basic Mode API only provides the basic functionalities for supporting RSerPool in applications, the Enhanced Mode API includes full support of the ASAP Session Layer (i.e. the Control Channel as well as connection maintenance and failover). The next two paragraphs shortly describe the Enhanced Mode API for the PU and PE side; more details can be found in Dreibholz (2005b).

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 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 Stevens et al. (2003).

To reduce the effort of a programmer to adapt a program to RSerPool, the approach for the Enhanced Mode API has been to mimic the Unix sockets API. Algorithm 5 presents an example: First, a session is created 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: “MyPool”).

After establishment of the session, it can be used for the application protocol. In line 6 to 9, the example application downloads a file by sending a request (using *rsp_send()*) and receiving the file (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 using the *rsp_close()* call (line 12).

Pool Element Side A non-RSerPool server application usually creates a socket (again using the *socket()* call), binds it to a specific port number (using the *bind()* call, e.g. TCP port 80 for a web server), puts the socket into the “listen” mode (using the *listen()* call) and accepts incoming connections 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 using *send()* and *recv()* calls. The connection is closed by a call to *close()*. For more details, see Stevens et al. (2003).

As for the PU, the PE-side Enhanced Mode API also mimics the Unix sockets API. An example is given in algorithm 6: first, a RSerPool socket is created (line 21) and the PE is registered to a pool given by its PH (here: “MyPool”). The RSPLIB library will automatically take care of re-registrations. After registering the PE, a loop waits for incoming sessions (using *rsp_poll()* in line 27), accepts them (using *rsp_accept()* in line 31) and creates new threads to serve them.

A thread function which handles a session is presented in line 1 to 16: first, it checks whether the first message received by *rsp_recv()* (line 3) is a state cookie. In this case, a saved session state is restored and the first command is read (line 6 to 7). After that, the loop from line 9 to 14 handles commands and saves the current session states as state cookies (using *rsp_send_cookie()* in line 12; 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 using *rsp_close()*.

The Enhanced Mode API also allows the UDP-like programming model and *poll()*/*select()*-based implementations. Due to space limitations, these programming schemes are not explained here. Details can be found at Dreibholz (2006c) and in Silverton et al. (2005).

5.7 The Demonstration System

As part of the RSPLIB prototype, a demonstration system has been created, in order to illustratively present the functionalities of RSerPool in a real-time distributed computing scenario (see subsection 3.6.5). It has been introduced in Dreibholz and Rathgeb (2005a), Dreibholz (2004a) and described in more detail in Dreibholz (2005b, 2006b). A screenshot of the demonstration system can be found in figure 5.4. The system consists of the following three parts:

Fractal Pool Element: A Fractal PE provides a computation service for Mandelbrot fractal graphics. It can be turned into a so called “unreliable mode” which breaks a transport connection after a certain number of transmitted packets to simulate a session failure.

Fractal Pool User: The Fractal PU is a graphical application which requests the computation of fractal graphics from the pool. The result is presented in a window (see the right-hand side of fig-

Algorithm 6 An Example for a Pool Element using the Enhanced Mode API

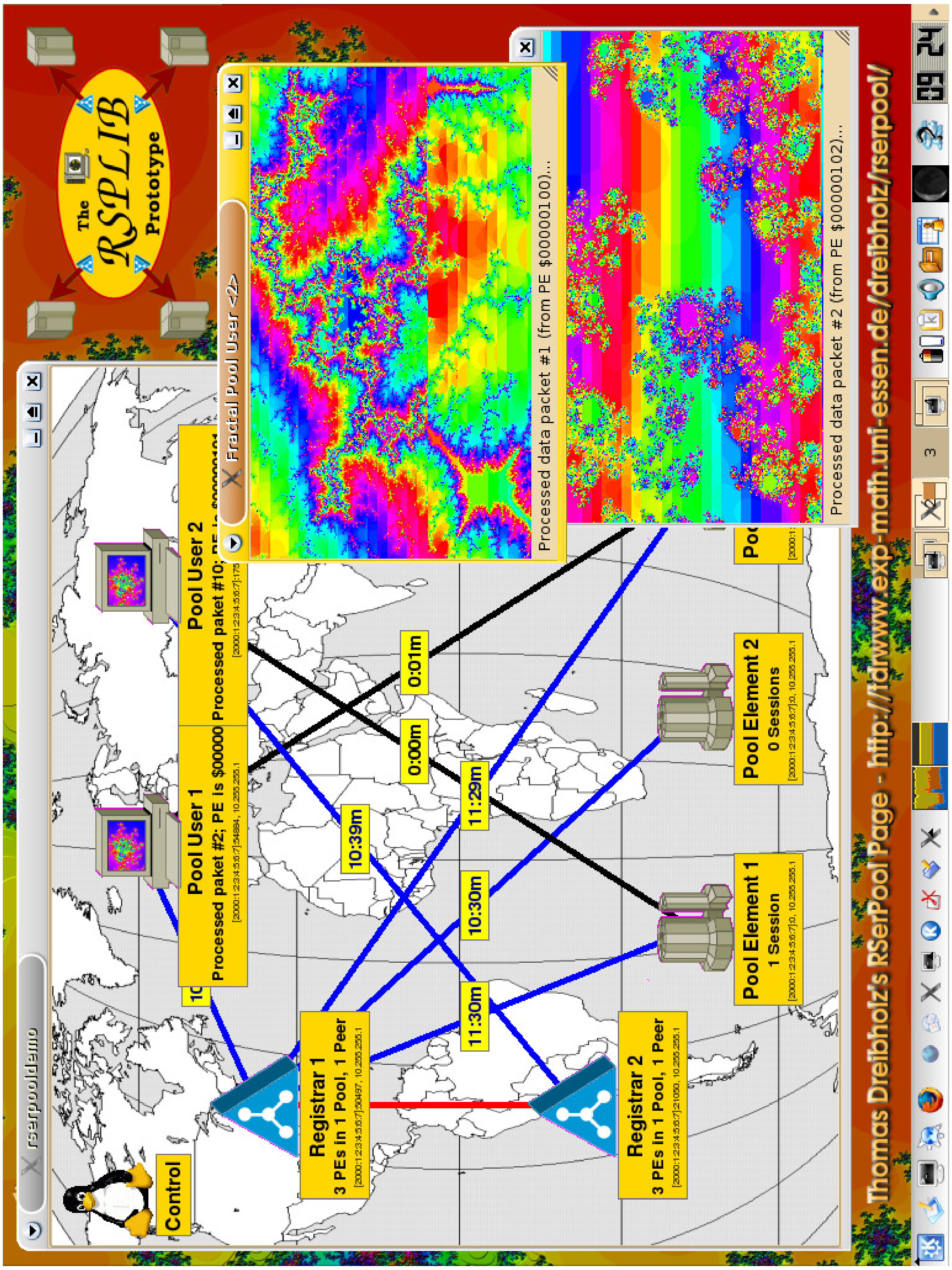
```
void serviceThread(session)
{
   rsp_recv(session, command, ...);
   if(command is a cookie) {
      /* Got a cookie -> restore session state */
      Restore state;
      rsp_recv(session, command, ...);
   }
   do {
      /* Handle commands from pool user */
      Handle command;
      rsp_send_cookie(session, 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, "MyPool", ...);

   /* 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);
}
```



The RSPLIB Prototype
rserpooldemo
Control
Registrar 1
3 PEs in 1 Pool, 1 Peer
Registrar 2
3 PEs in 1 Pool, 1 Peer
Pool User 1
Processed paket #2; PE is $000000
Pool User 2
Processed paket #10; PE is $0000010
Pool Element 1
1 Session
Pool Element 2
0 Sessions
10:39m
11:29m
10:30m
11:30m
0:00m
0:01m
Fractal Pool User <2>
Processed data packet #1 (from PE $00000100)...
Processed data packet #2 (from PE $00000102)...
Thomas Dreibholz's RSerPool Page - http://tdrwww.exp-math.uni-essen.de/dreibholz/rserpool/

Figure 5.4: The RSPLIB Demonstration System

ure 5.4), where the image is continuously building up. Failovers are exemplified by a slight colour change. That is, the observer will see a continuous calculation progress and notice possibly frequent failures of the calculating PEs only by the colour changes.

Demo Tool: A user interface to start and stop PEs, PUs and PRs is provided by the Demo Tool (see the left-hand side of figure 5.4). It also presents the states (active or inactive) of each component and the connection status and runtime for each association.

A special feature of the demonstration system is that each component may run on its own host. That is, a demonstration scenario may consist of different PCs; component failures and dynamic pool reconfiguration can be presented by disconnecting PCs from the network and reconnecting them again.

5.8 The Prototype Implementation Validation

To validate the correctness of the prototype implementation, the debugging software VALGRIND[7] – which has already been used to debug the handlespace management implementation as described in section 4.5 – has been used again. In order to detect problems, long-term tests (i.e. up to several days) have been performed, using the demonstration system described in section 5.7; each component has been executed by VALGRIND. In case of errors reported by VALGRIND, the tests have been restarted after bug-fixing.

In order to validate the message flow and the functional correctness of the components, simple tests of specific functionalities (like aborting a PR and observing the takeover procedure) have been performed. The resulting log output as well as the contents of the RSerPool messages transmitted have been verified manually. For the purpose of viewing the contents of messages, the network sniffer software WIRESHARK[8] (formerly called ETHEREAL) – which includes packet dissectors for various protocol types including SCTP as well as ASAP and ENRP – has been used intensively.

5.9 A Survey of Other RSerPool Implementations

After having explained the RSPLIB prototype implementation in detail, this section gives a short survey of other RSerPool implementations which are currently available or under development. It concludes with some notes on interoperability tests.

5.9.1 Motorola

As described by Silverton and Tüxen (2005), the Networks Business Unit of Motorola Inc. is developing a closed-source implementation of RSerPool. It currently includes an ASAP library running under Linux, FreeBSD and Solaris. Support for automatic configuration and the Enhanced Mode functionality is still under development. The Motorola PR implementation currently realizes the ASAP part only (without automatic configuration).

5.9.2 Cisco

An implementation of the RSerPool protocols by Cisco – for inclusion into their Cisco™ Internet Operating System IOS for routers – is currently under development. As of July 2006, the implementation

[7]See Seward and Nethercote (2005), Valgrind Developers (2005).
[8]See Lamping et al. (2006), Wireshark (2006).

Figure 5.5: The First Interoperability Tests at the 60th IETF Meeting

is still in an early development stage and includes an ASAP library (running under FreeBSD) without support for the automatic configuration feature. The PR implementation – which will be a part of future IOS versions – has not been started yet. However, a quick project progress is expected, due to the growing interest of router vendors in the RSerPool protocol framework.

5.9.3 Münster University of Applied Sciences

As part of a Bachelor's thesis, an Open Source PR implementation under the Berkeley Software Distribution[9] (BSD) license is currently under development at the Münster University of Applied Sciences in Münster, Germany. According to Tüxen and Dreibholz (2006a), the goal of this project is to develop a PR for research on the synchronization behaviour of the ENRP protocol. It is therefore not planned to realize the automatic configuration functionalities of the ASAP and ENRP protocols. The PR will run under different operating systems with kernel SCTP implementations, including Linux, FreeBSD and MacOS X.

5.9.4 Interoperability Tests

Protocol interoperability tests are an important step in the IETF standardization, since the IETF relies on running code. That is, before a Working Group Draft defining a protocol can become a RFC, two independently developed protocol implementations have to prove interoperability.

The very first interoperability tests of two ASAP implementations, between our own prototype RSPLIB and the proprietary Motorola implementation, have taken place at the 60th IETF Meeting in San Diego, California/U.S.A. on August 4 and August 6, 2004. These historic moments in RSerPool standardization have been captured in pictures and are presented in figure 5.5.

In July/August 2006, the first official RSerPool interoperability test – called Bakeoff – has been realized in conjunction with the 8th SCTP Bakeoff in Vancouver, Canada (see also Wagner (2006)). At this meeting, the RSPLIB prototype implementation has been successfully tested for interoperability with the Cisco ASAP library. In particular, the RSPLIB has also been tested with different SCTP implementations – including our own userland implementation SCTPLIB as well as the kernel

[9]See Open Source Initiative (1999).

implementations of Linux, FreeBSD and MacOS X. Rules for the evaluation of the RSerPool implementation interoperability testing are specified as Internet Draft in Dreibholz and Tüxen (2006).

5.10 Summary

In this chapter, the RSPLIB prototype implementation has been presented. Based on a research project between our group and Siemens, it has become the world's first complete, Open Source, multi-platform RSerPool implementation and is now the reference implementation of the IETF RSerPool WG. In the first part of the presentation, the important design decisions of the RSPLIB prototype have been shown: the independence from the underlying operating system and easy portability to new platforms. The three main parts of the implementation have been introduced in the following: the PR implementation, the PU/PE library and the demonstration system. Since the PU/PE library provides the connection between RSerPool and its applications, it has been explained in more detail. In particular, the two-part API has been presented with some pseudo-code examples: the Basic Mode API providing the core RSerPool functionalities (e.g. PE registration handling and handle resolutions) only, and the Enhanced Mode API including the full Session Layer features. Finally, a survey of other RSerPool implementations has been given, together with some notes on the interoperability testing among these implementations.

Chapter 6

The RSPSIM Simulation Model

An introduction to the RSerPool simulation model RSPSIM is provided in this chapter. At first, a short overview of the model's motivation and history is given; this is followed by the definition of the requirements for the model. After that, the simulation environment including the reasons for the choices of simulation and post-processing tools are described, followed by a presentation of the simulation model itself.

6.1 Introduction

After the development of the RSerPool prototype implementation RSPLIB (see chapter 5) had been started, there has been a growing demand for the research on performance aspects of RSerPool. In particular, the intention has been to evaluate the behaviour of different pool policies as part of the IETF standardization activities and to verify the ideas for the usage of RSerPool for real-time distributed computing (see subsection 3.6.5). But performing such analyses using a prototype implementation is very difficult, since effects discovered and possible problems are not easy to reproduce. Therefore, it has been necessary to develop a simulation model.

The simulation model – which is simply called RSPSIM for "RSerPool Simulation" – is described in the following sections. Its development is supported by the Deutsche Forschungsgemeinschaft (DFG) since October 01, 2004.

6.2 The Requirements for the Simulation Model

Before designing a simulation model, it is first necessary to define its goals. For the RSerPool simulation RSPSIM, the goal has been a performance evaluation of the RSerPool functionalities. That is, it has been necessary to model the RSerPool protocols and component functionalities. This has included the handlespace management of the PR and the PU-side cache, the policies (see section 3.11) and the Session Layer behaviour – in particular the client-based state sharing as described in subsubsection 3.9.5.2.

Furthermore, the requirements for the handlespace management have included the possibility to maintain large pools, e.g. for the application of real-time distributed computing as described in subsection 3.6.5. In order to compare policies and evaluate improvements, it has also been an important demand to be able to add new policies easily. Since the handlespace requirements have been similar for the RSPLIB prototype as well as for the RSPSIM simulation model, it has been decided to create a common handlespace management implementation to be used for both systems. This handlespace management approach is described separately in chapter 4.

When the goals of the RSPSIM simulation model have been defined, it has also been useful to explicitly state the non-goals: it has not been a goal to simulate the Transport Layer protocol, i.e. SCTP. Evaluating effects of SCTP and IP – like multi-homing for network path redundancy – would have exceeded the scope of this thesis (which is RSerPool performance). Related work on the subject of Transport Layer resilience with SCTP can be found in Jungmaier (2005), Jungmaier, Rathgeb and Tüxen (2002), Conrad et al. (2002), Jungmaier et al. (2000b,a).

6.3 The Simulation Framework

In this section, the simulation framework is introduced. This includes reasoning the choices for a discrete event simulation toolkit as well as for a package to perform the statistical post-processing of obtained results. The actual description of the simulation model RSPSIM follows in section 6.4.

6.3.1 A Discussion of Simulation Packages

Before a simulation model can be created, it is obviously necessary to choose a simulation system. Possible choices of discrete event simulation packages have been a LISP-based simulation package as well as OPNET MODELER, NS-2 and OMNET++. In the following subsubsections, the features of these packages are shortly summarized. This is followed by motivating the choice for the RSPSIM simulation model.

6.3.1.1 LISP-based Simulation Package

The first possible choice for a simulation system has been an Open Source simulation package for Common LISP, which is maintained by the University of Applied Sciences at Münster, Germany (see Tüxen (2003)). This package had already been used by us for simulating a lightweight QoS device[1], therefore some experience with this package had already been available. On the other hand, the simulation package is very rudimentary; it does not provide any additional functionalities except the discrete event simulator itself, some random number generators and classes to collect statistics. For a large project like RSPSIM, using this package would have required to self-develop fundamental functionalities like vector and scalar output as well as scenario and configuration management. That is, using the LISP-based package has been no realistic option for the RSPSIM simulation model.

6.3.1.2 OPNET MODELER

Due to an academic contract, we also had access to the commercial simulation system OPNET MODELER (see OPnet Technologies (2003)). It does not only include a discrete event simulation system with graphical user interface, but also provides powerful functionalities for the post-processing of obtained results data. It includes a huge library of complete models for all kinds of networking protocols and even models for existing hardware (like router and switch models of various vendors); own models are realized in ANSI-C. OPNET MODELER is also a well-known simulation system which is in particular frequently used in ITU standardization.

On the other hand, licensing has been an issue. OPNET MODELER licenses are highly expensive and furthermore time-limited. That is, after expiration of the license the simulator refuses to run and

[1]See Dreibholz, IJsselmuiden and Adams (2005), Dreibholz et al. (2004), Dreibholz, Smith and Adams (2003).

the developed model becomes useless. The requirement of an expensive license to run a simulation also prevents many interested people from actually using the simulation model. This is in particular an important issue in IETF standardization, where for such reasons Open Source simulation systems are preferred.

A particularly negative experience of working with OPNET MODELER has been that it is – despite its huge licensing costs – far away from being bug-free. In multiple cases, bugs causing program crashes have been experienced in the Closed Source parts of the simulation system. Since such bugs are impossible to track down (since there is no source code), it has been necessary to find workarounds by trial-and-error tests and asking for help in support forums. This has been a very annoying and time-consuming procedure.

6.3.1.3 NS-2

The NS-2 (Network Simulator 2, see NS-2 (2003)) simulation system is an Open Source discrete event simulation package which is well-known and also frequently used in IETF standardization. It already includes models for a variety of network protocols, many more models are available separately as Open Source. Unlike OPNET MODELER, it does not include specific models for existing hardware like routers or switches.

NS-2 simulations are usually written in a combination of C++ modules and oTcl (Object-Tcl, see Wetherall and Lindblad (1995)) scripts, where the oTcl part is usually used to define the simulation scenario as well as for parametrization. The performance-critical simulation objects are realized in C++. The choice of object-oriented programming languages simplifies the creation of simulation models: it is easily possible to create derived classes of models and hereby to realize specialised variants of the simulation objects.

All simulation output is written to so called trace files; these files are used to post-process the simulation results. Optionally, the program NAM (Network Animator, included in the NS-2 package) can display an animation of the simulation scenario, including packet flow and queue lengths.

6.3.1.4 OMNET++

OMNET++ (Objective Modular Network Testbed in C++, see Varga (2005b,a)) is another Open Source discrete event simulation system, but significantly newer than NS-2 and therefore not that widespread. Nevertheless, there are already models for the most important networking protocols available; although, the library of models for OMNET++ is still significantly smaller than for NS-2 or OPNET MODELER. Own simulation objects are implemented in C++.

A very interesting and useful feature of OMNET++ is to have the possibility to either create a command-line (*Cmdenv* environment) or GUI version (*Tkenv* environment) of a simulation model by simply linking it to the appropriate environment library. During model development, this allows the user to run the model using the GUI and view an animation of the packet flow as well as to inspect things like the contents of queues during the run. It is furthermore possible to interrupt the simulation and perform single steps to check the behaviour of the model. Finally, to perform the actual sets of simulation runs using scripts, it is sufficient to simply link the model with the command-line environment.

Another useful feature of the OMNET++ package is the availability of code and Makefile generators. In OMNET++, these generators take over frequently recurring tasks like the creation of classes for messages and modules (i.e. objects of a simulation scenario, see subsection 6.3.3 for details) from their definitions as well as creating a Makefile for the simulation's complete sources. These functionalities allow a simulation developer to concentrate on the simulation subject, rather than to take care for how to create the appropriate structures for the underlying simulation system.

Due to its powerful code and Makefile generators and its excellent documentation, the time required for initial training before being able to actually realize a simulation model is short. Therefore, we have already used OMNeT++ for student training sessions and gained experience with this simulation system. In particular, the experience has been that it is also very stable – since starting to use it, no bugs in its simulator classes have been discovered.

6.3.1.5 Conclusion

After intensively testing all four simulation systems presented in the previous subsubsections, the only applicable choices for a simulation package to realize the RSPSIM simulation model have been NS-2 or OMNeT++: while the LISP simulation package clearly lacks of features, the licensing issue of OPNET MODELER would have put too many restrictions on the usability of the model.

From the remaining set of possible choices – NS-2 and OMNeT++ – the OMNeT++ package finally has been selected (starting with version 2.3, now using version 3.2), due to its powerful code generators as well as its more advanced user interface and configuration options. While the code generators allow a quick development progress, the GUI-based user interface is a great help during debugging and validation of models (see also section 6.5). In particular, the animation and object monitoring features of OMNeT++ can be used *during* the simulation run – in contrast to NS-2's NAM program, which can only process a trace file.

6.3.2 Statistical Post-Processing

For the statistical post-processing of simulation data, there had been the choice between the two Open Source packages GNU OCTAVE (see Eaton (2003)) and GNU R (see R Development Core Team (2005)). Post-processing in case of the RSPSIM simulation model mainly means to collect a large number of statistics from the simulation output, perform some kind of data aggregation and manipulation (like the calculation of confidence intervals), and finally plot figures as PDF files.

For the task of collecting, aggregating and manipulating statistics data, the features of both packages – GNU OCTAVE and GNU R – are quite similar. The main difference are the possibilities for creating graphical output. While GNU OCTAVE relies on GNU PLOT (see Williams and Kelley (2003)) to create graphical output, GNU R provides its own set of graphics primitives and high-level plotting functions. GNU R offers a wide variety of directly usable functions to plot graphs; it is also possible to use the graphics primitives to create own plotting functions with customized features (like calculating confidence intervals) and appearance. This possibility to highly customize plotting has been the reason to choose GNU R (version 2 series) for the statistical post-processing tasks.

6.3.3 An Overview of the OMNeT++ Discrete Event Simulator

To understand the description of the simulation model in the following sections, it is first necessary to introduce some basic terminology and concepts of the OMNeT++ discrete event simulator. For a detailed introduction and tutorials of OMNeT++, see its documentation in Varga (2005b).

Central element of an OMNeT++ simulation model is a *Network*. A network consists of *Modules*, each module may be either a *Compound Module* or a *Simple Module*. A Compound Module consists of at least one sub-module, each sub-module may again be a Compound Module or a Simple Module. The Simple Module is atomic, its functionalities require implementation (in C++ language) by the user. Interfaces between modules are called *Gates*. A gate may be connected by a *Connection* to an other module's gate or the gate of its parent Compound Module. Gates are used for the directed transport of *Messages*, i.e. a gate can be either used for the transmission or reception of messages (depending on its type: input gate or output gate). Messages follow the path given by the gates' connections. A connection may introduce delay, bandwidth limitation and bit errors[2].

Timers are realized by scheduling the transmission of a message to a module itself. A timer (i.e. a scheduled message) is stored in the *Future Event Set* (FES); it arrives at the module when the simulation time reaches its schedule time stamp.

The definition of messages is done in an OMNeT++-specific definition language. Networks as well as modules with their gates and connections are specified in OMNeT++'s NEtwork Description language (NED). The code generators of OMNeT++ use these definitions and create C++ classes. The only task of the user is to implement the custom functionalities of Simple Modules in derived classes.

Implementing the actual behaviour of Simple Modules can be realized in two ways: the first possibility is as a *Finite State Machine* (FSM), where the module handles incoming messages according to its current state. The handling of a message possibly changes the module's state. States which are only left upon the reception of a message (e.g. when a timer expires) are denoted as *Stable State*; on the other hand, a *Transient State* is immediately left. The second possibility to implement the functionalities of a Simple Module is by using cooperative threads. Since the first approach – FSMs – has been taken to implement the simulation model, cooperative threads are not described here. For more details on this subject, see Varga (2005b).

A compiled simulation can use parameters read from an input file (called .ini file, due to its suffix) to parametrize a network with its modules. For example, an own .ini file for each simulation run could specify different seeds and load parameters for a simulated client application. The output of a simulation run is a vector file including all recorded vectors as well as a scalar file including all written scalars.

6.3.4 The Simulation Tool Chain

Based on a script executed by GNU R and the RSPSIM simulation model linked with the Cmdenv environment, a tool chain to actually run a simulation and perform the post-processing has been created. At first, the GNU R script defines the parameter space to simulate. That is, a set of values is specified for each simulation parameter (e.g. *parameter1Set*={1,2,5}, *parameter2Set*={"LeastUsed", "Random"}, ...) and the number of runs for each combination of parameters is defined. For each run and parameter combination, an .ini file as well as an entry in a Makefile is written. The Makefile entry actually executes the simulation model program using the corresponding .ini file as input and compresses the results (scalars, vectors, debug output) using the BZIP2 compression software (see Seward (2005)) to save disk space. Finally, an "all" entry is created in the Makefile which includes all simulation runs as well as creating a set of GNU R input files (to be explained later).

Actually performing the simulation runs simply means to invoke MAKE (see Free Software Foundation (2003)) on the created Makefile. A useful property of MAKE is its option to parallelize the simulation runs on multi-CPU and/or multi-core machines. In particular, the simulations of this thesis have been run on dual-core Pentium IV CPUs, where MAKE has been able to utilize both cores with different simulation runs.

[2]Bit errors are introduced by setting an error flag in the message object. The actual message remains unmodified, the receiver has to take care of the error bit and implement appropriate reactions.

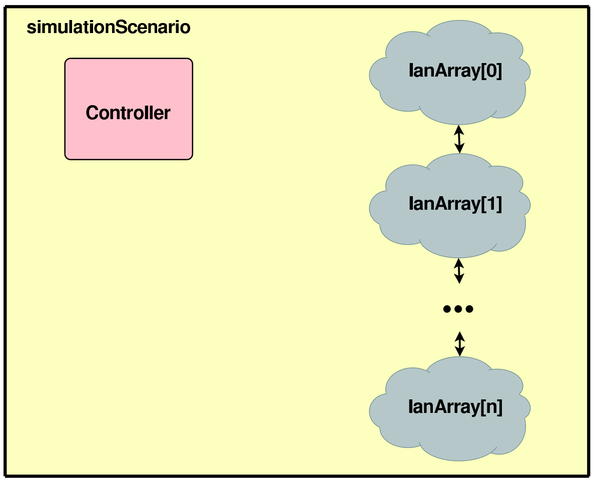


Figure 6.1: The Simulation Scenario Network

After successfully performing all simulation runs, the generated and compressed scalar output files are collected by a self-developed collection tool called `createsummary`. It stores the collected result statistics (e.g. for system utilization and request handling speed, to be explained in section 8.5) into separate input files in the GNU R data format (i.e. simple ASCII tables). All output files are BZIP2-compressed on-the-fly using LIBBZIP2 (see Seward (2005)), since they may also become quite large otherwise.

Taking the compressed data files as input, other GNU R scripts actually plot the results data. For plotting, custom output procedures have been created which may not only generate multi-page, multi-column and multi-row two-dimensional graphs with multiple curves per graph, but also allow to specify parameters for creating title and legend texts as well as allow to choose among coloured, grey-scale or black-and-white drawing. In particular, the plotting procedures also take care of computing and plotting the confidence intervals. The created plots are exported as PDF files.

6.4 The Simulation Model

In this section, the RSerPool simulation model RSPSIM is presented.

6.4.1 The Network

As explained in subsection 6.3.3, the first step to define an OMNET++ simulation scenario is to set up a network. The network used for the RSPSIM simulation model is shown in figure 6.1; it consists of a *Controller Module* (which will be explained in subsubsection 6.4.2.1) and an interconnected array of *LAN Modules*.

A LAN module is a Compound Module consisting of the sub-modules presented in figure 6.2: PUs, PEs and PRs are attached to a switch. The switch itself is connected to input and output gates of the LAN module. That is, a LAN can be connected to other modules; in particular it can be connected to other LANs as shown in the scenario network in figure 6.1.

Before providing the actual description of the modules for PR, PE and PU components as well as for the switch, it is first necessary to introduce some basic modules which are the building blocks.

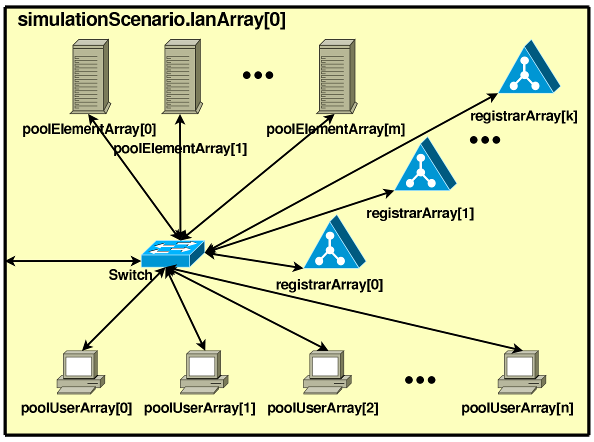


Figure 6.2: The LAN Module

6.4.2 The Foundation Modules

The building blocks of the RSPSIM simulation components are the Controller Module, the Transport Node module and the Registrar Table module. These three modules are described in the following.

6.4.2.1 The Controller Module

The *Controller Module* is a Simple Module which is responsible for the following tasks:

- Collecting global statistics (like the global PE utilization),
- Resetting statistics data after an initial startup phase,
- Triggering the writing of statistics (scalars) to the output file after reaching a predefined simulation time and
- Finally stopping the simulation.

Every RSPSIM simulation scenario requires exactly one instance of the Controller module, i.e. it is a singleton object[3].

6.4.2.2 The Transport Node Module

As explained in section 6.2, it is not the goal of the RSPSIM simulation model to build IP or SCTP networks. Therefore, realizing a fully-featured SCTP/IP stack including a routing protocol would be unnecessary and inefficient. The *Transport Node Module* therefore realizes a simple Layer 4 transport of messages via a network of interconnected Transport Node modules, based on Dijkstra's shortest path algorithm (see Cormen et al. (1998)) provided by OMNeT++. This module therefore becomes an integral part of all network device models.

A Transport Node consists of an array of Network Layer gates, a local Network Layer address (i.e. a unique ID for the Transport Node) and an array of Application Layer gates. Application Modules connected to Application Layer gates can bind their corresponding gate to a certain port number.

[3]The singleton design pattern is used to restrict the instantiation of a class to one object.

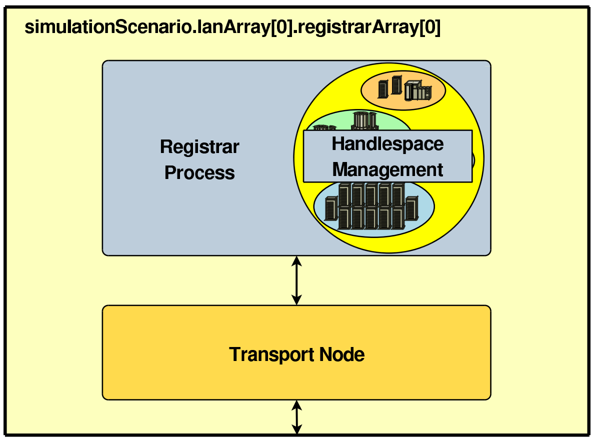


Figure 6.3: The Registrar Module

Incoming messages are either passed to the corresponding Application Layer port (if the message's destination address is the local Transport Node's address) or are forwarded out of the appropriate Network Layer gates according to the Dijkstra algorithm. For efficiency, shortest path computations are stored in a cache, so that the algorithm is only called if the destination is still unknown.

The switch of the LAN module presented in section 6.1 (see also figure 6.2) is simply an instance of the Transport Node module providing an appropriate number of input and output gates.

It is important to note that – although implementing a full SCTP/IP stack is not a goal of the RSPSIM simulation model – it should be easily possible to replace the Transport Node module with a SCTP/IP module.

6.4.2.3 The Registrar Table Module

The third foundation module of the RSPSIM simulation model is the *Registrar Table Module*. It realizes the Registrar Table (see subsubsection 3.7.1.1) for PE and PU modules. It includes a gate to a Transport Node to receive PR announces as well as a gate to an ASAP PU/PE Process module (to be explained later). The PU or PE Process module can request a PR identity using a *ServerHuntRequest* message; a randomly selected PR is returned by a *ServerHuntResponse* message.

6.4.3 The RSerPool Modules

Using the foundation modules described in subsection 6.4.2, it is now possible to define the actual RSerPool component modules for PR, PE and PU. These modules are introduced in the following. Since a detailed description of the RSerPool component interaction and the protocols has already been provided in chapter 3, the models' workflows are in the focus of these descriptions.

6.4.3.1 The Registrar Module

An illustration of the compound *Registrar Module* is presented in figure 6.3. Clearly, it includes a Transport Node module (see subsubsection 6.4.2.2) for its network communication functionalities. Connected to the Transport Node module is the *Registrar Process Module*, which provides the actual PR functionalities and therefore has to be introduced in more detail. Since the management of a

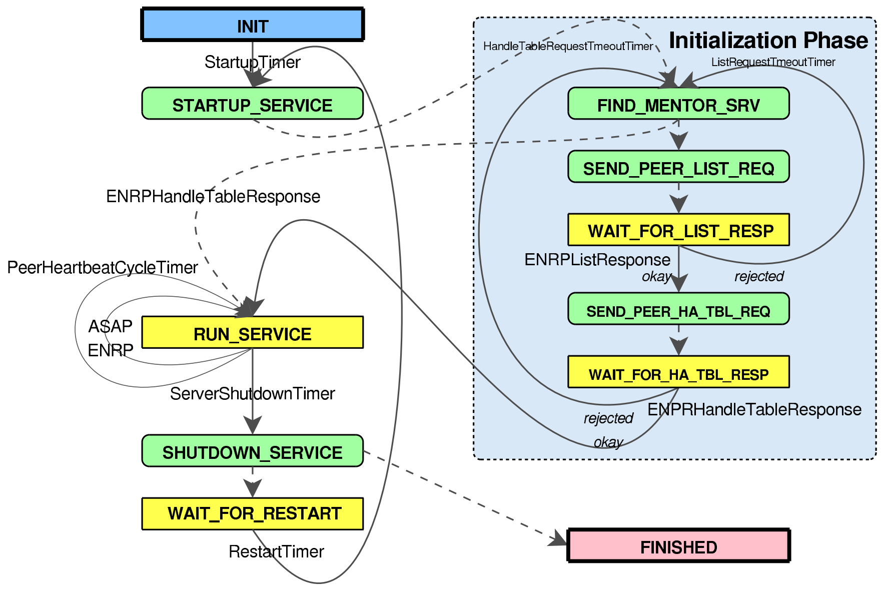


Figure 6.4: The Registrar Process Finite State Machine

handlespace is obviously the main task of a PR, the Registrar Process Module requires a handlespace management implementation. As already explained in section 6.2, the RSPSIM simulation model reuses the implementation introduced in chapter 4.

The actual PR functionalities of the Registrar Process module are realized as a FSM, which is illustrated in figure 6.4. In order to enhance the clearness of the presentation, the following notation is used for this and the following FSMs: stable states are shown in angular boxes, while transient states use rounded ones.

A Registrar Process goes into the STARTUP_SERVICE state upon expiry of its *StartupTimer*. This state initializes the Registrar module's Transport Node sub-module to receive ASAP and ENRP traffic. Upon startup, the PR has to find a mentor PR and download its peer list and handlespace (see subsubsection 3.10.2.4 and subsection 3.10.3 for details). Finally, the PR goes into the RUN_SERVICE state, i.e. the PR goes into normal operation. Upon expiry of the *ShutdownTimer*, the PR deactivates the reception of ASAP and ENRP traffic and waits for a restart in the state WAIT_FOR_RESTART. A newly expiring StartupTimer repeats the whole procedure.

6.4.3.2 The Pool Element Module

The building modules of the *Pool Element Module* are presented in figure 6.5. Clearly, a PE requires network access. Therefore, a Transport Node module (see subsubsection 6.4.2.2) is the basis of the PE model. Furthermore, each PE requires a Registrar Table to maintain its list of PRs. The Registrar Table module provides its service to the *Pool Element ASAP Module*, which offers the actual PE-side ASAP functionalities to the *Application Server Process Module*. This Application Server Process module performs the actual service of the PE, like the compute service being presented later in section 8.3.

Since the main ASAP functionality is located in the Pool Element ASAP Process module, it is necessary to show this module in more detail. Therefore, figure 6.6 presents an overview of its FSM. Upon startup, the Pool Element ASAP Process goes into the WAIT_FOR_APPL state and waits for

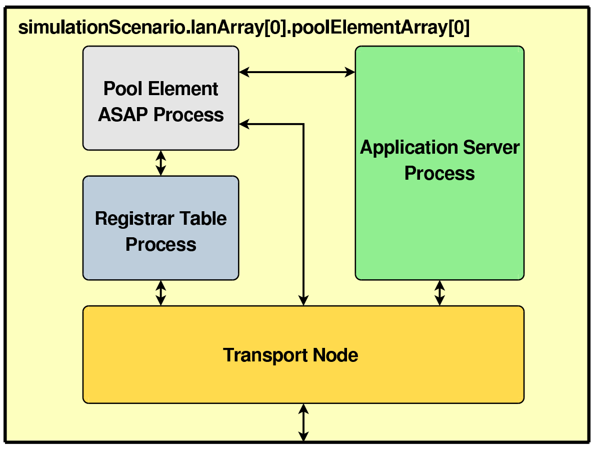

simulationScenario.lanArray[0].poolElementArray[0]
Pool Element ASAP Process
Application Server Process
Registrar Table Process
Transport Node

Figure 6.5: The Pool Element Module

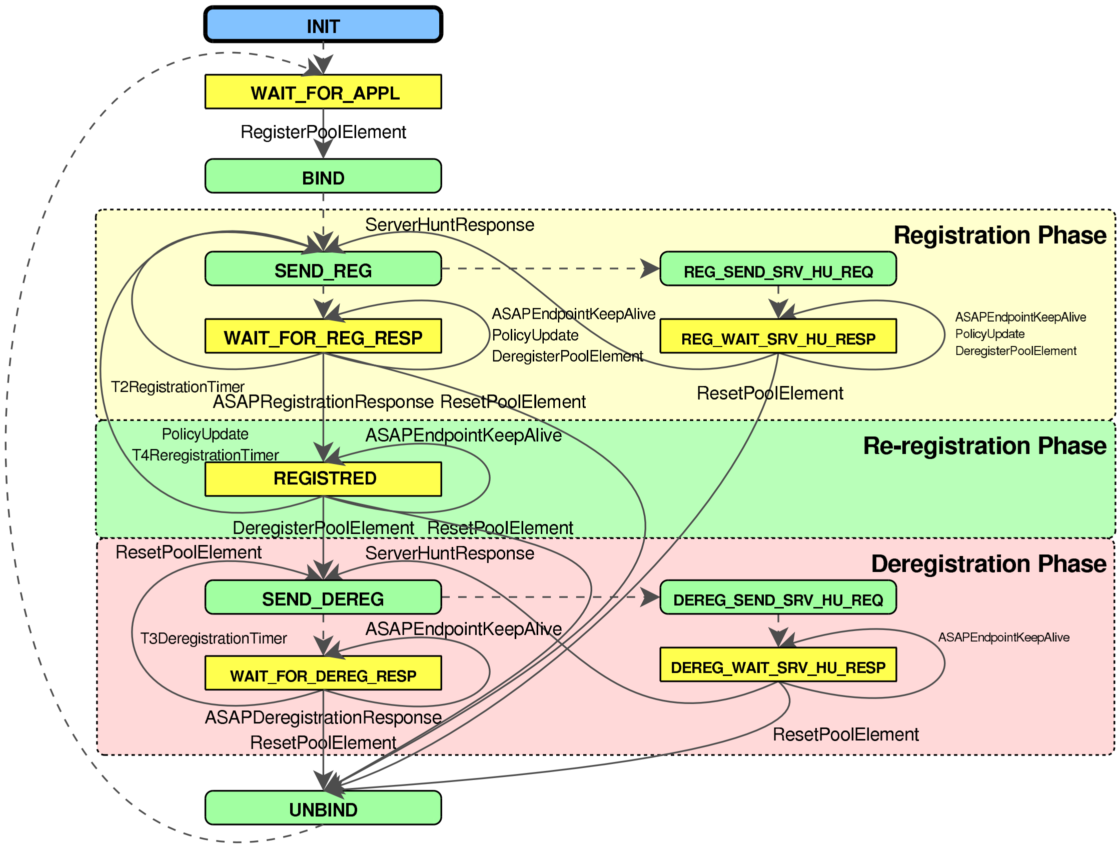


Figure 6.6: The Pool Element ASAP Finite State Machine

commands from the Application Server Process module. The Application Server Process module can initiate the registration of the PE using a *RegisterPoolElement* message. First step of the registration process is to tell the Transport Node module to allocate a new port (to be explained below) for the ASAP communication in the BIND state. After that, the registration at a PR is processed; this possibly requires a server hunt procedure by asking the Registrar Table module for a PR identity (using a ServerHuntRequest message). After successful registration (i.e. reception of an ASAPRegistrationResponse from the PR), the Pool Element ASAP Process confirms the registration to the Application Server Process using a *RegisterPoolElementAck* message and goes into the REGISTERED state.

The registration procedure is repeated every re-registration interval (controlled by the ReregistrationTimer). Upon reception of a *DeregisterPoolElement* message from the Application Server Process module, a deregistration is performed. This deregistration process possibly includes to choose a new PR by a server hunt procedure. After successful deregistration (i.e. reception of an ASAPDeregistrationResponse from the PR), the Pool Element ASAP Process confirms the deregistration to the Application Server Process using a *DeregisterPoolElementAck* message and the ASAP communication port is released by the Transport Node module (UNBIND state).

An Application Server Process module can update its policy information using a *PolicyUpdate* message. This event may also happen during waiting for a registration response from the PR (state WAIT_FOR_REG_RESP) or during a server hunt in order to register (REG_WAIT_SRV_HU_RESP). Therefore, it is necessary in these cases to re-schedule the re-registration to "immediately".

It is furthermore possible for an Application Server Process to "fail" at any time by sending a *ResetPoolElement* message. In this case, no deregistration is processed, but instead the Pool Element ASAP Process confirms the reset using a *ResetPoolElementAck* message and goes immediately into the UNBIND state. That is, the ASAP endpoint of the PE becomes unreachable and PR-H and PUs have to detect the unavailability of the PE by their own mechanisms. Since a restart of the PE means binding the Pool Element ASAP Process – as well as the Application Server Process – to new ports, it is ensured that "reincarnations" of the PE can be distinguished. This scheme models the behaviour of the RSPLIB prototype implementation, where ports for the ASAP instance and the application service are randomly chosen. If a PE is restarted, it will almost certainly become reachable under different port numbers.

6.4.3.3 The Pool User Module

The last RSerPool module is the *Pool User Module*, which provides the PU model. Figure 6.7 shows its building modules. As for the Pool Element Module, it consists of a Transport Node module (see subsubsection 6.4.2.2) for its network communication and a Registrar Table module (see subsubsection 6.4.2.3) for maintaining its list of PRs. The actual application of the PU (e.g. the compute service being presented later in section 8.3) is performed by the *Application Client Process Module*. All PU-side ASAP functionalities are provided by the *Pool User ASAP Module*.

Figure 6.8 presents the FSM of the Pool User ASAP module. Upon creation, the Pool User ASAP Process immediately goes into the state WAIT_FOR_APPL, which obviously waits for requests from the Client Application module. Commands for purging the PU-side cache (*CachePurge* message) and reporting an unreachable PE (*EndpointUnreachable* message) can be handled immediately. A request for a handle resolution (*ServerSelection* message) is processed by the PU-side cache in the SEL_PE_FROM_CACHE state. If the cache is empty, querying a PR is required. If currently no PR is available, a server hunt procedure (i.e. requesting a PR from the Registrar Table module using a ServerHuntRequest message) is necessary first. Finally, it is again tried to select a PE in the SEL_PE state.

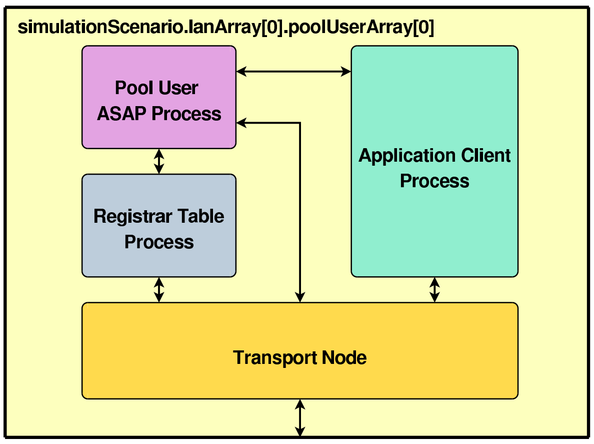

simulationScenario.lanArray[0].poolUserArray[0]
Pool User ASAP Process
Registrar Table Process
Application Client Process
Transport Node

Figure 6.7: The Pool User Module

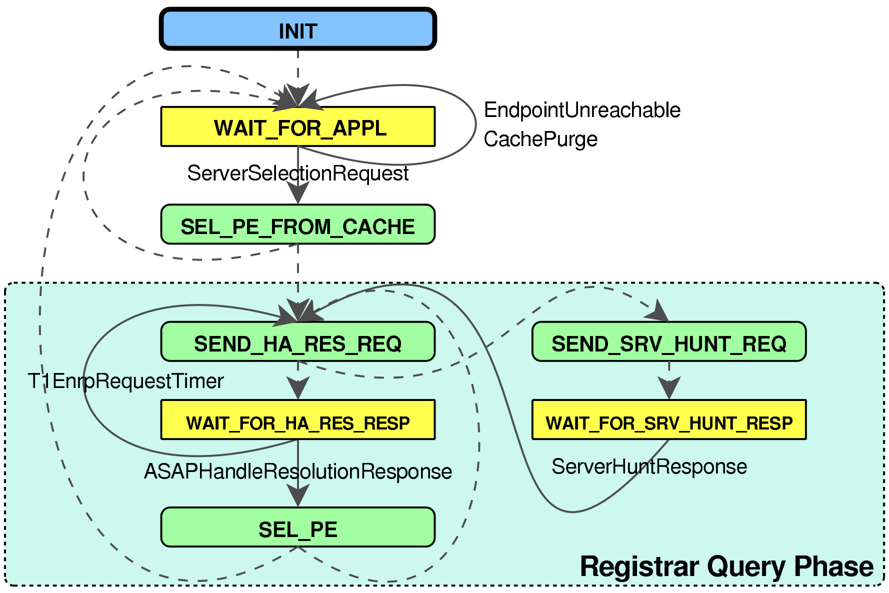


Figure 6.8: The Pool User ASAP State Machine

6.5 The Simulation Model Validation

To validate the correctness of the simulation model's implementation code, the debugging software VALGRIND[4] – which has already been used to debug the handlespace management implementation as described in section 4.5 and the RSPLIB prototype implementation as described in section 5.8 – has been intensively used again. That is, some simulation runs with the simulation model executed by VALGRIND have been performed, checking that no errors have been detected.

In order to validate the functional correctness of the models, simple simulations of specific functionalities (like registering a PE in a certain scenario setup) have been performed. Then, the output of the simulation model has been checked for correctness manually. For these tests, the GUI-based Tkenv environment of the OMNET++ system has been very handy: it allows to stop the simulation at arbitrary points to check the state of the modules and the message flow. Afterwards, the simulation can be resumed until the next interesting point for validation.

6.6 Summary

In this chapter, the RSerPool simulation model RSPSIM has been described. The motivation to design and create this model has been the demand for performance evaluations, which are – due to a wide parameter range – difficult to realize in a lab setup using the RSPLIB prototype. The first part of this chapter has described the requirements for the simulation model and provided a survey of possible simulation systems and post-processing packages. After that, the choices of packages actually used for the RSPSIM simulation model – OMNET++ and GNU R – have been motivated. In the following, a short introduction to OMNET++ has been given before the actual description of the simulation model itself. In particular, the actual application model for the PE and PU module is replaceable; the application model used for the performance simulations will be described in section 8.4. Finally, a short overview of the model validation has been given.

[4]See Seward and Nethercote (2005), Valgrind Developers (2005).

Chapter 7

Handlespace Management Performance

MANAGING the handlespace using the approach presented in chapter 4, the implementation effort is mainly reduced to the storage of sorted sets. However, it is not obvious which data structure is most suitable to manage such sets. Therefore, the goal of this chapter is to measure and evaluate the handlespace management performance using different set implementations first. After that, the handlespace management's scalability to high numbers of PEs and pools is evaluated.

7.1 Introduction

Handlespace management is a crucial part of every RSerPool system: it is not only integral duty of every PR (see subsection 3.7.1), it is also required for maintaining the PU-side cache (see subsection 3.7.3). Optimizing the performance of its implementation therefore becomes advantageous for many components of a RSerPool system.

7.2 The Performance Metric

To evaluate the performance of the handlespace management implementation described in chapter 4, it is first necessary to define a performance metric. The chosen metric is simply the time a realistic CPU (to be discussed below) requires to perform the basic operations on a handlespace of a certain size, referring to the number of pools and PEs. These basic operations are defined as follows:

Registration/Deregistration Operations: Registration of a PE (due to explicit registration by a PR-H or learned via ENRP, see subsubsection 3.9.2.1) and deregistration of a PE (due to explicit deregistration, but also due to an expired Lifetime or a Keep-Alive timeout, see subsubsection 3.9.2.3).

Re-Registration Operations: Update of a PE's registration, in particular also of its policy information (see subsubsection 3.7.1.2).

Handle Resolution Operations: Handle resolution for PUs (see subsubsection 3.7.1.4).

Timer Operations: The handling of the Keep-Alive timer (PR-H, see subsubsection 3.7.1.3) and the Lifetime Expiry timer (non-PR-H, see subsubsection 3.9.2.1).

Synchronization Operations: The handlespace synchronization procedure (i.e. a step-wise traversal as described in subsection 3.10.5).

Every RSerPool scenario requires at least two PRs to make it operational and to provide a certain level of redundancy. Usually, every network of a certain size also contains routers, therefore it seems to be realistic that routers also host PR processes to avoid the costs for installing and maintaining dedicated devices. The routers which are available today already provide many more services than just routing: from DHCP (Dynamic Host Configuration Protocol) over DNS and HTTP proxies to VoIP gateways. Therefore, it is quite realistic that router vendors like Cisco will also offer a PR service in future devices.

Routers strongly rely on hardware support to cope with performance-critical tasks like the routing itself and checksum calculations. Furthermore, it is also possible to have TCP provided in hardware by a so called TCP Offload Engine (TOE). For details on TOEs and performance comparisons, see Mogul (2003), Rangarajan et al. (2002), Tomonori and Masanori (2003). Like for TCP, it is therefore also realistic to assume that SCTP Offload Engines (SOE) become widespread.

Since the intention of this work is to evaluate RSerPool and the performance of its handlespace management, the influence of the message transport is completely neglected and the focus is laid on evaluating the throughput of the handlespace operations.

7.3 The Measurement Setup

In order to perform a handlespace performance evaluation, it has been necessary to choose a realistic setup. Since the intention is to possibly run PR processes on routers, the selected system should have been equipped with a CPU power realistic for a router. That is, instead of using the newest and most powerful desktop PC, it has been decided to use an AMD Athlon-based PC containing a 1.3 GHz CPU – a system which has been state of the art about five years ago (in 2001). It is assumed that the computation power of such a CPU is realistic for the upcoming access router generation.

The handlespace management approach explained in chapter 4 mainly reduces the effort of maintaining a handlespace to the efficient handling of sorted sets. Therefore, the most performance-critical part is the efficiency of the set storage algorithms (see also subsection 4.4.7). In order to evaluate the handlespace management approach, the following data structures and algorithms for the maintenance of sorted sets have been realized:

Linear List: A linear list is the most obvious structure to store a set. For the implementation, a doubly-linked ring list is used, i.e. it supports the traversal in forward and backward directions. Furthermore, it allows for the insertion of a new node before or after – and the removal of – a *known* node in $O(1)$ time.

Binary Tree: The implementation of the binary tree uses iterative implementations of the insertion, lookup and removal functions. That is, no recursion – which would obviously be less efficient – is used here. The general implementation of the three operations is similar to the functions provided in LIBDICT (see Mela (2005)), with some important modifications for improving the handlespace management performance:

- To improve the speed of finding the next or previous node according to the sorting order, all nodes are also *back-linked*. That is, a node does not only link to its child nodes but also back to its parent node.

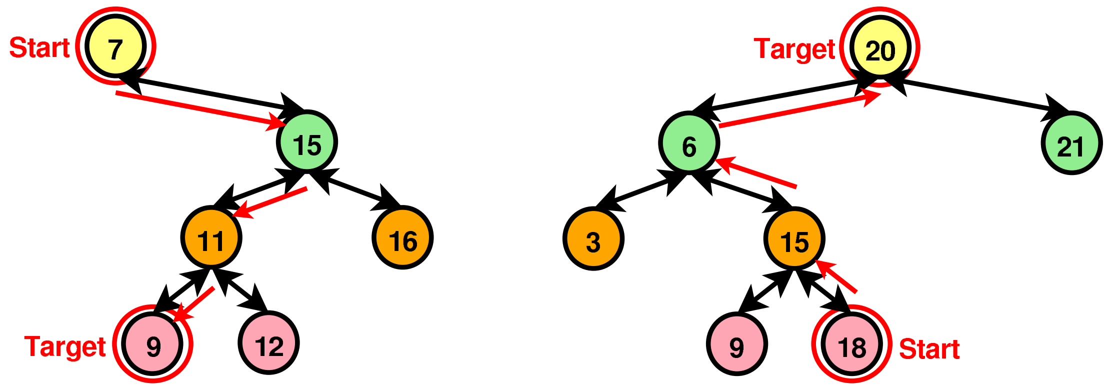


Figure 7.1: Finding the Successor of a Node in a Binary Tree

In particular, the back-linking simplifies the lookup of successor nodes in the tree. If looking for the successor of a node n, the following two cases can occur:

1. The node n possesses a right subtree. In this case, the node looked for is the right subtree's leftmost child. An example is provided on the left-hand side of figure 7.1.
2. Node n has no child nodes. In this case, the chain of ancestor nodes must be traversed – using the back-linked references – until the parent node's link is a left one. The right-hand side of figure 7.1 provides an example for this case.

- In order to efficiently handle the lookup of nodes based on the Weight Sum construct defined in subsubsection 4.4.2.1, it is possible to specify a positive integer weight constant w_i for each node i. Each node also includes a weight sum W_i, which is the sum of its own weight and the child nodes' weight sums. Therefore, the root node's weight sum $W_{\text{root}} = \sum_i w_i$ is equal to the sum of all nodes' weights as required by equation 4.2. Now, for any number $r \in [1, \ldots, W_{\text{root}}] \subset \mathbb{N}$, a uniquely identified node of the tree can be reached in a maximum number of steps equal to the depth of the node. An example is provided in figure 7.2, where r=19 chooses a node out of the tree with W_{root}=25. Note, that the weight sum maintenance with non-recursive insertion and removal functions (as it is realized for the implementation) requires back-linking as described above.

Treap: The treap implementation is basically equal to the binary tree, except for ensuring the treap constraint, as defined in subsection 4.4.7, upon insertion and removal by the rotation of nodes.

Red-Black Tree: As for the treap implementation, the red-black tree realization is also basically equal to the binary tree. Again, the only exception is the enforcement of the red-black tree constraints, as defined in subsection 4.4.7, by applying rotations upon insertion and removal of nodes.

Finally, to evaluate whether the idea of leaf-linking the tree's nodes (see subsection 4.4.7) provides a performance improvement, appropriately modified versions of the binary tree, treap and red-black tree implementations have been provided. To link the nodes, the implementation of the doubly-linked ring list has been used. That is, the tree structure becomes an index into the list; inserting before or after a known node into – or removing a known node from – the doubly-linked ring list itself is possible in $O(1)$ time.

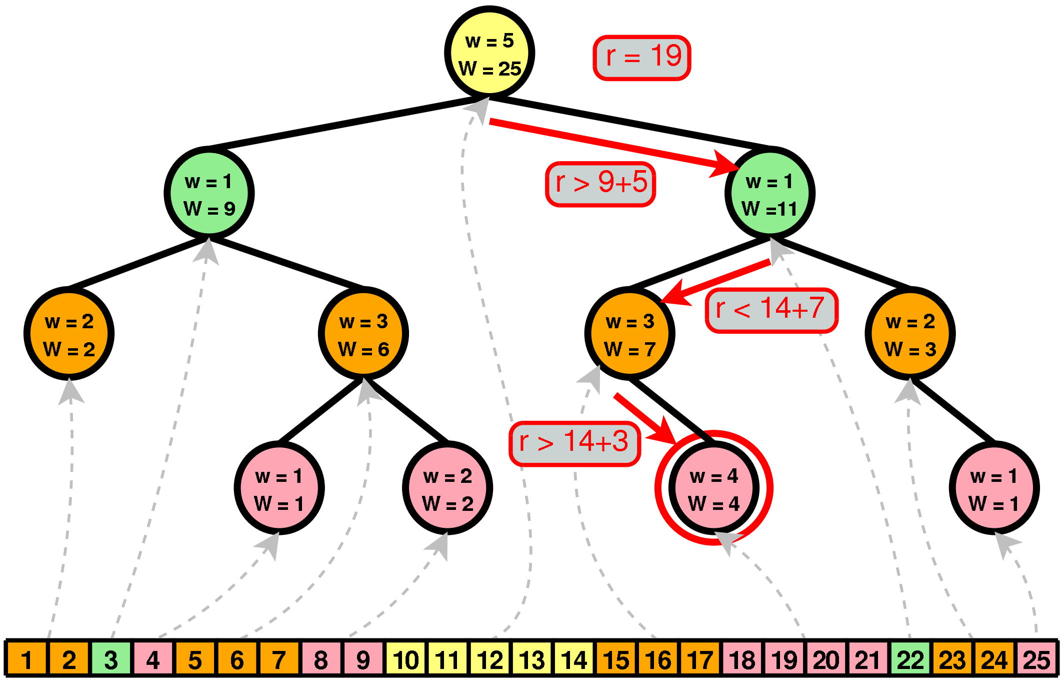


Figure 7.2: Using Selection by Weight Sum in a Binary Tree

The measurement program developed for the analysis of the handlespace management performance first creates a handlespace consisting of the given number of pools and pool elements. After that, the actual operations as described in section 7.2 are processed and their throughput is measured. All measurements have been performed using the already mentioned 1.3 GHz Athlon system, running Mandriva Linux 10.2 in runlevel 3 (i.e. without graphical user interface and any other services, in order to perform the measurements without disturbance by other processes). The sources have been compiled with GCC version 4.0.1, all possible optimizations have been turned on (option `-O3`).

The actual throughput measurement runtime for each parameter setting has been 5s. To ensure the necessary statistical accuracy, all measurements have been repeated 18 times. The plots show the average values of these measurement runs, together with their 95% confidence intervals. The legend in the plots provides the variable settings for each curve of the plot. Variable bindings can be found right-justified above the plots. Different settings of the first variable result in different colours/shades of the corresponding curves; different settings of a second variable are represented by different line styles (e.g. solid and dotted). The axis colour is unique for each output unit, in order to differentiate between comparable and non-comparable plots. More details on the post-processing of the obtained results are presented in subsection 6.3.2.

7.4 The Operations Throughput of Different Storage Algorithms

In the first set of performance simulations, the behaviour of the handlespace management under different set storage implementations for small handlespace sizes is evaluated. In particular, also inappropriate algorithms are presented for comparison. The scalability of the handlespace management to large scenarios for a choice of applicable set storage algorithms is shown later in the subsequent

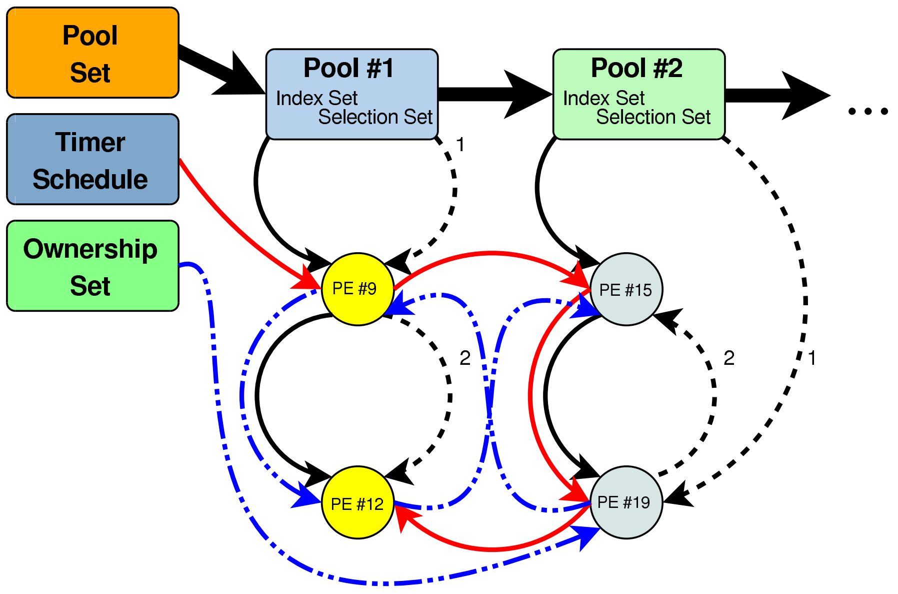


Figure 7.3: The Complete Handlespace Structure

section 7.5.

7.4.1 Registration and Deregistration Operations

The first operation evaluated is the combination of registrations and deregistrations as explained in section 7.3. Since it is not possible to only run registrations or deregistrations alone – the handlespace would grow or shrink – the performance of both operations together is observed: each time a registration/deregistration operation is invoked, the size of the pool is checked:

- If the number of PEs is equal to the configured pool size, a randomly selected PE will be deregistered.
- Otherwise, a new PE will be registered.

That is, the handlespace size remains steady. This behaviour is also realistic, since there is no deregistration without registration and every registered PE will – sooner or later – also be deregistered.

Upon a registration, the handlespace management has to perform the following tasks on the handlespace structure (illustrated in figure 7.3):

- Creation of the PE structure,
- Finding the pool given by the PH (a pool will be created if it is not already existing),
- Checking whether the new PE is compatible to the pool (referring to its policy information and transport parameters, see subsubsection 3.7.1.2),
- Linking the PE structure into the pool's Index Set (sorted by PE ID),
- Linking the PE structure into the pool's Selection Set (sorted by the sorting order of the pool's policy) and

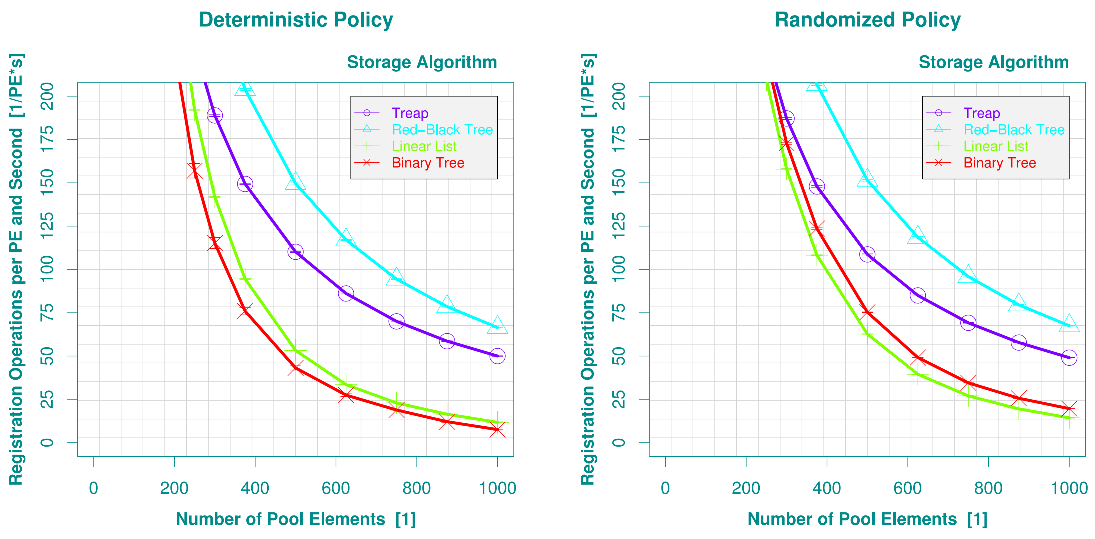


Figure 7.4: The Throughput of the Registration/Deregistration Operations

- Linking the PE structure into the Ownership Set (sorted by Home-PR ID/PE ID/PH, see subsection 4.4.5).

A registration operation is equivalent for both, registration via an ASAP Registration or an ENRP Handle Update message (see also subsubsection 3.7.1.2). The necessary timer handling (Keep-Alive or Lifetime Expiry) is evaluated separately in subsection 7.4.3.

The deregistration operation requires the following tasks:

- Finding the pool given by the PH,
- Finding the PE given by the PE ID within the pool,
- Unlinking the PE structure from the Index Set, Selection Set and Ownership Set,
- Disposing the PE structure and
- Removing the pool if it became empty.

As for the registration operation, a deregistration operation is equivalent for both, deregistration using an ASAP Deregistration or an ENRP Handle Update message (see also subsubsection 3.7.1.2). Note again that the timer handling is evaluated separately in subsection 7.4.3.

Figure 7.4 presents the throughput of the combined registration/deregistration operations (as explained above) per PE and second for deterministic (left-hand side; here: Round Robin) and randomized (right-hand side; here: Weighted Random) policies in a handlespace consisting of a single pool and the given number of PEs.

The obvious assumption on the performance of the registration/deregistration operation would be that there is no significant difference for distinct policies. While this can be verified for the linear list and the balanced trees (red-black and treap), there is a significant gap between the performance of the binary tree for a deterministic policy and a randomized policy: for example at 500 PEs, the randomized policy side achieves about 75 operations per PE and second, while less than 50 operations can be handled on the deterministic policy side. The reason for this behaviour is the Selection Set: for the randomized Weighted Random policy, the order of this set is irrelevant. Therefore, the set is sorted by PE ID to ensure uniqueness (see subsubsection 4.4.2.3). Since the PE ID is a random number, the nodes are inserted into the binary tree unsystematically. In this case, the depth of the binary tree can be assumed to be within $O(\log n)$. On the other hand, in case of the deterministic Round Robin policy, the PEs are inserted systematically: new nodes go to the end of the set, resulting in the binary tree to degenerate to a linear list. Clearly, the effort to manage a tree is slightly higher than for a linear list, which results in the lower performance of the tree.

As expected, the linear list and the balanced trees show no significant performance differences for varying policies and the performance of the balanced trees is clearly significantly better than the operations throughput of the linear list. Furthermore, the performance of the balanced red-black tree is quite constantly by about 25 operations per PE and second better than the performance of the treap. That is, instead of only trying to keep the probability of being unbalanced as low as possible (as it is realized by the treap using randomization), it is useful to add some more complexity in the storage operations to guarantee an optimal tree structure.

7.4.2 Re-Registration Operations

The next operation to be evaluated is the re-registration. A re-registration operation is equivalent for both, re-registration using an ASAP Registration or an ENRP Handle Update message. The necessary timer handling (Keep-Alive or Lifetime Expiry) is evaluated separately in subsection 7.4.3. For a re-registration, the handlespace management has to perform the following tasks on the handlespace structure (see figure 7.3 for an illustration):

- Finding the pool given by the PH,
- Finding the PE structure given by the PE ID and
- Checking whether the updated information is still compatible to the pool (referring to its policy information and transport parameters, see subsubsection 3.7.1.2) and
- If the update changes the policy information:
  - Unlinking the PE structure from the Selection Set,
  - Updating the PE information (including policy information and transport addresses) and
  - Re-linking the PE structure into the Selection Set again.

In case of a changed ownership (i.e. the PE has changed its PR-H), it is furthermore necessary to update the PE's location in the Ownership Set (i.e. by unlinking, updating and re-linking). Since the ownership change occurs quite rarely (only in case of a takeover), and the corresponding results do not provide any new insights compared to the adaptive policy case, they are omitted in the following analysis.

Since an updated policy information implies a removal from and re-insertion into the Selection Set, it is necessary for the performance evaluation to distinguish between adaptive policies (i.e. the policy information changes) and non-adaptive policies. Figure 7.5 presents the throughput of re-registration operations per PE and second for non-adaptive policies (left-hand side; here: Round Robin) and adaptive policies (right-hand side; here: Least Used). For the adaptive Least Used policy, each re-registration updates the policy information with a randomly chosen load value.

As expected, the linear list provides the worst performance for both classes of policies. Since the non-adaptive policy case only means PE structure lookup, there is no performance difference for

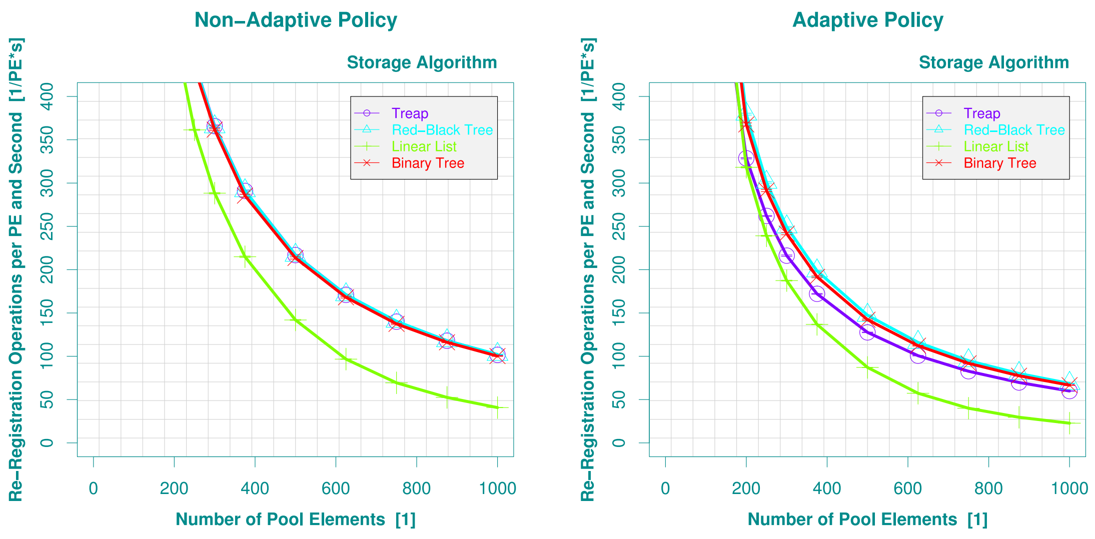


Figure 7.5: The Throughput of the Re-Registration Operation

the three tree-based variants. Furthermore, since the load value for the adaptive Least Used policy is chosen randomly, this results in an unsystematic insertion into the Selection Set. Therefore, no degeneration of the binary tree as described in subsection 7.4.1 can be observed. For the adaptive policy case, the binary and red-black trees provide almost the same and best performance. The somewhat lower operations throughput for the treap is a result of trying to optimize the structure of the tree where it is not necessary (otherwise, the performance of the binary tree would have been significantly worse). This results in time consumed for actually unnecessary node rotations.

In summary, as it has already been observed for the registration/deregistration operation in subsection 7.4.2, the best performance is provided by the red-black tree.

7.4.3 Timer Handling

As described in subsection 4.4.3, all timers are stored in the same Timer Schedule. While a non-PR-H only has to schedule a PE entry's expiration (Lifetime Expiry Timer), a PR-H has to schedule a Keep-Alive Transmission Timer. Upon its expiration, an ASAP Endpoint Keep-Alive message is sent to the PE and a Keep-Alive Timeout Timer is scheduled. Usually, the PE will acknowledge the keep-alive message and the timer can be unscheduled. After that, a new transmission timer has to be scheduled.

To evaluate the timer handling performance of the handlespace management, it is useful to define the timer operation as follows:

- For owned PEs, a timer operation consists of:
  - Linking the PE structure into the Timer Schedule with the timer type set to Keep-Alive Transmission Timer,
  - Unlinking the PE structure from the Timer Schedule,
  - Linking the PE structure into the Timer Schedule with the timer type set to Keep-Alive Timeout Timer and
  - Unlinking the PE structure from the Timer Schedule again.

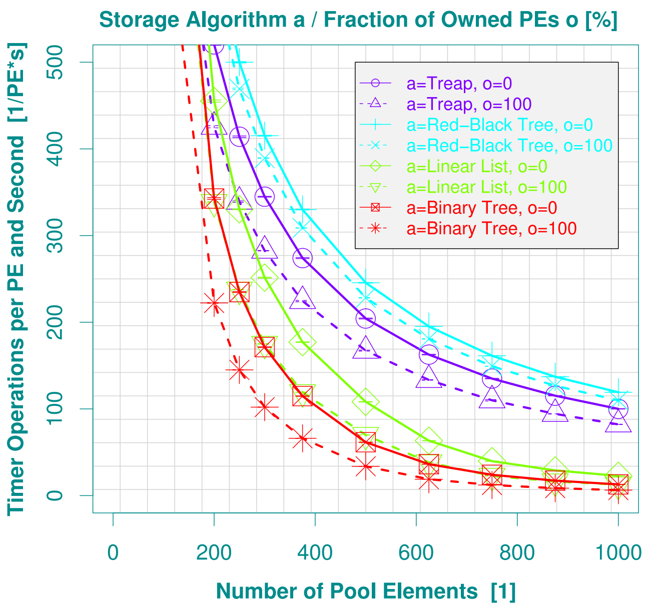


Figure 7.6: The Throughput of the Timer Handling Operation

- For not-owned PEs, a timer operation consists of:
  - Linking the PE structure into the Timer Schedule with the timer type set to Lifetime Expiry Timer and
  - Unlinking the PE structure from the Timer Schedule.

Figure 7.6 presents the throughput of timer operations per PE and second in a handlespace consisting of a single pool and the given number of PEs, for varying set storage algorithms and fractions of owned PEs. As fractions of owned PEs, 0% (none) and 100% (all) have been chosen, in order to make the difference between these two extremes visible.

For both fractions of owned PEs, the balanced trees (treap and red-black) clearly provide a significantly better performance than linear list and binary tree. Again, the operations throughput of the treap is somewhat lower compared to the red-black tree. However, the most interesting observation is the performance of the binary tree: it is even outperformed by the linear list. The reason for this effect is the systematic insertion and removal behaviour of the Timer Schedule: timers are scheduled for a future event, i.e. they are usually appended to the end of the schedule. On the other hand, expiring timers are removed from the top of the schedule. This results in a degeneration of the binary tree to (nearly) a linear list. Since the management of a binary tree is somewhat more complex than simply holding a linear list, the performance of the binary tree used for the Timer Schedule is worse than for a linear list itself. And unlike the registration/deregistration case, where the performance drop in the Selection Set is compensated by faster PE lookups in the Index Set (see subsection 7.4.1), no such compensation effect is available here.

Obviously, the timer operation for owned PEs is slightly more costly than for not-owned PEs, since in fact two timers have to be managed. While the throughput difference for the red-black tree between 0% and 100% of owned PEs remains smaller than 25 operations per PE and second for 200 to 1,000

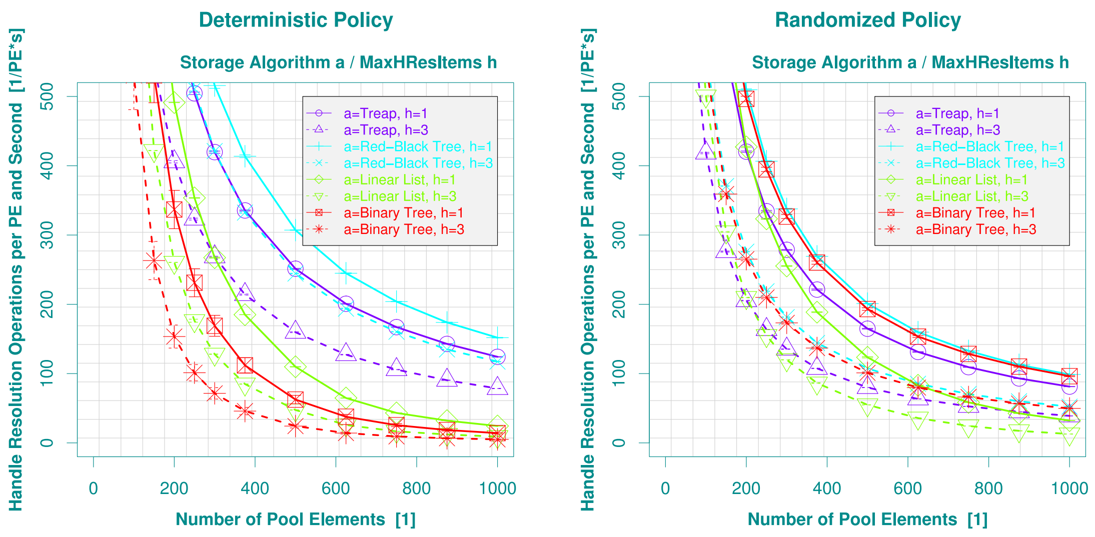


Figure 7.7: The Throughput of the Handle Resolution Operation

PEs, the gap grows to up to about 50 operations per PE and second for the three other algorithms. The reason for this small difference – despite of having to perform doubled work when dealing with the two Keep-Alive timers – is the CPU's cache: after unlinking the Keep-Alive Transmission Timer, its former structure is turned into the Keep-Alive Timeout Timer and vice versa. Therefore, the structure and parts of the Timer Schedule's nodes can be expected to be still situated in the CPU's cache.

7.4.4 Handle Resolution Operations

The next operation to be evaluated is the handle resolution.

7.4.4.1 Introduction

As described in subsection 4.4.1, the default selection procedure is to simply take the first element from the Selection Set. This is suitable for the deterministic policies like Least Used or Round Robin. But on the other hand, this simple selection procedure cannot be used for the randomized policies as defined in subsubsection 4.4.2.5. In this case, the weight-sum based selection is used. Clearly, the computational complexity of the "take from the top" selection procedure is $O(1)$, while the randomized variant depends on the size of the selection set (i.e. $O(n)$, where n is the size of the Selection Set). Using the tree-based approach presented in section 7.3, the processing complexity is $O(\log n)$, under the assumption of a balanced tree.

In order to realize the handle resolution operation, it is necessary to introduce two variables:

- $\mathrm{MaxHResItems}$ defines the number of PE identities which should be returned upon a handle resolution. That is, if $\mathrm{MaxHResItems}$=3, a handle resolution should return 3 PE identities. Of course, if the pool only consists of 2 PEs, only 2 PE identities can actually be returned.

- $\mathrm{MaxIncrement}$ specifies the maximum number of selected PE identities for which a status update is performed upon selection. As defined in subsubsection 4.4.2.1, a selection usually means to unlink the PE entry from the Selection Set, perform e.g. a sequence number update and re-link the PE into the Selection Set again. Limiting the number of PE entries which have to pass the unlink - update - re-link cycle is in particular useful for the Round Robin policies, as it will be described in subsection 8.8.1.

To process a handle resolution, the handlespace management has to perform the following tasks on the handlespace structure (see figure 7.3):

- Finding the pool given by the PH,
- Selecting up to MaxHResItems PE structures from the Selection Set,
- Unlinking up to MaxIncrement of the selected PE structures from the Selection Set,
- Updating the selection information (in particular the PE sequence number as defined in subsubsection 4.4.2.1) of the unlinked (and only of the these) PE structures and
- Re-linking the removed PE structures into the Selection Set again.

Due to the diversities of the selection procedures, the handle resolution results shown in figure 7.7 have been separated into a deterministic policy part (left-hand side; here: Round Robin) and a randomized policy part (right-hand side; here: Weighted Random). In the presented scenario, the case of $\text{MaxHResItems} \leq \text{MaxIncrement}$ is evaluated, i.e. all selected PE structures have to be unlinked, updated and re-linked. An advice on how to appropriately configure the MaxIncrement parameter is given later in section 8.8 and section 8.9.

7.4.4.2 An Unlimited Setting of MaxIncrement

As expected, the worst performance in the deterministic policy scenario is provided by the binary tree and the linear list. Again, the binary tree's operations throughput is worse than the performance of the linear list, due to the degeneration effect already observed for the timer and registration operations (see subsection 7.4.3 and subsection 7.4.2). However, for the randomized scenario, the performance of the binary tree is nearly as high as for the best algorithm: the red-black tree. In this scenario (Weighted Random policy), the Selection Set is ordered by the randomly chosen PE ID and no degeneration occurs.

Comparing the curve for MaxHResItems h=1 to the results of MaxHResItems h=3, the operations throughput for the deterministic policy only slightly decreases: from about 300 to about 250 operations per PE and second for using a pool of 500 PEs and the red-black tree. On the other hand, the performance nearly halves – from about 200 to only about 100 operations per PE and seconds – for the randomized policy. Clearly, this is the result of the more expensive selection procedure.

7.4.4.3 A Reduced Setting of MaxIncrement

As being explained later in section 8.8 and section 8.9, the parameter MaxIncrement can be smaller than MaxHResItems, i.e. more PE identities are returned than are actually passed through the unlink - update - re-link cycle. This is usually the case if multiple PE identities should be returned to support the PU's caching functionality, under the assumption that only a limited number of PEs (e.g. only one) is actually used for processing a request. Figure 7.8 shows the operations throughput using MaxHResItems=10 for a deterministic policy (left-hand side; here: Round Robin) and a randomized policy (right-hand side; here: Weighted Random). Since linear list and binary tree do not provide a reasonable performance, their results are omitted here. The settings of MaxIncrement have been "unlimited" and the minimum useful value. For the deterministic policy scenario, the smallest value

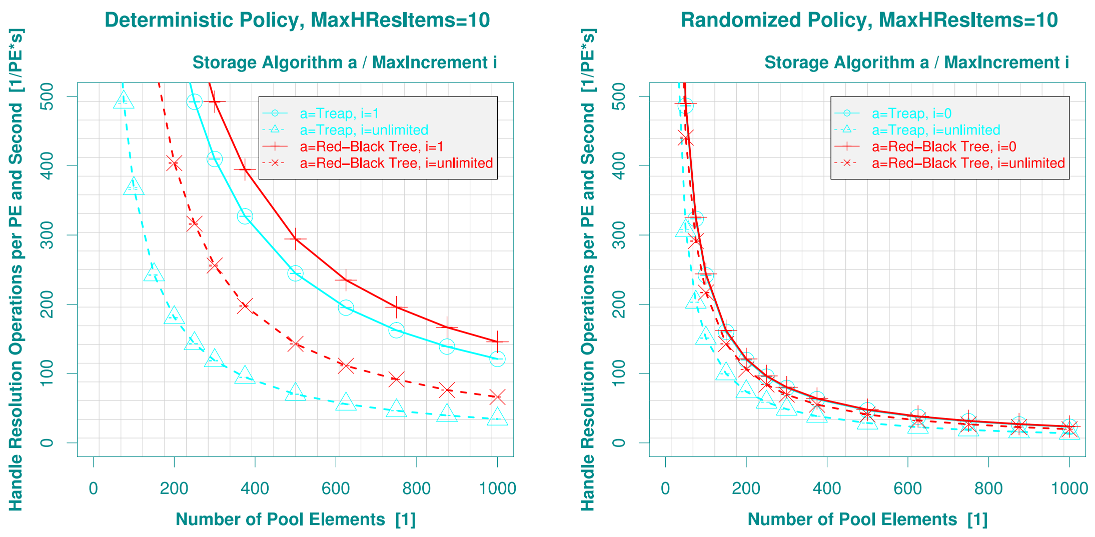


Figure 7.8: The Handle Resolution Throughput for a Variation of MaxIncrement

is 1 – otherwise, always the same list of PE identities would be returned. In case of a randomized policy, the minimal setting is 0.

For the deterministic policy, a reduction of $\mathrm{MaxIncrement}$ results in a significant performance gain: from about 140 to almost 300 operations per PE and second at 500 PEs for the red-black tree. The throughput increment is larger for the treap: from about 80 to 240 operations per PE and second. Clearly, a reduction of the unlink - update - re-link cycles – which are less efficient for the treap – provides a higher speed-up compared to the red-black tree. Nevertheless, the performance using a treap is always lower.

Comparing the results of the deterministic policy and the randomized policy, the choice of the parameter $\mathrm{MaxIncrement}$ only slightly changes the operations throughput of the randomized policy. The reason is the behaviour of the Weighted Random policy: the unlink - update - re-link cycle does not change the position of the entry. Therefore, it is re-inserted at the same place. Since all nodes in the tree leading to a just selected node can be assumed to be still in the CPU's cache, the unlinking and re-linking procedure is quite fast.

7.4.4.4 Summary

In summary, the red-black tree again achieves the best performance, while the treap's operations throughput is somewhat lower. A performance gain can be achieved by configuring the parameter $\mathrm{MaxIncrement}$ appropriately (i.e. smaller than $\mathrm{MaxHResItems}$), in particular for deterministic policies. The handle resolution operation for randomized policies is more costly, due to the more complex selection procedure ($O(\log n)$ vs. $O(1)$).

7.4.5 Synchronization Operations

The last operation to be evaluated is the synchronization. A synchronization operation denotes the step-wise traversal of the complete handlespace in order to obtain the necessary references to the PE structures for creating ENRP Handle Table Response messages as described in subsection 3.10.3. A synchronization operation performs the following tasks:

- Obtaining the first up to MaxElementsPerHTRequest references to PE structures.
- Saving PH and PE ID of the last PE reference obtained to resume the handlespace traversal.
- As long as the end of the handlespace data has not been reached yet:
  - Finding the subsequent PE entry of the PE whose identity (PH and PE ID) has been stored.
  - Obtaining the next up to MaxElementsPerHTRequest references to PE structures.
  - Saving PH and PE ID of the last PE reference obtained to resume the handlespace traversal.

There are two possibilities to select the PE identities to be included: all PEs or only the PEs owned by the PR itself. Since the "own PEs only" option is only a traversal on a subset of the PEs (based on the Ownership Set, see subsection 4.4.5) from the perspective of the synchronization operation, this case is neglected here. Instead, the performance analysis concentrates on the most time-consuming task: the full traversal.

First, it is necessary to specify the step size MaxElementsPerHTRequest for the traversal. Since the Handle Table Response message (see figure 3.34 for an illustration) – like all messages as defined in section 3.8 – only allows a size of 64 KBytes, the maximum useful step value is limited by the minimum size of the Pool Element Parameter as defined in subsubsection 3.9.2.1 (an illustration of this parameter is presented in figure 3.19): the minimum size of this parameter type is 60 bytes, consisting of 20 bytes for header, PE ID, Home-PR ID and Registration Life as well as at least 16 bytes for a User Transport Parameter (with a single IPv4 address), at least 16 bytes for an ASAP Transport Parameter (with a single IPv4 address) and at least 8 bytes for a Policy Parameter (e.g. Round Robin; see also figure 3.20 and Tüxen and Dreibholz (2006b)). Assuming that all PEs are in the same pool (i.e. only a single Pool Handle Parameter is needed; see also figure 3.18), an ENRP Handle Table Response may consist of up to about 1,050 PE identities. Clearly, it is useful to fill a message up to the limit. Therefore, a step size of 1,024 and a size of 128 for comparison are evaluated. The smaller value should cover realistic cases of multi-homed IPv4+IPv6 PEs and also the need for multiple PHs.

Figure 7.9 presents the operations throughput in operations per second for a varying number of PEs in a single pool. Note, that the synchronization operation is a global operation; therefore, it is not useful to show an operations per PE and second value here. Since a number of 1 to 1,000 PEs would not be very interesting – the first step of size 1,024 already returns all PE references – the number of PEs is varied between 1 and 5,000 here. Clearly, for less than 1,000 PEs the operations throughput for all storage algorithms significantly exceeds more than 15,000 synchronizations per second (except for the linear list and the small step size – with still more than 6,000 operations per second). This is by orders of magnitude more than sufficient for any realistic scenario of that handlespace size.

However, for larger scenarios the linear list becomes quite inefficient: for each additional step necessary, all PE structures traversed before have to be visited again and again. This effect is confirmed by the curve for a step size of 128: there is a high performance drop compared to the large step size. On the other hand, the tree-based algorithms are quite fast and achieve more than 850 synchronizations per second at 5,000 PEs. Furthermore, the difference between the step sizes of 128 and 1,024 is small. The performance of the binary tree is only very slightly worse than for treap and red-black tree: the Index Set, which is used for the PE ID lookup, is ordered by the random PE ID (see also subsection 7.4.1). An additional reason for the high performance of the tree-based implementations is the back-linking of the nodes as described in section 7.3: iterating to the next node is quite efficient, especially if the tree is balanced.

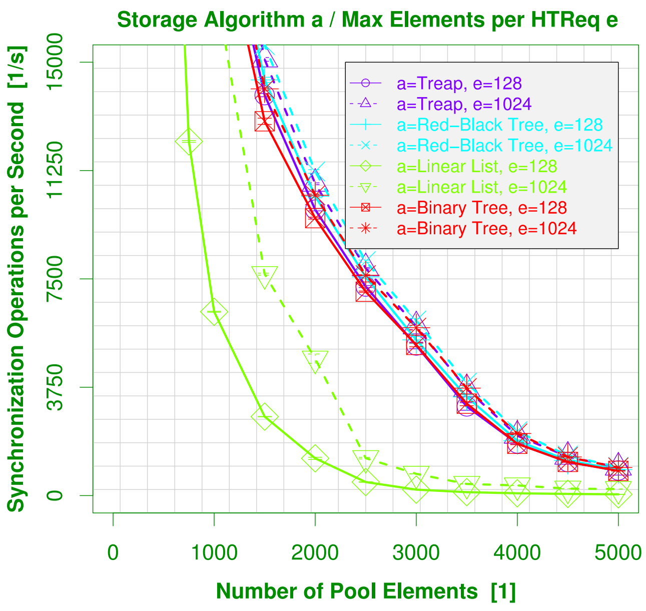


Figure 7.9: The Throughput of the Synchronization Operation

7.4.6 Summary and Conclusion

In summary, the first set of performance measurements has shown that the handlespace management approach is quite efficient in scenarios of up to 1,000 PEs – as long as an appropriate set storage algorithm is used: a balanced tree. While the treap – a randomized tree – implementation is somewhat simpler, the more complex red-black tree can guarantee balancing and therefore provides the best operations throughput.

While it is obvious that the naïve approach of using linear lists results in a low performance, simple binary trees have shown to be surprisingly inefficient: due to systematic insertion and removal in the Selection Set and the Timer Schedule, these binary trees can degenerate to linear lists. Such a degeneration results in an even worse performance than for a linear list itself.

After these interesting results, the next step of the performance analysis is to evaluate the scalability of the handlespace management approach to much larger pools.

7.5 The Scalability of the Number of Pool Elements

In this section, the performance of the handlespace management approach is evaluated for scenarios containing up to 100,000 PEs. Such scenarios seem to be realistic for large-scale real-time distributed computing pools as described in subsection 3.6.5. As already shown in section 7.4, a linear list or an unbalanced binary tree as storage implementations are completely unsuitable. Therefore, only the useful implementations – based on red-black tree and treap – are evaluated here.

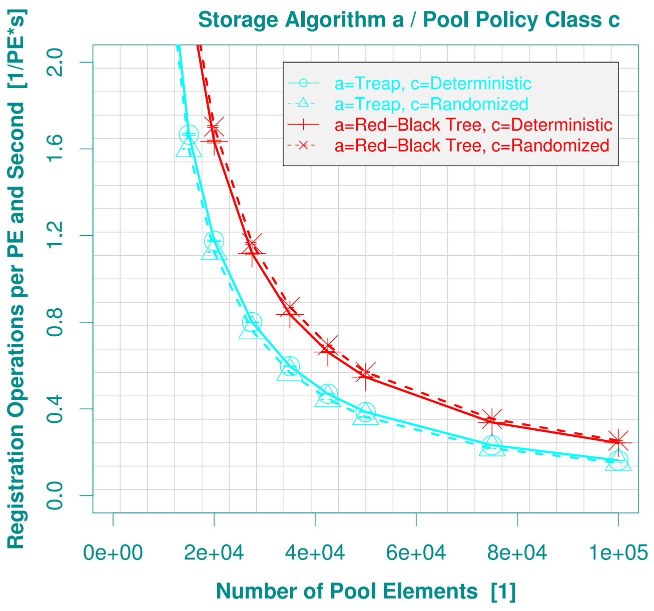


Figure 7.10: The Scalability of the Registration/Deregistration Operation

7.5.1 Registration and Deregistration Operations

The first operation to be evaluated in the scalability analysis is the combined registration/deregistration operation as defined in subsection 7.4.1. Again, it is useful to differentiate between deterministic and randomized policies.

Figure 7.10 presents the operations per PE and second for a range of PEs between 1 and 100,000. As expected from the results in subsection 7.4.1, no significant difference between deterministic policies (here: Round Robin) and randomized policies (here: Weighted Random) can be observed. Furthermore, as it has already been observed for the smaller scenarios, the red-black tree with its guarantee for being balanced provides a somewhat better operations throughput than the treap: about 0.55 operations per PE and second vs. about 0.4 operations per PE and second for 50,000 PEs.

In summary, the handlespace management approach is even applicable in a scenario of 100,000 PEs: at this number of PEs, each PE could register or deregister about every 3 seconds (more than 0.3 operations per PE and second, using a red-black tree). Assuming a realistic PE lifetime of as short as a few minutes, the registrations and deregistrations do not impose a significant load on the CPU.

7.5.2 Re-Registration Operations

The next operation of the scalability analysis is the re-registration operation as defined in subsection 7.4.2. Again, it is useful to differentiate between adaptive and non-adaptive policies. Figure 7.11 shows the operations per PE and second for a range of PEs between 1 and 100,000. For the adaptive policy (here: Least Used), each re-registration updates a PE's policy information with a randomly chosen load value. On the other hand, the non-adaptive policy (here: Weighted Random) does not modify the policy information.

As expected, the performance difference between treap and red-black tree is small, but the red-

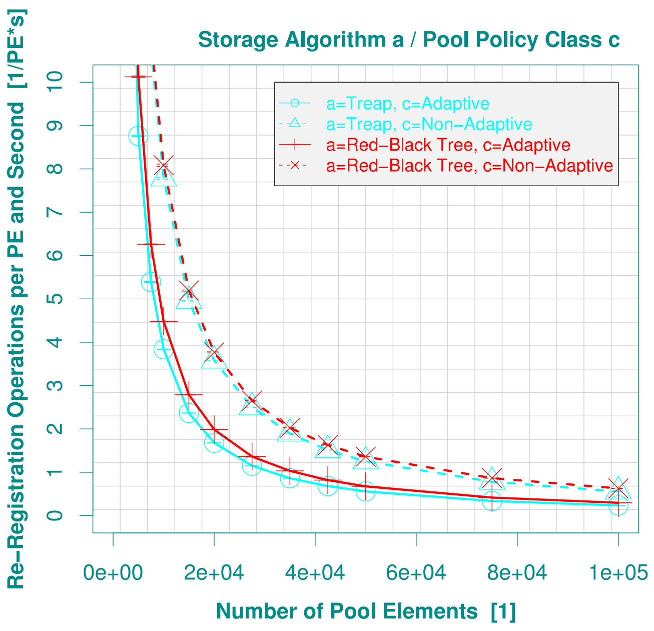


Figure 7.11: The Scalability of the Re-Registration Operation

black tree is a little bit more efficient. For the non-adaptive policies, both algorithms achieve an operations throughput of about 0.8 operations per PE and second for a pool of 100,000 PEs. Assuming a PE lifetime of as short as a few minutes, the achieved throughput for the non-adaptive policies is clearly sufficient: According to section 5.1 of Stewart, Xie, Stillman and Tüxen (2006a), the re-registration interval is defined as to the lower value of 10 minutes or the PE's Registration Life parameter (see subsubsection 3.9.2.1) minus 20s.

However, re-registrations may occur more frequently in case of an adaptive policy: whenever the PE decides that its policy information changes have to be propagated into the handlespace, e.g. on a load change for the Least Used policy, the PE performs a re-registration. As shown in figure 7.11, the operations throughput for 50,000 PEs still achieves about 0.8 operations per PE and second, while it drops to about 0.3 operations per PE and second for 100,000 PEs. That is, care has to be taken that the number of re-registrations does not overload the handlespace management if very large pools use an adaptive policy.

7.5.3 Timer Handling

The scalability of the operations throughput for the timer operation as defined in subsection 7.4.3 is presented in figure 7.12. Again, the results for the two extreme settings of the fraction of owned PEs – 0% (none) and 100% (all) – are shown.

As expected from the results obtained for the re-registration operation shown in the previous subsection 7.5.2, the best performance is provided by the red-black tree. The difference between 0% and 100% of owned PEs is quite small; however is can be assumed that the timer frequency for owned PEs is higher: in this case, the two Keep-Alive timers are managed (see also subsection 7.4.3). The frequency of the timer handling depends on the timeout settings of the Keep-Alive Transmission and Timeout timers. On the other hand, the timer of a not-owned PE is the Lifetime Expiry Timer, which

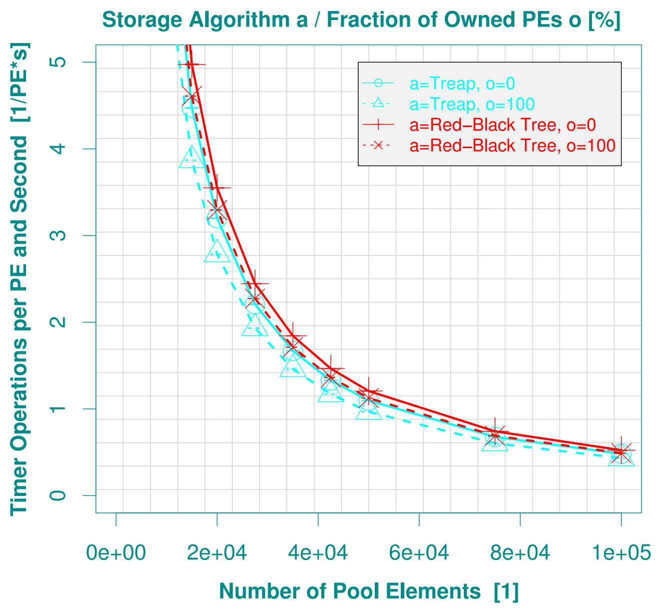


Figure 7.12: The Scalability of the Timer Handling Operation

depends on the PE's Registration Life setting (see subsubsection 3.9.2.1) and is obviously longer.

Nevertheless, the timer handling operation is fast enough to handle scenarios of 100,000 PEs at about 0.6 operations per PE and second. This is clearly sufficient for not-owned PEs (e.g. a realistic Registration Life could be in the range of a few minutes) and also for owned PEs (e.g. the Keep-Alive Transmission timer setting could be 60s and the timeout could be set to 10s in a pool of 100,000 PEs). Note further, that in large scenarios the owned PEs are usually distributed among the PRs, i.e. a PR is only the PR-H of a fraction of the PEs in the operation scope.

7.5.4 Handle Resolution Operations

The performance of the handle resolution operation as defined in subsection 7.4.4 is presented in figure 7.13. Again, it is distinguished between deterministic policies (i.e. using the default "take from the top" selection procedure as introduced in subsection 4.4.1) and randomized policies (i.e. using the weight sum based selection as described in section 7.3). In the presented scenario, $\mathrm{MaxIncrement}$ has been set to $\mathrm{MaxHResItems}$.

As already expected from the results in subsection 7.4.4, the best operations throughput is reached using the red-black tree: for 50,000 PEs and using a deterministic policy (here: Round Robin), about 2.5 operations per PE and second are achieved for $\mathrm{MaxHResItems}$=1, while there is still a throughput of more than 1.75 operations per PE and second for $\mathrm{MaxHResItems}$=3. On the other hand, in case of a randomized policy (here: Weighted Random), the performance drops from more than 1.0 operations per PE and second for $\mathrm{MaxHResItems}$=1 to about 0.5 operations per PE and second for $\mathrm{MaxHResItems}$=3. For an increased number of PEs, the operations throughput becomes significantly smaller.

Having a look at the results for $\mathrm{MaxIncrement}<\mathrm{MaxHResItems}$ and $\mathrm{MaxHResItems} = 10$ presented in figure 7.14, the expectations from subsection 7.4.4 are confirmed for large pools: for

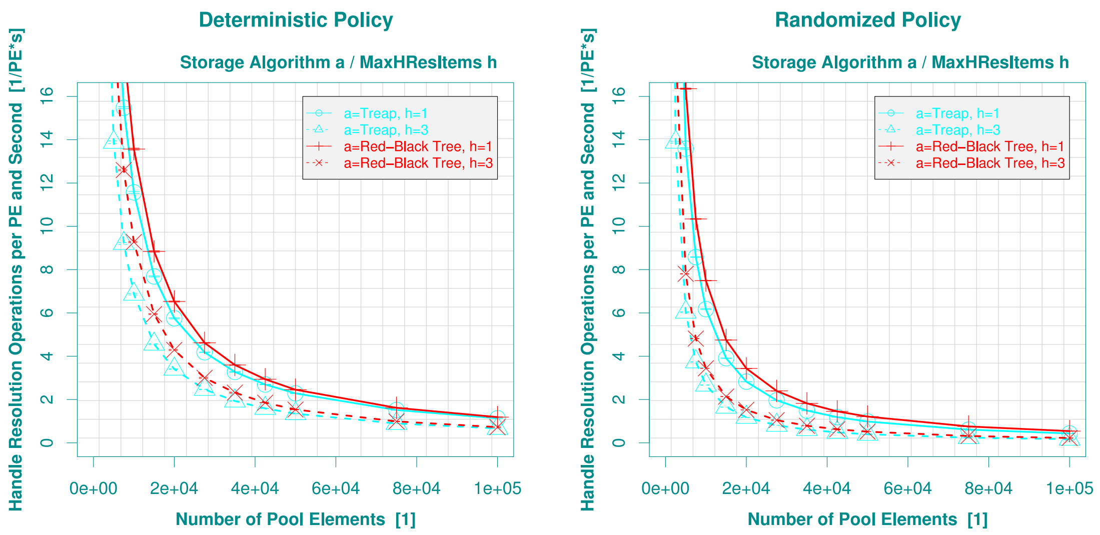


Figure 7.13: The Scalability of the Handle Resolution Operation

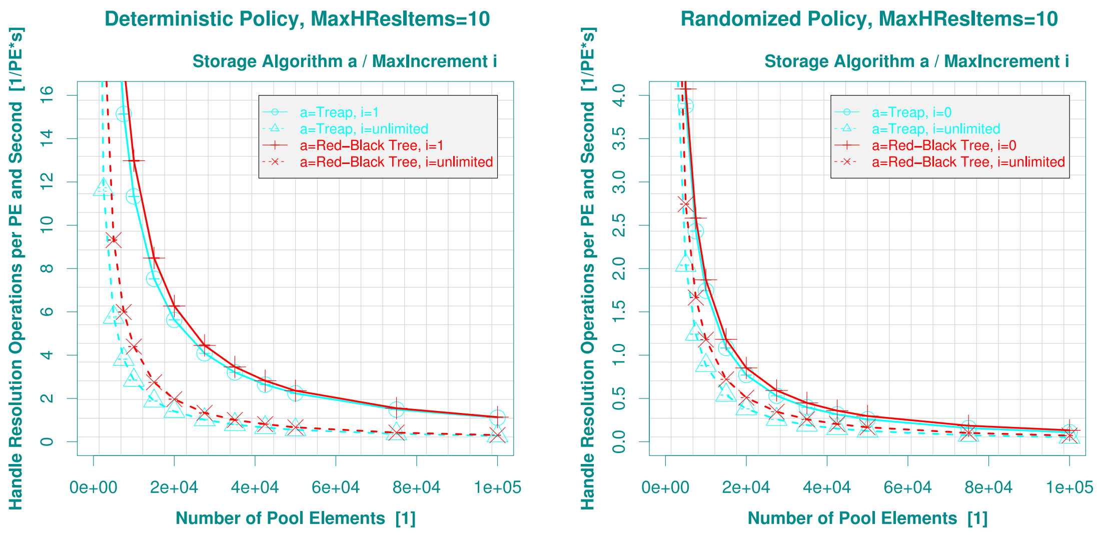


Figure 7.14: The Handle Resolution Scalability of a Variation of MaxIncrement

the deterministic policy (shown on the left-hand side; here: Round Robin), there is a significant performance difference between the curves for different settings of $\mathrm{MaxIncrement}$. In a pool of 50,000 PEs, a setting of 1 still achieves 2.5 operations per PE and second, while an "unlimited" setting drops the operations throughput to about 0.5 (red-black tree). The results for the randomized policy (presented on the right-hand side of figure 7.14) show the expected results: requiring all selected PE identities to run through the unlink - update - re-link cycle drops the operations throughput from about 0.3 operations per PE and second to about 0.2 operations per PE and second. Again, since the locations of the PE structures within the tree do not change here, the CPU's cache has accelerated the unlink - update - re-link cycle (see also subsection 7.4.4).

Clearly, the operations throughput for the randomized policy is significantly lower compared to the deterministic one (e.g. 2.5 vs. 0.3 operations per PE and second for $\mathrm{MaxHResItems}$=10 and the lowest possible $\mathrm{MaxIncrement}$ in a pool of 50,000 PEs and using red-black trees). However, despite of a low operations throughput in scenarios of large pools (e.g. more than 10,000 PEs), it is easily possible to scale a RSerPool system by adding additional PRs. That is, a pool of 100,000 PEs managed by 10 PRs can reduce the per-PR handle resolution load to $\frac{1}{10}\mathrm{th}$ – under the assumption that the request load of the PUs is approximately equally distributed among the PRs of the operation scope.

Finally, a comparison between red-black tree and treap again shows the expected result: the red-black tree is somewhat more efficient than the treap.

7.5.5 Synchronization Operations

The last operation to be evaluated is the synchronization operation as defined in subsection 7.4.5. For the smaller scenarios in this subsection (up to 5,000 PEs), only a small performance difference between the two realistic step sizes 128 and 1,024 has been observed. This observation can also be confirmed for pool sizes of up to 100,000 PEs as shown in figure 7.15. As it has already been observed for the previous operations, the red-black tree provides the best performance.

For a scenario of 100,000 PEs, it is still feasible to perform about 30 full synchronizations per second. This is by orders of magnitude more than necessary in any realistic scenario: in a well-planned RSerPool system of that size, there are probably at least 10 PRs – for redundancy reasons and to distribute the handle resolution and monitoring load. That is, the usual case is to only request the owned PEs in case of a detected handlespace checksum difference during the handlespace audit procedure (see subsection 3.10.5). Assuming that 100,000 PEs are approximately distributed equally among 10 PRs, such a handle table response may include about 10,000 PEs. For this table size, more than 500 operations per second are possible. The full handlespace synchronization procedure is only necessary on a PR startup, to request the complete handlespace content from a Mentor PR (see subsection 3.10.3). In a realistic scenario, this case might occur e.g. once within several days.

7.5.6 Summary

As a result of the scalability analysis in this section, it has been shown that the red-black tree is the appropriate data structure to store the handlespace data. In any case, a treap has a reduced performance, since it can only approximate a balanced tree structure instead of assuring it.

For the handlespace management, it has been shown that it is possible to manage a large-scale pool of up to 100,000 PEs if carefully deployed:

- For adaptive policies in large pools, care has to be taken of the rate of re-registrations, e.g. by a PR parameter or – the better solution – by a reasonable application design.

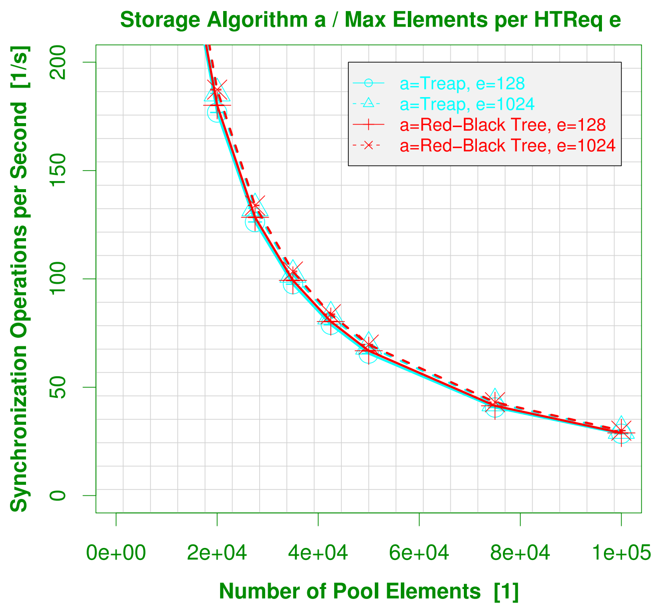


Figure 7.15: The Scalability of the Synchronization Operation

- Distributing the number of PEs and PUs among all PRs of the operation scope is useful: this reduces the per-PR workload of dealing with handle resolution requests and timer handling (Keep-Alive timers).

7.6 The Scalability of the Number of Pools

While only a single large pool as been evaluated in section 7.4 and section 7.5, it is furthermore necessary to analyse the handlespace management's scalability to a higher number of pools.

7.6.1 Analysis and Evaluation

Figure 7.16 presents the logarithm (base 10) of the number of handle resolution operations (see also subsection 7.4.4) per PE and second for a varying number ratio between pools and PEs, in handlespaces consisting of 100, 1,000 and 10,000 PEs. Showing the logarithm instead of the actual value has been chosen in order to fit the operations throughput of these very different pool sizes into a single plot. Parameters for the handle resolution have been $\mathrm{MaxHResItems}$=1 and $\mathrm{MaxIncrement}$=1, the policy has been Round Robin.

Clearly, the results mainly reflect what can be expected from the previous simulation results: using treaps leads to a reduced performance compared to the usage of red-black trees. Furthermore, distributing the number of PEs into a larger number of pools does not result in a severe performance degradation: at 100 PEs, the operations throughput keeps quite constant at slightly above 1,600 handle resolutions per PE and second – for all PEs in a single pool up to an own pool for each PE (red-black tree). For a handlespace consisting of 1,000 PEs stored in red-black trees, the throughput is reduced from about 150 operations per PE and second (for a single pool) down to about 140 (for an own pool

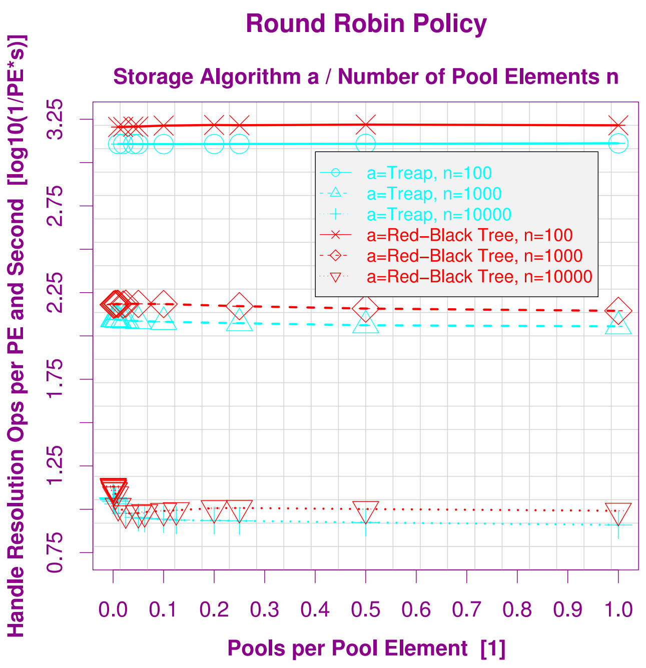


Figure 7.16: The Scalability of the Number of Pools

for each PE) – a reduction by less than 7%.

However, the reduction becomes larger for 10,000 PEs: from about 13.5 operations per PE and second to about 10.0 (i.e. a reduction by about 26%; red-black tree). Clearly, this is a result of the differences between Pool Set and Index Set: as explained in subsection 4.4.6, the maximum PH size has been limited to 32 bytes. In order to sort the entries of the Pool Set, a byte-wise comparison of PHs is necessary. For the measurements, randomized PHs of the maximum size have been constructed. Clearly, the PH comparison is more expensive than simply comparing 32-bit PE sequence numbers as it is needed in the Selection Set for the Round Robin policy (see subsubsection 4.4.2.2).

The results for registration, re-registration, timer and synchronization operations are very similar to the handle resolution results provided in this subsection. Since their results do not provide any new insights, they are omitted here.

7.6.2 Outlook on Future Scalability Enhancements

For realistic RSerPool scenarios, it can be expected that the number of pools in an operation scope is proportional to the number of deployed RSerPool applications. From the current perspective, values of up to 10 seem to be realistic, while cases of up to 100 might sometimes be useful. Currently, the only described application scenario with the requirement for a significantly larger number of pools is the RSerPool-based mobility support for SCTP, as described in subsection 3.6.6.

Since the number of pools is assumed to be very small, no additional effort for a speed-up of the pool lookup has been taken yet. As part of future work on the optimization of the handlespace management, it might be considered to use a hash table for the pool lookup. However, in this case care has to be taken to appropriately choose a hash function. A suitable hash function must meet the following requirements:

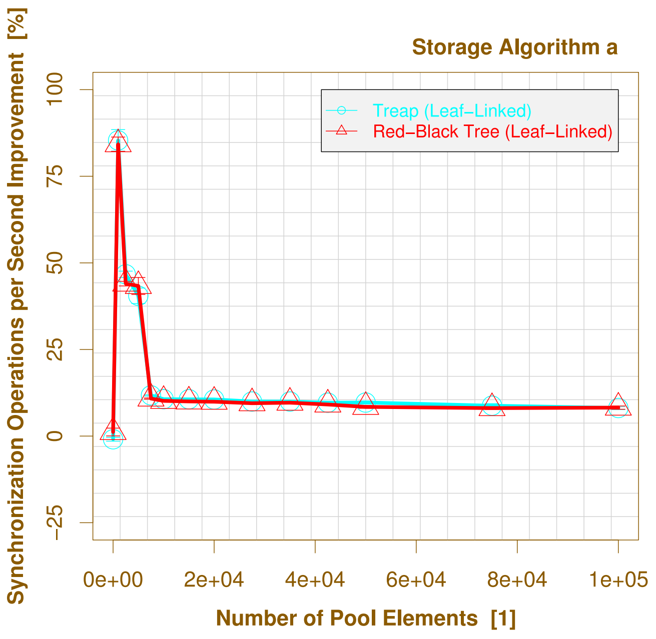


Figure 7.17: Using Leaf-Linked Trees for the Synchronization Operation

1. It has to be fast (of course) and

2. The hashing may not degenerate, even if an attacker creates pools with specially-crafted PHs.

The second requirement refers to so called computational complexity attacks, which are described in Crosby and Wallach (2003). The solutions presented there are so called universal hash functions.

A reasonable limitation of the current handlespace management is that the maximum PH length has been set to 32 bytes, in order to simplify its storage (see also subsection 4.4.6 for more details). While at the moment this limit seems to be justified and sufficient, special applications in the future might wish to use larger PHs. In this case, a hash function might also become useful for a smaller number of pools, due to the cost of comparing long PHs.

7.7 Using Leaf-Linked Trees

As it has been shown by the measurements presented in section 7.4 and section 7.5, the handlespace management approach – in combination with an appropriate set storage algorithm – already achieves a good performance. As explained in section 7.3, the back-linking technique should allow to easily find the predecessor or successor of any given node. However, as explained in subsection 4.4.7, leaf-linking, i.e. using the tree only for indexing into a doubly-linked linear list, may achieve another performance gain. To insert a new node before or after a known node into, or to remove a known node from a doubly-linked linear list is possible in $O(1)$ time. A balanced tree can ensure the quick lookup of nodes to insert before or after, as well as of nodes to be removed.

The main benefit of the leaf-linking technique can be expected from the synchronization operation (see subsection 7.4.5), while a minor speed-up may also be expected from the handle resolution operation (see subsection 7.4.4) in combination with a deterministic policy. The speed-up for the synchronization (in % compared to the conventional implementation) is shown in figure 7.17. Clearly, a significant performance improvement for scenarios of up to 10,000 PEs can be observed: by up to about 85% at 2,500 PEs. The reason for a higher performance improvement in small scenarios is that the average tree depth to PE number ratio $\frac{\text{treeDepth}}{\text{peNumber}}$ is higher if the pool size is small: e.g. $\frac{10}{1{,}000} = 0.01$ for 1,000 PEs, but only $\frac{16}{65{,}000} \approx 0.000246$ for 65,000 PEs. For rising ratios (i.e. more shallow trees), it can be expected that the number of nodes to visit for the back-linking technique described in section 7.3 also rises; this explains the better performance if leaf-linking is used.

For more than 7,500 PEs, the speed improvement for leaf-linked trees is still about 10%. As expected, no significant difference between treap and red-black tree is observable.

While leaf-linking clearly provides a significant speed-up for the synchronization operation, there is unfortunately a performance drop for the other four operations: the operations throughput decreases by about 2% for the treap and even by about 5% for the red-black tree. The reason for the treap's smaller decrement is its less-optimal balancing. That is, the depth of the treap is somewhat higher, requiring a few more node visits to find a predecessor or successor node without leaf-linking.

In summary, leaf-linking decreases the handling speed performance for both, treap and red-black tree: it only achieves a speed-up for the synchronization operation (which is rare) at the cost of an increased runtime for the other operations (which occur frequently).

7.8 Summary

As the measurements have shown, the appropriate algorithm for the data storage in the handlespace is the red-black-tree. For realistic scenarios of PE lifetime, re-registration and timer intervals, it is easily feasible to maintain a handlespace of up to 100,000 PEs on a router CPU, i.e. a CPU having a computation power comparable to the used 1.3 GHz Athlon. In this case, the handlespace management would not impose an overly significant load on the CPU.

The handlespace management approach is scalable for both, the number of PEs and the number of pools. However, for a large number of pools and long PHs, it may be considered to add a hash table for the pool lookup.

Chapter 8

RSerPool Performance Results

FOR a well-designed RSerPool scenario, it can usually be expected that component failures occur very rarely. That is, in 99.9. . . % of its runtime the system is in "normal" operation. Clearly, the performance of the system in this important case is most crucial for a system's cost-benefit ratio. The goal of this chapter is therefore to evaluate the performance of RSerPool systems in scenarios without failures. Failure cases are analysed in chapter 9.

8.1 Introduction

In order to perform evaluations of a RSerPool system based on the RSPSIM simulation model described in chapter 6, it is clearly the first step to design and realize an appropriate application model. This application model will be introduced in the subsequent sections. After that, suitable and realistic performance metrics – for service provider and service user – will be defined (see section 8.5).

To understand the effects of various network and RSerPool system parameters, it is crucial to know the implications of different workload compositions: request frequency and duration as well as parallelism. Therefore, the initial set of simulations presented in section 8.7 will evaluate these parameters first. After that, it is possible to answer important questions on the RSerPool system performance by performing further simulations:

The Policy Performance: Which pool policy achieves the best system performance, under which circumstances? Are there any pitfalls in policy configuration?

The PU-Side Handle Resolution Cache: What are the general effects of the PU-side handle resolution cache on the system performance? Under which circumstances can it provide a benefit, and how is it configured appropriately?

Coping with Network Delay: How does network delay affect the system performance? Is it possible to improve it in scenarios of globally distributed pools?

Handling Heterogeneous Server Capacities: What happens if the pool consists of PEs having different capacities? Is it possible to improve the system performance in such scenarios?

In particular, it is also a goal of the simulations and evaluations to identify critical parameter spaces to provide guidelines for dimensioning efficient RSerPool systems.

8.2 The Requirements for the Application Model

Before any performance evaluations can be performed, it is first necessary to create an application model. The application model to be designed has to meet the following requirements:

1. It has to model typical availability-sensitive Internet applications and
2. Client-based state sharing (see subsubsection 3.9.5.2) for failover support has to be integrated.

In order to design an application model fulfilling these requirements, it is necessary to have a look at some common Internet applications in order to identify their generic behaviour.

The most common Internet application protocol is the HTTP protocol (see also Fielding et al. (1999)): in order to download a file, it is requested using a GET command specifying the file name and optionally a byte range (e.g. to only get the data from the 10,000th byte to the end of the file). Using the byte range option, it is possible to continue an interrupted download. By applying the concept of client-based state sharing as described in subsubsection 3.9.5.2, this task could be handled by the Session Layer. The main resources consumed by the HTTP download are clearly I/O and network bandwidths; they are shared among all currently running sessions.

Another typical, availability-sensitive application is real-time distributed computing (see subsection 3.6.5). An application like the fractal graphics computation service provided by the demonstration system of the RSPLIB prototype implementation (see section 5.7 for a detailed description) requests the computation of a fractal image from a server. In case of a server failure, the calculation can be resumed using a state cookie. That is, it already includes the concept of client-based state sharing. The main resource consumed by a distributed computing system is computation power, hence the name. This resource is shared among all currently active sessions.

In summary, most Internet applications request a certain amount of resources (e.g. bandwidth or computation power) from a server. These resources are shared among all sessions. Session resumption in case of a failover could be realized by client-based state sharing.

8.3 The Design of the Application Model

Goal of the application model to be designed is to model the generic application properties which have been identified in section 8.2: a resource is requested from a server, it is shared among all active sessions; checkpoints set by state cookies provide the possibility to resume an interrupted session on another server. Since only the control path (i.e. the handling of sessions) is interesting, the data path is omitted. That is, the actual transmission of result data (e.g. web pages or calculated graphics data) is out of scope and therefore not included in the application model.

The message sequence of the application model's protocol – called *CalcAppProtocol* – during normal operation is presented in figure 8.1: a client can request the processing of a *Request* from a server using a *CalcAppRequest* message. Each request has a certain *Request Size* (RequestSize), which is given in the abstract unit of *Calculations*. A calculation denotes an arbitrary, application-specific unit: for a HTTP service the request size could denote a file size, while it could mean a number of operations to be performed for a real-time distributed computing job. Requests are generated in a certain *Request Interval* (RequestInterval) and are appended to the *Request Queue* as illustrated in figure 8.2. The requests of the request queue will be processed sequentially, i.e. each time the processing of a request is completed, the next one in the queue (if there is any) will be provided to a newly selected server for processing (like printing jobs in a printer's queue).

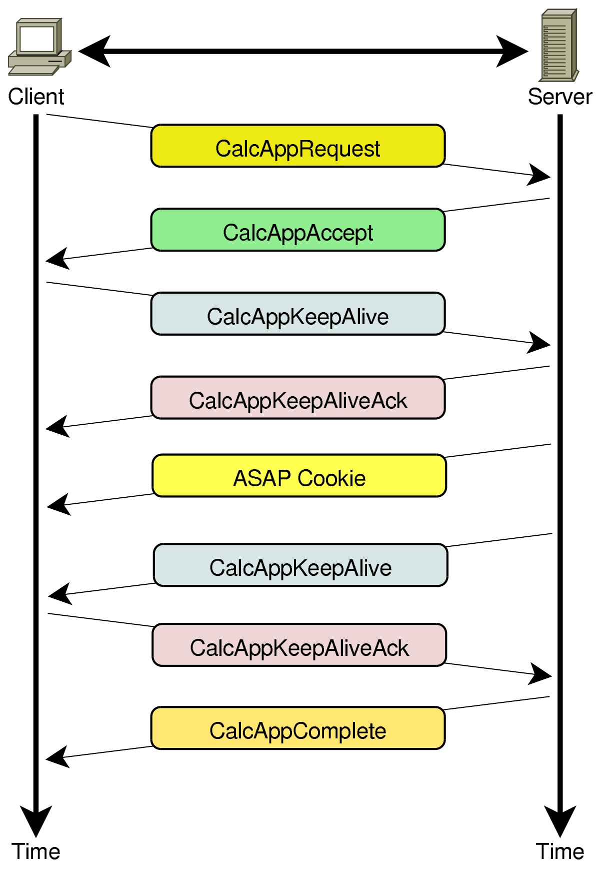


Figure 8.1: The CalcAppProtocol Message Sequence for Normal Operation

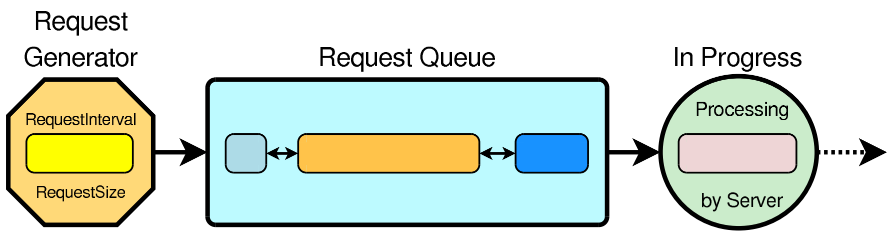


Figure 8.2: The Request Generation and Handling at the Client Side

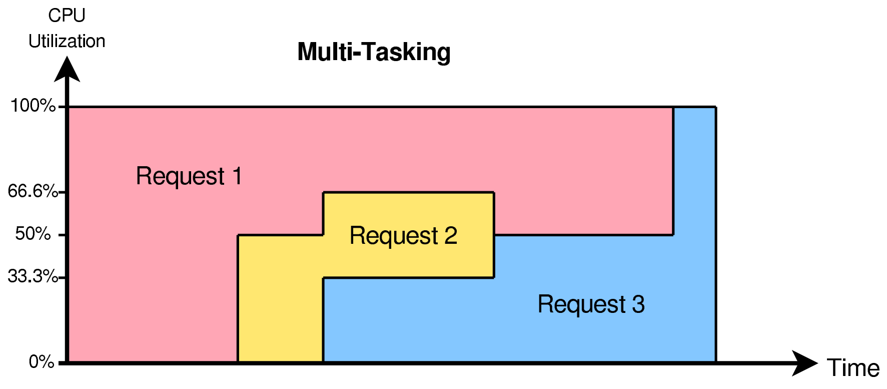


Figure 8.3: The Multi-Tasking Behaviour of the Server Side

A server can accept a new request if the number of currently active sessions does not exceed its configured limit MaxRequests. In this case, the acceptance of the request is signalled to the client using a *CalcAppAccept* message. On the other hand, if the client receives a *CalcAppReject* message, the request has been rejected and another server has to be chosen after waiting a certain time span given by the parameter RequestRetryDelay. The reason to wait is a consequence of the experience with the real-time distributed computing system described in Dreibholz, Rathgeb and Tüxen (2005), Zhang (2004): if currently no server can accept the request, the network would be flooded with server selection (i.e. handle resolution), request and reject messages.

The *Capacity* (Capacity) of a server is shared among all currently active sessions in a *Multi-Tasking* manner. Clearly, the unit for the capacity is calculations per second, i.e. depending on the application this could e.g. mean bandwidth (bytes per second) or processing operations per second. An example for the handling of requests in a real-time distributed computing system is illustrated in figure 8.3: at first, request #1 is the only active request. That is, this request can utilize the server's full capacity. When request #2 is started, it has to share the capacity (each request is handled with half of the server's CPU power). After starting a third session, each request can only be handled with one third of the capacity and so on.

During the processing of a request, both sides – i.e. the client and the server – send *CalcApp-KeepAlive* messages in a regular interval given by the parameter SessionKeepAliveInterval. The peer side has to acknowledge a CalcAppKeepAlive using a *CalcAppKeepAliveAck* message. If there is no answer within the time interval given by the parameter SessionKeepAliveTimeout, the peer side is assumed to be dead. For the server side, this simply means to remove the session; the client side has to perform a failover.

To actually support a failover, client-based state sharing is used. That is, the server side sends a state cookie each time one of the two following conditions applies:

1. The number of calculations consumed by the session after sending the last cookie or starting the session has exceeded CookieMaxCalculations calculations.

2. The time elapsed after sending the last cookie or starting the session has exceeded the number of seconds given by CookieMaxTime.

The message flow of a failover is presented in figure 8.4: after the failure of server #1 has been detected, the new server #2 is chosen and it is tried to resume the session by sending the state cookie to

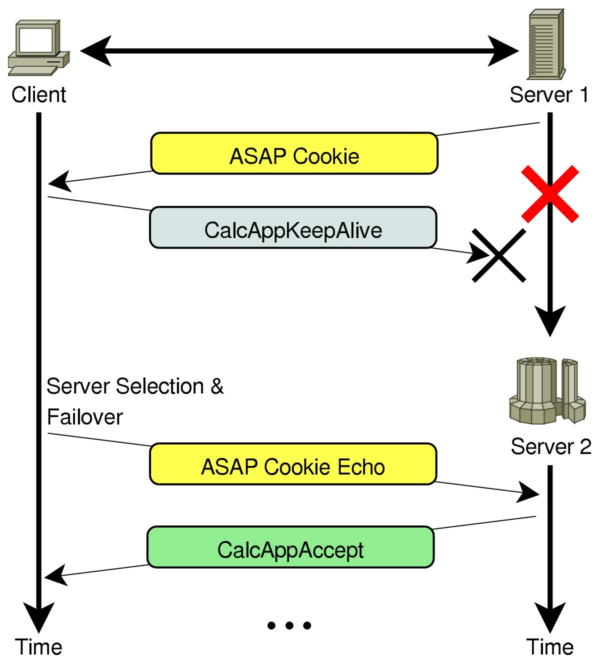


Figure 8.4: The CalcAppProtocol Message Sequence for a Failover

the new server using an ASAP Cookie Echo message. Server #2 has accepted the request and resumes the session at the checkpoint specified by the state cookie. Clearly, the calculations processed by the old server #1 between sending its last cookie and the failure are lost and have to be processed again by the new server.

A server can perform a so called “clean shutdown” by sending an ASAP Cookie directly followed by a *CalcAppAbort* message. That is, the client is told that the server cannot continue processing the request because it is shutting down. Due to the up-to-date state cookie, a new server does not need to re-process any calculation.

Finally, when the processing of a request is completed, it is signalled to the client by a *CalcApp-Complete* message.

8.4 The Implementation of the Application Model

To actually realize a PU and PE for the calculation application, the server and client sides have been implemented in form of a Server Application Process and a Client Application Process module for the RSPSIM simulation model, as described in subsubsection 6.4.3.2 (PE module) and subsubsection 6.4.3.3 (PU module). That is, the server registers into a pool using the Pool Element ASAP Process module and the client utilizes the Pool User ASAP Process module for server selection. The application-related parameters of PU and PE are summarized in table 8.1.

8.5 The Performance Metrics

Given the design of the application model introduced in section 8.3, there are two viewpoints to actually define performance: the service provider’s perspective referring to the pool of PEs and the

Parameter Name	Component	Description
Capacity	PE	Capacity of the server [Calculations/s]
MaxRequests	PE	Maximum number of simultaneous requests
CookieMaxCalculations	PE	Maximum calculations before sending a state cookie
CookieMaxTime	PE	Maximum time before sending a state cookie
RequestSize	PU	Size of requests [Calculations]
RequestInterval	PU	Interval of requests [s]
RequestRetryDelay	PU	Delay before retrying request distribution [s]
SessionKeepAliveInterval	both	Session Keep-Alive Interval [s]
SessionKeepAliveTimeout	both	Session Keep-Alive Timeout [s]

Table 8.1: The Calculation Application Pool Element and Pool User Parameters

service user's perspective concerning the PUs. Therefore, two performance metrics are being defined in the following sections – one for the service provider side and one for the service user side.

8.5.1 Performance from the Service Provider's Perspective

For a service provider, the primary goal – and therefore the main performance metric – is obviously to optimize the *System Utilization* (see also Dreibholz and Rathgeb (2005c,e)), which is defined as

$$\mathrm{SystemUtilization} = \frac{\mathrm{UsedResources}}{\mathrm{AvailableResources}}. \qquad (8.1)$$

That is, how many of the available calculations have been actually used to process work? In the usual case, a server pool is designed for a certain average *Target System Utilization* of e.g. 80%. If the utilization is lower, expensive server capacity is wasted. On the other hand, a too high utilization reduces the system's capability to cope with temporary overload and the resilience in case of component failures.

A secondary goal of the service provider is to keep the pool management cost low. As described in chapter 7 and Dreibholz and Rathgeb (2005b), pool management mainly means handlespace management. The secondary performance metric of the service provider side is therefore to keep the handlespace operations' runtime in an acceptable range.

8.5.2 Performance from the Service User's Perspective

The goal of the service user is clearly to get his requests handled as quickly as possible. That is, the performance metric on the service user side is to maximize the *Handling Speed* (see also Dreibholz and Rathgeb (2005c)), which is influenced by the items shown in figure 8.5:

Queuing Delay: After a request has been generated, it is enqueued into the request queue to wait until all previously created requests have been processed (see section 8.3).

Startup Delay: When the request is dequeued from the request queue, a PE has to be selected (possibly requiring a RTT between PU and PR for a handle resolution) and the request has to be proffered to the selected PE and be accepted (requiring a RTT between PU and PE). Possibly, the PE rejects the request and the procedure has to be repeated until a PE accepts the request.

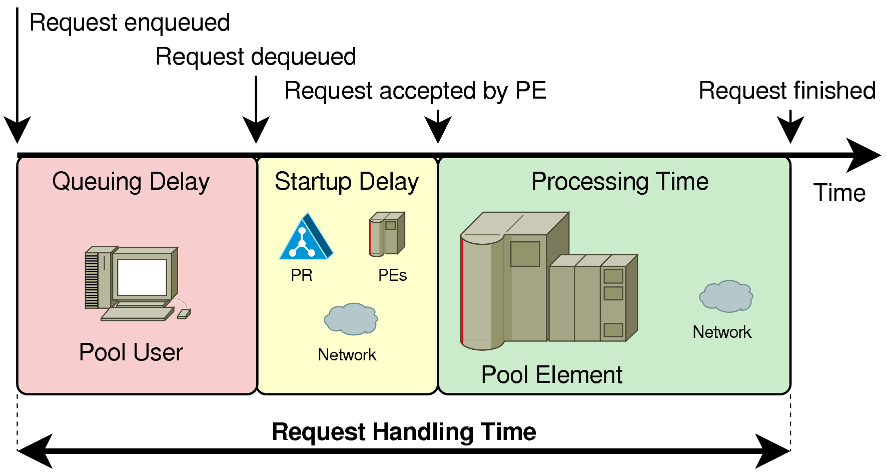


Figure 8.5: The Definition of the Request Handling Time

Processing Time: The processing time is the time span between the acceptance of a request by a PE (reception of the CalcAppAccept at the PU) and its completion (reception of the CalcAppComplete message at the PU). In case of failovers, the processing delay not only incorporates the actual time required to process the request (the so called *Goodput Time*) but also the time for failover (i.e. selecting and contacting a new PE, as well as re-processing calculations not saved by the latest state cookie; this is the so called *Failover Time*).

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

$$d_{\text{Handling}} = d_{\text{Queuing}} + d_{\text{Startup}} + d_{\text{Processing}}. \tag{8.2}$$

Given a request's size, it is possible to calculate the request's *Handling Speed* (in calculations/s) as follows:

$$\text{HandlingSpeed} = \frac{\text{RequestSize}}{d_{\text{Handling}}}. \tag{8.3}$$

The performance metric for the pool user is clearly to maximize the average handling speed for the requests in the system. That is, the sum of all completed requests' sizes divided by the total time consumed for their handling should be as high as possible.

In order to also measure the speed a PE requires to process a request of a given size, the *Processing Speed* is defined as follows:

$$\text{ProcessingSpeed} = \frac{\text{RequestSize}}{d_{\text{Processing}}}. \tag{8.4}$$

In order to make handling speed and processing speed values independent of the average PE capacity, it is practical to normalize them. Therefore, it is useful to divide the processing and handling speed (in calculations/s) by the average PE capacity (in calculations/s) and represent the resulting ratio in %. For example, a processing speed of 100% means that the request has been processed at the full average rate; if the request has got half of the average capacity, the processing speed is 50%.

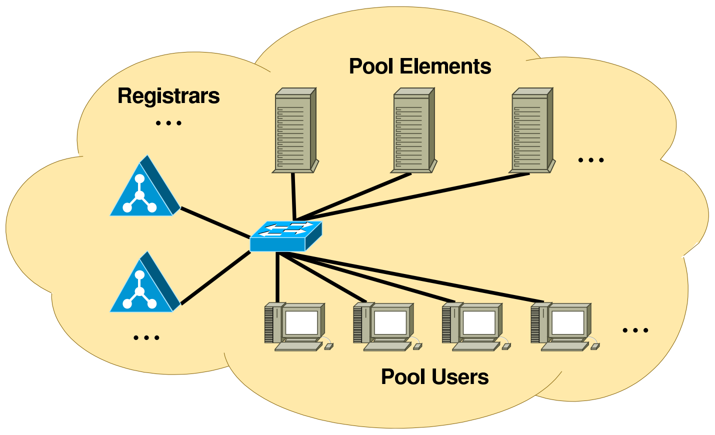


Figure 8.6: The Performance Simulation Scenario

Furthermore, the size of requests can be represented by the ratio between the request size in calculations and the average PE capacity (in calculations/s): $\frac{\text{RequestSize}}{\text{peCapacity}}$. This ratio will be denoted as request size:PE capacity ratio and represents the runtime (in seconds) required for exclusively processing a request of the given size on a PE providing the average capacity.

8.6 The Simulation Scenario Setup and Results Presentation

The simulation scenario setup used for the following performance simulations is presented in figure 8.6. It consists of one or more LAN modules (see subsection 6.4.1), each including a variable number of PR, PE and PU modules connected to a switch (i.e. a Transport Node module, see subsubsection 6.4.2.2). Unless otherwise specified, the following system parameters are used for the simulations presented in the subsequent sections:

- An actual simulated real-time of 120 minutes, after a startup phase of 15 minutes for initializing and starting up all components (all statistics are reset after the startup phase);
- No component latencies (the latency of the handlespace management is negligible, as shown in section 7.4 as well as in section 7.5 and in Dreibholz and Rathgeb (2005b));
- Latency-free network links (the effects of network delay are evaluated in subsection 8.10.1);
- A network consisting of a single LAN module (LANs interconnected via WAN links are evaluated in subsection 8.10.1);
- 1 PR (since failure scenarios are not evaluated here; see section 8.9 for the effects of varying the number of PRs);
- 10 PEs, each providing a capacity of 10^6 calculations/s (heterogeneous server capacity scenarios are evaluated in section 8.12);
- The number of simultaneous requests per PE is not limited, i.e. requests will not be rejected;
- A target system utilization (see subsection 8.5.1) of 80%;

- Negative exponential distribution for request intervals and request sizes, since the goal is a generic parameter sensitivity analysis which is independent of specific application types (see subsection 8.5.1) and
- The PU-side handle resolution cache is turned off (i.e. the stale cache value is set to 0s; the cache parameters are evaluated in subsection 8.11.1).

For the load-based pool policies, *Load* is defined as the current number of simultaneously handled requests; a re-registration is performed on every load change.

Every simulation consists of 24 runs – each using a different seed value – to ensure a sufficient statistical accuracy. The resulting plots have been generated using GNU R as described in subsection 6.3.2 and show the average values and their 95% confidence intervals.

Unless otherwise specified, the results presentation in this and the following chapter is structured as follows:

- A results figure contains two plots; the left-hand plot shows the average system utilization curves (provider's performance metric) and the right-hand plot presents the average request handling speed (user's performance metric). The axis colour is unique for each output unit, in order to differentiate between comparable and non-comparable plots.
- The handling speed is being normalized by the average PE capacity and presented in percent. That is, if a request has been handled with a speed of $7.5*10^5$ calculations/s and the average PE capacity is 10^6, the resulting value is 75%.
- The legend shows the variable settings for each curve of the plot. The bindings of the variables and their units in square brackets (if useful) can be found right-justified above the plot.
- To enhance clearness, different settings of the first variable result in different colours/shades of the corresponding curves. Different settings of a second variable are represented by different line styles (e.g. solid and dotted).

For example, the plots of figure 8.8 include the following two variables:

1. The pool policy p (with the policy name, of course) and
2. The request size:PE capacity ratio s (in seconds).

8.7 Understanding the Impact of the Load Parameters

In order to evaluate the performance of RSerPool systems, it is mandatory to well understand the influence of different load parameters. Therefore, this section first quantifies load and then shows initial simulation results to identify critical parameter ranges. These results have also been published in Dreibholz and Rathgeb (2005c).

8.7.1 Quantifying Workload

As described in section 8.3, two PU-side parameters define the load created by a PU:

- The request size ($\mathrm{RequestSize}$) and
- The request interval ($\mathrm{RequestInterval}$).

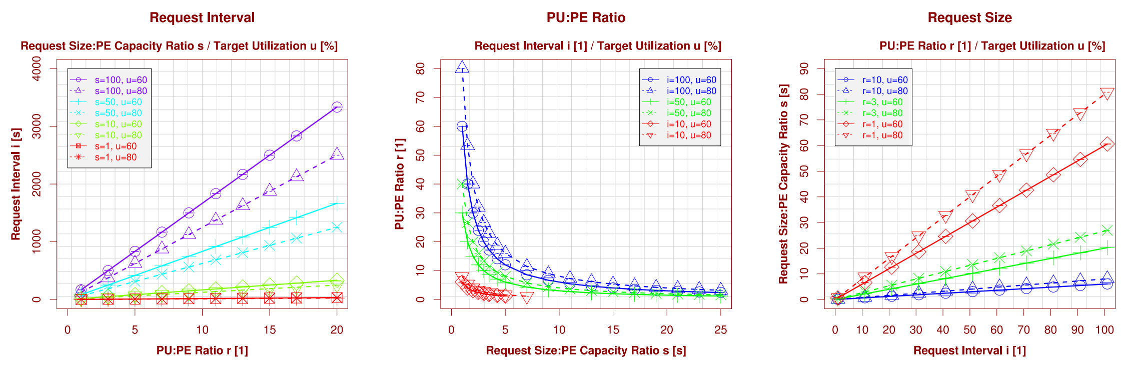


Figure 8.7: The Coherence of the Three Workload Parameters

Given the number ratio between PUs and PEs, the so called *PU:PE Ratio* (puToPERatio) as well as the (average) capacity per PE, it is possible to calculate the system's utilization:

$$\mathrm{SystemUtilization} = \mathrm{puToPERatio} * \frac{\frac{\mathrm{RequestSize}}{\mathrm{RequestInterval}}}{\mathrm{peCapacity}}. \tag{8.5}$$

The load fraction generated by a single PU is given by the following formula:

$$\mathrm{puLoad} = \frac{\mathrm{RequestSize}}{\mathrm{RequestInterval} * \mathrm{peCapacity}}. \tag{8.6}$$

A well-designed RSerPool system is provisioned for a certain target system utilization (denoted as $\mathrm{TargetSystemUtilization}$, e.g. 60% or 80%) as described in section 8.5. That is, under the assumption of a fixed average PE capacity, given values for any two of the three workload parameters (i.e. $\mathrm{puToPERatio}$, $\mathrm{RequestSize}$ and $\mathrm{RequestInterval}$), the third parameter can be calculated using equation 8.5. This leads to the following three equations:

$$\mathrm{RequestInterval} = \frac{\mathrm{puToPERatio} * \mathrm{RequestSize}}{\mathrm{TargetSystemUtilization} * \mathrm{peCapacity}}, \tag{8.7}$$

$$\mathrm{puToPERatio} = \frac{\mathrm{RequestInterval} * \mathrm{TargetSystemUtilization} * \mathrm{peCapacity}}{\mathrm{RequestSize}}, \tag{8.8}$$

$$\mathrm{RequestSize} = \frac{\mathrm{RequestInterval} * \mathrm{TargetSystemUtilization} * \mathrm{peCapacity}}{\mathrm{puToPERatio}}. \tag{8.9}$$

To make the coherence among the three workload parameters clear, examples are provided in figure 8.7: the left-hand side shows the request interval based on equation 8.7, the middle part the PU:PE ratio based on equation 8.8 and the right part the request size based on equation 8.9. Note, that the request size has been normalized by the (average) PE capacity to make it independent from the capacity setting.

In the following three subsections, the performance of RSerPool systems is evaluated for different policies and variations of the workload parameters described above.

8.7.2 Variation of PU:PE Ratio and Request Size

For the first simulation, the PU:PE ratio r has been varied from 1 to 20 for request size:PE capacity ratios s from 1 to 100. For each pair of both values, the request interval has been calculated using equation 8.7 described in subsection 8.7.1 (see the left part of figure 8.7 for an illustration). The results of the simulation are presented in figure 8.8.

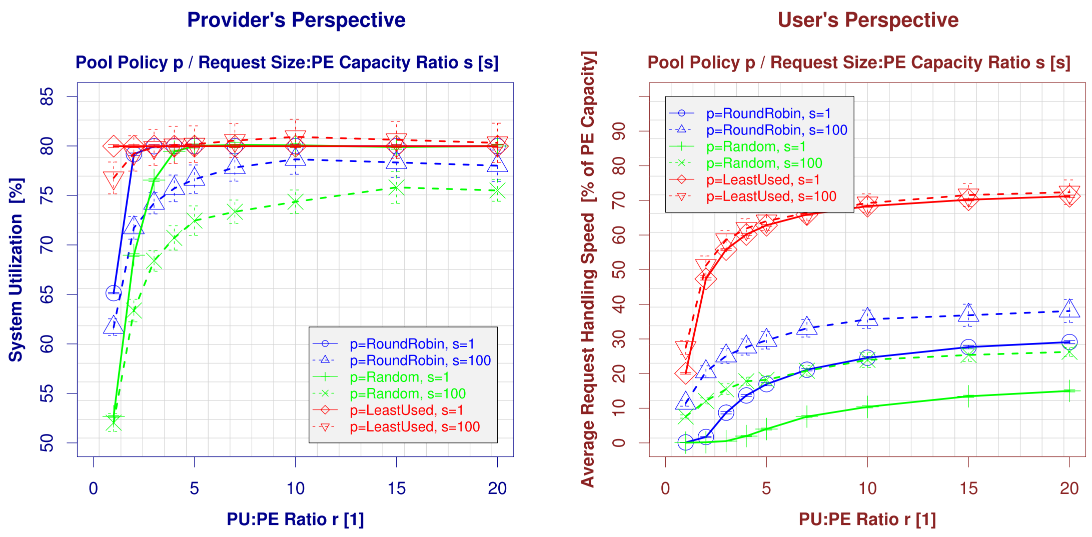


Figure 8.8: The Variation of PU:PE Ratio and Request Size

8.7.2.1 The Impact on the System Utilization

As shown, the PU:PE ratio r – giving the degree of parallelism in request handling – has a significant impact on the utilization: at r=1, the utilization is at 53% for the Random policy and between 60% and 65% for Round Robin. Using the Least Used policy, it nearly reaches 80%. The utilization difference for the policies becomes significantly smaller if r increases: for r=5, the difference is about 6% and for r=10, it decreases to about 3%.

The reason for this behaviour (i.e. a decreasing utilization difference for a rising PU:PE ratio r) is the number of requests processed simultaneously by the PEs: a PU:PE ratio r=1, there should be exactly one PE for every PU. That is, each PU expects to occupy a PE exclusively, which processes its requests during 80% (target utilization) of its runtime (see also equation 8.5 and equation 8.6). Each time a "bad" PE is selected for a request, one PE remains idle while another one has to split its capacity up to handle two requests simultaneously. Obviously, this behaviour is most frequent if PEs are chosen randomly. For Round Robin selection, the PE just selected should be chosen again only after having used every other PE before. This method already achieves a significant improvement over the Random policy. Finally, Least Used has the knowledge of the PEs' current load states; therefore – except for the rare cases of simultaneous selection – the least-loaded PE can be used. This is the reason for the superior performance of the Least Used policy.

Observing the utilization for a variation of the request size:PE capacity ratio s, only minor differences are shown. Even for a change of two orders of magnitude as presented in figure 8.8 (left-hand side), the utilization between s=1 (solid lines) and s=100 (dotted lines) only decreases by about 3% for the Least Used policy, about up to 4% for Round Robin about 1% for Random. The reason for the utilization decrement is that longer requests increase the impact duration of the selection decision: for example, a heavily-loaded PE may become idle within the next few seconds while a 75%-loaded one – putatively the appropriate choice – may stay at this load level for quite some time. This effect is illustrated with an in figure 8.9: "bad" selections for large requests (left-hand side) may last for quite some time, while the effect of such a selection for small requests (right-hand side) is short. An analogon is presented in figure 8.10: it is more wasteful to store large stones in a bucket than storing small pebbles – although the volumes of both sets of stones are equal. Clearly, the probability of a

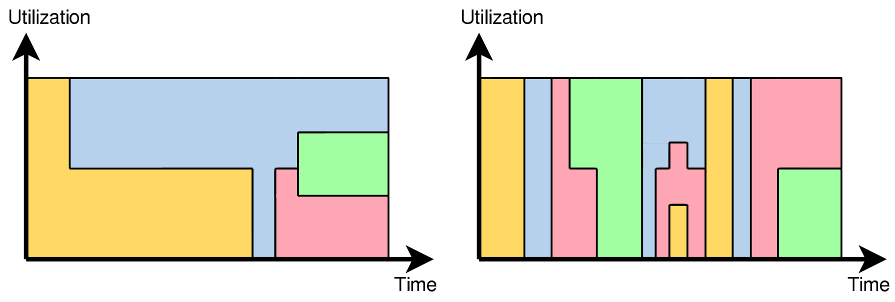


Figure 8.9: A Request Scheduling Example

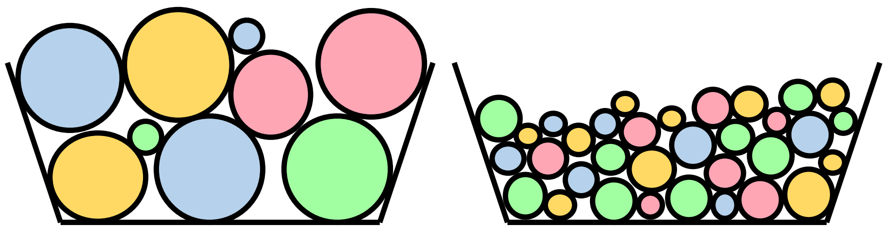


Figure 8.10: An Analogon for Request Scheduling

non-optimal assignment is highest for the Random policy and lowest for Least Used, explaining the performance differences among these policies.

8.7.2.2 The Impact on the Handling Speed

The results for the handling speed shown in figure 8.8 (right-hand side) reflect the results for the system utilization: since the per-PU load (see equation 8.6) becomes higher with a lower PU:PE ratio, a "bad" selection decision leads to queuing of requests. Clearly, this "request jam" significantly contributes to the handling speed loss.

While the utilization for higher request sizes s decreases, the handling speed increases: for example 100 requests of size s=1 are affected 100 times by the queuing delay, while one large request of s=100 is affected only once – for the same number of calculations to be processed. Clearly, this results in a significant handling speed gain.

Unlike the utilization curves for the different policies, the handling speed does not converge to a certain value for all policies if the PU:PE ratio r becomes high enough. Instead, a significant gap among the handling speeds of the Random, Round Robin and Least Used policies remains: Random and Round Robin frequently make "bad" assignments, leading to low handling speeds and therefore to longer request queues of the PUs. While the per-PU load (see equation 8.6) decreases with r, and therefore lowers the penalty of queuing delay (leading to an improved handling speed), the probability to make non-optimal selection decisions remains significantly higher for Random and Round Robin than for the Least Used policy.

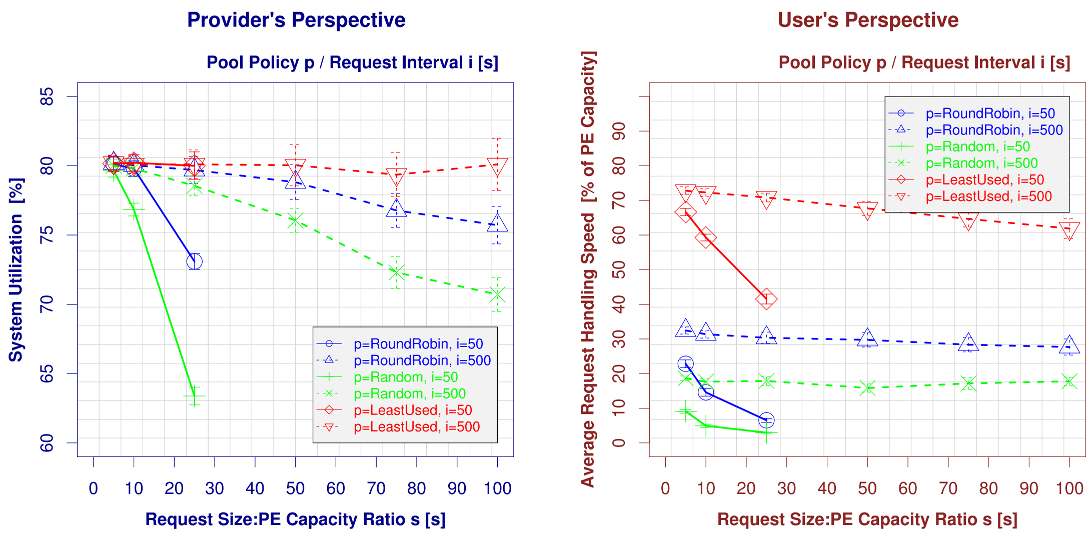


Figure 8.11: The Variation of Request Size and Request Interval

8.7.3 Variation of the Request Size and Request Interval

In the next simulation, the request size:PE capacity ratio s has been varied between 5 and 100 for values of the request interval i between 1s and 500s. Using equation 8.8 described in subsection 8.7.1, the PU:PE ratio has been calculated. Note, that the smaller the request size, the higher the PU:PE ratio (see also the middle part of figure 8.7). The results of the simulation are shown in figure 8.11.

Obviously, the utilization is highest for small values of the request size:PE capacity ratio s, since here the PU:PE ratio r is highest. As already observed in the analysis of the PU:PE ratio in subsection 8.7.2, the difference among the policies becomes small for high values of r, due to the parallelism of the request handling. Setting the request interval i to a higher value leads to a higher PU:PE ratio r. Therefore, the utilization decrement rate for rising s becomes smaller. Compare the results for i=50 (solid lines) with the curves for i=500 (dotted lines) at s=25: the PU:PE ratio at s=25 has already reached 1 for i=50 and therefore the gap among the three policies as explained in subsection 8.7.2 can be observed. On the other hand, the PU:PE ratio r is 16 for i=500; therefore, the difference among the policies is fairly small at only about 2%.

The handling speed results shown on the right-hand side of figure 8.11 reflect the observations for the utilization curves. r sinks with a rising request size s, i.e. the possibility to compensate "bad" selections by parallelism becomes smaller and the handling speed decreases. Obviously, this effect becomes smaller for higher request intervals i. Again, as explained in subsection 8.7.2, there is a huge difference among the three policies, due to their different selection decision qualities.

8.7.4 Variation of the Request Interval and PU:PE Ratio

The request interval is the third and last workload parameter. It has been evaluated in the range from 1s to 500s for PU:PE ratios from 1 to 10. The request size has been calculated using equation 8.9 described in subsection 8.7.1; the higher the request interval, the higher the request size (see also the right-hand side of figure 8.7). Figure 8.12 presents the results of the simulation.

With increasing request interval i, the request size:PE capacity ratio s also increases. As already explained in subsection 8.7.2 and subsection 8.7.3, this leads to a small decrement of the utilization,

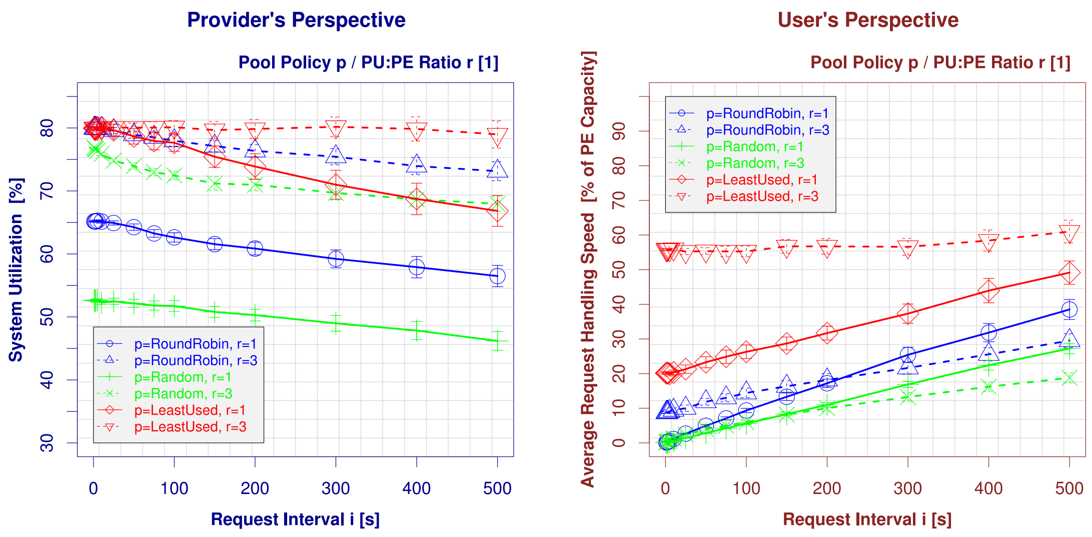


Figure 8.12: The Variation of Request Interval and PU:PE Ratio

due to the longer durability of selection decisions – in particular, it extends the impact of "bad" choices. Obviously, the most influencing factor is the PU:PE ratio r: while the policies' utilizations for r=1 differ in the range of about 20% to 25%, the variation already sinks to about 10% for r=3 for the reason of parallelism as explained in subsection 8.7.2.

The handling speed curves presented on the right-hand side of figure 8.12 mainly reflect the results for the system utilization. But while the utilization slightly decreases, the handling speed slowly increases. The reason is that s increases with i. Therefore, as explained in detail in subsection 8.7.2, the number of requests decreases and the processing time $d_{\text{Processing}}$ in equation 8.2 gains more importance over the queuing delay d_{Queuing}. This implies a handling speed improvement (see also equation 8.3). Note, that the handling speed curves for Round Robin and Random at r=1 exceed the curves at r=3 for sufficiently high values of i. Here, the gain by larger (and therefore fewer) requests at r=1 oversteps the gain by parallelism at r=3.

8.7.5 Summary

In summary, three important workload parameters have been identified:

- The PU:PE ratio,
- The request size and
- The request interval.

For a given target system utilization and PE capacity, any one of these parameters can be calculated if values for the other two are provided (using equation 8.5).

It has been shown that the PU:PE ratio is the most critical workload parameter. A small PU:PE ratio means a high per-PU load; appropriately scheduling the requests in this case is most difficult and "bad" PE selections lead to a reduced system utilization and handling speed. Varying the request size:PE capacity ratio together with the PU:PE ratio, it can be observed that the utilization for longer requests is worse (since it is even more difficult to achieve a "good" scheduling), but the handling speed increases (since a single but long request is affected by queuing delay only once, while n small requests are affected n times). The performance results for the two other workload parameter combinations can be deduced.

Comparing the three evaluated pool policies, it has been shown that usually the adaptive Least Used policy provides a better performance than the non-adaptive Round Robin. And Round Robin still achieves a better performance than Random. Since Round Robin is a very simple, non-adaptive policy, it seems to be a good choice to make it the default policy of RSerPool (i.e. make it the policy that must be supported by every RSerPool implementation). However, the Round Robin policy has some pitfalls that must be taken care of.

8.8 The Fallacies and Pitfalls of Round Robin Selection

This section presents effects of the Round Robin based policies that lead to a severe service degradation in case of unsuitable parameter choices.

8.8.1 Pitfalls of the Round Robin List Pointer

The first performance degradation effect, which has been published by us in Dreibholz, Rathgeb and Tüxen (2005), occurs if the Round Robin policy is used with an inappropriate setting of the $\mathrm{MaxIncrement}$ parameter. As explained in subsection 7.4.4, this parameter specifies – in case of the Round Robin policy – by how many steps the selection pointer into the Round Robin list is advanced when k PE identities of the pool are returned by the PR upon a handle resolution request. That is, if the PR returns $k \leq \mathrm{MaxHResItems}$ PE identities, the pointer is advanced by $l = \min(k, \mathrm{MaxIncrement})$ only.

Considering a Round Robin selection, the naïve assumption is clearly to use a configuration of $\mathrm{MaxIncrement}$=$\mathrm{MaxHResItems}$ and therefore to advance the list pointer by as many elements as actually returned. However, this can lead to a severe performance degradation, as shown in figure 8.13 for having varied $\mathrm{MaxIncrement}$ in pools of 10 and 24 PEs with a setting of $\mathrm{MaxHResItems}$=∞ (i.e. all PE identities of the pool are returned upon a handle resolution request). This example simulation has used a target system utilization of 60%, a request size of 10^7 calculations (negative exponential distribution) and a PU:PE ratio of 2.

The reason for the observed systematic performance drops is the behaviour of the PUs: they do not use their cache, i.e. their stale cache value has been set to 0s. Then, from the list returned by the PR only the first PE is actually used for processing a request. That is, if the number of PEs in the pool is an integer multiple of the number of entries in the list sent back by the PR, specific PEs will be systematically overloaded while the others will be hardly used. Assume, e.g. that the pool consists of the pool elements PE #1 to PE #6 and that the configured number of PE identities delivered by the PR in a handle resolution response – $\mathrm{MaxHResItems}$ – is 3. Now, the first handle resolution query to the PR returns the set {PE #1, PE #2, PE #3}, the following one will return {PE #4, PE #5, PE #6}. Then, new handle resolutions again start with {PE #1, PE #2, PE #3} and so on. In the worst case, the pool size is smaller than $\mathrm{MaxHResItems}$. Then, the reply is always the same (i.e. the complete pool). From the list received from the PR, the PU selects again one PE to establish the application connection to. According to Round Robin selection, this selected PE will always be the first one of the list. That is, only PE #1 and PE #4 will be used to handle requests while the four other PEs remain idle.

The example shown in figure 8.13 makes this degradation effect clear: for the pool of 10 PEs,

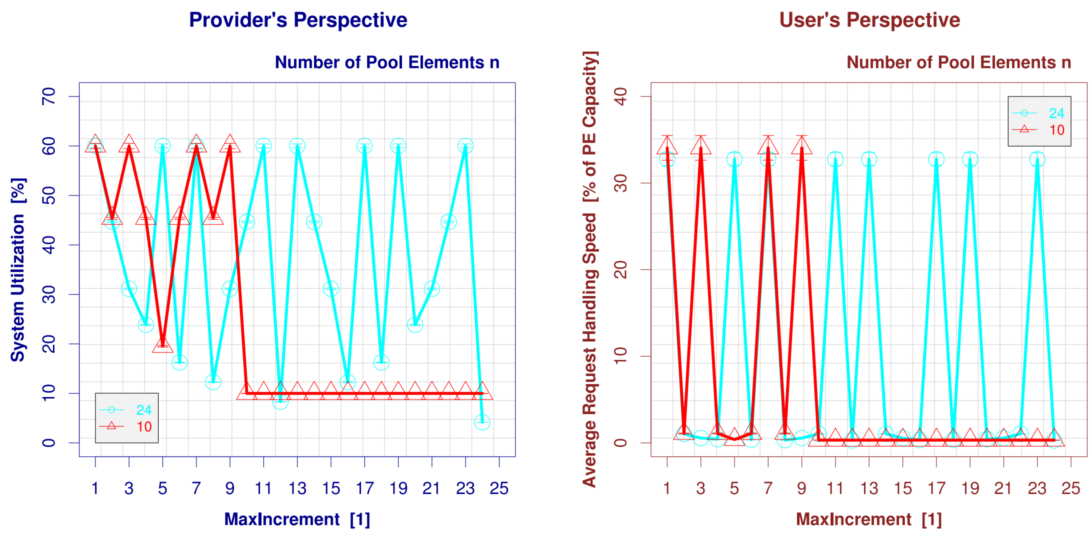


Figure 8.13: The Performance for a Fixed Step Size

the worst case is clearly MaxIncrement≥10. Note, that MaxIncrement settings above 10 cause no changes, since it is not possible to return more than 10 elements. A local performance minimum can be observed at MaxIncrement=5. In this case, only the first PE and sixth PE are used. Setting MaxIncrement to an even number results in using a set of 5 PEs only (the odd-numbered PEs). The same effect can be observed for the pool of 24 PEs if MaxIncrement is set to a divisor of 24. In particular, local performance minima can be found at a setting of 12 (only two PEs are used) as well as 6 and 18 (only four PEs will be used).

There are two possible approaches to solve the problem of the Round Robin degradation: to set MaxIncrement=1, i.e. always advancing the list pointer by one, or to randomize the number of steps. The first case is already shown in figure 8.13: here, it can be observed that a setting of 1 for MaxIncrement solves the problem. That is, the system utilization is at 60% as desired and the best possible handling speed is reached.

In the second approach, the pointer advancement is randomized by choosing a random step size $\sigma \in_R \{1, \ldots, \mathrm{MaxIncrement}\}$ for each handle resolution call. As shown by the results presented in figure 8.14, the utilization becomes independent of the MaxIncrement setting. However, the handling speed is negatively affected by settings larger than 1. The reason for this effect is that for $\sigma > 1$, at least one PE is skipped for the current Round Robin round. That is, the round finishes earlier and increases the probability that an already loaded PE of the list gets another request. Due to the increased load of this PE, all of its requests receive a decreased processing speed. This leads to a reduction of the average handling speed.

It is important to note that using the PU-side cache creates additional instances which independently perform Round Robin selections. That is, it is not possible to achieve a global Round Robin selection in this case. Nevertheless, setting MaxIncrement=1 or randomizing the number of steps ensures that each other selection component performs its own, local Round Robin selection using a different starting point.

At the time of the simulations, the RSerPool standards documents had only specified some policy names; details on how to implement and appropriately configure them had been missing. As result of the experience in implementing the policies (as part of the handlespace management approach

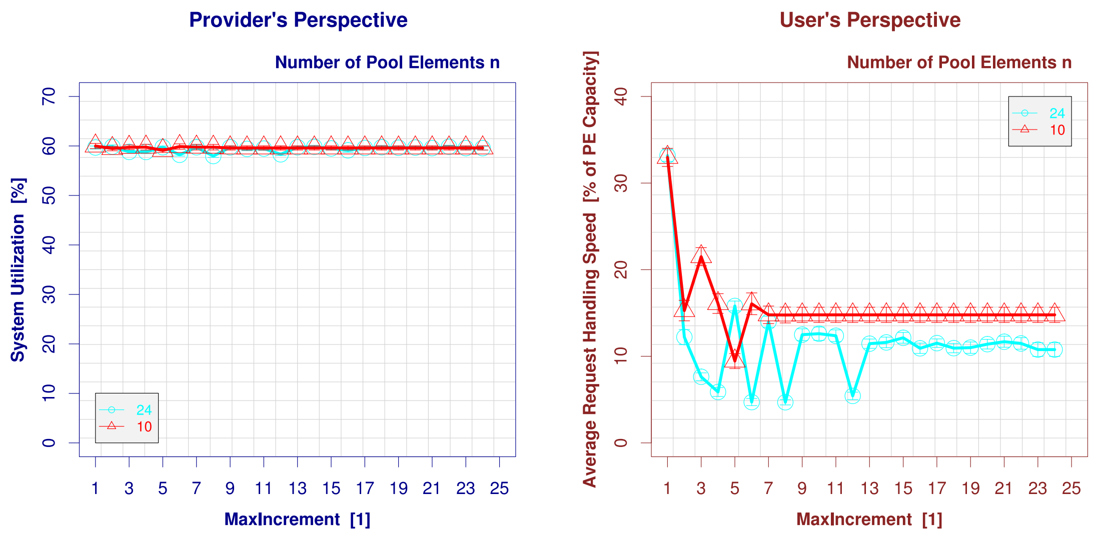


Figure 8.14: The Performance for a Randomized Step Size

described in chapter 4) and configuring them, the Internet Draft Tüxen and Dreibholz (2005) (Individual Submission) has been submitted to the IETF. This document has included an exact definition of the policies as well as of the pitfalls to avoid (in particular, the Round Robin problem described above). After a discussion at the 60th IETF Meeting 2004, this document has become the Working Group Draft Tüxen and Dreibholz (2006b) of the IETF RSerPool WG and is now official part of the RSerPool standards documents.

8.8.2 Pitfalls of the Weighted Round Robin Policy

The degradation effect of the Round Robin policy can also be transferred to the Weighted Round Robin policy. But this degradation effect is not the only problem of Weighted Round Robin; this policy induces additional problems which have been explained by us in Dreibholz and Rathgeb (2005e): considering the policy performance results for homogeneous server capacity scenarios as shown in section 8.7, Round Robin clearly provides a better performance than the Random policy. Unfortunately, it is a fallacy to assume that the Weighted Round Robin policy provides a better performance than Weighted Random in scenarios of heterogeneous server capacities (to be evaluated in section 8.12).

The first problem of Weighted Round Robin is its systematic selection. Consider a pool of 3 PEs with weights w_{PE_1}=2, w_{PE_2}=3 and w_{PE_3}=10. In this case, the selection order of a Weighted Round Robin round would be as follows:

$$\underbrace{\mathrm{PE}_1 \Rightarrow \mathrm{PE}_2 \Rightarrow \mathrm{PE}_3}_{\text{2 times}} \Rightarrow \underbrace{\mathrm{PE}_2 \Rightarrow \mathrm{PE}_3}_{\text{once}} \Rightarrow \underbrace{\mathrm{PE}_3}_{\text{7 times}}$$

Obviously, after PE_1 and PE_2 have been selected their given number of times, the Weighted Round Robin selection overloads PE_3. That is, if a PR selects PE_3 for 8 different PUs, 8 PUs simultaneously use PE_3 while other PEs may remain idle.

The second problem of the Weighted Round Robin policy is that the weights have to be integers: let PE_4 have 3.5 times the capacity of PE_5 and PE_6 1.75 times the capacity of PE_4. In this case, the

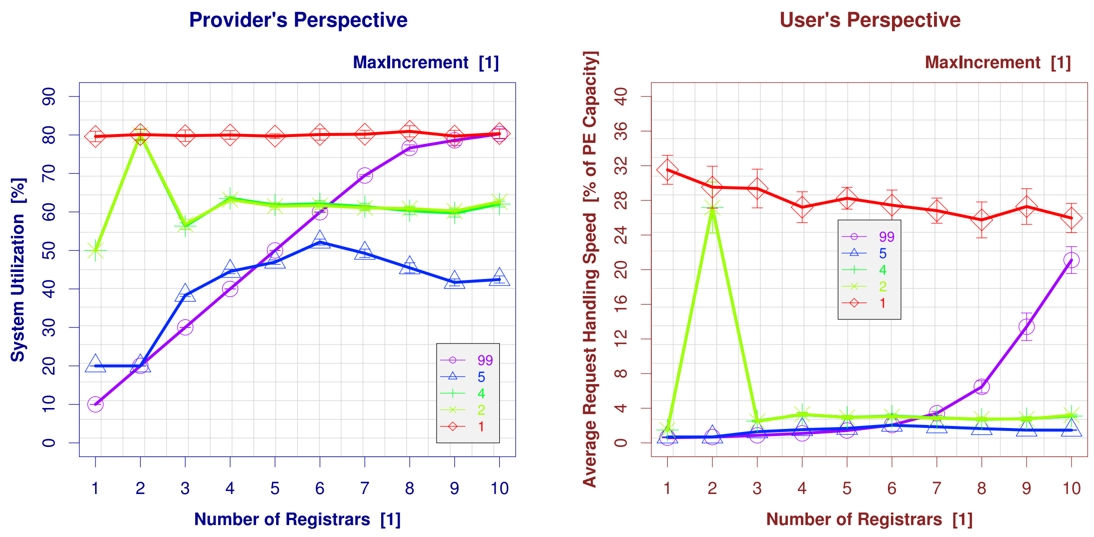


Figure 8.15: The Effects of Varying the Number of Registrars for the Round Robin Policy

resulting weights could be w_{PE_4}=4, w_{PE_5}=7 and w_{PE_6}=14. In the end of the selection round, PE_6 would be consecutively selected 8 times. Clearly, the higher the factor to extend the weights to integers, the more worse the selection problem described above.

Simulations for scenarios of heterogeneous server capacity distributions, which will be presented later in section 8.12, have shown that the performance of Weighted Round Robin is much worse than the results for Round Robin and even for Random (except for the homogeneous case where Weighted Round Robin is equal to Round Robin). Since Weighted Round Robin is completely useless in the described scenarios, curves for this policy have been omitted.

8.8.3 Summary

As a guideline on the usage of the Round Robin policy, the MaxIncrement parameter of the PR should be set to 1. This not only avoids the degradation problem, but also results in a performance gain for the handlespace management as shown in subsection 7.4.4 and subsection 7.5.4. Appropriately configured, the Round Robin policy usually provides a better performance than Random – as shown in section 8.7. Therefore, it is justified to make Round Robin the default policy of RSerPool, i.e. this is the only policy whose implementation is mandatorily required by the policies draft Tüxen and Dreibholz (2006b).

Although the Round Robin policy provides a better performance than Random, it is a pitfall to assume Weighted Round Robin to be performing better than Weighted Random. If considering to use Weighted Round Robin, extreme care has to be taken of the weight settings (which must be integers) and its systematic selection behaviour.

8.9 The Impact of the Number of Registrars

An important parameter for redundancy is the number of PRs. In section 8.6 it has been noted that this number does not significantly influence the system performance. This assertion has to be substantiated.

PRs synchronize their handlespace copies using the ENRP protocol. Neglecting the network delay (analysed later in section 8.10), it is obvious that the number of PRs does not affect the Least Used and Random policies, since they are "stateless" (see also subsection 3.11.1). However, Round Robin is "stateful", i.e. the selection of the next PE depends on the previous choices. As already shown in subsection 8.8.1, the system performance of the Round Robin policy highly depends on the setting of $\mathrm{MaxIncrement}$: some PEs may be systematically skipped, while other ones are overloaded.

In order to show the impact of the $\mathrm{MaxIncrement}$ setting when the number of PRs is increased, a simulation has been performed. The number of PRs has been varied between 1 and 10 for different settings of $\mathrm{MaxIncrement}$. The PU:PE ratio has been 25, the request size:PE capacity ratio has been 10 and the simulated real-time has been 60 minutes. PEs and PUs have been distributed equally among the PRs. The results of the simulation for the Round Robin policy are presented in figure 8.15.

Obviously, setting $\mathrm{MaxIncrement}$ to 1 is also useful to avoid the Round Robin problem (which has been explained in subsection 8.8.1) for scaling the number of PRs. In this case, the utilization remains unaffected, while only the handling speed slightly decreases with the number of PRs. The reason for this decrease is that the round robin selection on independent components differs from a global round robin selection.

On the other hand, using inappropriate $\mathrm{MaxIncrement}$ settings, the performance severely suffers: PEs are again systematically skipped, depending on the value of $\mathrm{MaxIncrement}$ and the number of PRs. An interesting observation is the peak for $\mathrm{MaxIncrement}$=2 and 2 PRs: each PR has its own Selection Set, ordered by PE sequence number (see subsubsection 4.4.2.2). That is, the 2 PRs use distinctly sorted round robin lists. So, PEs skipped by the first PR can be selected by the other one. This results in the good performance of this parameter set. However, this is not a common case, as shown for different settings of PR number and $\mathrm{MaxIncrement}$.

The performance results of Least Used and Random are not affected by the number of PRs. Therefore, plots for these policies have been omitted.

8.10 The Challenge of Network Delay

After having evaluated the performance of different pool policies in scenarios without network delay, latency is now added to the links as described in section 8.6 and the effects are evaluated.

8.10.1 General Effects of Network Delay

The first simulation with network delay enabled (based on our paper Dreibholz and Rathgeb (2005c)) is going to show the general effects introduced by the latency. Network latency has the highest impact on the system performance in the following two cases:

1. The PU:PE ratio r is small, so that the per-PU load (see equation 8.6) is large and

2. The ratio between delay and request size:PE capacity ratio s is high.

The first case reduces the system's capability to absorb the impact of "bad" selection decisions (see subsection 8.7.2); in the second case the delay decreases the handling speed (see equation 8.3). A simulation for a simulated real-time of 60 minutes has been performed in order to present these effects: the ratio between component RTT and request size:PE capacity ratio, $\frac{\text{RTT}}{\text{request size:PE capacity ratio}}$, has been varied between 0.0 to 1.0, for the PU:PE ratio r between 1 and 3. The request size:PE capacity ratio s has been 1. Figure 8.16 presents the results.

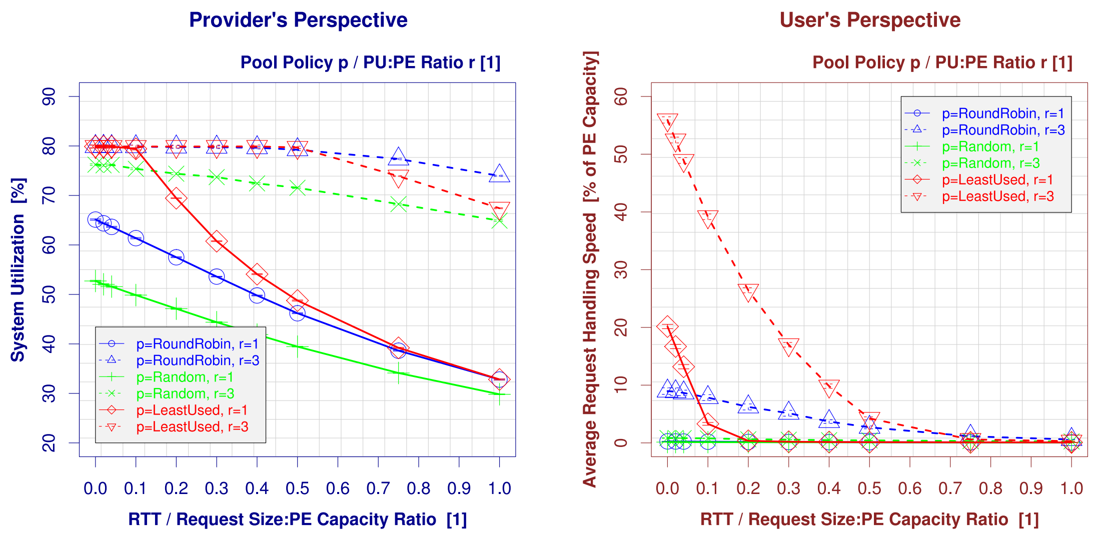


Figure 8.16: The General Effects of Network Delay

Obviously, the PU:PE ratio r has the main impact on the utilization for a rising delay: while the curves only slightly decrease for r=3, there is a steep descent for r=1. As explained in subsection 8.7.2, the lower the PU:PE ratio r, the higher the capacity share a single PU expects from its PE. That is, at r=1 a PE is expected to exclusively provide 80% of its capacity to request processing. Only 20% remain to absorb both, the delay of communications and the speed degradation due to simultaneously handled requests.

An important observation can be made for Least Used: while the utilization of Least Used converges to the curve of Round Robin for r=1, it converges to the curve of Random for higher values of r. The reason is clear: while for r=1 Round Robin provides the best chance to get an unloaded PE, the probability of Round Robin to make a good choice decreases with rising parallelism (see subsection 8.7.2).

The handling speed results – presented on the right-hand side of figure 8.16 – reflect the observations for the utilization. Clearly, the network delay has a significant impact on the handling speed for small settings of s (here: s=1), since it affects the handling speed (see equations 8.2 and equation 8.3) by the communication of PU and PR (handle resolution) and PU and PE (request handling).

In summary, the impact of network delay on the system utilization is small – except for the critical parameter range of a small PU:PE ratio. However, there is a significant effect on the handling speed in case of request sizes having a processing time (see equation 8.4) of about or less than the latency. For this parameter range, counter-measures seem to be useful.

8.10.2 Creating Distance-Aware Policies

In general, if the requests are short, it is useful to keep the network latency between PU and PE as small as possible. But critical applications may require their pools to be distributed globally, in order to continue their service even in case of localized disasters (e.g. earthquakes or tsunamis). An example is provided in figure 8.17: if some PEs in Asia become unavailable (here: 2 PEs), it is still possible to compensate the reduced capacity by PEs in the other regions. Furthermore, it is possible to dynamically increase the capacity in the other regions (here: one PE in America and one PE in Europe).

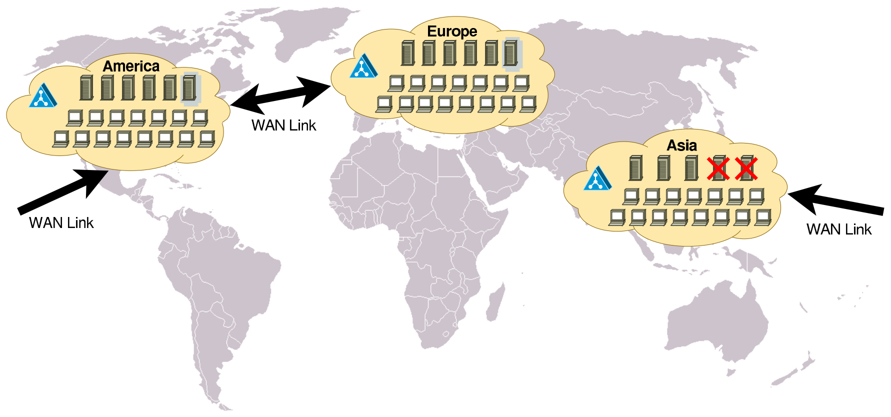


Figure 8.17: A Scenario of Globally Distributed RSerPool Components

Clearly, it is necessary to define policies which incorporate a distance measure. But in order to define such policies, it is obviously necessary to quantify *Distance* first.

8.10.2.1 How to Quantify Distance?

For the description of the distance, two approaches have been considered: the usage of geographical position information and delay measurements. Both approaches will be discussed in the following.

Geographical Distance Of course, the most obvious approach to describe a distance is to obtain the geographical coordinates of the components and calculate the distances among them. That is, given two components A and B with their latitudes δ_A and δ_B as well as their longitudes λ_A and λ_B, it is simply possible to calculate the orthodrome L (i.e. the length of the shortest path on the earth's surface) as follows:

$$\begin{aligned}\zeta &= \arccos(\sin\delta_A \sin\delta_B + \cos\delta_A \cos\delta_B \cos(\lambda_B - \lambda_A)) \\ L &= \frac{\zeta}{360^\circ} * 40,000\text{km}\end{aligned}$$

While such a distance calculation is rather simple, this approach has some decisive disadvantages: at first, a small geographical distance does not imply a small network latency. For example, a satellite link delay between two systems in French Polynesia may have a higher latency than a wired, intercontinental connection between Europe and North America – although the geographical distance is by an order of magnitude smaller. For more details on the relationship between distance and latency in the Internet, see also Subramanian et al. (2002).

A further disadvantage of using geographical coordinates and distances as a metric for server selection is the need for a GPS (Global Positioning System) device being built into every component, in order to obtain its position. Otherwise, the system would drop the support for mobility (which might be acceptable) and also introduce a possible point of misconfiguration (which may be critical). Consider e.g. servers claiming to be nearby located to a set of far-away clients – the result would possibly be a severe performance degradation.

Measured Delay The appropriate way to describe distances for server selection would be clearly based on up-to-date measurements of the network delay. This approach has the disadvantage that these measurements have to be performed continuously. However, the SCTP protocol (see subsection 2.4.3) already calculates a smoothed RTT for its paths. Furthermore, it is possible to query the RTT via the standard SCTP API (using the SCTP_STATUS option, see Stewart, Xie, Yarroll, Wood, Poon and Tüxen (2006)).

By default, the calculated RTT has a small restriction: a SCTP endpoint waits up to 200ms before acknowledging a packet, in order to piggyback the acknowledgement chunk with payload data. In this case, the RTT would include this latency. Using the option SCTP_DELAYED_ACK_TIME, the maximum delay before acknowledging a packet can be set to 0ms (i.e. "acknowledge as soon as possible"). After that, the RTT approximately consists of the network latency only. Using the RTT, the end-to-end delay between two associated components is approximately $\frac{\mathrm{RTT}}{2}$.

In real networks, there may be negligible delay differences: for example, the delay between a PU and PE #1 is 5ms and the latency between the PU and PE #2 is 6ms. From the service user's perspective, such minor delay differences are negligible and furthermore unavoidable in Internet scenarios. Therefore, the distance parameter between two components A and B can be defined as follows:

$$\mathrm{Distance}_{A\leftrightarrow B} = \mathrm{DistanceStep} * \text{round}\left(\frac{\frac{\mathrm{RTT}}{2}}{\mathrm{DistanceStep}}\right) \tag{8.10}$$

That is, the distance parameter rounds the measured delay to the nearest integer multiple of the constant $\mathrm{DistanceStep}$.

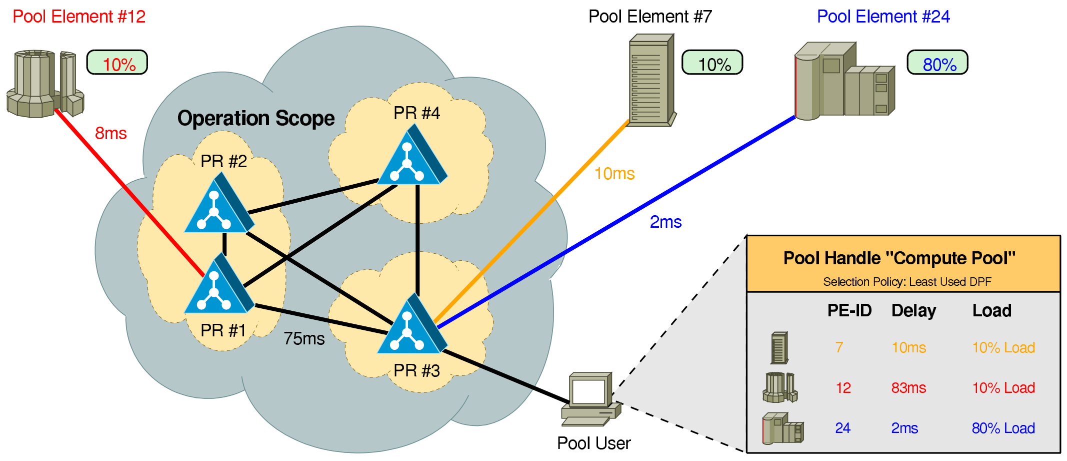
8.10.2.2 An Environment for Distance-Aware Policies

In order to define a distance-aware policy, it is necessary to define a basic rule: PEs and PUs choose "nearby" PRs. Since the operation scope of RSerPool is restricted to a single organization, this condition can be met easily by appropriately locating PRs. A PR-H can measure the delay of the ASAP associations to each of its PEs. As part of its ENRP updates to other PRs, it can report this measured delay together with the PE information. A non-PR-H receiving such an update simply adds the delay of the ENRP association with the PR-H to the PE's reported delay. Now, each PR can approximate the distance to every PE in the operation scope using equation 8.10. Note, that delay changes are propagated to all PRs upon PE re-registrations, i.e. the delay information (and the approximated distance) dynamically adapts to the state of the network.

An example for the described distance propagation environment is provided in figure 8.18: PE #12 is connected to its PR #1 with a delay of 8ms. PR #1 is connected to PR #3 over ENRP with a delay of 75ms. For PR #3, the delay to PR #12 is 8ms+75ms=83ms. PR #3 is the PR-H of PE #7 and PE #24, i.e. they appear in PR #3's handlespace with their connection delays of 10ms and 2ms. Since the PU asks its nearby PR #3 for handle resolution, it will receive the view of PR #3. Using equation 8.10 and a setting of $\mathrm{DistanceStep}$=50ms, the distance of PE #7 and PE #24 is 0ms, while it is 100ms for PE #24.

8.10.2.3 The Definition of Distance-Aware Policies

The policies suitable to be extended with distance information are Least Used and Weighted Random. Trying to adapt Weighted Round Robin would lead to the problem of non-integer weights, as described in subsection 8.8.2.



Pool Handle "Compute Pool"
Selection Policy: Least Used DPF

PE-ID	Delay	Load
7	10ms	10% Load
12	83ms	10% Load
24	2ms	80% Load

Figure 8.18: An Example for the Distance-Aware Policy Environment

Least Used with Distance Penalty Factor The Least Used policy can be adapted as follows: instead of only taking the load value into account for server selection, the new load value Load^* is computed by increasing the PE's reported value Load by a distance-dependent *Distance Penalty Factor* (DPF) as follows:

$$\mathrm{Load}^* = \mathrm{Load} + \underbrace{\mathrm{Distance} * \mathrm{LoadDPF}}_{\text{Distance Penalty Factor}}.$$

The constant $\mathrm{LoadDPF}$ describes the load units per millisecond by which a PE's reported load value is incremented for every millisecond of the measured network delay. That is, the unit for $\mathrm{LoadDPF}$ is ms^{-1}. Due to the DPF parameter, the new policy is denoted as Least Used with DPF (LU-DPF). It simply selects the PE with the lowest value of Load^*. If there are multiple lowest-valued PEs, round robin selection is applied among them.

Weighted Random with Distance Penalty Factor The Weighted Random with DPF (WRAND-DPF) policy can be defined by calculating an adapted weight constant Weight^* from the actual weight value Weight – again using a distance penalty factor – as follows:

$$\mathrm{Weight}^* = \max\{1, \mathrm{Weight} * (1 - \underbrace{\mathrm{Distance} * \mathrm{WeightDPF}}_{\text{Distance Penalty Factor}})\}.$$

Clearly, the constant $\mathrm{WeightDPF}$ defines the fraction by which the actual weight is decreased for each millisecond of the measured network delay. The unit for $\mathrm{WeightDPF}$ is therefore ms^{-1}. Since it does not make sense[1] to have a weight of less than one, Weight^* is defined as the maximum value of 1 and the adapted weight. The actual PE selection is analogous to Weighted Random, using Weight^* as weight.

Note, that the sorting of the PEs in the Selection Set is still per-PR rather than per-PU for both new DPF-based policies, as it has been realized for the other policies presented in subsection 4.4.2. That is, the handlespace and policy management approach as described in section 4.4 – based on keeping sorted sets of PEs – can still be used. This allows a PR to efficiently store the handlespace data and perform operations on it.

[1]A weight of 0 would result in a probability of 0% for a PE to be selected. Clearly, if the PE is the only one in the pool, this would cause a problem.

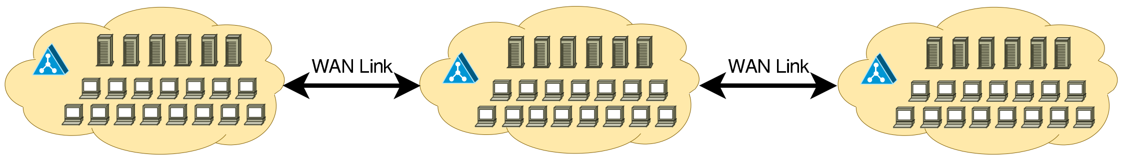


Figure 8.19: The Distance-Aware Policies Proof of Concept Simulation Setup

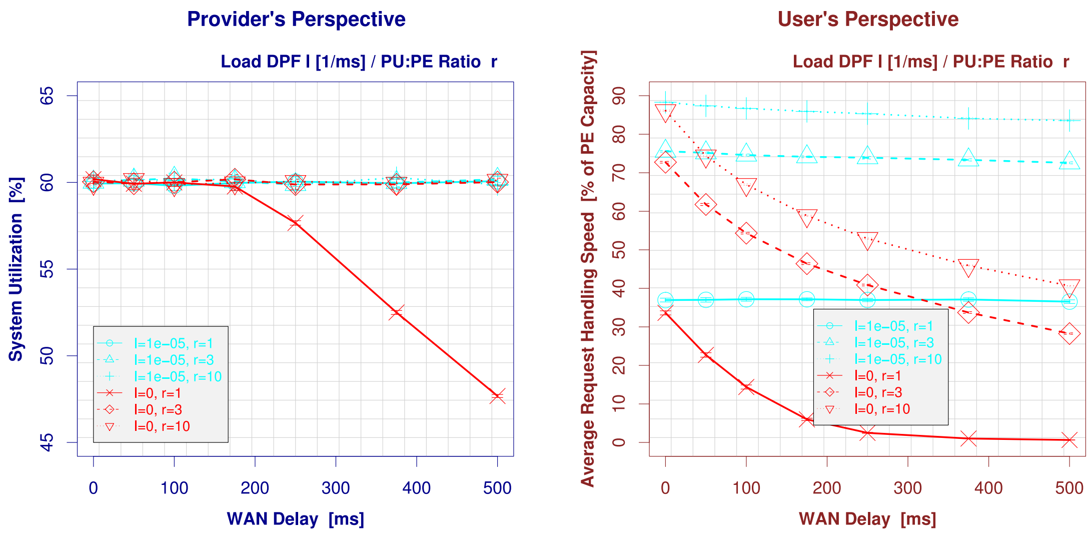


Figure 8.20: A Proof of Concept for the Least Used with DPF Policy

8.10.3 A Proof of Concept

In order to provide a proof of concept for the new distance-sensitive policies, simulations have been performed for a scenario having consisted of 3 LANs, as shown in figure 8.19. Each LAN has included 1 PR and 10 PEs. While the inter-component LAN delay has used a fixed setting of 10ms, the WAN delay has been varied from 0ms to 500ms (these settings are based on PlanetLab measurements and will be motivated in detail in subsubsection 8.10.5.2). The PUs have used a request size:PE capacity ratio of s=1 (i.e. if processed exclusively, the processing time of an average request is 1s), the PU:PE ratio r has been varied between 1 and 10 for a target system utilization of 60%. DistanceStep has been 1ms, the simulated real-time has been 15 minutes.

The performance results of the Least Used with DPF policy for the LoadDPF settings 0 (i.e. the policy is similar to plain Least Used) and $1*10^{-5}$ (this parameter will be evaluated in detail in subsection 8.10.4) are shown in figure 8.20. Clearly, for LoadDPF=0, the utilization is negatively affected by the WAN delay if the PU:PE ratio r is small (1 in this case). This corresponds to the results observed and explained in subsection 8.10.1. However, for LoadDPF=$1*10^{-5}$, the utilization is not affected by the delay any more. A similar effect can also be observed for the request handling speed: while for LoadDPF=0 the handling speed severely suffers due to the WAN delay (as expected from the results in subsection 8.10.1), the network latency only very slightly decreases the handling speed for LoadDPF=$1*10^{-5}$. The small descent is caused by the selection of remote PEs if all local PEs have a higher load.

Figure 8.21 presents the results of the Weighted Random with DPF policy for having varied

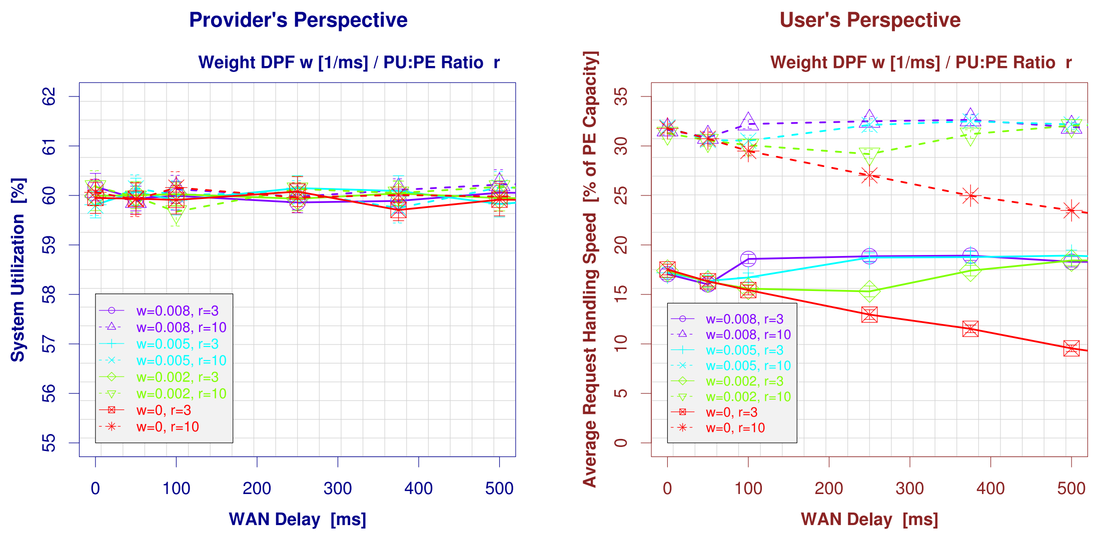


Figure 8.21: A Proof of Concept for the Weighted Random with DPF Policy

WeightDPF between 0 (i.e. the policy behaves like regular Weighted Random) and $8*10^{-3}$ (this parameter will be evaluated in detail in subsection 8.10.4). Note, that a curve for a PU:PE ratio of 1 has been omitted, since it is not useful for this policy. Clearly, the results for the utilization are insignificant if the PU:PE ratio is high enough. However, the handling speed is decreasing if the PE selection does not take care of the network latency (i.e. WeightDPF=0). The higher the WeightDPF, the smaller the influence of the delay on the handling speed. That is, a sufficiently large WeightDPF setting is required in order to achieve a significant effect.

In summary, the proof-of-concept simulations have shown that the new policies – Least Used with DPF and Weighted Random with DPF – are useful. But how to appropriately configure their parameters LoadDPF and WeightDPF?

8.10.4 The Appropriate Choice of Parameters

Since the use cases of the new policies are globally distributed pools which have to cope with localized disasters, the configurations of the parameters LoadDPF (for Least Used with DPF) and WeightDPF (for Weighted Random with DPF) have been evaluated with respect to situations where PEs of a region become unavailable and remote PEs should be used instead. Therefore, the DPF parameters have been varied in a scenario of 3 LANs (again, as illustrated in figure 8.19), with each LAN having contained 1 PR and 12 PEs. In order to simulate a localized disaster, the number of PEs in the third LAN has been varied between 100% (i.e. all 12 PEs) and 25% (only 3 PEs). To compensate the capacity loss in the third LAN, additional PEs have been distributed equally between the other 2 LANs. That is, the overall capacity of the pool has always remained the same. In the scenario, an inter-component LAN delay of 10ms and a WAN delay of 150ms[2] have been used (to be justified in subsubsection 8.10.5.2). All other parameters (except for the PU:PE ratio r, see below) have been set as for the proof-of-concept simulation described in subsection 8.10.3.

The performance results for having varied the LoadDPF parameter of the Least Used with DPF policy for a PU:PE ratio of 3 are shown in figure 8.22. While there is no significant impact on the

[2]This is a realistic delay for an inter-continental WAN connection; see subsubsection 8.10.5.2.

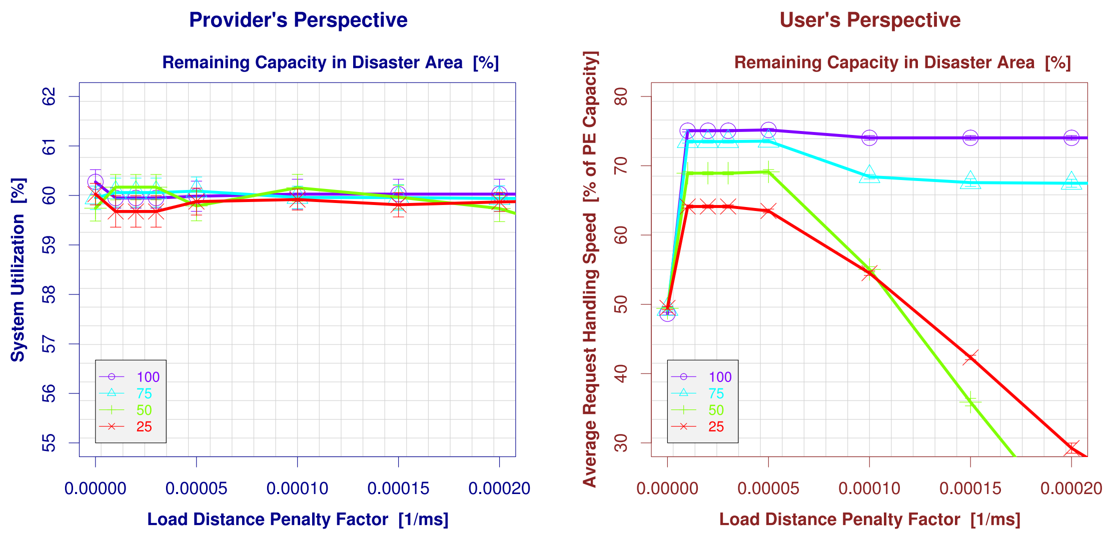


Figure 8.22: Finding a Reasonable Load DPF Setting for the Least Used with DPF Policy

utilization (since the parameter set is not critical), the handling speed is increased significantly – even for a very small LoadDPF setting (e.g. $1*10^{-5}$). If there is a choice among multiple least-loaded PEs, there will be a preference for a nearby one. As expected, a higher value of the LoadDPF setting has no impact if all PEs in the third LAN are available. In this case, there is sufficient capacity within the LANs to handle the requests locally. That is, the server selection mainly behaves as for three separate pools.

However, decreasing the number of PEs in the third LAN and increasing the number in the other LANs, the scenario becomes heterogeneous. For setting the LoadDPF parameter too high, the handling speed decreases and – for a sufficiently high setting (here: $15*10^{-5}$ for $\leq$50% PEs in the third LAN) – the handling speed is even outperformed by plain Least Used (i.e. a LoadDPF of 0). The higher the setting of LoadDPF, the less likely the usage of remote PEs – even if this would be desirable in the case of a localized disaster. That is, the resulting guideline for setting the LoadDPF parameter is rather simple: set it to a value slightly above 0 – e.g. $1*10^{-5}$. This setting gives the server selection a preference for the nearest PE – in case of having multiple least-loaded PEs. Furthermore, it also provides an improved performance in scenarios of localized disasters – by enabling the selection of remote PEs if necessary. That is, Least Used with DPF can achieve a significant performance benefit over plain Least Used.

Figure 8.23 presents the performance results for having varied WeightDPF between 0 and $8*10^{-3}$ for the Weighted Random with DPF policy and a PU:PE ratio of 10 (otherwise, the scenario would become critical too quickly to illustratively show the desired effects). The other parameters have remained as described for the other policy. As expected, WeightDPF has to be sufficiently large to show an improvement: for WeightDPF$\geq 3*10^{-3}$, the handling speed starts rising for 75% and 100% of the PEs in the third LAN. However, for a smaller number of PEs, the handling speed decreases quickly. In this case, the probability to select a remote PE becomes too small and a local "request jam" occurs in the third LAN. This effect can be verified by the reduced system utilization. That is, while Weighted Random with DPF provides a performance improvement in sufficiently homogeneous scenarios, its ability to cope with localized disasters is limited.

In summary, the simulative results for the Least Used with DPF policy are quite promising. But is

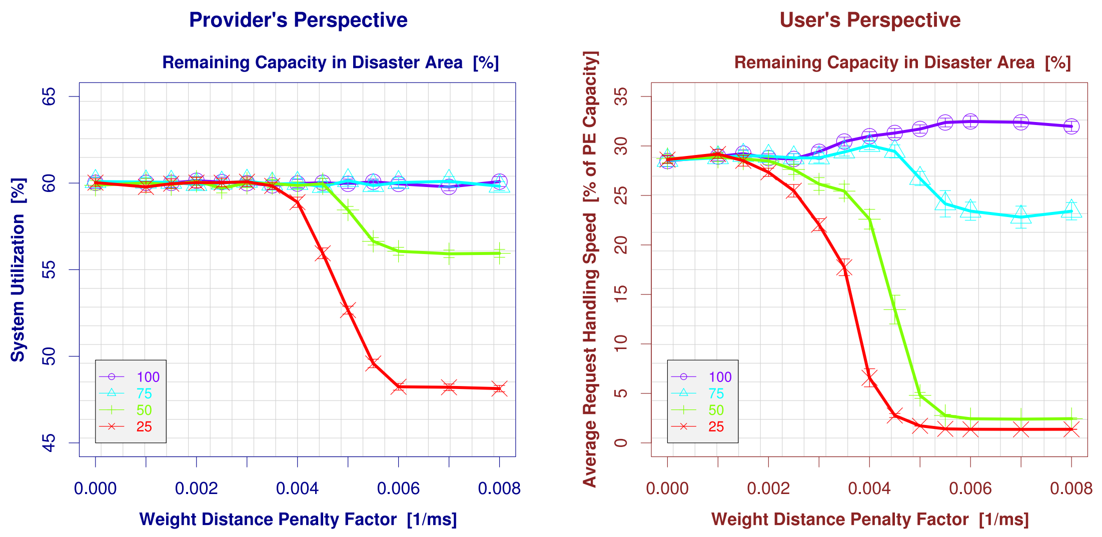


Figure 8.23: Finding a Reasonable Weight DPF Setting for the Weighted Random with DPF Policy

it also useful in reality? To answer this question, a performance evaluation using the RSPLIB prototype in a PLANETLAB scenario has been performed.

8.10.5 Experimental Validation using the PLANETLAB

The PLANETLAB is actually a set of globally distributed, Linux-based research hosts in the Internet. Its intention is to allow the test and validation of network software in real-world setups. A detailed introduction to the PLANETLAB can be found in Peterson and Roscoe (2006), Peterson et al. (2005), Peterson (2004), the basic ideas and concepts are explained in Chun et al. (2003), Roscoe (2002), Peterson and Roscoe (2002), Peterson et al. (2002). Therefore, the following subsubsection only gives a brief introduction to the PLANETLAB environment used for the following measurements.

8.10.5.1 The PLANETLAB Environment

Having access to a set of PLANETLAB nodes (which is called *Slice*), it is possible to log into these hosts using the Secure Shell (SSH) software. SSH provides interactive and script-based execution of remote programs, as well as the transfer of files between hosts. Based on SSH, shell scripts have been developed for the following tasks:

- `create-binaries` copies an archive of the RSPLIB prototype implementation to a PLANETLAB node, compiles it and sends an archive of the created executables back.
- `install-binaries` simultaneously installs the executables on a given set of PLANETLAB nodes.
- `create-hosts-files` tests a given set of PLANETLAB nodes for usability (to be explained below).
- A set of scenario scripts can set up certain configurations of PRs, PEs and PUs. Furthermore, these scripts collect the created statistics and use the tool `createsummary` (which has been developed for the simulation model, see subsection 6.3.4) to create input files for GNU R.

Again, GNU R (see subsection 6.3.2) has been used for the statistical post-processing and plotting of the results.

While it is possible to assign all of the about 650 PLANETLAB nodes to a slice, the number of nodes which are really accessible at a given time is about 30%. Since the resources of a node (CPU power, memory and harddisk space) are shared among all slices, there are usually some nodes which are highly overloaded (e.g. they are extremely slow or spend most of their time swapping memory to and from harddisk). In order to find suitable nodes for an experiment, the following tests are performed by the `create-hosts-files` script:

1. Logging into the node (i.e. it is verified that the node is accessible),
2. Invoking `ping` to test the network connection to http://www.planet-lab.org (i.e. it includes a DNS resolution of the hostname) with a timeout of 5s,
3. Creating a file consisting of $16*10^6$ bytes of random numbers (i.e. it is verified that the file system is writeable) and
4. Computing the SHA-1[3] hash of the test file (i.e. the computation power of the node is tested).

If these four operations are successfully completed within 36s, the node is considered to be usable (for comparison: the Athlon 1.3 GHz PC, used for the handlespace management performance evaluation in chapter 7, performs these tasks in about 9s).

Since the PLANETLAB nodes do not support SCTP directly (i.e. by their Linux kernel), our userland implementation SCTPLIB (see section 5.3) has been used. Due to technical limitations of the PLANETLAB software, all SCTP traffic has to be tunnelled via UDP. This SCTP over UDP encapsulation is defined in Tüxen and Stewart (2005). Except for a slight increase in overhead (additional 8 bytes for the UDP header per packet), it does not impose any functional restriction.

In order to perform PLANETLAB-based performance evaluations of the new policy, the RSPLIB prototype (see chapter 5) has been used to realize PE and PU application implementations. These implementations provide and use the same compute service as described for the simulation model in section 8.3.

Using the application's PU and PE implementations as well as the PR of the RSPLIB prototype package, systematic tests in different PLANETLAB scenarios have been performed in order to validate the correctness of the programs. While the RSPLIB prototype as well as the underlying SCTPLIB and SOCKETAPI implementations have already shown to be working quite reliably in long-term lab tests using the demonstration system (see also section 5.8), an Internet scenario is different. Here, unforeseeable packet losses and transmission delays may occur. Therefore, a significant fraction of the time required to realize the measurement setup has been to also debug the SCTPLIB and SOCKETAPI implementations. For example, there have been bugs like a forgotten unscheduling of timers for already removed associations or wrong reactions to unexpected packet retransmissions – which have both caused segmentation faults. Such bugs – which occur quite rarely and are therefore difficult to reproduce – have been identified by long-term PLANETLAB tests with more than 100 components over several days, collecting the core dumps created on segmentation faults and analysing the core dumps and log files. After bug-fixing, these tests have been repeated. The result of these time-consuming debugging sessions – which have taken weeks – has been a significantly improved robustness of all three components: the RSPLIB prototype as well as the SCTPLIB and SOCKETAPI packages.

[3]Secure Hash Algorithm 1 (192 bits), see Eastlake and Jones (2001).

Interval	Network State	LU-DPF	LU	Improvement
1m - 14m	Normal Operation I	2.22s $\pm$ 0.07	2.55s $\pm$ 0.06	12.9%
16m - 29m	Failure in Asia	2.44s $\pm$ 0.07	2.55s $\pm$ 0.06	4.3%
31m - 45m	Added Backup Capacity	2.32s $\pm$ 0.06	2.46s $\pm$ 0.06	5.7%
46m - 49m	Failure Resolved	2.23s $\pm$ 0.13	2.46s $\pm$ 0.12	9.3%
51m - 64m	Normal Operation II	2.21s $\pm$ 0.06	2.69s $\pm$ 0.07	17.8%

Table 8.2: The Average Request Handling Time Results of a First Trial

8.10.5.2 The Measurement Setup

The measurement setup for the following performance evaluations has consisted of components distributed into three regions (see also figure 8.17): Europe (mainly Germany), America (U.S.A., mainly West Coast) and Asia (mainly Japan). Each region has contained one PR, which has been used by the 5 PEs and 15 PUs of the region. As for the simulation, the PEs have used a capacity of 10^6 calculations/s.

Tests using `ping` and `traceroute` have shown latencies between 5ms to 15ms within the regions; the inter-region delay varies between about 75ms to 150ms between Europe and America and America and Asia, as well as about 250ms to 350ms between Europe and Asia (routed via the U.S.A.). The delays between any two endpoints have not shown a significant variation. That is, it can be assumed that there has been sufficient bandwidth available. This is also realistic for RSerPool scenarios, since all components belong to a single operation scope (e.g. a company) and QoS mechanisms can therefore be applied easily (e.g. WAN connections via DiffServ-based VPN links using an appropriate SLA). Based on the delay experiments, $\mathrm{DistanceStep}$ has been set to 75ms.

8.10.5.3 The Results of a First Trial

In order to validate the setup functionality and to show the essential effects of a PLANETLAB measurement, the results of a first trial are presented in this subsubsection. A long-term experiment will be analysed in the following subsubsection 8.10.5.4.

The measurements of the first trial have had a runtime of 65 minutes, with the following actions: at t_1=15min, 2 of the 5 Asian PEs have been turned off; at t_2=30min, two additional PEs have been turned on – one in America, the other one in Europe. At t_3=45min, the failure in Asia has been repaired. Both PEs are again added to the pool, increasing its total capacity. The two additional PEs in Europe and America have been turned off at t_4=50min. The application has used the following workload parameters: an average request size of 10^6 calculations and an average request interval of 7.5s (both using negative exponential distribution).

In order to make the performance results of the Least Used policy and the Least Used with DPF policy (using a $\mathrm{LoadDPF}$ setting of $1*10^{-5}$) comparable, the runs for both policies have been performed simultaneously. That is, two pools have been set up – one for Least Used, the other one for Least Used with DPF. On each of the PE hosts, two PE instances have been started: one having registered into the Least Used pool, the other one having registered into the Least Used with DPF pool. Analogously, each PU host has run two PU instances: one having used the Least Used pool, the other one having used the Least Used with DPF pool. Due to the simultaneous execution, it has been ensured that both measurements have been affected equally by temporal variations of the Internet's QoS conditions. Therefore, the results of both policies are comparable.

Interval	Network State	LU-DPF	LU	Improvement
1m - 14m	Normal Operation I	2.17s ± 0.05	2.63s ± 0.05	17.5%
16m - 29m	Failure in Asia	2.55s ± 0.02	2.78s ± 0.02	8.3%
31m - 45m	Added Backup Capacity	2.54s ± 0.02	2.71s ± 0.02	6.3%
46m - 49m	Failure Resolved	2.35s ± 0.06	2.55s ± 0.03	7.8%
51m - 64m	Normal Operation II	2.08s ± 0.02	2.47s ± 0.02	15.8%

Table 8.3: The Average Request Handling Time Results of 22 Measurements

The resulting average request handling times and their 95% confidence intervals are presented in table 8.2. Note, that the table shows the average over intervals beginning one minute after and ending one minute before a system condition change, since the latency to log into a PLANETLAB node to start or stop a component may take up to about 30s. Furthermore, small deviations of the hosts' clocks may be possible. Comparing the results for Least Used and Least Used with DPF, the Least Used with DPF policy provides a significant handling speed gain: between 12.9% and 17.8% for the two phases of normal operation and still around 5% during the failure in Asia. When the two PEs in Asia come back again, the gain rises to 9.3%.

To further illustrate the behaviour of the two policies and the effects of temporal network QoS changes in the Internet, figure 8.24 shows the handling speed of each handled request for the Least Used with DPF policy; figure 8.25 presents the same for Least Used. Each request is represented by a line starting at the request's queuing time and ending at its completion time; for each PU, a different colour/shade is used. The position of a request on the y-axis shows its handling speed; the average handling speed (presented by the thick line) has been computed as

$$\text{AverageHandlingSpeed} = \frac{\sum_i \text{RequestSize}_i}{\sum_i d_{\text{Handling}_i}}$$

for each request i being completed in the corresponding interval. Comparing the plots for Least Used with DPF and plain Least Used, it is clearly observable that there are significantly more requests in the range of higher handling speeds. Furthermore, the varying network QoS conditions in the Internet are observable by small dents and peaks (e.g. 23min to 26min or 43min to 45min) – they appear for both policies in the same time intervals.

8.10.5.4 The Results of a Long-Term Measurement

In order to achieve a more accurate analysis, the measurement run described in subsubsection 8.10.5.3 has been repeated 22 times, covering a total experiment runtime of about 35 hours. The resulting average values of these measurements are presented in table 8.3. These results have also been published in Dreibholz and Rathgeb (2007).

Comparing the results for Least Used and Least Used with DPF, the Least Used with DPF policy provides a significant handling speed gain: between 17.5% and 15.8% for the two phases of normal operation and still around 8% during the failure and its resolution in Asia. That is, the expectations from the first measurement in subsubsection 8.10.5.3 are met.

It is important to note that the performance in the "failure resolved" state is lower than for the "normal operations" states, although there are additional PEs in America and Europe: the over-capacity in these regions attracts the assignment of requests from Asia. This effect – the assignment of requests

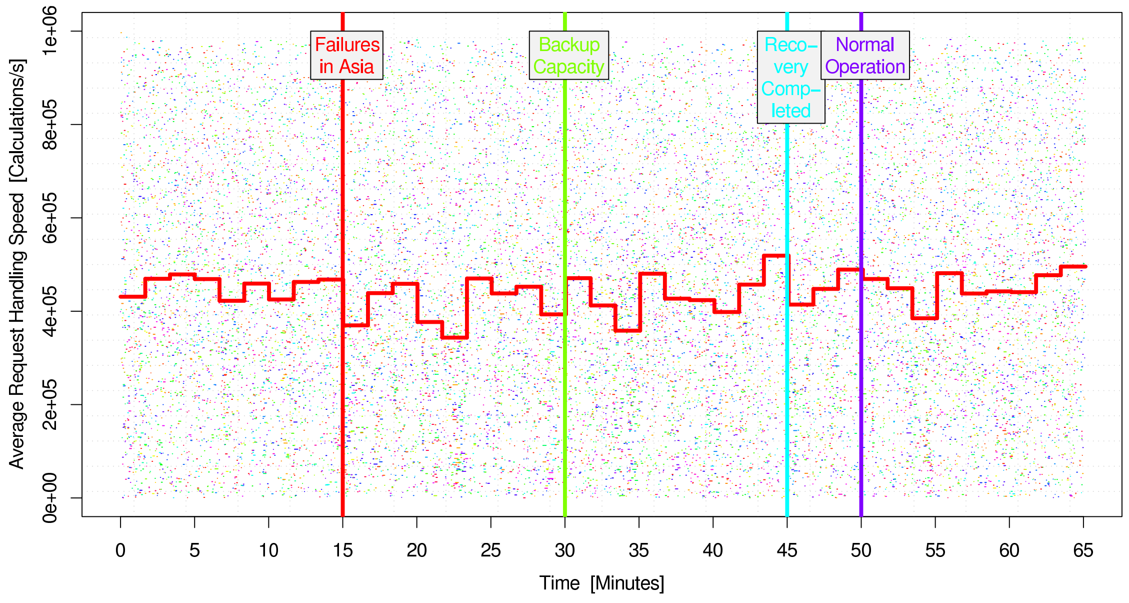


Figure 8.24: Experimental Results of the Least Used with DPF Policy

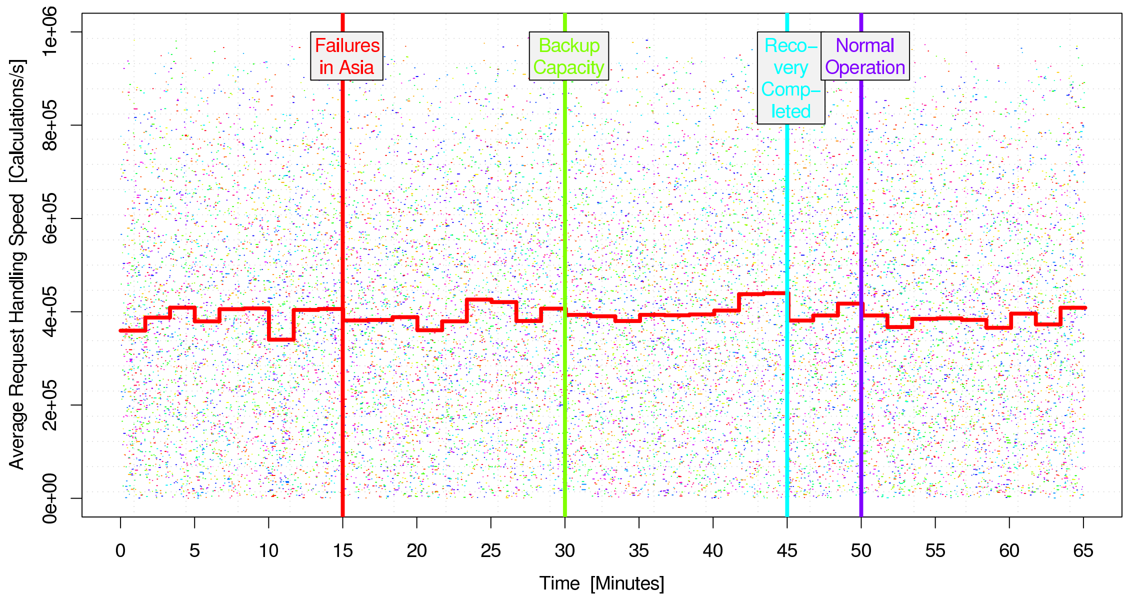


Figure 8.25: Experimental Results of the Least Used Policy

to slightly-loaded servers – is a property of all load-based policies; trying to avoid it by simply using a high $\mathrm{LoadDPF}$ setting would not lead to a performance improvement, as shown in subsection 8.10.4.

In summary, the measurements have shown that the new Least Used with DPF policy is also working as intended under realistic conditions in the real Internet. However, as part of future work, additional measurements with different workload parameter sets should be performed to validate the usefulness of the new policy under a broader parameter range.

8.10.6 Summary

In summary, it has been shown that distance-aware policies can significantly improve the performance of highly distributed RSerPool systems. The approach of PR-based delay measurements as distance metric allows to reuse the efficient handlespace management approach presented in chapter 4. The new policies Least Used with DPF and Weighted Random with DPF are realizations of distance-aware policies. While Least Used with DPF only has to introduce a small preference for local PEs (by a low setting of $\mathrm{LoadDPF}$) to improve the performance even for scenarios of localized disasters, Weighted Random with DPF requires a significantly higher influence of the distance (i.e. a higher $\mathrm{WeightDPF}$ setting) to achieve a performance benefit. This results in a reduced performance in a localized disaster situation. Therefore, the usage of this policy has to be planned carefully. Finally, tests in the PLANETLAB have shown that the new Least Used with DPF policy also provides the expected results in the real Internet.

8.11 Configuring the PU-Side Cache Parameters

Depending on the frequency of handle resolutions and their costs (RTT between PU and PR, network bandwidth), it can be beneficial to use the PU-side handle resolution cache as described in subsection 3.7.3. That is, after querying a PR for a handle resolution, the content of the PR's response is stored in the cache for the time given by the stale cache value parameter. Subsequent handle resolutions may be directly satisfied from the cache, avoiding the overhead to query the PR again.

8.11.1 General Effects of the PU-Side Cache

To be effective, the stale cache value has to be configured to a value greater than the request interval i. A simulation to present the general effects of the cache – again based on our paper Dreibholz and Rathgeb (2005c) – has been performed using i=1s and PU:PE ratios r between 1 and 3, for a target system utilization of 60% (for a larger setting the effects would be too strong). The stale cache value:request interval ratio c is a measure for how many times the cache can be utilized for a handle resolution before having to query a PR again. It has been varied from 0 to 10. $\mathrm{MaxHResItems}$ has been ∞ (i.e. all 10 PE identities are returned upon a handle resolution) and the simulated real-time has been 60 minutes. Figure 8.26 presents the results of the simulation.

As expected, the cache does not affect the results of the Random policy, neither the utilization nor the handling speed. Clearly, the utilization of the Least Used policy quickly decays for rising c at r=1, due to a high per-PU load (see equation 8.6) and therefore a high penalty of "bad" selection decisions. Caching results means using out-of-date load information. Therefore, inappropriate decisions become more likely for larger values of c. For higher values of the PU:PE ratio r, the situation becomes much better, since the per-PU penalty of a "bad" choice is reduced by parallelism (see subsection 8.7.2).

The reason for the decaying utilization of the Round Robin policy is the same as for the number of PRs (see section 8.9): each component selects independently using the round robin strategy. But

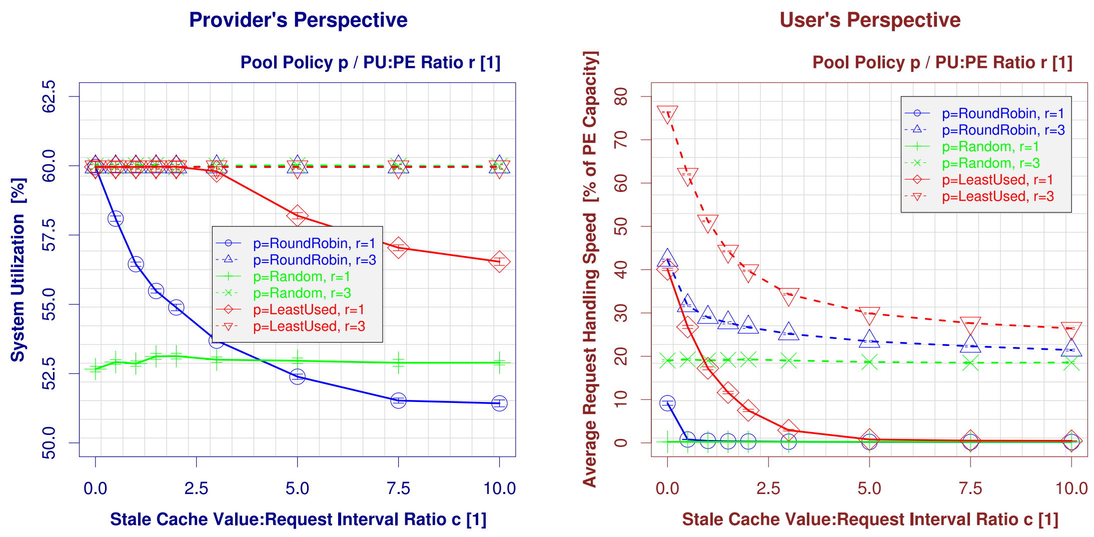


Figure 8.26: The General Effects of using the PU-Side Cache

viewed globally, the selections differ from the desired round robin behaviour. For r=1, it can be observed that the utilization of the Round Robin policy even slightly falls short of the Random policy's utilization for c greater than 4 – the higher the value of c, the more independent are the round robin selections, the more the global view of the selections adapts to random.

Although the utilization penalty for using the cache becomes small if the PU:PE ratio r is high enough, the request handling speed is severely affected for policies other than Random: using the Least Used or Round Robin policies, there remains a significant loss in handling speed. This is caused by long request queues due to an inappropriate PE selection.

As a conclusion, using the cache for a policy other than Random does not make much sense. The gain of saving some overhead messages on handle resolution comes at a high price on system performance. So, in which cases can the cache become valuable?

8.11.2 When to Use the PU-Side Cache?

The PU-side cache can become useful in the following situations:

1. The request size:PE capacity ratio s is small, therefore the network delay for the handle resolution at the PR significantly contributes to a reduction of the handling speed.

2. A PE may reject a request with a certain probability a_{PE}, implying the need for an additional handle resolution to find a new PE. This is e.g. the case in telephone signalling, if the PE's request queue is full and it currently cannot accept any more request.

For a given rejection probability a_{PE}, the total rejection probability after having performed n trials is $a_{\text{Total}} = (a_{\text{PE}})^n$. To reach a given value of a_{Total} (e.g. 0.05 for an acceptance probability of 95%), the number of trials n can be computed as follows:

$$n(a_{\text{Total}}, a_{\text{PE}}) = \max(1, \lceil log_{a_{\text{PE}}}(a_{\text{Total}}) \rceil). \qquad (8.11)$$

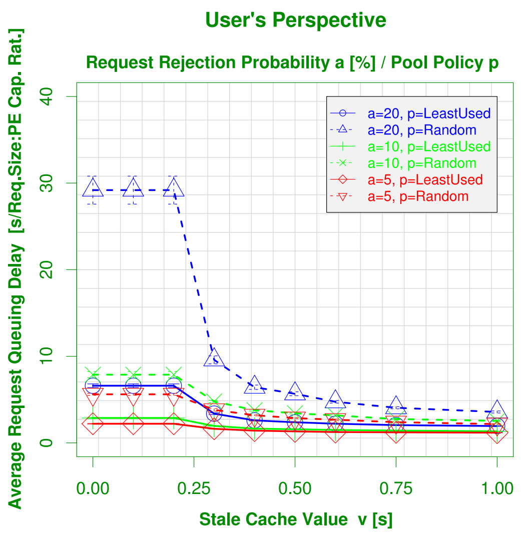


Figure 8.27: An Example for an Effective Cache Usage

It is now possible to configure the cache to cover the calculated number of n trials. That is, the time required to perform n trials can be computed as follows:

$$d(a_{\text{Total}}, a_{\text{PE}}) = \text{RTT}_{\text{PU}\leftrightarrow\text{PR}} + (n(a_{\text{Total}}, a_{\text{PE}}) - 1) * \text{RTT}_{\text{PU}\leftrightarrow\text{PE}}. \tag{8.12}$$

Using the resulting delay as stale cache value, the trials are covered by the cache.

To show the effectiveness of the cache, an example simulation having used a PU:PE ratio of r=3, a request interval of i=1 and a target system utilization of 60% has been performed. The resulting request size:PE capacity ratio, based on equation 8.9, has been s=0.2 (such frequent and short transactions are typical for telecommunications signalling) for a component RTT of 200ms. Again, MaxHResItems has been ∞ and the simulated real-time has been 60 minutes.

Figure 8.27 presents the resulting queuing delay (see subsection 8.5.2 for the definition) for varying stale cache values from 0s (no cache) to 1s (equal to the request interval) and request rejection probabilities a_{PE} from 0.0 (0%) to 0.2 (20%). The system utilization is stable at 60% for all settings; therefore, a figure is omitted. Plots for the Round Robin policy are also not shown, since the results are quite similar.

Obviously, the cache has a huge impact on the reduction of the queuing delay (and therefore on an improvement of the handling speed). In the extreme case of using the Random policy and a_{PE}=0.2, a stale cache value of only 400ms reduces the average queuing delay from 28s to less than 7s ($d(0.05, 0.20)$=400ms); for a_{PE}=0.1, the queuing delay of Random drops from 8s to 4s at the same stale cache value ($d(0.05, 0.10)$=400ms). Clearly, the reductions for the Least Used policy are smaller, since the initial delay of the Least Used policy without cache is much smaller than for Random. Furthermore, the load values become inaccurate due to the cache. But nevertheless, there is still a significant queuing delay reduction by several seconds.

8.11.3 Summary

In summary, the general guideline on the PU-side cache usage is to avoid it – except for the Random policy, of course. However, on the conditions of

- Costly handle resolutions (referring to network delay) at a PR, as well as
- A certain probability that the handle resolution has to be repeated (due to rejection of the request by the chosen PE, or if the selected PE is unreachable),

the handle resolution cache – with a reasonably configured stale cache value – can achieve a significant performance benefit.

8.12 The Effects of Heterogeneous Server Capacities

After having analysed the behaviour of RSerPool in scenarios of homogeneous server capacities, it is clearly necessary to also have a look at scenarios where some PEs are more powerful than others. The goal of this section is therefore to provide more insights into the implications of such scenarios and to identify critical configurations and parameter ranges. A subset of these results has been published in Dreibholz and Rathgeb (2005e).

8.12.1 Server Capacity Distribution Scenarios

In the following subsubsections, the effects of heterogeneous server capacities are analysed for different capacity distributions.

8.12.1.1 A Single Powerful Server

The most obvious scenario of heterogeneous server capacities is probably to have a designated powerful server. Such a scenario is likely if there is a high-capacity server to do the usual work and a set of older and slower ones to provide failure protection by redundancy. In order to present the effects introduced by heterogeneous server capacities, it is useful to vary the degree of heterogeneity. Therefore, the variable κ is defined as the capacity ratio between the powerful server and a slow server:

$$\kappa = \frac{\mathrm{Capacity}_{\mathrm{FastPE}}}{\mathrm{Capacity}_{\mathrm{SlowPE}}}. \tag{8.13}$$

For example, a value of κ=5 means that the fast server has five times the capacity of a slow one. For the following simulation, κ is varied between 1 and 8. In order to keep the overall pool capacity constant, the resulting server capacities are being normalized (scenarios of an increased overall pool capacity are evaluated later in subsection 8.12.2). That is:

$$\mathrm{PoolCapacity} = \underbrace{1 * (\kappa * \mathrm{Capacity}_{\mathrm{SlowPE}})}_{\text{Capacity of the Fast PE}} + \underbrace{(\mathrm{PEs} - 1) * \mathrm{Capacity}_{\mathrm{SlowPE}}}_{\text{Overall Capacity of the Slow PEs}}.$$

Using a fixed setting of PoolCapacity, the PE capacities can be calculated as follows:

$$\begin{aligned}
\mathrm{Capacity}_{\mathrm{SlowPE}}(\kappa) &= \frac{\mathrm{PoolCapacity}}{1 * (\kappa - 1) + \mathrm{PEs}}, && (8.14)\\
\mathrm{Capacity}_{\mathrm{FastPE}}(\kappa) &= \kappa * \mathrm{Capacity}_{\mathrm{SlowPE}}. && (8.15)
\end{aligned}$$

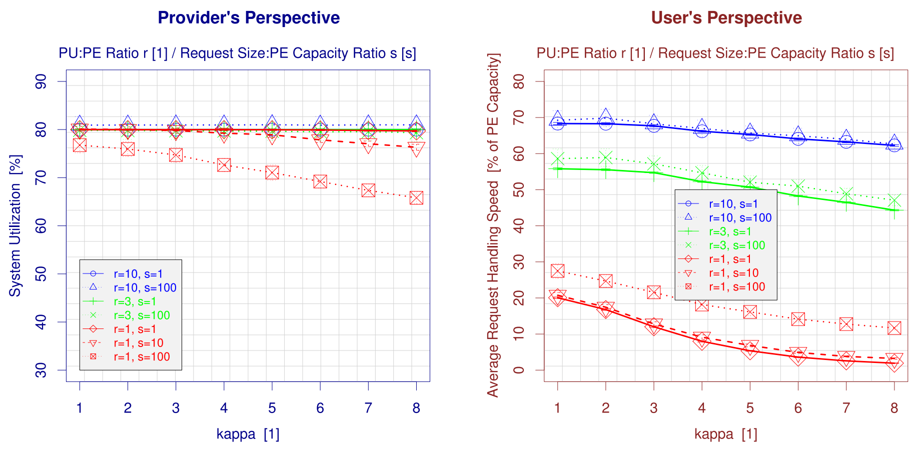


Figure 8.28: A Single Powerful Server: Using the Least Used Policy

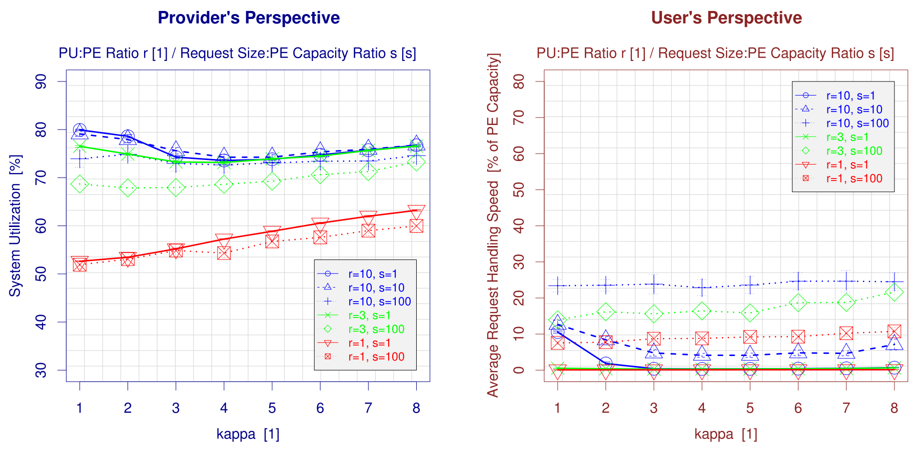


Figure 8.29: A Single Powerful Server: Using the Weighted Random Policy

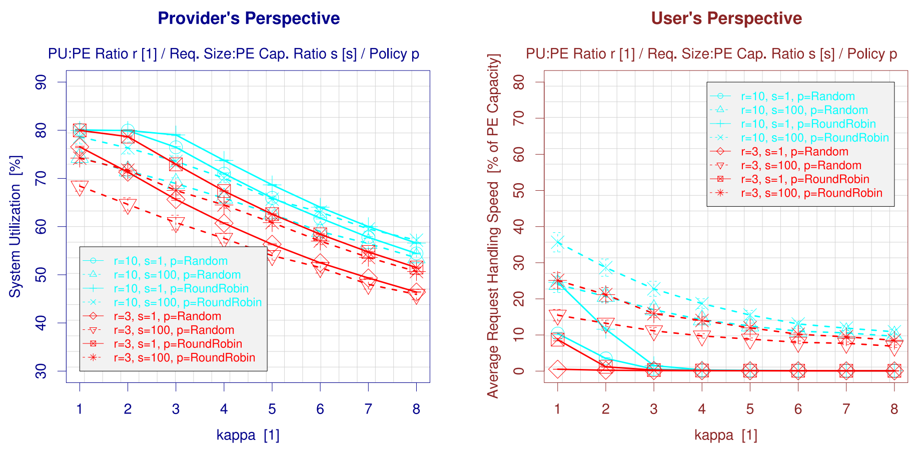


Figure 8.30: A Single Powerful Server: Using Inappropriate Policies

Results for the Least Used Policy Figure 8.28 presents the performance results for the Least Used policy. κ has been varied for different settings of the PU:PE ratio r (from 1 to 10) and of the request size:PE capacity ratio s between 1 and 100. The target system utilization has been 80%. Clearly, the utilization is only negatively affected for a critical PU:PE ratio (r=1) in conjunction with a large request size:PE capacity ratio (s=100). As already observed and explained for the workload parameter analysis in subsection 8.7.2, mapping a long request to an inappropriate PE results in a "request jam" at the corresponding PU and therefore leads to a decreased system utilization. In the case of a dedicated fast server, a "bad" PE selection means mapping a request to a slow server. That is, it obviously takes longer to process the request.

The results for the utilization are reflected by the average handling speed results: the handling speed is slightly decreasing with κ, since the capacity – and therefore the processing speed – of the slow servers decreases with rising κ (since the overall pool capacity remains constant). But while the reduction for r=1 is highest (e.g. from 20% at κ=1 to about 5% at κ=8), the descent becomes significantly smaller with a higher PU:PE ratio r: for r=10, the handling speed only degrades from 68% at κ=1 to about 65% at κ=8. That is, as already observed for the workload parameter analysis in subsection 8.7.2, the PU:PE ratio has the most significant influence on the system performance. And again, the influence of the request size:PE capacity ratio s is fairly small if r is high enough: while at r=1 a handling speed difference of 8% to 10% can be observed between s=1 and s=100, the difference is only about 1% for r=10. For a sufficiently high PU:PE ratio r, it is easy to compensate "bad" selections by using a fast PE for the next request. Furthermore, using longer requests results in a better handling speed, since the influence of the queuing delay is reduced (see also subsubsection 8.7.2.2).

Results for the Weighted Random Policy The performance results for the Weighted Random policy are presented in figure 8.29. Clearly, the weight w_i of each PE i is its capacity Capacity_i (see also subsubsection 4.4.2.5). As expected from the workload parameter analysis in subsection 8.7.2, the PU:PE ratio r is critical for the non-adaptive (Weighted) Random policy. If r is low, the utilization is lowest. This effect is amplified by a large request size:PE capacity ratio s – mapping large requests is more difficult.

A rising heterogeneity of the capacities (i.e. an increased value of κ) simplifies the distribution of requests: there is a single powerful server which concentrates most of the pool's capacity. However, while this observation holds for the critical parameter range (r low), it can be observed for the curves of r=10 and s between 1 and 100, that the utilization decreases until about κ=4 and then starts rising again. The reason for this effect is that the capacity of the slower servers becomes low, but there is still a certain probability that they get requests.

In order to further evaluate this effect, it is necessary to have a look at the request handling speed. Here, the request size:PE capacity ratio s becomes the important parameter: as already observed in subsection 8.7.2, larger requests are better for the handling speed – a single large request is affected by the queuing delay only once, splitting it up into 100 short requests results in 100 occurrences of queuing delay. Using small requests (here: s=1 and s=10), the system performance does not reach an acceptable value until a certain PU:PE ratio r is reached (here: r=10). However, the handling speed still quickly decays with rising κ (see the plots for r=10 and s=1 or s=10). That is, using Weighted Random in heterogeneous scenarios requires a reasonably high PU:PE ratio r in order to provide an acceptable performance. In this case, a handling speed increasing slightly with κ can be expected – the more powerful the fast PE, the higher the possible speed it can process its requests with.

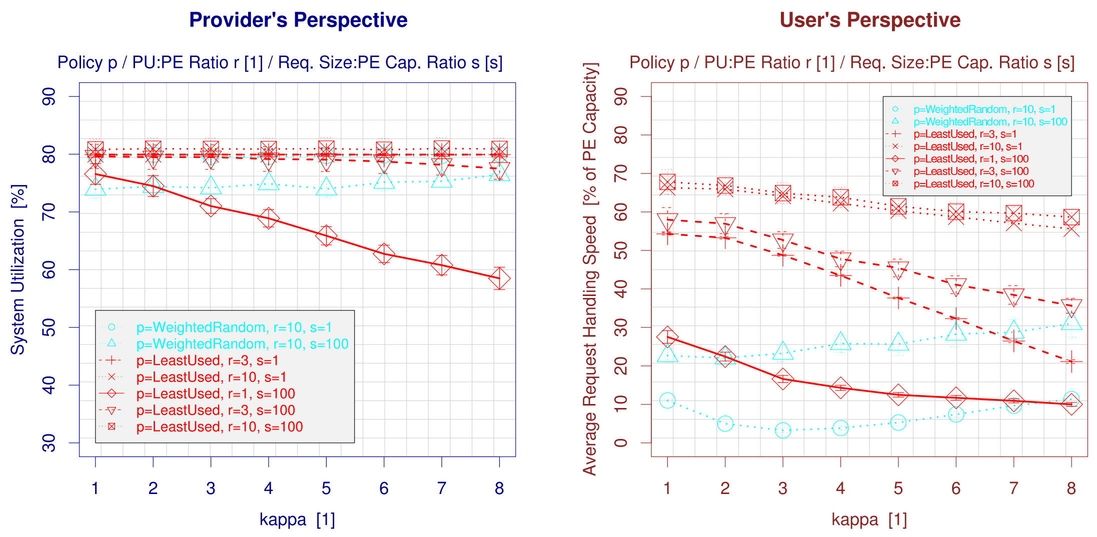


Figure 8.31: One Third Powerful Servers

Results for the Round Robin and Random Policies For comparison, figure 8.30 shows the performance results for the Round Robin and Random policies. Clearly, since these policies do not have any information about different server capacities (like Weighted Random) or load states (like Least Used), the performance of both policies severely degrades in heterogeneous scenarios (i.e. for $\kappa > 1$). Therefore, the Round Robin and Random policies are unsuitable for heterogeneous server capacity scenarios.

Summary In summary, it has been shown that the policies Least Used and Weighted Random are able to cope with the existence of a dedicated fast server. Again, the PU:PE ratio is the most critical workload parameter for the system performance. In particular if using the Weighted Random policy, the PU:PE ratio has to be sufficiently high to achieve a reasonable performance.

8.12.1.2 Multiple Powerful Servers

After having evaluated the scenario of a single dedicated fast server in the previous subsubsection 8.12.1.1, it is clearly useful to analyse a scenario containing multiple powerful servers. To show the essential results, a scenario of 9 PEs including 3 fast ones has been used. Again, κ defines the capacity ratio between fast and slow servers. That is, the overall capacity $\mathrm{PoolCapacity}$ of the pool is given by:

$$\mathrm{PoolCapacity} = \underbrace{\mathrm{PEs}_{\mathrm{FastPE}} * (\kappa * \mathrm{Capacity}_{\mathrm{SlowPE}})}_{\text{Overall Capacity of the Fast PEs}} + \underbrace{(\mathrm{PEs} - \mathrm{PEs}_{\mathrm{Fast}}) * \mathrm{Capacity}_{\mathrm{SlowPE}}}_{\text{Overall Capacity of the Slow PEs}}.$$

Using a fixed setting of $\mathrm{PoolCapacity}$, the PE capacities can be calculated as follows:

$$\mathrm{Capacity}_{\mathrm{SlowPE}}(\kappa) = \frac{\mathrm{PoolCapacity}}{\mathrm{PEs}_{\mathrm{Fast}} * (\kappa - 1) + \mathrm{PEs}}, \quad (8.16)$$

$$\mathrm{Capacity}_{\mathrm{FastPE}}(\kappa) = \kappa * \mathrm{Capacity}_{\mathrm{SlowPE}}. \quad (8.17)$$

The other parameters have remained as for the scenario of a single fast server described in subsubsection 8.12.1.1.

Figure 8.31 shows the results of the simulation for the Weighted Random and Least Used policies. As it has already been shown and explained in subsubsection 8.12.1.1, the Random and Round Robin policies – as well as Weighted Random for a too small PU:PE ratio – are useless for heterogeneous scenarios. Therefore, the results for these policies and parameter settings have been omitted.

As expected from the utilization results of the previous simulation, only a small PU:PE ratio r in combination with a high request size:PE capacity ratio s is critical: r=1 and s=100 for Least Used, as well as r=10 and s=100 for Weighted Random. Comparing these results for the critical parameter settings to the results of the "single fast server" scenario (see figure 8.28 for the Least Used policy and figure 8.29 for the Weighted Random policy; note the different axis scaling for the handling speed), a slight utilization decrease for Least Used and an utilization increase for Weighted Random can be observed. These effects are the results of the changed capacity distribution: having multiple fast servers, most of the pool's capacity is concentrated on these servers. This leads to a smaller capacity of a slow server for a fixed setting of κ.

For example, the slow server capacity for κ=5 is about 455,000 calculations/s if there are three fast servers (see equation 8.16), while a slow server still has a capacity of about 714,000 calculations/s in the "single fast server" scenario (see equation 8.14).

Due to the smaller capacities of the slow servers, it is more difficult for Least Used to reach a high utilization in the "three fast servers" scenario: a slow server will be selected if it only has the lowest load value. This leads to a decreased handling speed. The higher the heterogeneity of the pool, the more significant the effect. In a critical parameter range (i.e. a low PU:PE ratio and a high request size:PE capacity ratio), this leads to queuing of requests ("request jam") and therefore to a reduced utilization.

On the other hand, having multiple fast servers is beneficial for the Weighted Random policy: since $w_i = \text{Capacity}_i$, the probability of mapping a request to the very slow PEs becomes small, resulting in an improved request handling speed and utilization compared to the "single fast server" scenario. Furthermore, as already observed in subsubsection 8.12.1.1, the speed is improved with an increased heterogeneity of the pool (i.e. a higher value of κ).

8.12.1.3 Linear Capacity Distribution

The next capacity scenario to be analysed is a linear distribution. Such a distribution is likely if the pool consists of multiple generations of servers. For example, a pool could consist of 10 PEs, where the first one had been installed five years ago and a new one – containing state-of-the-art hardware – has been added every 6 months. The capacities of the other servers are linearly distributed between the capacity of the oldest (i.e. slowest) PE ($\text{Capacity}_{\text{SlowestPE}}$) and the capacity of the newest (i.e. fastest) PE ($\text{Capacity}_{\text{FastestPE}}$). The ratio between these two capacities defines the scaling factor γ:

$$\gamma = \frac{\text{Capacity}_{\text{FastestPE}}}{\text{Capacity}_{\text{SlowestPE}}}.$$

That is, for γ=3 the fastest server has three times more capacity than the slowest one. Again, while scaling γ, the resulting server capacities are being normalized to keep the overall pool capacity constant (non-normalized scenarios are analysed later in subsection 8.12.2). That is, the overall capacity

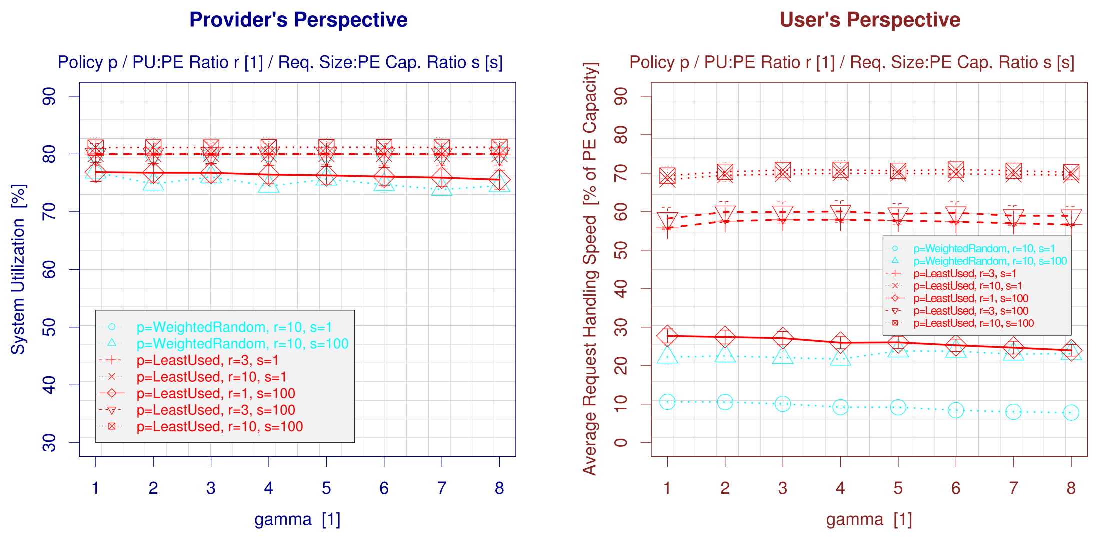


Figure 8.32: Using a Linear Capacity Distribution

of the pool, PoolCapacity, can be calculated as follows:

$$\begin{aligned}
\text{PoolCapacity} &= \underbrace{\sum_{i=1}^{\text{PEs}} \left(\underbrace{\frac{(\gamma-1) * \text{Capacity}_{\text{SlowestPE}}}{\text{PEs}-1}}_{\text{Capacity Gradient}} *(i-1) \right)}_{\text{Additional PE Capacities}} + \underbrace{\sum_{i=1}^{\text{PEs}} \text{Capacity}_{\text{SlowestPE}}}_{\text{Base PE Capacities}} \\
&= \left[\frac{(\gamma-1) * \text{Capacity}_{\text{SlowestPE}}}{\text{PEs}-1} * \sum_{i=1}^{\text{PEs}-1} i \right] + \text{Capacity}_{\text{SlowestPE}} * \text{PEs} \\
&= \text{Capacity}_{\text{SlowestPE}} * \left(\frac{(\gamma-1) * \text{PEs}}{2} + \text{PEs} \right).
\end{aligned}$$

That is, given a fixed value of PoolCapacity, $\text{Capacity}_{\text{SlowestPE}}$ can be calculated as follows:

$$\begin{aligned}
\text{Capacity}_{\text{SlowestPE}} &= \frac{\text{PoolCapacity}}{(\gamma-1) * \frac{\text{PEs}}{2} + \text{PEs}} \\
&= \frac{\text{PoolCapacity}}{\text{PEs} * (\frac{\gamma-1}{2} + 1)}.
\end{aligned}$$

Finally, the capacity of PE i is:

$$\text{Capacity}_i = \underbrace{\frac{(\gamma-1) * \text{Capacity}_{\text{SlowestPE}}}{\text{PEs}-1}}_{\text{Capacity Gradient}} *(i-1) + \underbrace{\text{Capacity}_{\text{SlowestPE}}}_{\text{Base Capacity}}.$$

In order to perform the simulation, the setup described in subsubsection 8.12.1.1 has been used with the PE capacity distribution as described above. The results are presented in figure 8.32. Clearly, the distribution of the capacities in a linear scenario is very systematic: for each slower PE, there will also be a faster one. A PU being served by a slow PE will be served by a fast one soon. On

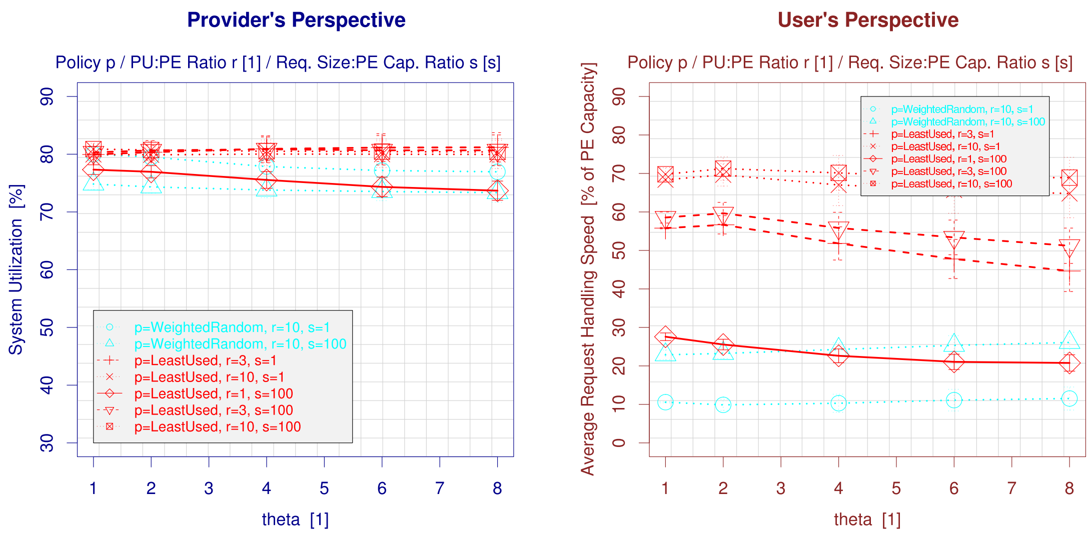


Figure 8.33: Using a Uniform Capacity Distribution

average, the difference to a homogeneous scenario (i.e. γ=1) becomes almost insignificant. Again, as already observed and explained for the fast server scenarios in subsubsection 8.12.1.1 and subsubsection 8.12.1.2, the only case of a critical parameter range is a small PU:PE ratio r in combination with a large request size:PE capacity ratio s.

After having evaluated scenarios using deterministic capacity distributions, it is also useful to examine randomized ones. Such analyses will be provided by the following two subsubsections.

8.12.1.4 Uniform Random Capacity Distribution

For the first randomized capacity distribution scenario, the server capacities have been uniformly randomized. Such a scenario is likely if currently unused PCs are added to a compute pool for real-time distributed computing (see subsection 3.6.5). Clearly, there is a certain lower bound for the minimum capacity (set by the administrator) and an upper bound defined by the state-of-the-art in PC technology. The ratio between the minimum and maximum capacities is given by the factor ϑ, which is defined as follows:

$$\vartheta = \frac{\text{Capacity}_{\text{FastestPE}}}{\text{Capacity}_{\text{SlowestPE}}}.$$

Then, for each PE i, its capacity Capacity_i is randomly chosen (using an uniform distribution):

$$\text{Capacity}_i \in_R [\text{Capacity}_{\text{SlowestPE}}, \ldots, \vartheta * \text{Capacity}_{\text{SlowestPE}}] \subset \mathbb{N}.$$

As for the previous simulations, the resulting capacities have to be normalized, i.e. the average system capacity remains constant for all settings of ϑ (see subsection 8.12.2 for non-normalized scenarios).

In order to show the essential effects, a simulation using an uniform capacity distribution has been performed. For each parameter setting, 64 runs have been performed with different seeds to achieve a sufficient statistical accuracy. All other parameters have remained as described in subsubsection 8.12.1.1. The results are presented in figure 8.33.

Again, as observed and explained for the simulations of the previous subsubsections, the utilization is affected by critical parameter ranges of the PU:PE ratio r (small) and the request size:PE capacity ratio s (large) only. In a critical parameter range, the utilization sinks slightly with a rising ϑ.

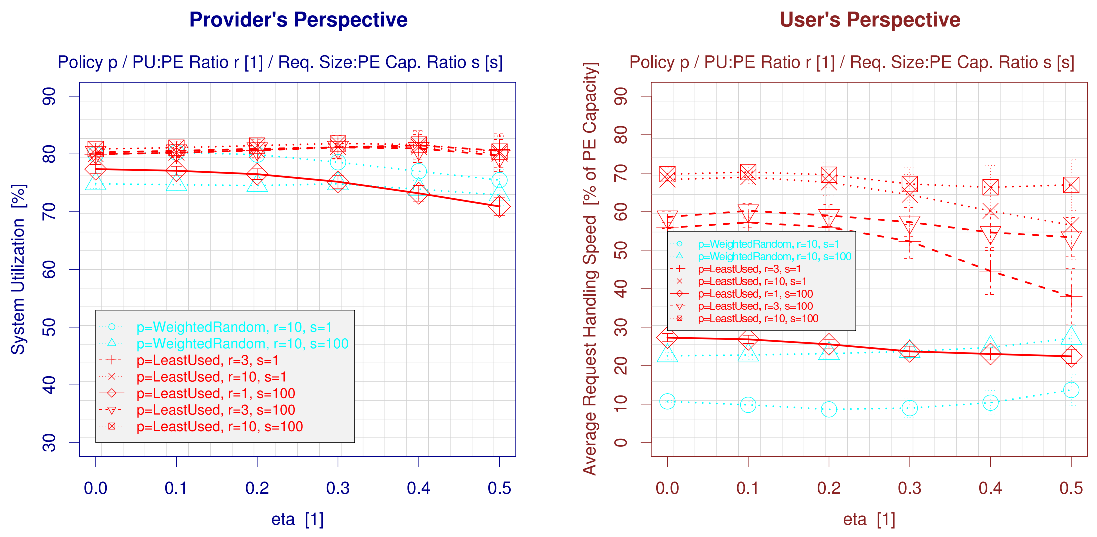


Figure 8.34: Using a Truncated Normal Capacity Distribution

On the handling speed side, however, the curves for Least Used are obviously negatively affected by a rising value of ϑ (e.g. from 59% at ϑ=1 to 52% at ϑ=8 for r=3, s=100), while an increased heterogeneity slightly increases the performance of Weighted Random (e.g. from 22% at ϑ=1 to about 26% at ϑ=8 for r=10, s=100). The reason is clear: the higher the heterogeneity (i.e. a large setting of ϑ), the higher the probability for the existence of very slow PEs. While for Weighted Random the probability for the selection of such a slow PE sinks with its capacity, Least Used will select such a PE if it just has a small load. Clearly, such a selection will lead to a low handling speed.

8.12.1.5 Truncated Normal Random Capacity Distribution

Finally, the last capacity distribution scenario has used a truncated normal random distribution with a given average PE capacity $\mathrm{Capacity}_{\mathrm{AveragePE}}$ (10^6 calculations/s) and the standard deviation given by $\eta * \mathrm{Capacity}_{\mathrm{AveragePE}}$. Obviously, capacities cannot be negative. Therefore, the capacity selection is truncated by enforcing a lower limit (10^4 calculations/s). Finally, the PE capacities have to be normalized in order to keep the overall system capacity constant.

Figure 8.34 shows the results of the simulation for the truncated normal distribution. The setup has remained the same as having used for the uniform distribution (see subsubsection 8.12.1.4). The results for the utilization are quite similar compared to the uniform results: in the critical parameter range (low PU:PE ratio r, large request size:PE capacity ratio s), the utilization decreases slightly with η.

However, the handling speed curves for Least Used are obviously negatively affected by a rising value of η (e.g. from 59% at η=0 to 52% at η=0.5 for r=3, s=100), while an increased heterogeneity slightly increases the performance of Weighted Random (e.g. from 22% at η=0 to about 26% at η=0.5 for r=10, s=100). Again, the higher the heterogeneity of the pool (a large setting of η), the more probable the existence of low-capacity PEs. And – as explained in the previous subsubsection – Least Used may map requests to such PEs (if they have the lowest load), while for Weighted Random the weights and therefore the selection probabilities become small.

8.12.1.6 Summary

In summary, the simulations presented in the previous subsubsections have outlined the following important effects:

- The policies Round Robin and Random are obviously unsuitable in heterogeneous server capacity scenarios, since they neither have information on PE load states nor on their capacities.
- Weighted Random is capable to cope with heterogeneous server capacities, although the PU:PE ratio has to be sufficiently high. For a rising heterogeneity of the pool, Weighted Random is also able to make use of very fast PEs to increase the overall request handling speed.
- Least Used clearly achieves the best performance, but it can be observed that the overall request handling speed decreases with a rising heterogeneity of the pool – in particular, if the capacities are highly unbalanced (e.g. in the scenarios of dedicated fast servers or for the truncated normal distribution). This is a result of the fact that Least Used selects by PE load rather than by PE capacity: even a very slow PE will get requests as long as its load state is the lowest.

In particular, the observation for the Least Used handling speed leads to the following important question: which policies are able to make best use of an increased pool capacity (i.e. in a non-normalized capacity scenario)? This question will be answered in the following subsection.

8.12.2 A Load-Increment-Aware Policy

An important feature of RSerPool is the possibility to dynamically add servers to a pool or remove them from in order to adapt a pool's capacity to a changed demand. In particular, this allows for an administrator to add capacity if the pool is highly utilized. But what happens with additional capacity if the pool is (temporarily) slightly loaded and removing some servers is not useful? Clearly, it is usually desirable that the spare capacity would result in an improvement of the request handling speed! But are the policies sufficiently equipped to handle such a scenario? Some basic ideas have already been proposed in Dreibholz, Rathgeb and Tüxen (2005), but a more general evaluation has still been missing. Therefore, the goal of this subsection is to provide an appropriate analysis.

8.12.2.1 A Single Powerful Server

Probably the most obvious scenario of adding capacity to a pool is to increase the capacity of a single dedicated server. This could e.g. mean to add more memory, increase the CPU speed or add another CPU. The ratio between the new and the original capacity of the pool can be given by the following equation:

$$\varphi = \frac{\mathrm{PoolCapacity}_{\mathrm{New}}}{\mathrm{PoolCapacity}_{\mathrm{Original}}}. \tag{8.18}$$

The variable φ is denoted as the *Capacity Scale Factor*; a value of φ=1 stands for no capacity change, while φ=4 denotes a quadruplicated capacity.

In case of a single powerful server, changing φ results in varying the capacity of the designated PE only. That is, the capacity increment of the whole pool, denoted as $\Delta_{\mathrm{Pool}}(\varphi)$, can be calculated as follows:

$$\Delta_{\mathrm{Pool}}(\varphi) = \underbrace{(\varphi * \mathrm{PoolCapacity}_{\mathrm{Original}})}_{\mathrm{PoolCapacity}_{\mathrm{New}}} - \mathrm{PoolCapacity}_{\mathrm{Original}}. \tag{8.19}$$

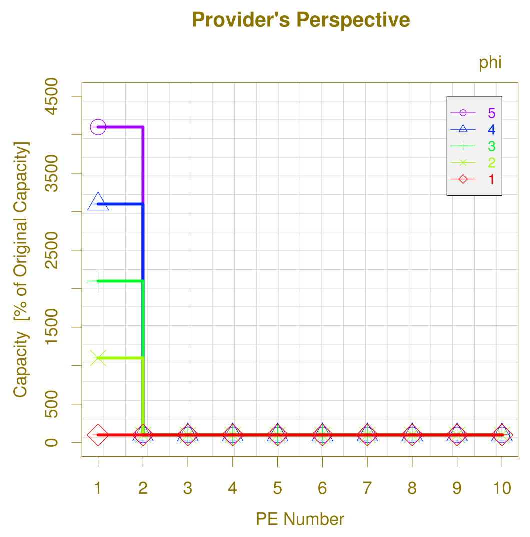


Figure 8.35: Server Capacities of the Single Powerful Server Scenario

Then, the capacity of the powerful PE, $\text{Capacity}_{\text{FastPE}}(\varphi)$, can simply be calculated as:

$$\text{Capacity}_{\text{FastPE}}(\varphi) = \frac{\text{PoolCapacity}_{\text{Original}}}{\text{PEs}} + \Delta_{\text{Pool}}(\varphi).$$

The capacities of all other PEs remain unchanged, i.e.:

$$\text{PoolCapacity}_{\text{SlowPE}}(\varphi) = \frac{\text{PoolCapacity}_{\text{Original}}}{\text{PEs}}.$$

Figure 8.35 illustrates the capacity for each of the 10 PEs for having varied the value of φ.

To show the essential effects of varying the fast server's capacity, simulations have been performed using the policies Round Robin, Random, Weighted Random, Least Used and Priority Least Used (to be introduced below) in a scenario of PU:PE ratios r from 1 to 10 and a request size:PE capacity ratio of s=10 for a target system utilization of 90% in the heterogeneous case. Note, that the request load being created by the PUs remains constant. The performance results are presented in figure 8.36.

The results for the average system utilization are not very surprising: the higher the capacity of the pool, the lower the utilization. Since the requested capacity remains constant and the policies are not used in critical parameter ranges (i.e. in particular, the PU:PE ratio for the non-adaptive policies is sufficiently high, see section 8.7), no significant utilization differences among the used policies can be observed.

Clearly, as it can be expected from the results in subsubsection 8.12.1.1, Round Robin and Random provide the lowest handling speed. That is, these policies are only useful in homogeneous scenarios. However, the performance of Least Used is surprisingly bad: at φ=5 and for r=3, the handling speed is only 90% for Least Used, while it is already about 500% for the Weighted Random policy. What is the problem of the Least Used policy in this scenario?

The reason for the bad performance of the Least Used policy is the fact that the selection decision is based on the current PE load only. Consider a powerful PE #1 loaded by 11% and a slow PE #2 loaded by 10%. Clearly, the Least Used policy would select PE #2, because it has the lowest load. But the new request may increase the slow PE #2's load by another 10%, while using PE #2 may only have

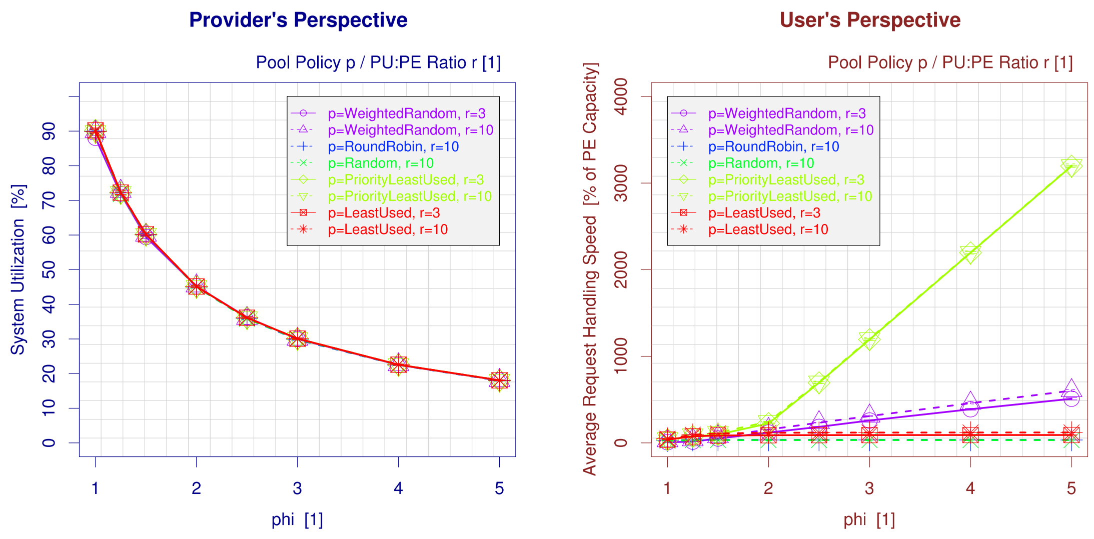


Figure 8.36: Increasing the Capacity of a Single Server

increased its load by 2%. That is, PE #2 would have been able to handle the request more quickly. The lack of Least Used to incorporate the aspect of different load increments for different servers has led to the definition of the Priority Least Used policy (see subsubsection 3.11.3.2 for its definition and subsubsection 4.4.2.4 for its implementation as part of the handlespace management), which has been defined by us in Dreibholz, Rathgeb and Tüxen (2005) as result of the described problem statement. Using Priority Least Used, each PE can specify its load increment, i.e. the number of load units the server's load is increased by an additional request. Upon selection, the PE having the lowest sum of load and load increment is chosen. That is, while Least Used takes the PE *currently* having the lowest load, Priority Least Used selects the PE having the lowest load *after* accepting a new request.

For the simulations in figure 8.36, the load increment $\hat{l}_i$ for each PE i is defined as:

$$\hat{l}_i = \frac{\text{Capacity}_i(\varphi)}{2.5 * 10^5}. \tag{8.20}$$

That is, using a PE of the average capacity (10^6 calculations/s, see section 8.6), a new request rises a PE's load by $\hat{l}_{\text{AveragePE}}$=25%. If the fast PE's capacity would be $5{*}10^6$ calculations/s, the load increment would only be $\hat{l}_{\text{FastPE}}$=5%.

Using the new Priority Least Used policy with the described setting of the load increment, a significant handling speed gain can be observed (see figure 8.36): about 3,250% for φ=5 (r=3 and r=10) vs. 600% for Weighted Random (r=10) and about 160% for Least Used (r=10). That is, the new Priority Least Used policy provides the desired functionality of increasing the handling speed in the scenario of a single designated server. But what about other scenarios of increased PE capacity?

8.12.2.2 Multiple Powerful Servers

Another realistic capacity increment scenario is to tune multiple servers instead of a designated one only. Therefore, the following simulation has evaluated the effect of equally increasing the power of one third (3) of 9 PEs. All other parameters have remained as for the previous simulation in subsubsection 8.12.2.1. Therefore, the capacity $\text{Capacity}_i(\varphi)$ of PE i has been defined as follows

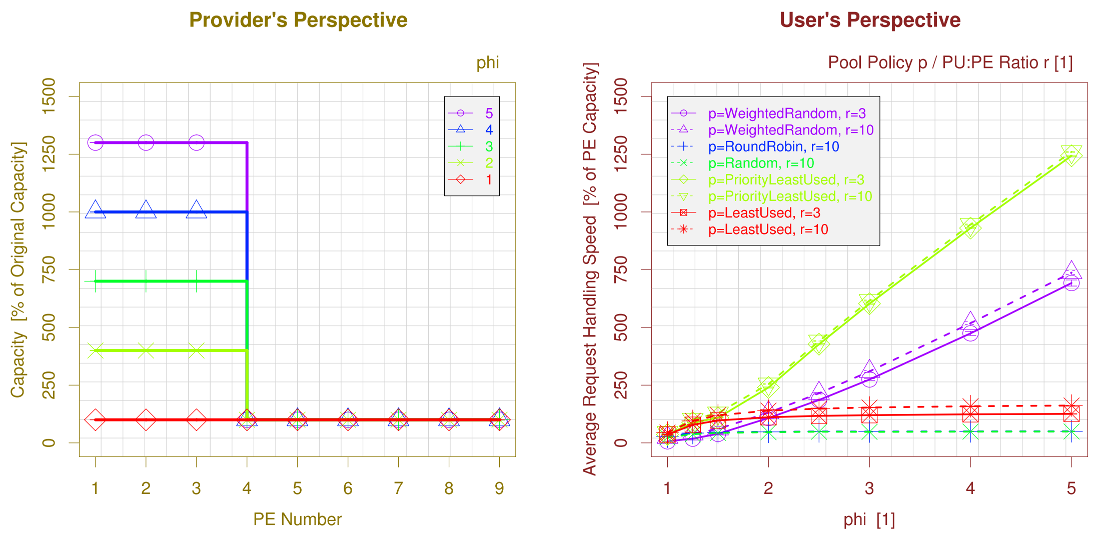


Figure 8.37: Increasing the Capacity of One Third of the Servers

(for the pool capacity increment $\Delta_{\text{Pool}}(\varphi)$ as defined by equation 8.19):

$$\begin{aligned}\Delta_{\text{FastPE}}(\varphi) &= \frac{\Delta_{\text{Pool}}(\varphi)}{\text{PEs}_{\text{Fast}}},\\ \text{Capacity}_i(\varphi) &= \begin{cases}\frac{\text{PoolCapacity}_{\text{Original}}}{\text{PEs}} + \Delta_{\text{FastPE}}(\varphi) & (i \leq \text{PEs}_{\text{Fast}})\\ \frac{\text{PoolCapacity}_{\text{Original}}}{\text{PEs}} & (i > \text{PEs}_{\text{Fast}})\end{cases}.\end{aligned}$$

Figure 8.37 shows the simulation results for the scenario having used 9 PEs, including 3 fast servers. Since the utilization results are quite similar to the previous simulation, they have been omitted here. Instead, the left-hand side of figure 8.37 presents the capacity of each server for varying settings of φ; the right-hand side shows the plots for the average request handling speed. As a general result for the handling speed, it can be observed that the behaviour of the policies is quite similar to the scenario of one designated powerful server. That is, while Round Robin and Random are in fact useless, the performance of Least Used is significantly outperformed by Weighted Random (e.g. a handling speed of more than 700% vs. about 125% for Least Used for a PU:PE ratio r=10; the request size:PE capacity ratio s is 10 for all simulations of this subsection). Again, a significant performance improvement is achievable by using the Priority Least Used policy with the load increment setting defined by equation 8.20.

Comparing the results with the observations for the “single fast server” scenario of subsubsection 8.12.2.1, the effects of the changed capacity distribution can be observed: while the single powerful PE incorporates almost the pool’s complete capacity (e.g. 4,100% of a slow server’s capacity for φ=5, see figure 8.35), the additional capacity is now divided up among three servers (i.e. 3 servers having 1,300% of a slow server’s capacity for φ=5, see the left-hand side of figure 8.37). In this case, the top handling speed is lower (e.g. 1,250% vs. 3,250% for Priority Least Used at φ=5), since the maximum possible request handling speed is limited by the processing speed of a fast PE. For the Least Used policy and also for Round Robin and Random, the fact that now one third of the servers are powerful ones becomes beneficial: their handling speed is increased, since the probability of mapping a request to a powerful PE is increased significantly (e.g. 160% vs. 125% for Least Used at φ=5 and r=10).

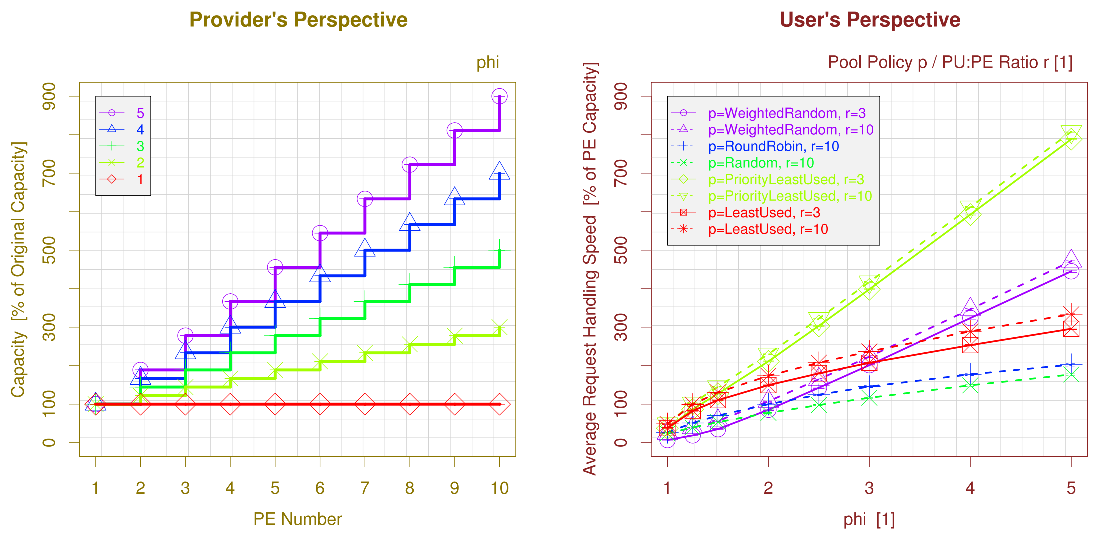


Figure 8.38: Increasing the Server Capacities Linearly

In summary, while it has been shown that an appropriate policy can make use of additional capacity to improve the handling speed in scenarios containing a set of powerful PEs, it is furthermore necessary to have a look at a scenario containing a less extreme capacity distribution.

8.12.2.3 Linear Capacity Distribution

In the final capacity scenario, the PE capacities are increased linearly. That is, while the capacity of the first PE remains constant, the capacities of the following PEs are increased with a linear gradient – for the pool to reach its desired capacity $\mathrm{PoolCapacity}_{\mathrm{New}}$ (see also equation 8.18). Therefore, the capacity $\mathrm{Capacity}_i(\varphi)$ of PE i is defined as follows (for the additional pool capacity $\Delta_{\mathrm{Pool}}(\varphi)$ as defined by equation 8.19):

$$\begin{aligned}\Delta_{\mathrm{FastestPE}}(\varphi) &= \frac{2 * \Delta_{\mathrm{Pool}}(\varphi)}{\mathrm{PEs}},\\ \mathrm{Capacity}_i(\varphi) &= \underbrace{\underbrace{\frac{\Delta_{\mathrm{FastestPE}}(\varphi)}{\mathrm{PEs}-1}}_{\text{Capacity Gradient}} *(i-1)}_{\text{Additional Capacity for PE } i} + \frac{\mathrm{PoolCapacity}_{\mathrm{Original}}}{\mathrm{PEs}}.\end{aligned}$$

A simulation using the linear capacity distribution has been performed. All other parameters have remained as for the fast server scenario of subsubsection 8.12.2.1. The results are presented in figure 8.38; the left-hand side shows the capacity for each of the 10 PEs for having varied φ, the right-hand side presents the average handling speed results. Again, a plot for the system utilization has been omitted, since it does not provide any new insights.

While the general ranking behaviour of the policies remains as observed for the two scenarios of fast servers (see subsubsection 8.12.2.1 and subsubsection 8.12.2.2), it is clearly visible that a linear capacity distribution results in smaller performance differences among the policies: for φ=5, the capacity of the fastest PE is only 900% of the slowest PE's one (see the left-hand side of figure 8.38), while it is 1,300% in the three-fast-servers scenario and even 4,100% for the dedicated fast server (see the left-hand sides of figure 8.37 and figure 8.35). That is, while the top request handling speed for requests is significantly lower (e.g. only about 800% for Priority Least Used and a PU:PE ratio r=10), the chance that a less-performing policy can reach a high handling speed is significantly improved. At φ=5 and r=10, the Random policy already achieves a handling speed of about 175%, while Round Robin even exceeds 200%. Least Used is able to reach about 330%, but is still being outperformed by Weighted Random with about 450%. Note, that the handling speed of Weighted Random only outperforms Least Used for $\varphi > 3$ – for smaller values of φ, its performance is significantly lower. Compared to one dedicated server (Least Used is already outperformed at φ=2), the performance of Least Used for a linear capacity distribution is significantly better.

As the main result, it has been observed that the linear scenario is significantly less critical – even inappropriate policies like Round Robin and Random are able to make use of the improved capacity: while it is still possible to map a request to a slower PE, the probability to map the next request to a faster one is the same (due to the linear capacity distribution).

8.12.2.4 Summary

In summary, it has been shown that the Weighted Random policy is able to make use of increased server capacities by increasing the request handling speed, while Round Robin and Random – as expected – are not useful in a heterogeneous capacity scenario. However, the performance of the Least Used policy falls significantly behind Weighted Random if the scenario is sufficiently heterogeneous: Least Used only takes the load states into account, but not the PEs' capacities. The new Priority Least Used policy solves this problem by allowing a PE to define a load increment constant.

As result of suggesting the Priority Least Used policy as part of our paper Dreibholz, Rathgeb and Tüxen (2005) as well as of the Internet Draft Tüxen and Dreibholz (2005) (Individual Submission) including the definition of pool policies, performance results have been presented at the 60th IETF Meeting (see Dreibholz (2004c)). After that, the draft has become the Working Group Draft Tüxen and Dreibholz (2006b) of the IETF RSerPool WG.

8.13 Summary

In this chapter, a generic application model as well as performance metrics have been introduced first. After that, simulative performance evaluations of RSerPool systems have been performed in order to identify critical parameter ranges. Important results of these analyses have been:

- An estimation of the workload parameter influence on the pool policy performance,
- The identification of fallacies and pitfalls for the Round Robin policies,
- The definition of distance-sensitive pool policies,
- Guidelines for the usage of the PU-side cache and
- An optimized Least Used policy for heterogeneous server capacity scenarios.

Results of these evaluations, optimizations and extensions have been contributed into the IETF's RSerPool standardization process.

Chapter 9

RSerPool Failure Scenario Results

GOAL of this chapter is the evaluation of failure scenarios, i.e. the use case for which RSerPool has been designed for. While in practise PEs as well as PUs and PRs can fail, the focus of the analyses in this chapter is the most important case: the failure of PEs.

9.1 Introduction

From the user's perspective (i.e. in the view of a PU), a PE failure is handled by the steps illustrated in figure 9.1: during normal operation, the application sets checkpoints for the session state to return to in case of problems (e.g. by sending state cookies if applying client-based state sharing for this task, see also subsubsection 3.9.5.2 and Dreibholz (2002)). However, all work performed by the server between the latest checkpoint and a PE failure will be lost[1].

The failure of the PE will be detected by the PU side using mechanisms being appropriate for the application. In the usual case, the failure detection is provided by timeouts, which could e.g. be based on SCTP heartbeats, features of the application protocol itself, or both. In case of the simulated calculation application, it is provided by the session keep-alive mechanism using CalcAppKeepAlive and CalcAppKeepAliveAck messages (see section 8.3).

[1]Of course, all work is lost if there is no checkpoint.

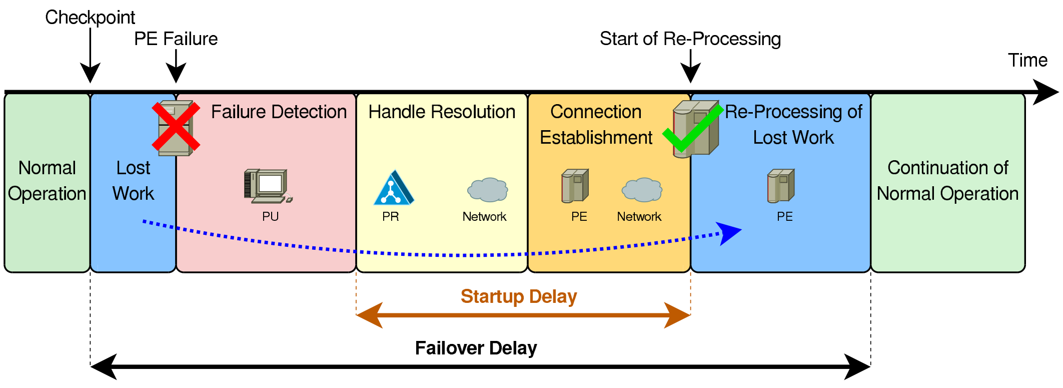


Figure 9.1: The Detection and Handling of a Pool Element Failure

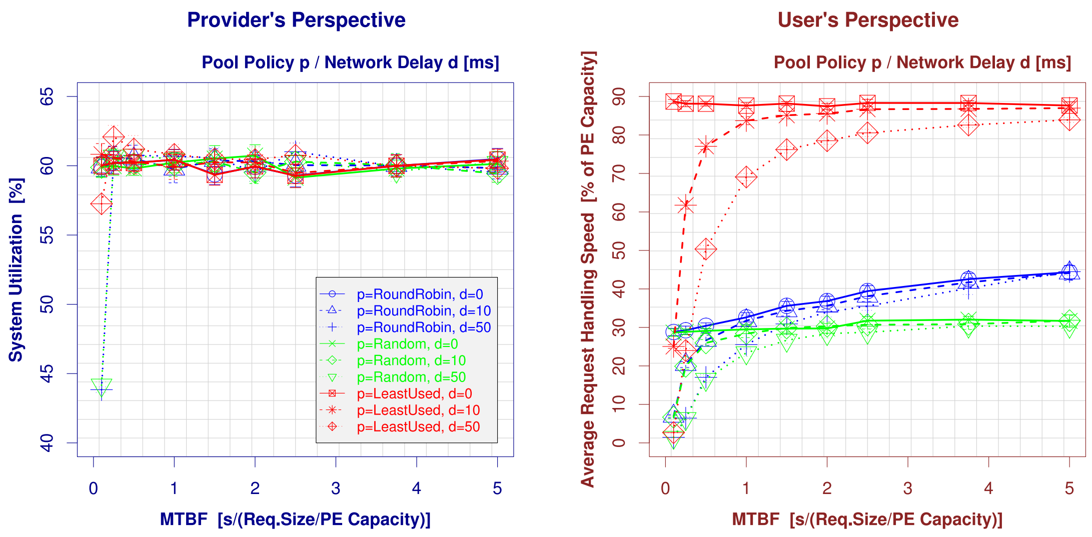


Figure 9.2: The Failover Performance of Dynamic Pools

After the detection of a server failure, a new PE has to be chosen (by applying the handle resolution procedure, see subsubsection 3.9.3.1), a connection to the selected PE has to be established, the session state saved by the latest checkpoint has to be restored and the lost work (i.e. the work not saved by a checkpoint) to be re-processed. After that, the session can continue its normal operation. The delay between leaving the normal operation state and reaching it again is denoted as *Failover Delay*.

Clearly, the goal of a well-provisioned RSerPool system is to minimize the failover delay. The approaches to reach this goal are to

- Reduce the time required for failure detection as well as failover initiation and
- To minimize the amount of lost work (and therefore to keep the re-processing effort as small as possible).

While the last item is related to the Application Layer and tightly connected to the mechanism used for session failover (e.g. the client-based state sharing as described in subsubsection 3.9.5.2), the first item is related to the Session Layer only. Therefore, it is useful to differentiate between these two parts of the failover procedure. In the following sections 9.2 and 9.3, the failure detection mechanisms are analysed. This is followed by an evaluation of the two most important failover mechanisms: the "abort and restart" principle in subsection 9.4.1 and client-based state sharing in subsection 9.4.2.

9.2 Using Dynamic Pools

A basic feature of the RSerPool architecture is its support for dynamic pools. That is, new PEs can register into a pool and registered PEs can deregister from their pool. In particular, this functionality allows for dynamically adapting a pool's capacity to the current user requirements. This feature is highly useful for application cases like real-time distributed computing, as described in subsection 3.6.5. While the deregistration of a PE in a dynamic pool is not actually a failure, it is nevertheless necessary to perform a failover: in case of a long-lasting session, it is usually not possible for the PE to finish all of its sessions. That is, a PE change due to an intentional deregistration can be viewed as "controlled failure" of a PE.

Such a controlled failure is the simplest case for a failover: the PE can tell its PUs about the oncoming shutdown, and the PUs can take the appropriate actions. In the usual case, this in particular means to set a checkpoint to seamlessly resume the session on a different PE. This avoids any lost work and re-processing effort. That is, a controlled failure only results in the costs (i.e. usually time) for handle resolution and resumption of the session on the new PE. Since the endpoint latencies are negligible, the system performance in such scenarios is mainly influenced by the network delay only.

To make the effects clear, an example simulation based on the scenario setup described in section 8.6 has been performed. The following parameters have been used: a target system utilization of 60%, a simulated real-time of 60 minutes, a PU:PE ratio of 10 and a request size:PE capacity ratio of 10 (i.e. an average-sized request is processed within 10s). The *Mean Time Between Failure* (MTBF) of the PEs has been varied between 0.1 and 5 times the request size:PE capacity ratio, having used a negative exponential distribution. That is, a value of 1 means that a PE goes down in an interval equal to the time required to exclusively process a request of the average size.

Figure 9.2 presents the performance results of the simulation. Clearly, the effects of PE failures for high settings of the MTBF are almost negligible. Therefore, this simulation has in particular used very low settings. A use case where small MTBF values may be likely is real-time distributed computing, where user PCs provide their capacity to a pool if idle and remove themselves when utilized again (see also subsection 3.6.5).

The left-hand side of figure 9.2 contains the plots for the system utilization. Clearly, the effects of dynamic pools are small, except in case of very low MTBF settings occurring in combination with network delay (here: a MTBF setting of 0.1 and an inter-component network latency of 50ms). In this case, the PUs spend a significant fraction of the handling time for handle resolution and session resumption (i.e. startup delay, see figure 9.1), leaving the PEs idle. Note, that the negative effect for the Least Used policy is smaller than for the Round Robin and Random policies (e.g. 57% vs. 44% utilization for a MTBF setting of 0.1 and a network delay of 50ms): as already observed and described in section 8.7, the load distribution performance of Least Used is significantly better. Therefore, it is easier to utilize the currently available resources.

However, from the user's performance perspective on the handling speed side – shown on the right-hand side of figure 9.2[2] – the effects of dynamic pools can be observed earlier (i.e. even for higher settings of the MTBF): as expected, the handling speed decreases with a smaller MTBF and higher network delay (i.e. more costly PE changes). However, an interesting observation is the behaviour of the Round Robin policy: while this policy provides a better performance than Random in the usual case (e.g. a handling speed of 40% vs. about 30% for Random for a MTBF value of 5; see also section 8.7 for a detailed evaluation of the two policies), the handling speed of Round Robin converges to the value of Random for lower MTBF values. The explanation is quite clear: the idea of Round Robin is to choose PEs in turn. However, for a low MTBF value, PEs appear and disappear frequently. That is, there is no stable list to select the PEs from. Instead, the choice becomes more and more random.

In summary, the "controlled failover" for dynamic pools is clearly the ideal case. An idea to further enhance the failover performance in case of very dynamic pools is that a PE going to shut down shortly issues a "shutdown soon" notification to its PUs some seconds before actually going out of service. This allows for the PUs to select a new PE and establish a connection to it in parallel, while the old PE is still active. Then, a new PE is already in stand-by when the old PE is actually finishing

[2]The legend has been omitted to enhance readability. It is equal to the legend of the plot on the left-hand side.

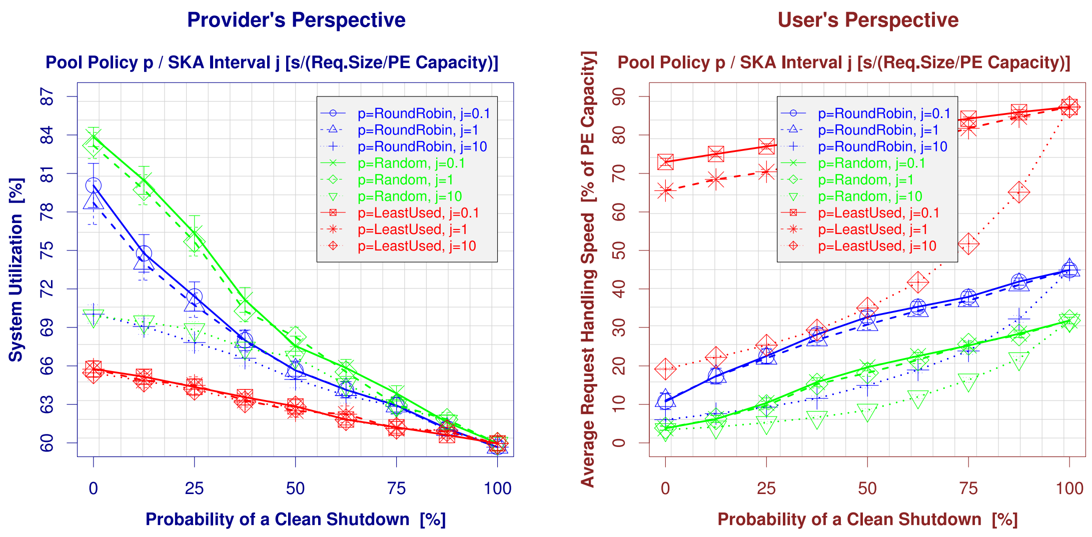


Figure 9.3: The Impact of Clean Shutdowns

its shutdown by setting the last checkpoint and tearing down the transport connection.

9.3 The Handling of Pool Element Failures

A short overview of the impacts of real PE failures has already been presented in Dreibholz and Rathgeb (2005d). However, this subject involves multiple parameters to be tuned, therefore it is necessary to have a more detailed look.

9.3.1 Introducing Server Failures

The first parameter to be evaluated is the probability that a PE performs a “clean” shutdown before disappearing (see also section 8.3). Such a proper shutdown includes a deregistration at its PR-H and the transmission of a state cookie for each session. In case of an “unclean” shutdown, the PE simply disappears without notification. In order to illustrate the essential effects, a simulation based on the scenario setup described in section 8.6 has been performed. Figure 9.3 shows the results of this simulation which has used the following parameters:

- A simulated real-time of 60 minutes,
- An average inter-component network delay of 10ms,
- A target system utilization of 60%,
- A PU:PE ratio of 10 and a request size:PE capacity ratio of 10,
- An average PE MTBF of 5 times the request size:PE capacity ratio (i.e. 50s here; negative exponential distribution),
- MAX-BAD-PE-REPORT has been set to 1 (to be evaluated later in subsection 9.3.4),

- A session keep-alive interval of 1s (to be evaluated in detail in subsection 9.3.2) and a session keep-alive timeout of 1s,
- An ASAP Endpoint Keep-Alive transmission interval of 1s (to be evaluated in detail in subsection 9.3.3) and an ASAP Endpoint Keep-Alive Ack reception timeout of 1s and
- A cookie-based failover has been realized using a cookie transmission interval of 10% of the average request size (i.e. at most about 10% of the work is lost in case of a PE failure; to be evaluated in detail in subsection 9.4.2).

The right-hand side of figure 9.3 shows the average request handling speed for having varied the probability of a clean shutdown, for the policies Least Used, Round Robin and Random. As second parameter, the session keep-alive interval has been varied. This keep-alive interval denotes the transmission interval of the CalcAppKeepAlive messages, as defined in section 8.3. It is given as fraction of the request size:PE capacity ratio. That is, an interval of 0.1 denotes a session keep-alive interval of 10% of the request processing time if the request is processed exclusively.

Clearly, the results for the handling speed are as expected: the higher the fraction of PEs silently disappearing, the lower the handling speed. Furthermore, the ranking of the three policies remains as observed in section 8.7. That is, Least Used is able to better distribute the workload, leading to a better compensation for lost work (e.g. a handling speed of still more than 70% vs. about 10% for Round Robin and 5% for Random at 0% probability for a clean shutdown). This effect can be verified by the results for the average system utilization shown on the left-hand side of figure 9.3: while the average system utilization for Least Used rises from 60% (i.e. the desired target utilization) for clean shutdowns to only about 66% for no clean shutdowns, the utilization for Round Robin reaches 80%, while the Random policy already exceeds 84%. The slower the handling speed, the higher the probability that a resumed session is interrupted again, the more calculations are lost and have to be re-processed.

Furthermore, the session keep-alive monitoring interval (here: in form of the CalcAppKeepAlive message transmission) is a crucial parameter for the handling speed: a too high value (e.g. 10) extends the time the PU requires to detect the PE failure. That is, while the PU does not cause server load (the increase of the system utilization remains small), the handling speed is strongly affected. For example, an interval of 10 instead of 1 decreases the handling speed of Least Used from more than 65% to less than 20% in the scenario of no clean shutdowns.

In summary, it has been observed that the ranking of the policies (Least Used is better than Round Robin is better than Random) is also valid in failure scenarios. Furthermore, the monitoring granularity of the session (here: the CalcAppKeepAlives) is crucial for the handling speed, due to its influence on the timeliness of the PE failure detection.

9.3.2 Session Monitoring by the Pool User

The process of monitoring a session directly influences the speed of detecting a PE failure (see also figure 9.1). Arbitrary mechanisms can be used to actually realize such a monitoring functionality (e.g. transaction timeouts or SCTP heartbeats); for the RSPSIM simulation model and its calculation application, this mechanism is provided by the session keep-alive mechanism using CalcAppKeepAlive messages (see section 8.3). In order to present the general effects introduced by a variation of the session keep-alive interval, the results of an example simulation are presented in figure 9.4. The simulation has been based on the scenario setup described in section 8.6, the following parameter settings have been used:

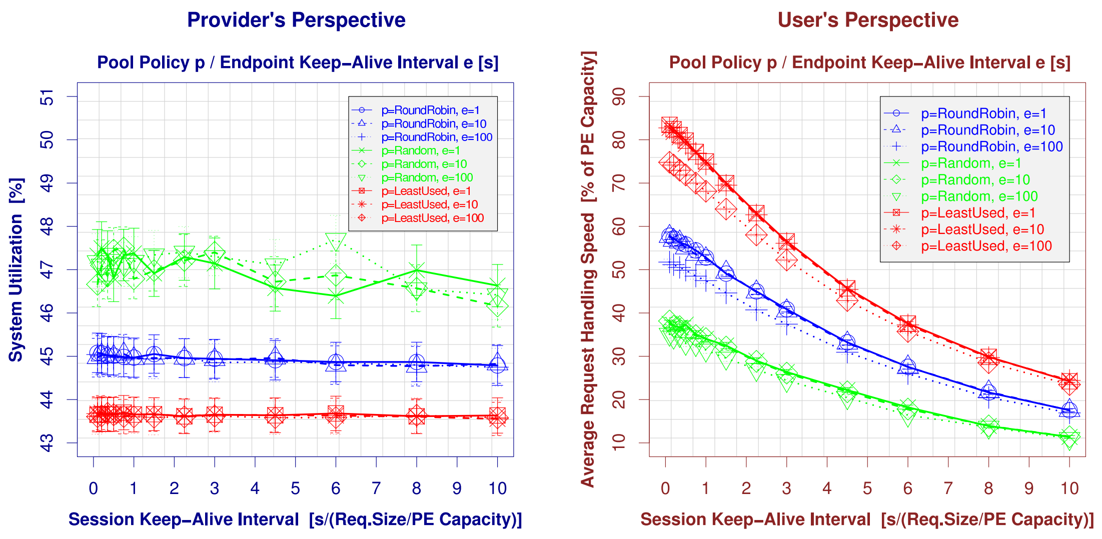


Figure 9.4: The Impact of the Session Keep-Alive Interval for Session Monitoring

- A simulated real-time of 60 minutes,
- An average inter-component network delay of 10ms,
- A target system utilization of 40% (in order to illustratively show the effect on the handling speed; for a larger value, it would already be too strong),
- A PU:PE ratio of 10 and a request size:PE capacity ratio of 10,
- An average PE MTBF of 5 times the request size:PE capacity ratio (i.e. 50s here; negative exponential distribution),
- A probability for a clean shutdown of 0%,
- MAX-BAD-PE-REPORT=∞ (to be evaluated later in subsection 9.3.4),
- A session keep-alive timeout of 1s,
- An ASAP Endpoint Keep-Alive Ack reception timeout of 1s and
- A cookie transmission interval of 10% of the average request size (to be evaluated in detail in subsection 9.4.2).

The effect of varying the session keep-alive interval (i.e. the transmission interval for CalcApp-KeepAlive messages) on the average system utilization, presented on the left-hand side of figure 9.4, is fairly small: neither the session monitoring granularity nor the transmission interval of the ASAP Endpoint Keep-Alive messages from the PR-Hs to the PEs have a significant influence on the system utilization. The reason is obvious: a PU waiting for failure detection does not consume PE capacity. Only the ranking of the policies can be observed, i.e. while Least Used achieves an utilization of 44%, the utilization of Round Robin is about 45.5% and about 47% for the Random policy. Again, the Least Used policy achieves a better load distribution, reducing the amount of lost work and therefore the need for an extensive re-processing effort.

While the results for the average handling speed, shown on the right-hand side of figure 9.4, obviously reflect the ranking of the policies, a significant performance loss is induced by a larger session monitoring granularity. Clearly, the more time a PU requires to detect a PE failure, the more time is needed to handle a request.

Having a look at the transmission interval of the ASAP Keep-Alive messages, only a very small influence can be observed. This parameter therefore needs a more detailed evaluation, which will be presented in the following subsection 9.3.3.

In summary, the frequent monitoring of a session is extremely crucial for the performance of a RSerPool system: the quicker a PE failure is detected, the shorter the failover delay. Note, that while sending keep-alive messages is a common mechanism to realize such a monitoring functionality, the application protocol itself may provide features to realize monitoring much more efficiently. Consider for example a database transaction application or the simple download of web pages. Here, monitoring could be simply realized as a timeout for issued transactions or download requests. That is, monitoring does not necessarily impose any additional traffic on the network for such applications.

While the monitoring of sessions is a must, RSerPool provides another feature to observe the health of PEs: the ASAP Endpoint Keep-Alive monitoring. But is this additional monitoring feature really required?

9.3.3 Pool Element Monitoring by the Home Registrar

The PE monitoring using ASAP Endpoint Keep-Alive messages (see subsubsection 3.7.1.3), performed by a PE's PR-H, is used to ensure that the handlespace content (i.e. the set of usable PEs) reflects reality with a high probability. While the session keep-alive monitoring granularity (as evaluated in subsection 9.3.2) reduces the time a PU requires to detect a PE failure, the Endpoint Keep-Alive monitoring interval reduces the startup time, i.e. the time required to start or resume a request on a selected PE (see figure 9.1). The startup time becomes negligible for a large request size:PE capacity ratio (i.e. the processing time dominates the startup time; see also the results of the simulation in subsection 9.3.2). But as the request size gets smaller, a fast startup gains increasing importance.

9.3.3.1 General Effects of the Endpoint Keep-Alive Monitoring

To illustrate the influence of the Endpoint Keep-Alive transmission interval, the results of an example simulation are presented in figure 9.5. This simulation has been based on the scenario setup described in section 8.6; the parameter settings have been as follows:

- A simulated real-time of 60 minutes,
- An average inter-component network delay of 10ms,
- A target system utilization of 25% (in order to have sufficient over-capacity to illustratively present the effects),
- A PU:PE ratio of 10 and a request size:PE capacity ratio of 1,
- An average PE MTBF of 2, 5 and 10 times the request size:PE capacity ratio (negative exponential distribution),
- A probability for a clean shutdown of 0%,
- MAX-BAD-PE-REPORT=∞ (to be evaluated later in subsection 9.3.4),

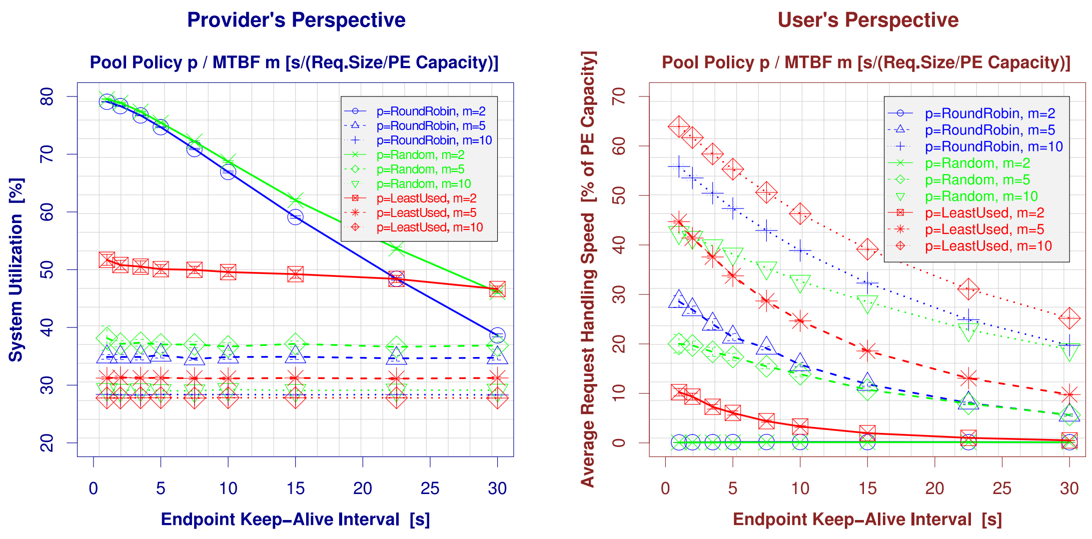


Figure 9.5: The Impact of the Endpoint Keep-Alive Interval for Pool Element Monitoring

- A session keep-alive interval of 1s and a session keep-alive timeout of 1s,
- An ASAP Endpoint Keep-Alive Ack reception timeout of 1s and
- A cookie transmission interval of 10% of the average request size (to be evaluated in detail in subsection 9.4.2).

The results for the average system utilization are presented on the left-hand side of figure 9.5. Again, the ranking of the policies (Least Used is better than Round Robin is better than Random), as already observed for the simulations of the previous subsections, can be noticed. The smaller the average MTBF and the worse the load distribution quality of the policy, the higher the system utilization due to the necessity to re-process lost work. Clearly, the impact of the ASAP Endpoint Keep-Alive transmission interval becomes insignificant if the average PE MTBF is sufficiently high (here: 5 or 10 times the request size:PE capacity ratio).

However, a significant impact on the utilizations of the Round Robin and Random policies can be observed for an average MTBF of 2 times the request size:PE capacity ratio: the utilization is sinking with a rising monitoring interval. The reason for this effect is that the PUs spend a significant fraction of their runtime waiting for service startup: the probability of the Round Robin and Random policies to be affected by a PE failure is significantly higher than for Least Used (since the request processing takes longer due to non-optimal distribution). So, after detecting a PE failure, there will be a second startup delay for contacting a newly chosen PE. If the new PE is still in service, it will start re-processing (i.e. the utilization is increased). Clearly, the probability for this case is much higher if the PR's monitoring interval is small. Otherwise, the new PE may be already out of service. Then, the PU waits for its session keep-alive timeout (no capacity is consumed, i.e. low utilization) and finally tries again.

The effects observed for the utilization are reflected by the results for the average handling speed, shown on the right-hand side of figure 9.5: the handling speed decreases with the ASAP Endpoint Keep-Alive transmission interval. The longer it takes to detect a PE failure, the higher the resulting startup time. This directly implies an increased handling time and therefore a decreased handling speed. Furthermore, it can be observed that the handling speed of Round Robin converges into the direction of the Random speed (because there is no stable list to select from) – as already being observed and explained for the results in section 9.2.

In summary, the Endpoint Keep-Alive mechanism is clearly necessary if the request size:PE capacity ratio is small and therefore the selection of an unavailable PE significantly decreases the handling speed due to an increased service startup time.

9.3.3.2 How Costly is the Endpoint Keep-Alive Monitoring?

Obviously, the PR-H-based monitoring causes network overhead – each ASAP Endpoint Keep-Alive message also has to be answered by an ASAP Endpoint Keep-Alive Ack. Furthermore, while the session keep-alive monitoring may utilize features of the Application Layer protocol, a smaller monitoring interval for the PR-H directly implies additional ASAP overhead traffic.

The per-second cost for the monitoring of a pool clearly depends on the size of the pool and the Endpoint Keep-Alive transmission interval. It can be described by the following formula:

$$c_{\text{EKAMonitoring}}(\text{PEs}, \text{EKAInterval}) = \underbrace{\underbrace{2 * c_{\text{Packet}}}_{\text{Keep-Alive and Acknowledgement}} * \frac{1}{\text{EKAInterval}}}_{\text{Costs per PE}} * \text{PEs}$$

In this formula, EKAInterval denotes the transmission interval (in seconds) and c_{Packet} the per-packet cost. Since each Endpoint Keep-Alive also has to be answered by an ASAP Endpoint Keep-Alive Ack message, this cost factor is doubled.

The size (and therefore the bandwidth cost) of an Endpoint Keep-Alive or Endpoint Keep-Alive Ack message depends on the size of the pool's PH (see subsubsection 3.9.2.2 for a detailed description of the message contents, as well as figure A.1 and figure A.2 for illustrations of the message formats). Assuming an upper limit of 32 bytes for a PH (see subsection 4.4.6 for a detailed discussion of this limit), the size of each of the Keep-Alive packet types could be in the range of 80 to 100 bytes (including IP header and SCTP overhead). Under the assumption that PEs choose a nearby PR-H (e.g. by automatic configuration as described in subsubsection 3.7.1.1), the actual distance between PR-H and PE should usually be small (see also subsubsection 8.10.2.2 for a detailed reasoning of this assumption). Therefore, in a well-designed RSerPool network, this usually means that monitoring traffic keeps local and therefore becomes rather inexpensive.

In reality, however, there are scenarios which are different from the usual case. For the Endpoint Keep-Alive traffic, such an exception could mean that the transport becomes costly (e.g. if the monitoring interval has to be very small, or if bandwidth is scarce). So, is there a possibility to reduce the ASAP monitoring overhead while keeping the handlespace content up-to-date?

9.3.4 Reducing the Pool Element Monitoring Overhead

As it has been shown in subsection 9.3.2, a PU must monitor its session with a PE in order to detect a PE failure. That is, if the PE is already monitored by its PUs anyway, a PR may utilize this feature to save on its own ASAP bandwidth: PUs can report PE failures to their PR using an ASAP Endpoint Unreachable message (see also subsubsection 3.9.3.2). A PR counts the number of failure reports for each PE; if the threshold MAX-BAD-PE-REPORT is reached (see subsubsection 3.7.1.4; the default value is 3), the PE is removed from the handlespace.

To show the effectiveness and essential effects of the failure reporting mechanism, the results of an example simulation are presented in figure 9.6. This simulation has been based again on the scenario setup described in section 8.6 and the following parameters have been used:

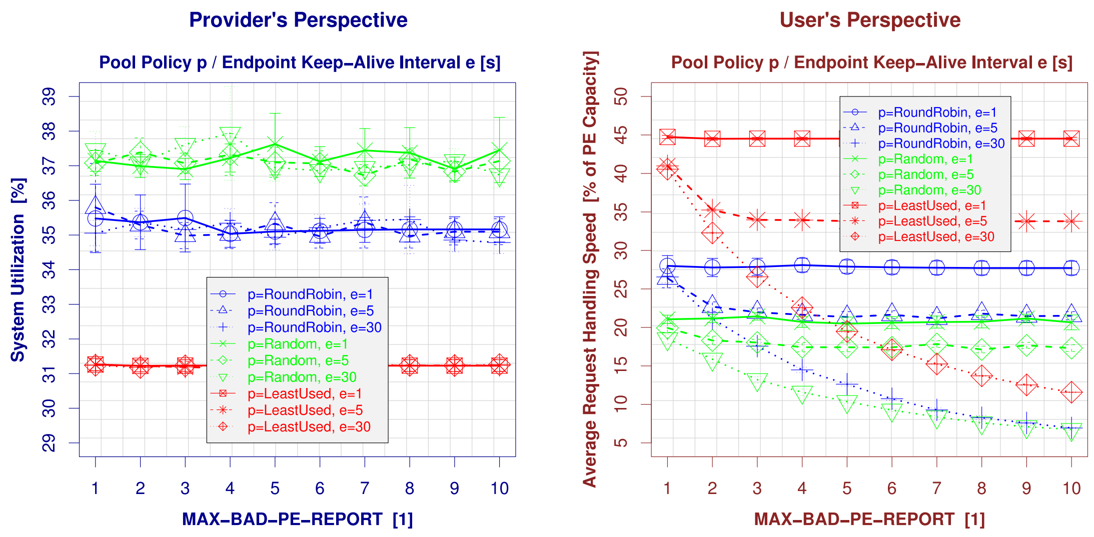


Figure 9.6: Utilizing Failure Reports from Pool Users for Pool Element Monitoring

- A simulated real-time of 60 minutes,
- An average inter-component network delay of 10ms,
- A target system utilization of 25% (for the results to be comparable to the simulation in subsection 9.3.3),
- A PU:PE ratio of 10 and a request size:PE capacity ratio of 1,
- An average PE MTBF of 5 times the request size:PE capacity ratio (negative exponential distribution),
- A probability for a clean shutdown of 0%,
- A session keep-alive interval of 1s and a session keep-alive timeout of 1s,
- An ASAP Endpoint Keep-Alive Ack reception timeout of 1s and
- A cookie transmission interval of 10% of the average request size (to be evaluated in detail in subsection 9.4.2).

Clearly, the results for the utilization – shown on the left-hand side of figure 9.6 – are not very surprising: while the policy ranking as explained in subsection 9.3.1 can be observed again, there is no significant impact introduced by the variations of MAX-BAD-PE-REPORT and the ASAP Endpoint Keep-Alive transmission interval. This coincides with the expectations from the simulation of subsection 9.3.3 for the non-critical MTBF setting (here: 5 times the request size:PE capacity ratio).

However, the setting of MAX-BAD-PE-REPORT has a significant impact on the average handling speed, as shown on the right-hand side of figure 9.6: while MAX-BAD-PE-REPORT has no impact if the ASAP Endpoint Keep-Alive interval is low (here: 1s), a higher setting of MAX-BAD-PE-REPORT in combination with a longer monitoring interval causes a significant performance drop. For example, the Least Used handling speed sinks from about 41% at MAX-BAD-PE-REPORT=1 to less that 12% at MAX-BAD-PE-REPORT=10.

But comparing the average handling speed results for monitoring intervals of 30s and 1s at MAX-BAD-PE-REPORT=1, only small differences can be noticed: 41% vs. 44% for Least Used, 27% vs. 28% for Round Robin and 18% vs. 21% for the Random policy. That is, although the ASAP monitoring traffic has been reduced to $\frac{1}{30}$th, there is only a small drop in performance.

Therefore, using MAX-BAD-PE-REPORT can greatly reduce the number of ASAP monitoring packets. In particular, if the application protocol already includes features for session monitoring (e.g. transaction timeouts), the PU-based monitoring may actually come for free. However, it would be a fallacy to generally recommend using MAX-BAD-PE-REPORT in favour of ASAP-based endpoint monitoring: the failure reports give PUs the power to impeach PEs from the handlespace. This is clearly a major security threat in scenarios of non-trustworthy PUs. Therefore, it is necessary to carefully plan the usage of failure reports for PE monitoring.

9.4 The Evaluation of Session Failover Mechanisms

After having evaluated the mechanisms to detect a PE failure in the previous section, two important mechanisms for the session failover are analysed: the abort-and-restart principle and client-based state sharing.

9.4.1 Abort and Restart

Clearly, the easiest mechanism for a session failover is to simply restart the session. This mechanism is therefore denoted as *Abort and Restart* principle. The most important application using this mechanism is the download of web pages using the HTTP protocol (see Fielding et al. (1999)). Since the "abort and restart" approach simply restarts a session from scratch, its performance is obviously depending on the request size:PE capacity ratio (i.e. being proportional to the amount of work lost in case of a PE failure) and of course on the average MTBF of the PEs.

To show the essential effects of applying "abort and restart", the results of an example simulation are presented in figure 9.7. Again, the simulation has been based on the scenario setup described in section 8.6, having used the following parameters:

- A simulated real-time of 60 minutes,
- An average inter-component network delay of 10ms,
- A target system utilization of 25% (to have over-capacity for the handling of PE failures),
- A PU:PE ratio of 10,
- An average PE MTBF of 2, 5 and 100 times the time required to exclusively process a request having a request size:PE capacity of 10 (here: 20s, 50s and 1000s; negative exponential distribution),
- A probability for a clean shutdown of 0%,
- MAX-BAD-PE-REPORT has been set to 1,
- A session keep-alive interval of 1s and a session keep-alive timeout of 1s,
- An ASAP Endpoint Keep-Alive transmission interval of 1s and an ASAP Endpoint Keep-Alive Ack reception timeout of 1s.

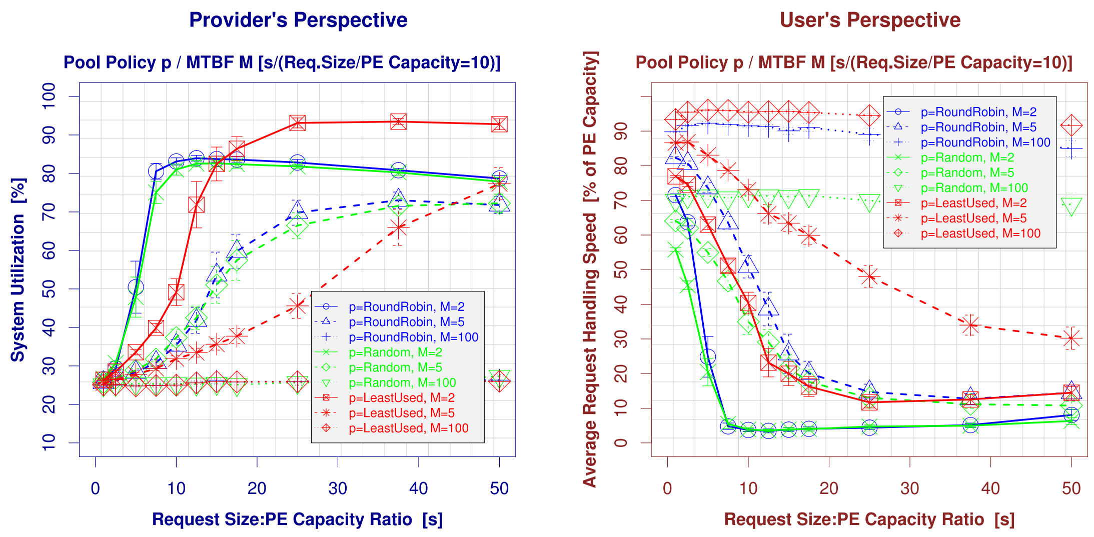


Figure 9.7: Using the “Abort and Restart” Principle for the Session Failover

As expected, the impact of using “abort and restart” on the average system utilization (shown on the left-hand side of figure 9.7) is small, as long as the PEs are sufficiently available (i.e. the MTBF is high enough): for a MTBF of 100 times the time required to exclusively process a request having a request size:PE capacity of 10, the utilization increment is almost invisible. Furthermore, the decrement of the handling speed (presented on the right-hand side of figure 9.7) is also small. Clearly, the rare necessity to restart a session has no significant impact on the system performance.

However, for a sufficiently small MTBF, the results change: at a MTBF of 5, a significant utilization rise – as well as a handling speed decrease – can be observed for the Round Robin and Random policies if the request size:PE capacity ratio is increased. The effect on the Least Used policy is smaller: as it has been explained for the workload parameter evaluations in section 8.7 and as it is expected from the dynamic pool performance results of section 9.2, this policy is able to provide a better processing speed due to superior request load distribution. That is, the probability for a request of a fixed size to be affected by PE failures is smaller if using Least Used instead of Round Robin and Random. Note furthermore that the utilization of Least Used almost reaches 95% for a MTBF of 2 and larger request size:PE capacity ratios, due to its better load distribution capabilities. In the same situation – i.e. high overload, of course – the Round Robin and Random policies only achieve an utilization of less than 85%.

In summary, while the abort-and-restart mechanism is fairly simple and useful in case of short transactions and sufficiently available servers, it is obvious that the longer the duration of the requests being processed, the more inefficient is a failover using the abort-and-restart approach. In such cases, it is instead useful to define checkpoints and allow for the session to be resumed from the latest checkpoint.

9.4.2 Client-Based State Sharing

Client-based state sharing, using state cookies as introduced in subsubsection 3.9.5.2, provides a simple mechanism for a server to set checkpoints to recover the session state from by sending the state in form of a state cookie to the client side. Clearly, the interval in which checkpoints are set (and therefore cookies are transmitted) limits the effort of re-processing.

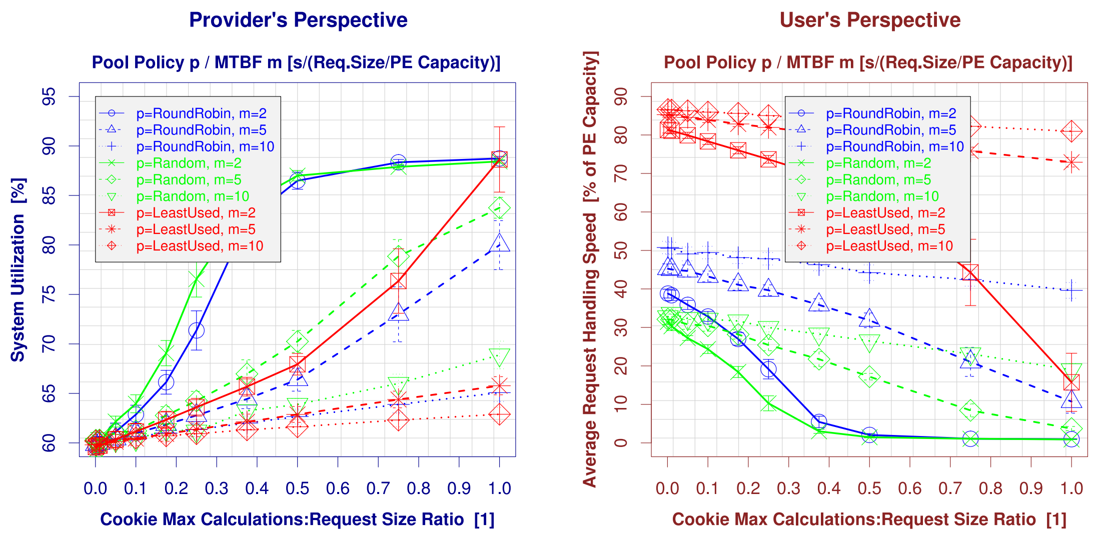


Figure 9.8: Using State Cookies for the Session Failover

9.4.2.1 General Effects of a Cookie-Based Failover

In order to present the essential effects of applying client-based state sharing for session failover, the results of an example simulation are presented in figure 9.8. As for the previous simulations, it has again been based on the scenario setup described in section 8.6 and has used the following parameter settings:

- A simulated real-time of 60 minutes,
- An average inter-component network delay of 10ms,
- A target system utilization of 60%,
- A PU:PE ratio of 10 and a request size:PE capacity ratio of 10,
- An average PE MTBF of 2, 5 and 10 times the request size:PE capacity ratio (negative exponential distribution),
- A probability for a clean shutdown of 0%,
- MAX-BAD-PE-REPORT has been set to 1,
- A session keep-alive interval of 1s and a session keep-alive timeout of 1s,
- An ASAP Endpoint Keep-Alive transmission interval of 1s and an ASAP Endpoint Keep-Alive Ack reception timeout of 1s.

Having a look at the results for the average system utilization (left-hand part of figure 9.8) and the average request handling speed (right-hand part of figure 9.8), the expected effects can be observed: the less frequent the transmission of cookies, the higher the system utilization due to the re-processing of lost work and the lower the request handling speed. Again, the ranking of the policies as explained in subsection 9.3.1 is shown: the Least Used policy is clearly better able to cope with critical parameter ranges (here: a small MTBF and a high cookie interval). Due to its better request distribution

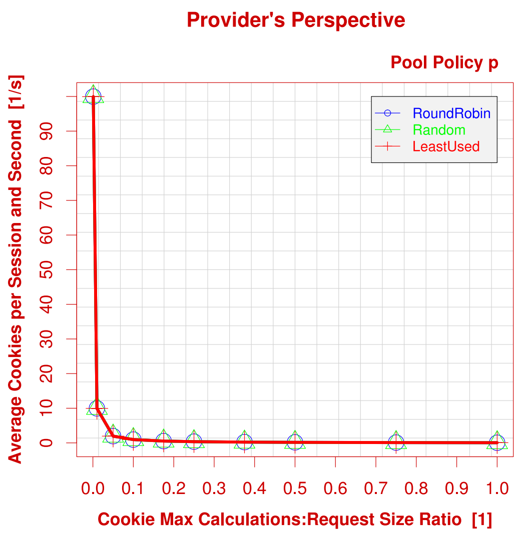


Figure 9.9: The Number of State Cookies for the Session Failover

capabilities, the resulting handling speed is higher. This reduces the probability that for a fixed MTBF a request is actually affected by a failure.

The most interesting curves are presented in figure 9.9; they show the number of cookies per request (only for the MTBF set to 2 times the request size:PE capacity ratio; the results for other MTBF settings are almost similar and have been omitted to enhance readability). Comparing the number of cookies per request with the system performance, it is clearly visible that a too small cookie interval provides no benefit and instead only causes a significant number of overhead packets. So, the resulting question clearly is: how to set the cookie interval while maintaining a certain level of service resilience in PE failure scenarios?

9.4.2.2 How Costly is a State Cookie?

The cost of transporting a state cookie clearly depends on the cookie size and therefore on the application storing its state into the cookie. In order to give an estimate of realistic cookie sizes, some applications are analysed shortly:

- For the fractal graphics application of the RSPLIB demonstration system (see section 5.7), the state cookie contains the image and calculation parameters, the latest calculation position and some identification information to verify the validity of the cookie. The resulting size is about 100 bytes.

- A download application could store a file name (e.g. about up to 250 bytes for a file in a deep directory hierarchy), some user authentication information (e.g. a user name and password), a cookie signature for security (e.g. 16 bytes for a SHA-256-based authentication code) and of course the latest file position. Therefore, cookie sizes of 300 or even more bytes are realistic.

- Dreibholz (2002) proposes to use state cookies in an e-shop system. In this case, the cookie contains a possibly long order list, authentication information and various other data related to the customer management. Cookie sizes of up to a few thousands of bytes seem to be realistic here.

In addition to the actual payload size of the cookie, the overhead for the ASAP message structure (see figure A.12 for an illustration) has to be added. Depending on the SCTP primary path's Maximum Transmission Unit (MTU), the ASAP Cookie message may have to be segmented (by the SCTP protocol) into several SCTP packets, all including an IPv4 or IPv6 header. The resulting costs for a cookie are therefore a function of the cookie size itself, the path MTU and the SCTP and Network Layer overhead.

9.4.2.3 Limiting the Amount of Cookie Traffic

As already shown in subsubsection 9.4.2.1, the performance of client-based state sharing depends on the pool policy. That is, sending cookies in a certain time interval results in different performance results for each policy. Using the Least Used policy, a request is – for a given MTBF – less frequently affected by a failure than for using Round Robin or Random, due to the fact that the handling speed of the Least Used policy is better. To recommend a cookie transmission interval, it is therefore useful to specify it in terms of calculations (multiples of the average request size) rather than time. Given the cookie transmission interval parameter $\mathrm{CookieMaxCalculations}$ (in the unit of calculations as multiple of the average request size; see also section 8.3), the average loss of calculations (and therefore the re-processing effort) per failure – denoted as AverageLossPerFailure – can be calculated as follows:

$$\mathrm{AverageLossPerFailure} = \frac{\mathrm{CookieMaxCalculations}}{2}. \tag{9.1}$$

That is, the server fails in the middle of a cookie interval on average. For example using an interval of half of the average request size, the calculations of 25% of an average request size are lost upon a failover and have to be re-processed.

Knowing the average loss per failure as well as the MTBF and average PE capacity, the goodput ratio of all performed calculations – i.e. the percentage of calculations not being lost due to failures – can be calculated as follows:

$$\mathrm{GoodputRatio} = \frac{(\mathrm{MTBF} * \mathrm{AverageCapacity}) - \mathrm{AverageLossPerFailure}}{\mathrm{MTBF} * \mathrm{AverageCapacity}}. \tag{9.2}$$

Using equation 9.1 and equation 9.2, it is possible to compute the cookie transmission interval $\mathrm{CookieMaxCalculations}$ (in units of calculations given as multiple of the average request size) necessary to reach a desired goodput ratio (e.g. 95%):

$$\mathrm{CookieMaxCalculations} = -2 * \mathrm{MTBF} * \mathrm{AverageCapacity} * (\mathrm{GoodputRatio} - 1). \tag{9.3}$$

A plot showing the cookies per request (i.e. $\frac{1}{\mathrm{CookieMaxCalculations}}$) is presented in figure 9.10. Clearly, the expectations of the initial simulation in subsubsection 9.4.2.1 are met: even for a MTBF of as low as 2 times the request size:PE capacity ratio, about 10 cookies per request are sufficient to reach a goodput ratio of 97.5%. However, trying to only slightly improve the goodput ratio results in a vast increase in the number of cookies.

The resulting general guideline to set the cookie transmission interval is therefore to estimate the PE MTBF and calculate the cookie interval using equation 9.3 for a reasonable goodput ratio in the range of about 97.5%.

9.4.2.4 Summary

In summary, it has been shown that the client-based state sharing approach provides an efficient mechanism for failover. If used with an appropriate cookie transmission interval, the overhead remains low while a good system performance is achieved.

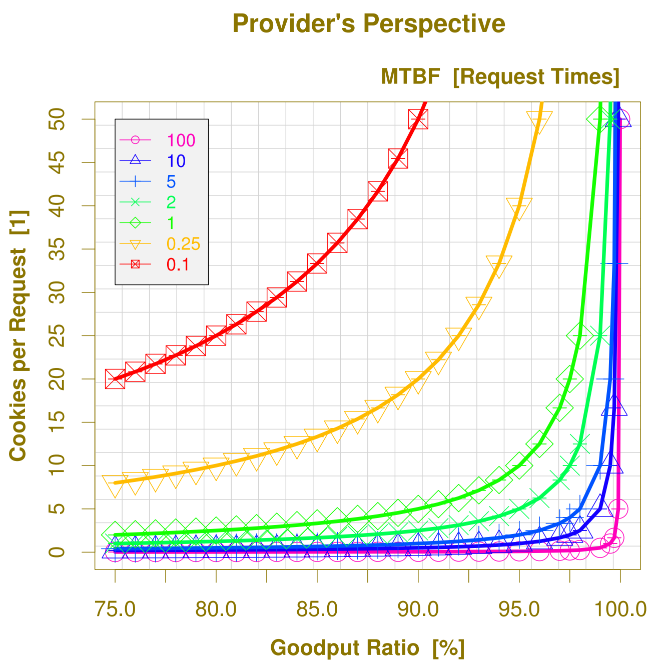


Figure 9.10: The Number of Cookies per Request

9.5 The Handling of Registrar Failures

The failure of PRs affects both, its PEs and PUs. In the usual case, both components detect the failure of a PR and simply select another one. Here, the automatic configuration feature of RSerPool becomes important: PUs and PEs can maintain an up-to-date list of PRs by listening to their announces. If the selection of a new PR is required, it is quickly possible to connect to another one and continue normal operation. Detailed evaluations of the PR hunt mechanism can be found in Uyar et al. (2004), Uyar, Zheng, Fecko and Samtani (2003b,a). In particular, these papers analyse the performance of PR changes in various scenarios. In summary, it is shown that the mechanism is quite effective and PR failures – which can be assumed to be significantly less frequent than PE failures – do not impose a significant performance loss in the usual case.

However, special cases may occur. In particular, the following scenarios need a more detailed evaluation:

- ENRP's auditing and synchronization features in case of PRs having very different views of the handlespace (e.g. due to network problems like congestion for ENRP traffic);
- The effects of takeovers on the system performance and
- The separation and reunion of network parts.

At first sight, the number of ENRP items to be evaluated looks small. But due to the dependency on other system parameters, there are a significant number of cases which have to be evaluated in order to completely cover the topic. For example, a takeover negotiation could fail for some reason. In this case, the PEs themselves detect their PR-H failure at their next re-registration (parameter: re-registration interval) or deregistration. If the policy is adaptive, a re-registration is also applied on policy information update (parameter: choice of policy). Nevertheless, the PEs of the failed PR still remain in the handlespace and are found and usable by PUs. PUs are affected by the PR failure only if PEs of the unavailable PR also fail. In this case, the PEs may be removed upon failure reports (parameter: MAX-BAD-PE-REPORT) or finally upon expiry of their Lifetime Expiry Timer (parameter: registration life).

Due to space and time limitations for this thesis, it is not possible to cover all cases which would clearly be necessary for an expressive analysis of the ENRP topic. Therefore, these items will be the subject of future work on the topic of RSerPool – to be processed in the continuation of our RSerPool project.

9.6 Summary

In this chapter, failure scenarios have been evaluated with the focus on PE failures. The performance of handling a server failure is influenced by two factors:

- The time required to detect a PE failure (and to find and contact a new PE), as well as
- The effort to re-process lost work using an appropriate session failover mechanism.

On the subject of the first item, it has been shown that RSerPool systems include efficient features to detect PE failures. After that, two important session failover mechanisms have been evaluated: the "abort and restart" principle and the client-based state sharing.

Chapter 10

Conclusion and Outlook

FINALLY, in the very last chapter of this thesis, the achieved goals and results – concerning handlespace management, performance evaluations, the prototype implementation and standardization – are summarized in the first part. After that, an outlook to interesting future work and open issues still to be evaluated is presented.

10.1 Achieved Goals and Obtained Results

10.1.1 The Prototype Implementation and the Simulation Model

As first goal of this thesis, the Open Source RSerPool prototype implementation RSPLIB (see chapter 5) has been designed and realized. In order to systematically evaluate the performance of RSerPool systems in a reasonable manner, also the RSPSIM simulation model has been created (see chapter 6). Furthermore, a generic application model used for both, the prototype implementation and the simulation model, has been developed (see section 8.3). Based on this application model, reasonable performance metrics for both, the service provider's perspective (system utilization) and the user's perspective (request handling speed), have been defined (see section 8.5) in order to evaluate RSerPool systems and proposed improvements.

But first, it has been found necessary to design an efficient handlespace management component as a foundation of both, the RSPLIB prototype and the RSPSIM simulation model.

10.1.2 The Handlespace Management

The developed handlespace management approach reduces the effort of maintaining a handlespace to the management of sorted sets, where PE structures are linked according to certain sorting orders (see section 4.4). Pool policies can be realized easily by defining appropriate sorting orders and selection procedures. By performance evaluations using a runtime-based performance metric, it has been shown in chapter 7 that the proposed handlespace management approach is even scalable to large pools of up to 100,000 PEs – with moderate CPU power requirements. Crucial for the efficiency of the handlespace management is the data structure which is actually used to realize the sets. It has been shown that red-black trees are the most suitable structure, while simple binary trees can degenerate and lead to an even worse performance than taking the naïve approach of using linear lists.

Using the new, sophisticated handlespace management approach, the RSPSIM simulation model has become capable of efficiently handling the scenarios necessary to perform reasonable evaluations of the RSerPool system performance.

10.1.3 The Performance of RSerPool Systems

The first part of the RSerPool system performance evaluations has focused on failure-free scenarios, which are (hopefully) the usual operating condition a system is working in for most of its runtime. Therefore, performance in this situation becomes crucial for the cost-benefit ratio of the provided service.

10.1.3.1 Workload Parameters

Using a generic application model and well-defined performance metrics, it has first been useful to evaluate the three basic workload parameters: the number ratio of PUs to PEs (PU:PE ratio), the request interval and the request size. Given a target system utilization and PE capacities, any one of the three workload parameters can be calculated if values for the other two are provided (see equation 8.5 in subsection 8.7.1).

The PU:PE ratio is the most critical workload parameter (see section 8.7): a small value means a high per-PU load. Appropriately scheduling the requests in this case is most difficult and inappropriate PE selections lead to a reduced system utilization and handling speed. Varying the request size together with the PU:PE ratio, it can be observed that the utilization for longer requests is worse (since it is even more difficult to achieve a "good" scheduling), but the handling speed increases (since a single but long request is affected by queuing delay only once, while n small requests are affected n times). The performance results for the two other workload parameter combinations (request size/request interval and request interval/PU:PE ratio) can be deduced.

A general observation on the pool policies is that the Least Used policy provides the best performance, in particular if the parameters are in a critical range – due to its knowledge about PE states (adaptive policy). Non-adaptive policies have a lower performance: since Round Robin tries to select PEs in turn, i.e. the recently selected PE will be chosen again after all other PEs have been used, its performance is usually better than simply applying Random selection. However, in critical parameter ranges, the performance of Round Robin tends to converge to the Random results.

10.1.3.2 Fallacies and Pitfalls of the Round Robin Policies

While Round Robin usually achieves a better performance than Random, some fallacies and pitfalls have been discovered for the Round-Robin-based policies (see section 8.8): a PR replying with multiple PE identities may not advance the Round Robin list pointer by more than one entry. Otherwise, PEs may be systematically skipped – leading to a severe performance degradation. Using the Weighted Round Robin policy, the problem of not having the possibility to specify non-integer weights adds a further problem. Expanding weights leads to an even worse performance than simply using Weighted Random selection. The guideline for this policy is therefore not to use it – except in special, carefully planned cases.

10.1.3.3 Coping with Network Delay

Network delay becomes crucial if it is large in comparison to the request processing time: in this case, the delay severely affects the handling speed. In order to cope with this problem, two distance-sensitive policies have been defined by adding a delay measurement functionality to the PR, based on the RTT already obtained by the SCTP protocol (see subsection 8.10.2). While the ability of the distance-sensitive Weighted Random policy to cope with localized disaster scenarios is limited, it has been shown for the distance-sensitive Least Used policy that it is also able to handle such scenarios (see subsection 8.10.2). After that, the applicability and usefulness of this new Least Used policy has been successfully validated in the real Internet by measurements using the RSPLIB prototype in a PLANETLAB scenario (see subsection 8.10.5).

10.1.3.4 Making Use of the PU-Side Cache

Except for the Random policy, using the PU-side cache results in a decreased system performance. While therefore the general guideline is to avoid it at all (except for the Random policy, of course), the cache becomes very useful if the startup time of requests (i.e. the time between starting a handle resolution and finally getting the request accepted by a server) has a significant contribution to the overall handling time. The usefulness of the cache in such a situation has been demonstrated by an example simulation (see subsection 8.11.2).

10.1.3.5 Scenarios of Heterogeneous Server Capacities

Using heterogeneous server capacities (see subsection 8.12.1), the Round Robin and Random policies clearly achieve a low performance, since the server selection for these policies has no knowledge about the different capacities. The Weighted Random policy can cope with such scenarios if the PU:PE ratio is high enough. Furthermore, it is also able to take advantage of very fast PEs to increase the request handling speed. However, the performance of Least Used – especially for the handling speed in critical parameter ranges – is significantly better. Since Least Used selects PEs based on the current server load, there is a slight performance degradation if the scenario becomes very heterogeneous. In this case, it is possible to select a very slow PE if only its load state is sufficiently low.

The latter observation has led to the following question: what happens in scenarios where the capacity is increased (see subsection 8.12.2), in particular if containing one or more dedicated powerful servers? For the Least Used policy, it has been shown that the request handling speed is significantly exceeded even by Weighted Random if the scenario contains sufficiently fast servers – due to the selection of slow PEs with a low load value. This deficiency has been overcome with the new Priority Least Used policy, which incorporates a PE load increment constant defining how much a new request increases the load of a PE. Using this additional information, a significant performance gain is achieved.

10.1.3.6 Pool Element Failure Detection Mechanisms

In the second part of the performance evaluations, PE failure scenarios have been analysed – which have been the main motivation for the creation of RSerPool. In the ideal case, a PE tells its PU that it is disappearing and sets a checkpoint (see section 9.2). Then, it is only necessary to find a new PE to continue the session. This scenario only becomes critical if the PE MTBF is very low and the time to find a new PE and connect to becomes overly long.

However, such an ideal case is unfortunately very rare and PEs may fail without notification. In this case, it is crucial that a PU is able to quickly detect the death of its PE. This detection mechanism (see subsection 9.3.2) can be based on SCTP heartbeats, keep-alive messages or on an application feature like the timeout for a transaction. In the latter case, this mechanism actually does not impose any additional overhead traffic on the network.

The PR-H-based ASAP Endpoint Keep-Alive monitoring of its PEs is important if the request startup time significantly contributes to the request handling time (see subsection 9.3.3), i.e. if there is a significant chance that a just selected PE is already out of service. In this case, a small Endpoint

Keep-Alive interval becomes crucial for the system performance. However, the smaller the interval, the larger the number of overhead messages. Using failure reports by PUs, a PR can remove a PE from the handlespace as soon as a certain threshold of failure reports has been reached. This mechanism is almost as effective as a frequent ASAP Endpoint Keep-Alive monitoring, but significantly saves on overhead bandwidth. Unfortunately, this mechanism can only be applied if the PUs are considered trustworthy. Otherwise, it would allow a malicious PU to impeach PEs and easily cause at least a severe service degradation.

The ranking of the policies, which has already been observed and explained for the failure-free case, also remains in failure scenarios: the Least Used policy provides the best performance, due to its knowledge of the PEs' load states. This leads to higher requests handling speeds and reduces the probability that – for a given average PE MTBF – a request is affected by a failure. While the performance of Round Robin is usually still better than for the Random policy, the handling speed of Round Robin converges into the direction of the Random speed in extreme cases (i.e. if there is no stable list of PE identities to select servers from in turn).

10.1.3.7 Session Failover Mechanisms

In case of a PE failure, all work performed by the PE after having set the last checkpoint is lost. The session failover mechanism used by the application has to ensure that this lost work is being re-processed by a newly chosen PE in order to resume the session. The simplest case for a session failover mechanism is the "abort and restart" principle (see subsection 9.4.1): no checkpoints are set, all work is lost and the session has to start from scratch. Clearly, this approach becomes inefficient for long requests in combination with a small PE MTBF.

Using client-based state sharing, which is provided by the ASAP Session Layer (see subsection 9.4.2), state cookies are used to set checkpoints. Using a reasonable interval for the transmission of state cookies (i.e. the definition of checkpoints), it has been shown in subsection 9.4.2 that an acceptable failover performance can be achieved while keeping the number of overhead packets low.

10.1.4 Standardization and Deployment of RSerPool

Finally, the last goal of this thesis has been to also bring results of the RSerPool implementation experiences and research into the standardization process of the IETF. The first step to reach this goal has been fulfilled by the development of the RSPLIB prototype implementation. This prototype is the world's first, fully-featured and Open Source (under GPL license) implementation of the RSerPool framework. It has become the reference implementation of the IETF RSerPool Working Group.

Furthermore, it has been possible to make contributions to the IETF standardization documents in form of three Working Group drafts (which will become RFCs – i.e. official standards documents – in the near future) and multiple Individual Submission drafts:

- Tüxen and Dreibholz (2006b) (Working Group draft) provides a specification of the RSerPool policies, in particular also including implementation guidelines for the Round Robin policies (to avoid the fallacies and pitfalls found as part of this thesis) and the Priority Least Used policy (a result of evaluating the heterogeneous server capacity scenarios).

- Silverton et al. (2005) (Working Group draft) describes the RSerPool API. This document is a direct result of the experiences made by implementing the RSPLIB prototype, as well as of discussions and interoperability tests at IETF meetings and the RSerPool Bakeoff event.

- Dreibholz, Mulik, Conrad and Pinzhoffer (2006) (Working Group draft) describes the SNMP Management Information Base (MIB) for RSerPool. It is also a result of experiences made with the RSPLIB prototype.

- Dreibholz, Coene and Conrad (2006) (Individual Submission draft) describes the usage of RSerPool for IP Flow Information Export (IPFIX) (see subsection 3.6.3). IPFIX is still under standardization and this application scenario might gain more interest as soon as some IPFIX implementations will be available and deployed.

- Dreibholz (2006a) (Individual Submission draft) proposes to use RSerPool in Distributed Computing scenarios (see subsection 3.6.5). This application scenario is an idea originating from the development of the RSPLIB demonstration system (see section 5.7) in conjunction with client-based state sharing (see subsubsection 3.9.5.2).

- Dreibholz and Pulinthanath (2006) (Individual Submission draft) proposes to use RSerPool for providing SCTP-based mobility in combination with the Dynamic Address Reconfiguration (Add-IP) extension (see subsection 3.6.6) of SCTP. It is partially also a result of our SCTP research (see also Dreibholz, Jungmaier and Tüxen (2003)).

- Dreibholz and Tüxen (2006) (Individual Submission draft) finally describes guidelines for the RSerPool implementation interoperability testing at RSerPool Bakeoff meetings. It is a direct result of the RSPLIB implementation experience and various discussions on the IETF's RSerPool mailing list and at IETF meetings.

As part of our standardization activities, the RSPLIB prototype implementation has been successfully tested for interoperability at the 60th IETF meeting as well as at the first official RSerPool Bakeoff.

Finally, to support the deployment of RSerPool, the RSPLIB prototype and its demonstration system have been presented at some leading conferences and events, including Dreibholz and Tüxen (2003) (Linux Conference Australia), Dreibholz (2004a) (IEEE International Conference on Network Protocols), Dreibholz and Rathgeb (2005a) (IEEE INFOCOM), Dreibholz and Rathgeb (2005b) (IEEE International Conference on Telecommunications), Dreibholz (2005b) (LinuxTag), Dreibholz and Rathgeb (2005c) (IEEE Local Computer Networks Conference) and Dreibholz and Rathgeb (2005e) (IEEE TENCON).

10.2 Outlook and Future Work

While this thesis has evaluated a lot of important and interesting aspects of RSerPool, there are still many more things to be analysed in detail as part of future work.

For the handlespace management, there may be application cases where the number of pools becomes large. As shown in section 7.6, the lookup time for a pool currently depends on the lookup time of a PH in a red-black tree. For a larger number of pools, an additional hash function to speed up the pool lookup might be considered. Furthermore, the implementation currently limits the PH size to 32 bytes; for larger PHs, a hash table may also be useful for a smaller number of pools.

The main subject of future work is clearly the ENRP protocol and its handlespace synchronization features. Open questions on this topic are the following:

- Is it possible to reduce the ENRP Update message overhead, e.g. by aggregating updates or performing selective updates only?

- What happens if the ENRP bandwidth is scarce (due to congestion) and the handlespace views of different PRs become heavily unsynchronized?
- How efficient is the separation and reunion of RSerPool networks handled, e.g. if WAN links between LAN clouds become unavailable?
- Which are useful settings for timeout parameters in real-world network scenarios?
- Can the number of ENRP associations be reduced? For example, instead of having an own association between each PR of an operation scope, a PR-H could only connect to some designated ones. These PRs could then forward the received information.

In particular, it is also necessary to validate simulative results for these questions in real-life, i.e. by measurements using the RSPLIB prototype in PLANETLAB scenarios.

A further subject of future work is to analyse additional state sharing approaches for the Application Layer: different approaches have to be evaluated for certain application scenarios, in order to find improvements and possibly to define extensions for the RSerPool protocols to support certain new mechanisms.

Finally, an important but still open topic of RSerPool is its security. While the current RSerPool standards documents simply move security to existing lower-layer technologies like TLS or IPsec (see section 3.13), this topic should receive some more attention on the RSerPool level itself. In particular, there is the need to clearly identify and evaluate weak points of the protocols and to define mechanisms to decrease the effectiveness of attacks. For example, the PE failure report feature gives PUs the power to impeach PEs. Simply limiting the acceptance rate of such failure reports could already make an attack significantly more difficult.

Appendix A

RSerPool Message Types and Parameters

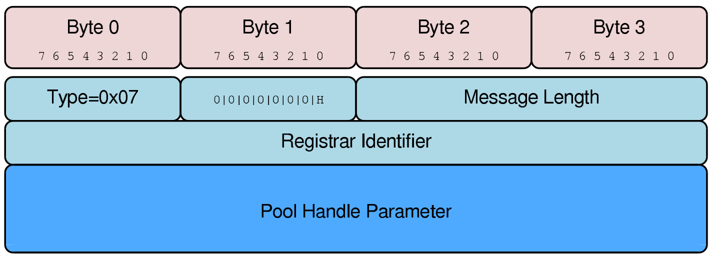


Figure A.1: The ASAP Endpoint Keep-Alive Message

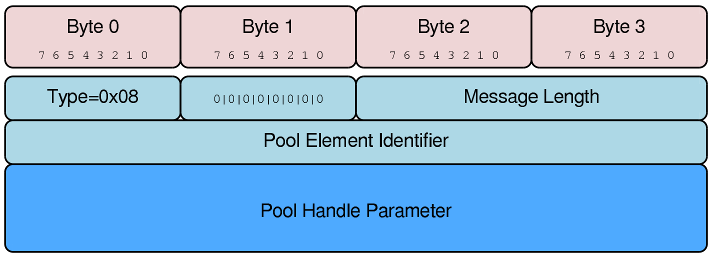


Figure A.2: The ASAP Endpoint Keep-Alive Ack Message

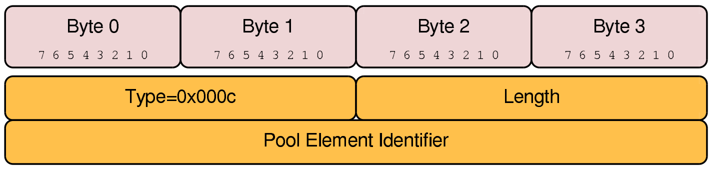


Figure A.3: The Pool Element Identifier Parameter

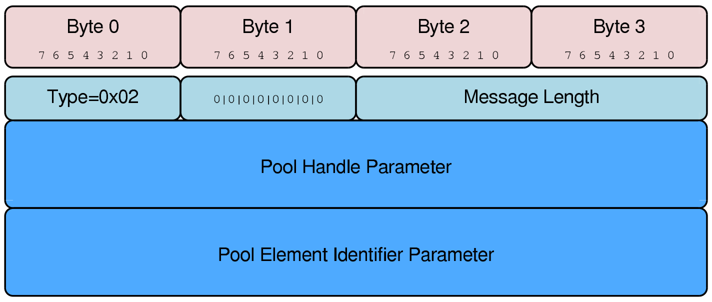


Figure A.4: The ASAP Deregistration Message

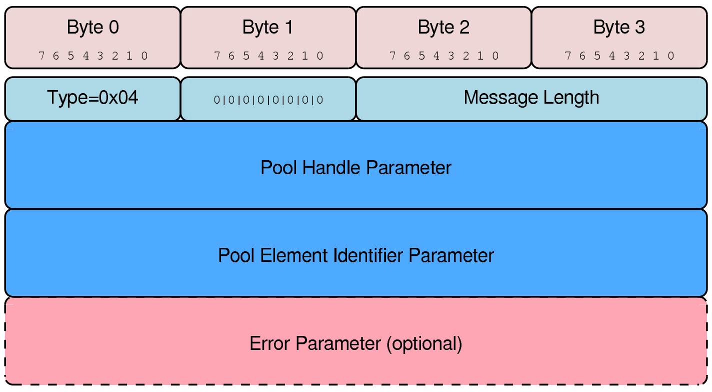


Figure A.5: The ASAP Deregistration Response Message

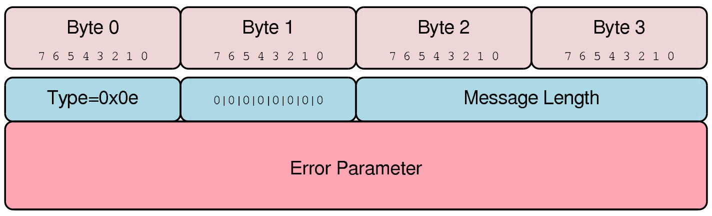


Figure A.6: The ASAP Error Message

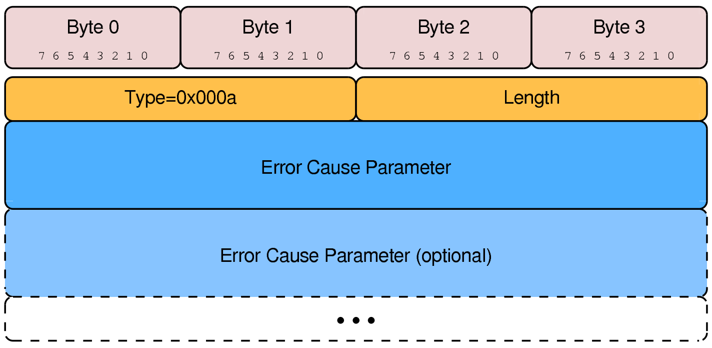


Figure A.7: The Error Parameter

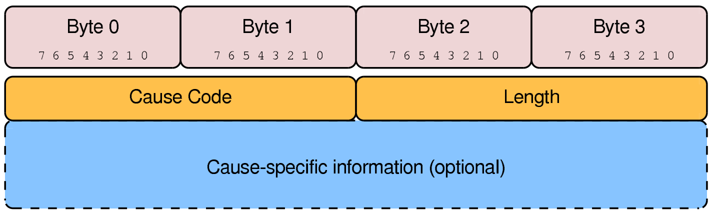


Figure A.8: The Error Cause Parameter

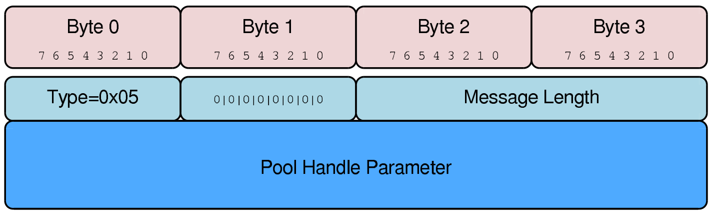


Figure A.9: The ASAP Handle Resolution Message

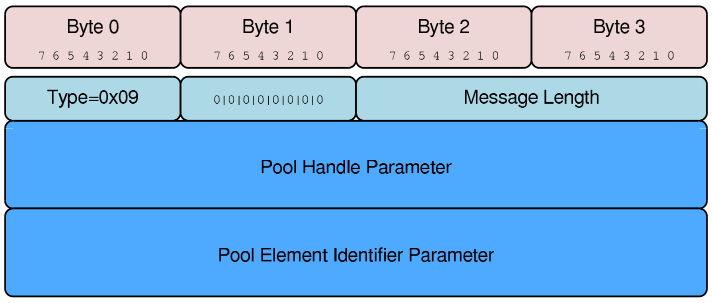


Figure A.10: The ASAP Endpoint Unreachable Message

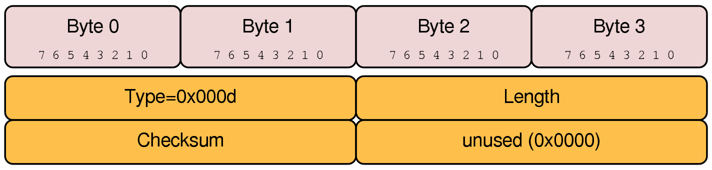


Figure A.11: The Checksum Parameter

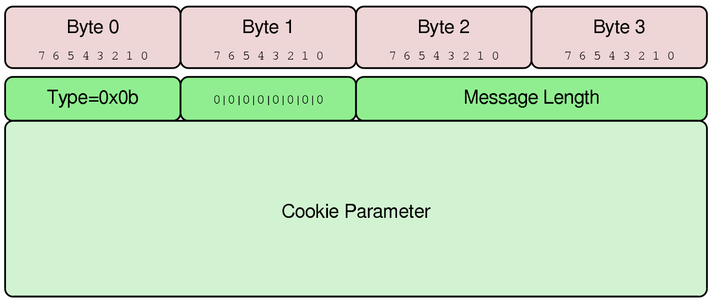


Figure A.12: The ASAP Cookie Message

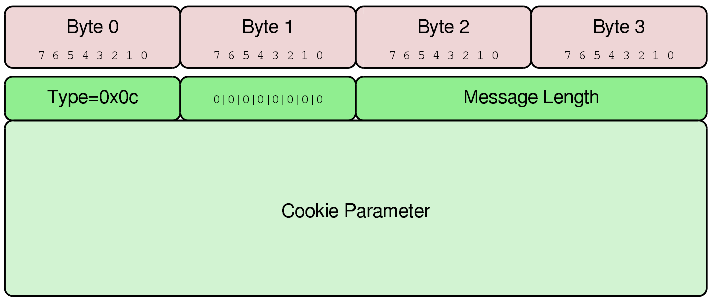


Figure A.13: The ASAP Cookie Echo Message

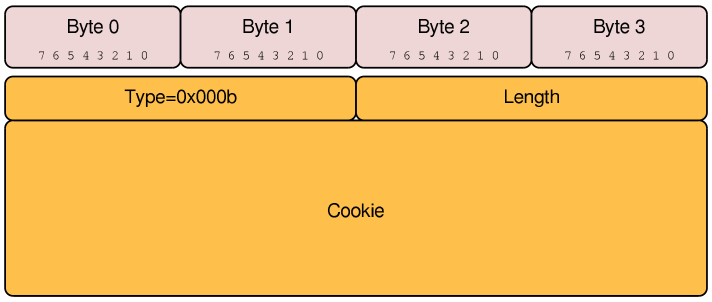


Figure A.14: The Cookie Parameter

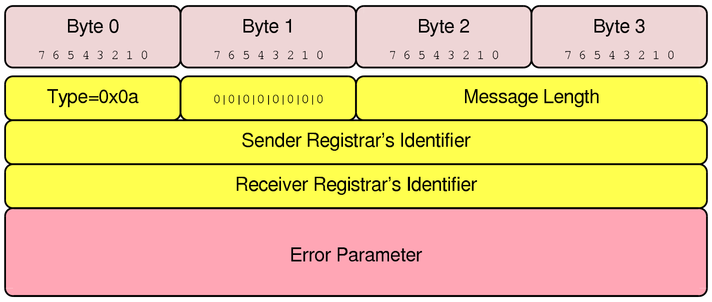


Figure A.15: The ENRP Error Message

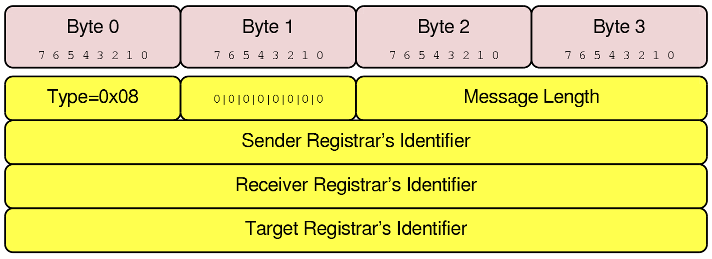


Figure A.16: The ENRP Init Takeover Ack Message

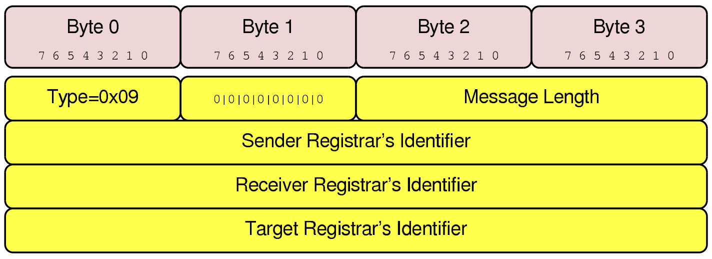


Figure A.17: The ENRP Takeover Server Message

List of Algorithms

List of Figures

List of Tables

Bibliography

Adelson-Velskii, G. M. and Landis, E. M.: 1962, An algorithm for the organization of information, *Soviet Mathematics Doklady*, Vol. 3, pp. 1259–1263. 4.4.7

Alvisi, L., Bressoud, T. C., El-Khashab, A., Marzullo, K. and Zagorodnov, D.: 2001, Wrapping Server-Side TCP to Mask Connection Failures, *Proceedings of the IEEE Infocom 2001*, Vol. 1, Anchorage, Alaska/U.S.A., pp. 329–337. ISBN 0-7803-7016-3.
URL: *http://citeseer.ist.psu.edu/alvisi00wrapping.html* 1.2.1, 1.2.4

Aragon, C. and Seidel, R.: 1989, Randomized search trees, *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pp. 540–545. 4.4.7

AROS Development Team: 2005a, AROS: Amiga Research Operating System.
URL: *http://www.aros.org* 4.4.8

AROS Development Team: 2005b, *AROS Application Development Manual*.
URL: *http://www.aros.org/documentation/developers/application-development.php* 4.4.8

Berger, E.: 2002, *Memory Management for High-Performance Applications*, PhD thesis, The University of Texas at Austin.
URL: *http://citeseer.ist.psu.edu/berger02memory.html* 5

Berger, E. and Browne, J. C.: 1999, Scalable Load Distribution and Load Balancing for Dynamic Parallel Programs, *Proceedings of the International Workshop on Cluster-Based Computing 99*, Rhodes/Greece.
URL: *http://citeseer.ist.psu.edu/berger99scalable.html* 1.2.2, 3.11.1

Berger, E., Zorn, B. and McKinley, K.: 2001, Composing High-Performance Memory Allocators, *SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah/U.S.A., pp. 114–124. ISBN 1-58113-414-2.
URL: *http://citeseer.ist.psu.edu/berger01composing.html* 5

Berger, E., Zorn, B. and McKinley, K.: 2002, Reconsidering Custom Memory Allocation, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) 2002*, Seattle, Washington/U.S.A., pp. 1–12.
URL: *http://citeseer.ist.psu.edu/698680.html* 5

Bhattacharjee, S., Ammar, M. H., Zegura, E. W., Shah, V. and Fei, Z.: 1997, Application-Layer Anycasting, *Proceedings of the IEEE Infocom '97*, Kobe/Japan, pp. 1388–1396. ISBN 0-8186-7780-5.
URL: *http://citeseer.ist.psu.edu/bhattacharjee97applicationlayer.html* 1.2.1

Bivens, A.: 2006, Server/Application State Protocol v1, *Technical Report Version 03*, IETF, Individual Submission. draft-bivens-sasp-03.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-bivens-sasp-03.txt* 3.6.4

Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J. and Wright, T.: 2003, Transport Layer Security (TLS) Extensions, *Standards Track RFC 3546*, IETF.
URL: *http://www.ietf.org/rfc/rfc3546.txt* 4, 9, 4

Bozinovski, M.: 2004, *Fault-tolerant platforms for IP-based Session Control Systems*, PhD thesis, Aalborg University, Aalborg/Denmark.
URL: *http://kom.aau.dk/~marjanb/thesis/PhDthesis.pdf* 3.6.2

Bozinovski, M., Gavrilovska, L. and Prasad, R.: 2003, A State-sharing Mechanism for Providing Reliable SIP Sessions, *Proceedings of the 6th International Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Services*, Vol. 1, Nis/Serbia and Montenegro, pp. 384–387.
URL: *http://kom.aau.dk/~marjanb/papers/telsiks03.pdf* 3.6.2

Bozinovski, M., Gavrilovska, L., Prasad, R. and Schwefel, H.-P.: 2004, Evaluation of a Fault-tolerant Call Control System, *Facta Universitatis Series: Electronics and Energetics* **17**(1), 33–44.
URL: *http://kom.aau.dk/~marjanb/papers/facta04.pdf* 3.6.2

Bozinovski, M., Renier, T., Schwefel, H.-P. and Prasad, R.: 2003, Transaction Consistency in Replicated SIP Call Control Systems, *Proceedings of the Communications & Signal Processing and Fourth Pacific-Rim Conference on Multimedia (ICICS-PCM 2003)*, Vol. 1, pp. 314–318.
URL: *http://kom.aau.dk/~marjanb/papers/icics03.pdf* 3.6.2

Bozinovski, M., Schwefel, H.-P. and Prasad, R.: 2004, Maximum Availability Server Selection Policy for Session Control Systems based on 3GPP SIP, *Proceedings of Seventh International Symposium on Wireless Personal Multimedia Communications*, Padova/Italy.
URL: *http://kom.aau.dk/~marjanb/papers/wpmc04.pdf* 3.6.2

Braden, R., Borman, D. and Partridge, C.: 1988, Computing the Internet Checksum, *Standards Track RFC 1071*, IETF.
URL: *http://www.ietf.org/rfc/rfc1071.txt* 2.4.1, 2.4.2, 3.10.5, 4.4.4

Cain, B., Deering, S., Kouvelas, I., Fenner, B. and Thyagarajan, A.: 2002, Internet Group Management Protocol, *Standards Track RFC 3376*, IETF.
URL: *http://www.ietf.org/rfc/rfc3376.txt* 2.3.1, 2.3.2

Cardellini, V., Colajanni, M. and Yu, P. S.: 2000, Geographic load balancing for scalable distributed Web systems, *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, San Francisco, California/U.S.A., pp. 20–27.
URL: *http://citeseer.ist.psu.edu/cardellini00geographic.html* 1.2.2

Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M. and Bowman, M.: 2003, PlanetLab: An Overlay Testbed for Broad-Coverage Services, *ACM SIGCOMM Computer Communication Review* **33**(3).
URL: *http://www.planet-lab.org/PDN/PDN-03-009/pdn-03-009.pdf* 8.10.5

Cisco Systems: 2000, Cisco™ Distributed Director.
URL: *http://www.cisco.com/cpropart/salestools/cc/pd/cxsr/dd/prodlit/dd_pa.pdf* 1.2.1, 3.1

Cisco Systems: 2004, Cisco™ IOS Reference Guide.
URL: *http://www.cisco.com/warp/public/620/1.pdf* 4.4.8

Coene, L.: 2002, Stream Control Transmission Protocol Applicability Statement, *Informational RFC 3257*, IETF.
URL: *http://www.ietf.org/rfc/rfc3257.txt* 2.4.3.1

Coene, L., Conrad, P. and Lei, P.: 2004, Reliable Server pool applicability Statement, *Internet-Draft Version 02*, IETF, RSerPool Working Group. draft-ietf-rserpool-applic-02.txt.
URL: *http://www.watersprings.org/pub/id/draft-ietf-rserpool-applic-02.txt* 3.6.4

Colajanni, M. and Yu, P. S.: 2002, A Performance Study of Robust Load Sharing Strategies for Distributed Heterogeneous Web Server Systems, *IEEE Transactions on Knowledge and Data Engineering* **14**(2), 398–414.
URL: *http://citeseer.ist.psu.edu/596241.html* 1.2.2

Conrad, P., Jungmaier, A., Ross, C., Sim, W.-C. and Tüxen, M.: 2002, Reliable IP Telephony Applications with SIP using RSerPool, *Proceedings of the State Coverage Initiatives 2002, Mobile/Wireless Computing and Communication Systems II*, Vol. X, Orlando, Florida/U.S.A. ISBN 980-07-8150-1.
URL: *http://www.exp-math.uni-essen.de/~ajung/papers/SCI2002_Reliable_IP_Telephony_with_SIP_and_RSerPool_16_07_2002.pdf* 3.6.2, 6.2

Conrad, P. and Lei, P.: 2005a, Services Provided By Reliable Server Pooling, *Internet-Draft Version 02*, IETF, RSerPool Working Group. draft-ietf-rserpool-service-02.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-ietf-rserpool-service-02.txt* 1, 3.9.5.1, 3.9.5.2

Conrad, P. and Lei, P.: 2005b, TCP Mapping for Reliable Server Pooling Enhanced Mode, *Internet-Draft Version 03*, IETF, RSerPool Working Group. draft-ietf-rserpool-tcpmapping-03.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-ietf-rserpool-tcpmapping-03.txt* 3.5

Cormen, T., Leiserson, C. and Rivest, R.: 1998, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts/U.S.A. ISBN 0-262-03141-8. 4.4.7, 6.4.2.2

Crosby, S. A. and Wallach, D. S.: 2003, Denial of service via Algorithmic Complexity Attacks, *Proceedings of the 12th USENIX Security Symposium*, Washington, DC/U.S.A., pp. 29–44.
URL: *http://www.cs.rice.edu/~scrosby/hash/CrosbyWallach_UsenixSec2003.pdf* 7.6.2

Davies, A.: 2004, Computational Intermediation and the Evolution of Computation as a Commodity, *Journal of Applied Economics* **36**(11), 1131–1142.
URL: *http://www.business.duq.edu/faculty/davies/research/EconomicsOfComputation.pdf* 3.6.5.1

Deering, S. and Hinden, R.: 1998a, Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification, *Standards Track RFC 2463*, IETF.
URL: *http://www.ietf.org/rfc/rfc2463.txt* 2.3.2

Deering, S. and Hinden, R.: 1998b, Internet Protocol, Version 6 (IPv6), *Standards Track RFC 2460*, IETF.
URL: *http://www.ietf.org/rfc/rfc2460.txt* 2.3.2, 3

Dierks, T. and Allen, C.: 1999, The TLS Protocol - Version 1.0, *Standards Track RFC 2246*, IETF.
URL: *http://www.ietf.org/rfc/rfc2246.txt* 4, 9, 4

Dreibholz, T.: 2001, *Management of Layered Variable Bitrate Multimedia Streams over DiffServ with Apriori Knowledge*, Masters Thesis, University of Bonn, Institute for Computer Science.
URL: *http://www.exp-math.uni-essen.de/~dreibh/diplom/index_e.html* 2.3.2

Dreibholz, T.: 2002, An Efficient Approach for State Sharing in Server Pools, *Proceedings of the 27th IEEE Local Computer Networks Conference*, Tampa, Florida/U.S.A., pp. 348–352. ISBN 0-7695-1591-6.
URL: *http://citeseer.ist.psu.edu/dreibholz02efficient.html* 1.2.1, 1.2.3, 1, 3.6.7, 3.9.5.2, 9.1, 9.4.2.2

Dreibholz, T.: 2004a, An Overview of the Reliable Server Pooling Architecture, *Proceedings of the 12th IEEE International Conference on Network Protocols 2004*, Berlin/Germany. Poster presentation.
URL: *http://citeseer.ist.psu.edu/dreibholz04overview.html* 5.7, 10.1.4

Dreibholz, T.: 2004b, draft-ietf-rserpool-policies-00.txt - Definition of Member Selection Policies, *Proceedings of the 61st IETF Meeting*, Washington, DC/U.S.A.
URL: *http://tdrwww.iem.uni-due.de/dreibholz/rserpool/rserpool-publications/IETF61.pdf* 3.11

Dreibholz, T.: 2004c, Member Selection Policies for the Reliable Server Pooling Protocol Suite, *Proceedings of the 60th IETF Meeting*, San Diego, California/U.S.A.
URL: *http://tdrwww.iem.uni-due.de/dreibholz/rserpool/rserpool-publications/IETF60.pdf* 3.11, 8.12.2.4

Dreibholz, T.: 2004d, Policy Management in the Reliable Server Pooling Architecture, *Proceedings of the Multi-Service Networks Conference 2004*, Abingdon, Oxfordshire/United Kingdom.
URL: *http://tdrwww.iem.uni-due.de/dreibholz/rserpool/rserpool-publications/MSN2004-Final-with-Examples.pdf* 3.6.5.2, 3.10.5, 4.4

Dreibholz, T.: 2005a, An IPv4 Flowlabel Option, *Internet-Draft Version 04*, IETF, Individual Submission. draft-dreibholz-ipv4-flowlabel-04.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-dreibholz-ipv4-flowlabel-04.txt* 2.3.2

Dreibholz, T.: 2005b, Das rsplib–Projekt – Hochverfügbarkeit mit Reliable Server Pooling, *Proceedings of the LinuxTag*, Karlsruhe/Germany.
URL: *http://tdrwww.iem.uni-due.de/dreibholz/rserpool/rserpool-publications/LinuxTag2005.pdf* 5.6.2.1, 5.6.2.2, 5.7, 10.1.4

Dreibholz, T.: 2006a, Applicability of Reliable Server Pooling for Real-Time Distributed Computing, *Internet-Draft Version 01*, IETF, Individual Submission. draft-dreibholz-rserpool-applic-distcomp-01.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-dreibholz-rserpool-applic-distcomp-01.txt* 3.6.5, 10.1.4

Dreibholz, T.: 2006b, rsplib – Eine Open Source Implementation von Reliable Server Pooling, *Proceedings of the Linuxtage in Essen*, Essen/Germany.
URL: *http://tdrwww.iem.uni-due.de/dreibholz/rserpool/rserpool-publications/Linuxtage2006.pdf* 5.7

Dreibholz, T.: 2006c, Thomas Dreibholz's RSerPool Page.
URL: *http://tdrwww.exp-math.uni-essen.de/dreibholz/rserpool* 1.3, 5.1, 5.6.2.2

Dreibholz, T., Coene, L. and Conrad, P.: 2006, Reliable Server pool use in IP flow information exchange, *Internet-Draft Version 02*, IETF, Individual Submission. draft-coene-rserpool-applic-ipfix-02.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-coene-rserpool-applic-ipfix-02.txt* 3.6.3, 10.1.4

Dreibholz, T., IJsselmuiden, A. and Adams, J. L.: 2004, Simulation of an advanced QoS protocol for mass content, *Second International Conference on Performance Modelling and Evaluation of Heterogeneous Networks (HET-NET)*, Ikley, West Yorkshire/United Kingdom.
URL: *http://tdrwww.iem.uni-due.de/dreibholz/flowrouting/flowrouting-publications/HET-NET2004-Paper.pdf* 1

Dreibholz, T., IJsselmuiden, A. and Adams, J. L.: 2005, An Advanced QoS Protocol for Mass Content, *Proceedings of the IEEE Conference on Local Computer Networks 30th Anniversary*, Sydney/Australia, pp. 517–518. ISBN 0-7695-2421-4.
URL: *http://citeseer.ist.psu.edu/dreibholz05advanced.html* 1

Dreibholz, T., Jungmaier, A. and Tüxen, M.: 2003, A new Scheme for IP-based Internet Mobility, *Proceedings of the 28th IEEE Local Computer Networks Conference*, Königswinter/Germany, pp. 99–108. ISBN 0-7695-2037-5.
URL: *http://citeseer.ist.psu.edu/dreibholz03new.html* 3.6.6, 10.1.4

Dreibholz, T., Mulik, J., Conrad, P. and Pinzhoffer, K.: 2006, Reliable Server Pooling: Management Information Base using SMIv2, *Internet-Draft Version 02*, IETF, RSerPool Working Group. draft-ietf-rserpool-mib-02.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-ietf-rserpool-mib-02.txt* 10.1.4

Dreibholz, T. and Pulinthanath, J.: 2006, Applicability of Reliable Server Pooling for SCTP-Based Endpoint Mobility, *Internet-Draft Version 00*, IETF, Individual Submission. draft-dreibholz-rserpool-applic-mobility-00.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-dreibholz-rserpool-applic-mobility-00.txt* 3.6.6, 10.1.4

Dreibholz, T. and Rathgeb, E. P.: 2005a, An Application Demonstration of the Reliable Server Pooling Framework, *Proceedings of the 24th IEEE INFOCOM*, Miami, Florida/U.S.A. Demonstration and poster presentation.
URL: *http://citeseer.ist.psu.edu/dreibholz05application.html* 5.7, 10.1.4

Dreibholz, T. and Rathgeb, E. P.: 2005b, Implementing the Reliable Server Pooling Framework, *Proceedings of the 8th IEEE International Conference on Telecommunications*, Vol. 1, Zagreb/Croatia, pp. 21–28. ISBN 953-184-081-4.
URL: *http://citeseer.ist.psu.edu/dreibholz05implementing.html* 1.2.4, 3.6.5.2, 3.10.5, 4.4, 8.5.1, 8.6, 10.1.4

Dreibholz, T. and Rathgeb, E. P.: 2005c, On the Performance of Reliable Server Pooling Systems, *Proceedings of the IEEE Conference on Local Computer Networks 30th Anniversary*, Sydney/Australia, pp. 200–208. ISBN 0-7695-2421-4.
URL: *http://citeseer.ist.psu.edu/dreibholz05performance.html* 2, 3.6.5, 8.5.1, 8.5.2, 8.7, 8.10.1, 8.11.1, 10.1.4

Dreibholz, T. and Rathgeb, E. P.: 2005d, RSerPool – Providing Highly Available Services using Unreliable Servers, *Proceedings of the 31st IEEE EuroMirco Conference on Software Engineering and Advanced Applications*, Porto/Portugal, pp. 396–403. ISBN 0-7695-2431-1.
URL: *http://citeseer.ist.psu.edu/dreibholz05rserpool.html* 1, 3.6.5, 9.3

Dreibholz, T. and Rathgeb, E. P.: 2005e, The Performance of Reliable Server Pooling Systems in Different Server Capacity Scenarios, *Proceedings of the IEEE TENCON '05*, Melbourne/Australia. ISBN 0-7803-9312-0.
URL: *http://citeseer.ist.psu.edu/733077.html* 2, 3.6.5, 8.5.1, 8.8.2, 8.12, 10.1.4

Dreibholz, T. and Rathgeb, E. P.: 2007, On Improving the Performance of Reliable Server Pooling Systems for Distance-Sensitive Distributed Applications, *Proceedings of the 15. ITG/GI Fachtagung Kommunikation in Verteilten Systemen*, Bern/Switzerland.
URL: *http://tdrwww.iem.uni-due.de/dreibholz/rserpool/rserpool-publications/KiVS2007.pdf* 8.10.5.4

Dreibholz, T., Rathgeb, E. P. and Tüxen, M.: 2005, Load Distribution Performance of the Reliable Server Pooling Framework, *Proceedings of the 4th IEEE International Conference on Networking*, Vol. 2, Saint Gilles Les Bains/Reunion Island, pp. 564–574. ISBN 3-540-25338-6.
URL: *http://citeseer.ist.psu.edu/dreibholz05load.html* 2, 3.6.5, 3.11, 3.11.2.1, 3.11.3.2, 8.3, 8.8.1, 8.12.2, 8.12.2.1, 8.12.2.4

Dreibholz, T., Smith, A. and Adams, J. L.: 2003, Realizing a scalable edge device to meet QoS requirements for real-time content delivered to IP broadband customers, *Proceedings of the 10th IEEE International Conference on Telecommunications*, Vol. 2, Papeete/French Polynesia, pp. 1133–1139. ISBN 0-7803-7661-7.
URL: *http://citeseer.ist.psu.edu/dreibholz03realizing.html* 1

Dreibholz, T. and Tüxen, M.: 2003, High Availability using Reliable Server Pooling, *Proceedings of the Linux Conference Australia*, Perth/Australia.
URL: *http://citeseer.ist.psu.edu/dreibholz03high.html* 3.6.5, 5.6.2.1, 10.1.4

Dreibholz, T. and Tüxen, M.: 2006, Reliable Server Pooling (RSerPool) Bakeoff Scoring, *Internet-Draft Version 00*, IETF, Individual Submission. draft-dreibholz-rserpool-score-00.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-dreibholz-rserpool-score-00.txt* 5.9.4, 10.1.4

Dykes, S. G., Robbins, K. A. and Jeffery, C. L.: 2000, An Empirical Evaluation of Client-Side Server Selection Algorithms, *Proceedings of the IEEE Infocom 2000*, Vol. 3, Tel Aviv/Israel, pp. 1361–1370. ISBN 0-7803-5880-5.
URL: *http://citeseer.ist.psu.edu/dykes00empirical.html* 1.2.2

Eastlake, D., Crocker, S. and Schiller, J.: 1994, Randomness Recommendations for Security, *Informational RFC 1750*, IETF.
URL: *http://www.ietf.org/rfc/rfc1750.txt* 3.7.2

Eastlake, D. and Jones, P.: 2001, US Secure Hash Algorithm 1 (SHA1), *Informational RFC 3174*, IETF.
URL: *http://www.ietf.org/rfc/rfc3174.txt* 4, 3

Eaton, J.: 2003, Octave Home Page.
URL: *http://www.octave.org* 6.3.2

Echtle, K.: 1990, *Fehlertoleranzverfahren*, Springer-Verlag, Heidelberg/Germany. ISBN 3-540526-80-3.
URL: *http://dc.informatik.uni-essen.de/Echtle/all/buch_ftv/* 1.2.3, 3.1, 3.12.1

Eisele, K.: 2002, *Design of a Memory Management Unit for System-on-a-Chip Platform "LEON"*, Master's thesis, Universität Stuttgart, Institut für Informatik, Rechnerarchitektur.
URL: *http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2013\&engl=* 4.4.8

Engelmann, C. and Scott, S. L.: 2005, Concepts for High Availability in Scientific High-End Computing, *In Proceedings of the High Availability and Performance Workshop (HAPCW) 2005*, Santa Fe, New Mexico/U.S.A.
URL: *http://citeseer.ist.psu.edu/745548.html* 1.2.3, 3.12.4

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and Berners-Lee, T.: 1999, Hypertext Transfer Protocol - HTTP/1.1, *Standards Track RFC 2616*, IETF.
URL: *http://www.ietf.org/rfc/rfc2616.txt* 2.1, 3.1, 8.2, 9.4.1

Free Software Foundation: 1991, GPL General Public License.
URL: *http://www.gnu.org/copyleft/gpl.html* 2

Free Software Foundation: 2003, GNU Make.
URL: *http://www.gnu.org/software/make/* 6.3.4

GNOME Project: 2001, GLib Reference Manual.
URL: *http://developer.gnome.org/doc/API/glib/* 4.2, 4.4.8

Gradischnig, K. D. and Tüxen, M.: 2001, Signaling transport over IP-based networks using IETF standards, *Proceedings of the 3rd International Workshop on the Design of Reliable Communication Networks*, Budapest/Hungary, pp. 168–174.
URL: *http://www.sctp.de/papers/drcn2001.pdf* 2, 3.6.1

Gradischnig, K., Kramer, S. and Tüxen, M.: 2000, Loadsharing – a key to the reliability of ss7-networks, *Proceedings of the Second International Workshop on the Design of Reliable Communication Networks*, Munich/Germany, pp. 216–221.
URL: *http://citeseer.ist.psu.edu/gradischnig00loadsharing.html* 2

Grams, T.: 1999, *Reliability and Safety - Zuverlässigkeit und Sicherheit*, University of Applied Sciences Fulda, Fulda/Germany.
URL: *http://www2.hs-fulda.de/~grams/Reliability/R\&S-Terms1.html* 3.1

Gray, J. and Siewiorek, D.: 1991, High-Availability Computer Systems, *IEEE Computer Magazine* **24**(9), 39–48.
URL: *http://citeseer.ist.psu.edu/727599.html* 1.1

Guibas, L. J. and Sedgewick, R.: 1978, A dichromatic framework for balanced trees, *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science*, New York/U.S.A., pp. 8–21. 4.4.7

Gupta, D. and Bepari, P.: 1999, Load Sharing in Distributed Systems, *Proceedings of the National Workshop on Distributed Computing*.
URL: *http://citeseer.ist.psu.edu/245239.html* 1.2.2

Harrington, D., Presuhn, R. and Wijnen, B.: 2002, An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks, *Standards Track RFC 3411*, IETF.
URL: *http://www.ietf.org/rfc/rfc3411.txt* 2.1

Hinden, R. and Deering, S.: 2006, IP Version 6 Addressing Architecture, *Standards Track RFC 4291*, IETF.
URL: *http://www.ietf.org/rfc/rfc4291.txt* 2.3.2

Hinden, R. and Haberman, B.: 2005, Unique Local IPv6 Unicast Addresses, *Standards Track RFC 4193*, IETF.
URL: *http://www.ietf.org/rfc/rfc4193.txt* 2.3.2

Hohendorf, C., Dreibholz, T. and Unurkhaan, E.: 2006, Secure SCTP, *Technical Report Version 01*, IETF, Individual Submission. draft-hohendorf-secure-sctp-01.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-hohendorf-secure-sctp-01.txt* 2.4.3.7

Huitema, C. and Carpenter, B.: 2004, Deprecating Site Local Addresses, *Standards Track RFC 3879*, IETF.
URL: *http://www.ietf.org/rfc/rfc3879.txt* 2.3.2

IETF: 2006, Internet Engineering Task Force.
URL: *http://www.ietf.org/* 2.2

IETF RSerPool WG: 2005, Ietf reliable server pooling working group.
URL: *http://www.ietf.org/html.charters/rserpool-charter.html* 3.1

Jungmaier, A.: 2005, *Das Transportprotokoll SCTP*, PhD thesis, Universität Duisburg-Essen, Institut für Experimentelle Mathematik.
URL: *http://miless.uni-duisburg-essen.de/servlets/DocumentServlet?id=12152* 2.4.3.1, 2, 2.4.3.5, 3.12.2, 5.1, 6.2

Jungmaier, A., Rathgeb, E. P., Schopp, M. and Tüxen, M.: 2001, A multi-link end-to-end protocol for IP-based networks, *AEÜ - International Journal of Electronics and Communications* **55**(1), 46–54.
URL: *http://tdrwww.exp-math.uni-essen.de/inhalt/forschung/sctp-aeu.pdf* 2.4.3.1

Jungmaier, A., Rathgeb, E. P. and Tüxen, M.: 2002, On the Use of SCTP in Failover-Scenarios, *Proceedings of the State Coverage Initiatives 2002, Volume X, Mobile/Wireless Computing and Communication Systems II*, Vol. X, Orlando, Florida/U.S.A. ISBN 980-07-8150-1.
URL: *http://tdrwww.exp-math.uni-essen.de/inhalt/forschung/sctp_fb/sctp-failover.pdf* 3.12.2, 6.2

Jungmaier, A., Rescorla, E. and Tüxen, M.: 2002, Transport Layer Security over Stream Control Transmission Protocol, *Standards Track RFC 3436*, IETF.
URL: *http://www.ietf.org/rfc/rfc3436.txt* 2.4.3.7

Jungmaier, A., Schopp, M. and Tüxen, M.: 2000a, Das Simple Control Transmission Protocol (SCTP) – Ein neues Protokoll zum Transport von Signalisierungsmeldungen über IP-basierte Netze, *Elektrotechnik und Informationstechnik – Zeitschrift des Österreichischen Verbandes für Elektrotechnik* **117**(6), 381–388.
URL: *http://tdrwww.iem.uni-due.de/inhalt/forschung/sctp_deutsch.pdf* 2.4.3.1, 2, 6.2

Jungmaier, A., Schopp, M. and Tüxen, M.: 2000b, Performance Evaluation of the Stream Control Transmission Protocol, *Proceedings of the IEEE Conference on High Performance Switching and Routing*, Heidelberg/Germany, pp. 141–148.
URL: *http://tdrwww.iem.uni-due.de/inhalt/forschung/sctp/ppframe.htm* 6.2

KAME: 2006, Webpage of the KAME project.
URL: *http://www.kame.net* 5.3

Kent, S. and Atkinson, R.: 1998a, IP Authentication Header, *Standards Track RFC 2402*, IETF.
URL: *http://www.ietf.org/rfc/rfc2402.txt* 2.3.2, 8, 3

Kent, S. and Atkinson, R.: 1998b, IP Encapsulating Security Payload (ESP), *Standards Track RFC 2406*, IETF.
URL: *http://www.ietf.org/rfc/rfc2406.txt* 2.3.2, 8

Kent, S. and Atkinson, R.: 1998c, Security Architecture for the Internet Protocol, *Standards Track RFC 2401*, IETF.
URL: *http://www.ietf.org/rfc/rfc2401.txt* 2.3.2, 8, 3

Kremien, O. and Kramer, J.: 1992, Methodical Analysis of Adaptive Load Sharing Algorithms, *IEEE Transactions on Parallel and Distributed Systems* **3**(6).
URL: *http://citeseer.ist.psu.edu/kremien92methodical.html* 1.2.2

Lamping, U., Sharpe, S. and Warnicke, E.: 2006, Wireshark User's Guide.
URL: *http://www.wireshark.org/download/docs/user-guide-a4.pdf* 8

León, J., Fisher, A. L. and Steenkiste, P.: 1993, Fail-safe PVM: A Portable package for Distributed Programming with Transparent Recovery, *Technical Report CMU-CS-93-124*, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania/U.S.A.
URL: *http://citeseer.ist.psu.edu/3289.html* 1.2.3

LKSCTP: 2006, Linux Kernel SCTP.
URL: *http://lksctp.sourceforge.net* 5.3

Loughney, J., Stillman, M., Xie, Q., Stewart, R. and Silverton, A.: 2005, Comparison of Protocols for Reliable Server Pooling, *Internet-Draft Version 10*, IETF, RSerPool Working Group. draft-ietf-rserpool-comp-10.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-ietf-rserpool-comp-10.txt* 1.3, 3.2

LVS Project: 2003, Linux Virtual Server.
URL: *http://www.linuxvirtualserver.org* 1.2.1, 3.1

Mela, F.: 2005, libdict.
URL: *http://freshmeat.net/projects/libdict/* 7.3

Mogul, J.: 2003, TCP offload is a dumb idea whose time has come, *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii/U.S.A., pp. 25–30. ISBN 1-931971-17-X.
URL: *http://citeseer.ist.psu.edu/657249.html* 7.2

Neely, M. S.: 1996, *An Analysis of the Effects of Memory Allocation Policy on Storage Fragmentation*, Master's thesis, University of Texas, Austin, Texas/U.S.A.
URL: *http://citeseer.ist.psu.edu/58412.html* 5

NIST: 2002, Secure Hash Standard (SHS), *Technical Report Federal Information Processing Standards Publication FIPS 180-2*, National Institute of Standards and Technology.
URL: *http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf* 4

NS-2: 2003, The Network Simulator NS-2.
URL: *http://www.isi.edu/nsnam/ns/* 6.3.1.3

Ong, L. and Yoakum, J.: 2002, An Introduction to the Stream Control Transmission Protocol (SCTP), *Informational RFC 3286*, IETF.
URL: *http://www.ietf.org/rfc/rfc3286.txt* 2.4.3.1

Open Source Initiative: 1999, BSD License.
URL: *http://www.opensource.org/licenses/bsd-license.php* 9

OPnet Technologies: 2003, OPnet Modeler.
URL: *http://www.opnet.com* 6.3.1.2

Peterson, L.: 2004, PlanetLab: Version 3.0, *Technical Report PDN–04–023*, PlanetLab Consortium.
URL: *http://www.planet-lab.org/PDN/PDN-04-023/pdn-04-023.pdf* 8.10.5

Peterson, L., Anderson, T., Culler, D. and Roscoe, T.: 2002, A Blueprint for Introducing Disruptive Technology into the Internet, *Proceedings of HotNets–I*, Princeton, New Jersey/U.S.A.
URL: *http://www.planet-lab.org/PDN/PDN-02-001/pdn-02-001.pdf* 8.10.5

Peterson, L., Bavier, A., Fiuczynski, M., Muir, S. and Roscoe, T.: 2005, Towards a Comprehensive PlanetLab Architecture, *Technical Report PDN–05–030*, PlanetLab Consortium.
URL: *http://www.planet-lab.org/PDN/PDN-05-030/pdn-05-030.pdf* 8.10.5

Peterson, L. and Roscoe, T.: 2002, PlanetLab Phase 1: Transition to an Isolation Kernel, *Technical Report PDN–02–003*, PlanetLab Consortium.
URL: *http://www.planet-lab.org/PDN/PDN-02-003/pdn-02-003.pdf* 8.10.5

Peterson, L. and Roscoe, T.: 2006, The Design Principles of PlanetLab, *Operating Systems Review* **40**(1), 11–16.
URL: *http://www.planet-lab.org/PDN/PDN-04-021/pdn-04-021.pdf* 8.10.5

Plank, J. S., Beck, M., Kingsley, G. and Li, K.: 1995, Libckpt: Transparent Checkpointing under Unix, *Proceedings of the USENIX Winter 1995 Technical Conference*, New Orleans, Louisiana/U.S.A., pp. 213–224.
URL: *http://citeseer.ist.psu.edu/plank95libckpt.html* 1.2.3, 3.12.3

Postel, J.: 1980, User Datagram Protocol, *Standards Track RFC 768*, IETF.
URL: *http://www.ietf.org/rfc/rfc768.txt* 2.4.1

Postel, J.: 1981a, Internet Control Message Protocol, *Standards Track RFC 792*, IETF.
URL: *http://www.ietf.org/rfc/rfc792.txt* 2.3.1

Postel, J.: 1981b, Internet Protocol, *Standards Track RFC 791*, IETF.
URL: *http://www.ietf.org/rfc/rfc791.txt* 2.3.1

Postel, J.: 1981c, Transmission Control Protocol, *Standards Track RFC 793*, IETF.
URL: *http://www.ietf.org/rfc/rfc793.txt* 2.4.2, 2.4.3.3

Postel, J. and Reynolds, J.: 1985, File Transfer Protocol (FTP), *Standards Track RFC 959*, IETF.
URL: *http://www.ietf.org/rfc/rfc959.txt* 2.1

R Development Core Team: 2005, *R: A language and environment for statistical computing*, R Foundation for Statistical Computing, Vienna/Austria. ISBN 3-900051-07-0.
URL: *http://www.R-project.org* 6.3.2

Ramalho, M., Xie, Q., Tüxen, M. and Conrad, P.: 2006, Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration, *Technical Report Version 15*, IETF, Transport Area Working Group. draft-ietf-tsvwg-addip-sctp-15.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-ietf-tsvwg-addip-sctp-15.txt* 2.4.3.7, 3.5, 3.6.6

Rangarajan, M., Bohra, A., Banerjee, K., Carrera, E. and Bianchini, R.: 2002, TCP Servers: Offloading TCP Processing in Internet Servers. Design, Implementation, and Performance, *Technical report*, Rutgers University.
URL: *http://citeseer.ist.psu.edu/rangarajan02tcp.html* 7.2

Rathgeb, E. P.: 1999, The MainStreetXpress 36190: a scalable and highly reliable ATM core services switch, *International Journal of Computer and Telecommunications Networking* **31**(6), 583–601.
URL: *http://tdrwww.exp-math.uni-essen.de/inhalt/forschung/cn_rathgeb.pdf* 3.1

Renier, T., Schwefel, H.-P., Bozinovski, M., Larsen, K., Prasad, R. and Seidl, R.: 2005, Distributed redundancy or cluster solution? An experimental evaluation of two approaches for dependable mobile Internet services, *Lecture Notes in Computer Science* **3335**. ISBN 978-3-540-24420-2.
URL: *http://kom.aau.dk/~marjanb/papers/isas_springer05.pdf* 3.6.2

Riegel, M. and Tüxen, M.: 2006, Mobile SCTP, *Technical Report Version 06*, IETF, Individual Submission. draft-riegel-tuexen-mobile-sctp-06.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-riegel-tuexen-mobile-sctp-06.txt* 2.4.3.7, 3.6.6

Rijsinghani, A.: 1994, Computing the Internet Checksum via Incremental Update, *Informational RFC 1624*, IETF.
URL: *http://www.ietf.org/rfc/rfc1624.txt* 3.10.5, 4.4.4

Rivest, R.: 1992, The MD5 Message-Digest Algorithm, *Informational RFC 1321*, IETF.
URL: *http://www.ietf.org/rfc/rfc1321.txt* 4

Roscoe, T.: 2002, PlanetLab Phase 0: Technical Specification, *Technical Report PDN–02–002*, PlanetLab Consortium.
URL: *http://www.planet-lab.org/PDN/PDN-02-002/pdn-02-002.pdf* 8.10.5

Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M. and Schooler, E.: 2002, SIP: Session Initiation Protocol, *Standards Track RFC 3261*, IETF.
URL: *http://www.ietf.org/rfc/rfc3261.txt* 3.6.2

Sadasivan, G., Brownlee, N., Claise, B. and Quittek, J.: 2006, Architecture for IP Flow Information Export, *Technical Report Version 10*, IETF, IP Flow Information Export WG. draft-ietf-ipfix-architecture-10.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-ietf-ipfix-architecture-10.txt* 3.6.3

Seidel, R. and Aragon, C.: 1996, Randomized search trees, *Algorithmica* **16**(4), 464–497.
URL: *http://citeseer.ist.psu.edu/seidel96randomized.html* 4.4.7

Seligman, E. and Beguelin, A.: 1994, High-Level Fault Tolerance in Distributed Programs, *Technical Report CMU-CS-94-223*, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania/U.S.A.
URL: *http://citeseer.ist.psu.edu/seligman94highlevel.html* 1.2.3

SETI Project: 2003, SETI@home: Search for Extraterrestrial Intelligence at home.
URL: *http://setiathome.ssl.berkeley.edu* 3.6.5

Seward, J.: 2005, *bzip2 - A program and library for data compression*, Snowbird, Utah/U.S.A.
URL: *http://www.bzip.org/1.0.3/bzip2-manual-1.0.3.html* 6.3.4

Seward, J. and Nethercote, N.: 2005, Using Valgrind to detect undefined value errors with bit-precision, *Proceedings of the USENIX'05 Annual Technical Conference*, Anaheim, California/U.S.A., pp. 17–30.
URL: *http://valgrind.org/docs/memcheck2005.pdf* 7, 7, 4

Silverton, A., Dreibholz, T. and Tüxen, M.: 2004, Private communication at the 61st IETF meeting, Washington, DC/U.S.A. 5

Silverton, A., Dreibholz, T., Tüxen, M. and Xie, Q.: 2005, Reliable Server Pooling Sockets API Extensions, *Internet-Draft Version 00*, IETF, RSerPool Working Group. draft-ietf-rserpool-api-00.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-ietf-rserpool-api-00.txt* 5.6.2, 5.6.2.2, 10.1.4

Silverton, A. and Tüxen, M.: 2005, Reliable Server Pooling Implementations, *Proceedings of the 63rd IETF Meeting*, Paris, France.
URL: *http://www3.ietf.org/proceedings/05aug/slides/rserpool-2/rserpool-3.ppt* 5.9.1

Stevens, W., Fenner, B. and Rudoff, A.: 2003, *Unix Network Programming*, Addison-Wesley Professional. ISBN 0-131-41155-1. 5.6.2.2, 5.6.2.2

Stewart, R., Arias-Rodriguez, I., Poon, K., Caro, A. and Tüxen, M.: 2006, Stream Control Transmission Protocol (SCTP) Specification Errata and Issues, *Informational RFC 4460*, IETF.
URL: *http://www.ietf.org/rfc/rfc4460.txt* 2.4.3.1

Stewart, R., Lei, P. and Tüxen, M.: 2006a, Stream Control Transmission Protocol (SCTP) Packet Drop Reporting, *Technical Report Version 04*, IETF, Individual Submission. draft-stewart-sctp-pktdrprep-04.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-stewart-sctp-pktdrprep-04.txt* 2.4.3.7, 3.12.2

Stewart, R., Lei, P. and Tüxen, M.: 2006b, Stream Control Transmission Protocol (SCTP) Stream Reset, *Technical Report Version 02*, IETF, Individual Submission. draft-stewart-sctpstrrst-02.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-stewart-sctpstrrst-02.txt* 2.4.3.7

Stewart, R., Ramalho, M., Xie, Q., Tüxen, M. and Conrad, P.: 2004, Stream Control Transmission Protocol (SCTP) Partial Reliability Extension, *Standards Track RFC 3758*, IETF.
URL: *http://www.ietf.org/rfc/rfc3758.txt* 2.4.3.7

Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L. and Paxson, V.: 2000, Stream Control Transmission Protocol, *Standards Track RFC 2960*, IETF.
URL: *http://www.ietf.org/rfc/rfc2960.txt* 2.4.3.1

Stewart, R., Xie, Q., Stillman, M. and Tüxen, M.: 2006a, Aggregate Server Access Protcol (ASAP), *Technical Report Version 13*, IETF, RSerPool Working Group. draft-ietf-rserpool-asap-13.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-ietf-rserpool-asap-13.txt* 3.3, 3.9, 3.9.2.1, 3.9.2.1, 3.9.2.3, 3.9.3.1, 3.9.4, 3.9.5.2, 7.5.2

Stewart, R., Xie, Q., Stillman, M. and Tüxen, M.: 2006b, Aggregate Server Access Protocol (ASAP) and Endpoint Handlespace Redundancy Protocol (ENRP) Parameters, *Internet-Draft Version 10*, IETF, RSerPool Working Group. draft-ietf-rserpool-common-param-10.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-ietf-rserpool-common-param-10.txt* 3.3, 3.8, 3.9, 3.9.2.1, 3.9.2.1, 3.9.2.3, 3.9.3.1, 3.9.5.2, 3.10, 3.10.2.1

Stewart, R., Xie, Q., Yarroll, Y., Wood, J., Poon, K. and Tüxen, M.: 2006, Sockets API Extensions for Stream Control Transmission Protocol (SCTP), *Internet-Draft Version 12*, IETF, Transport Area Working Group. draft-ietf-tsvwg-sctpsocket-12.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-ietf-tsvwg-sctpsocket-12.txt* 5.3, 5.4.1, 8.10.2.1

Stillman, M., Gopol, R., Sengodan, S., Guttman, E. and Holdrege, M.: 2005, Threats Introduced by RSerPool and Requirements for Security, *Internet-Draft Version 05*, IETF, RSerPool Working Group. draft-ietf-rserpool-threats-05.txt.
URL: *http://www.watersprings.org/pub/id/draft-ietf-rserpool-threats-05.txt* 3.13

Stone, J., Stewart, R. and Otis, D.: 2002, Stream Control Transmission Protocol (SCTP) Checksum Change, *Standards Track RFC 3309*, IETF.
URL: *http://www.ietf.org/rfc/rfc3309.txt* 2.4.3.1

Subramanian, L., Padmanabhan, V. and Katz, R.: 2002, Geographic Properties of Internet Routing, *Proceedings of the USENIX Annual Technical Conference*, Monterey, California/U.S.A., pp. 243–259. ISBN 1-880446-00-6.
URL: *http://citeseer.ist.psu.edu/subramanian02geographic.html* 8.10.2.1

Sultan, F., Srinivasan, K., Iyer, D. and Iftode, L.: 2002, Migratory TCP: Highly available Internet services using connection migration, *Proceedings of the ICDCS 2002*, Vienna/Austria, pp. 17–26.
URL: *http://citeseer.ist.psu.edu/sultan02migratory.html* 1.2.1, 1.2.4

Tanenbaum, A.: 1996, *Computer Networks*, Prentice Hall, Upper Saddle River, New Jersey/U.S.A. ISBN 0-13-349945-6. 2, 2.2

Tomonori, F. and Masanori, O.: 2003, Performance optimized software implementation of iSCSI, *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, Louisiana/U.S.A.
URL: *http://citeseer.ist.psu.edu/685948.html* 7.2

Tüxen, M.: 2001, The sctplib Prototype.
URL: *http://www.sctp.de/sctp.html* 5.1, 5.3

Tüxen, M.: 2003, LISP Simulation Package.
URL: *http://sctp.fh-muenster.de/sim.html* 6.3.1.1

Tüxen, M. and Dreibholz, T.: 2005, Reliable Server Pooling Policies, *Internet-Draft Version 00*, IETF, Individual Submission. draft-tuexen-rserpool-policies-00.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-tuexen-rserpool-policies-00.txt* 3.11, 8.8.1, 8.12.2.4

Tüxen, M. and Dreibholz, T.: 2006a, Private communication at the 8th SCTP Bakeoff, Vancouver/-Canada. 5.9.3

Tüxen, M. and Dreibholz, T.: 2006b, Reliable Server Pooling Policies, *Internet-Draft Version 02*, IETF, RSerPool Working Group. draft-ietf-rserpool-policies-02.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-ietf-rserpool-policies-02.txt* 2, 3.11, 3.11.1, 3.11.2, 3.11.3, 4.4.2.2, 7.4.5, 8.8.1, 8.8.3, 8.12.2.4, 10.1.4

Tüxen, M., Dreibholz, T., Silverton, A., Coene, L., Xie, Q., Stewart, R. and Lei, P.: 2004, Private communication at the 60th IETF meeting, San Diego/California, U.S.A. 5

Tüxen, M. and Stewart, R.: 2005, UDP Encapsulation of SCTP Packets, *Technical Report Version 00*, IETF, Individual Submission. draft-tuexen-sctp-udp-encaps-00.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-tuexen-sctp-udp-encaps-00.txt* 8.10.5.1

Tüxen, M., Stewart, R. and Lei, P.: 2005, Authenticated Chunks for Stream Control Transmission Protocol (SCTP), *Technical Report Version 03*, IETF, Transport Area Working Group. draft-tuexen-sctp-auth-chunk-03.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-tuexen-sctp-auth-chunk-03.txt* 2.4.3.7

Tüxen, M., Xie, Q., Stewart, R., Shore, M., Loughney, J. and Silverton, A.: 2006, Architecture for Reliable Server Pooling, *Technical Report Version 11*, IETF, RSerPool Working Group. draft-ietf-rserpool-arch-11.txt, work in progress.
URL: *http://www.watersprings.org/pub/id/draft-ietf-rserpool-arch-11.txt* 1.2.1, 1.2.4, 3.3, 3.6.1, 3.6.4

Tüxen, M., Xie, Q., Stewart, R., Shore, M., Ong, L., Loughney, J. and Stillman, M.: 2002, Requirements for Reliable Server Pooling, *Informational RFC 3227*, IETF.
URL: *http://www.ietf.org/rfc/rfc3237.txt* 1.3, 3.2

Unurkhaan, E.: 2005, *Secure End-to-End Transport - A new security extension for SCTP*, PhD thesis, University of Duisburg-Essen, Institute for Experimental Mathematics.
URL: *http://miless.uni-duisburg-essen.de/servlets/DocumentServlet?id=11981* 2.4.3.7

Uyar, Ü., Zheng, J., Fecko, M. A. and Samtani, S.: 2003a, Performance Study of Reliable Server Pooling, *Proceedings of the IEEE NCA International Symposium on Network Computing and Applications*, Cambridge, Massachusetts/U.S.A., pp. 205–212. ISBN 0-7695-1938-5.
URL: *http://www.cis.udel.edu/~fecko/papers_pdf/nca03_rsp_rev.pdf* 3.6.7, 9.5

Uyar, Ü., Zheng, J., Fecko, M. A. and Samtani, S.: 2003b, Reliable Server Pooling in Highly Mobile Wireless Networks, *Proceedings of the IEEE International Symposium on Computers and Communications*, Kemer-Antalya/Turkey, pp. 627–632. ISBN 0-7695-1961-X.
URL: *http://www.cis.udel.edu/~fecko/papers_pdf/iscc03_rev.pdf* 3.6.7, 9.5

Uyar, Ü., Zheng, J., Fecko, M. A., Samtani, S. and Conrad, P.: 2003, Reliable Server Pooling for Future Combat Systems, *Proceedings of the IEEE MILCOM Military Communications Conference*, Vol. 2, Boston, Massachusetts/U.S.A., pp. 927–932.
URL: *http://www.cis.udel.edu/~fecko/papers_pdf/milcom03_main_rev.pdf* 3.6.7

Uyar, Ü., Zheng, J., Fecko, M. A., Samtani, S. and Conrad, P.: 2004, Evaluation of Architectures for Reliable Server Pooling in Wired and Wireless Environments, *IEEE JSAC Special Issue on Recent Advances in Service Overlay Networks* **22**(1), 164–175.
URL: *http://www.cis.udel.edu/~fecko/papers_pdf/jsac04_rev.pdf* 1.2.1, 3.6.7, 9.5

Valgrind Developers: 2005, Valgrind Home.
URL: *http://www.valgrind.org* 7, 7, 4

Varga, A.: 2005a, OMNeT++ Discrete Event Simulation System.
URL: *http://www.omnetpp.org* 4.2, 4.4.8, 6.3.1.4

Varga, A.: 2005b, *OMNeT++ Discrete Event Simulation System User Manual - Version 3.2*, Technical University of Budapest/Hungary.
URL: *http://www.omnetpp.org/doc/manual/usman.html* 4.2, 4.4.8, 6.3.1.4, 6.3.3

Wagner, A.: 2006, Interoperability-Test.
URL: *http://www.cs.ubc.ca/~wagner/SCTP/* 5.9.4

Wetherall, D. and Lindblad, C.: 1995, Extending Tcl for dynamic object-oriented programming, *Proceedings of the Tcl/Tk Workshop*, Toronto/Canada, pp. 173–182. ISBN 1-880446-72-3.
URL: *http://citeseer.ist.psu.edu/wetherall95extending.html* 6.3.1.3

Wiesmann, M., Pedone, F., Schiper, A., Kemme, B. and Alonso, G.: 2000, Understanding Replication in Databases and Distributed Systems, *Proceedings of the 20th International Conference on Distributed Computing Systems*, Taipei/Taiwan, pp. 264–274.
URL: *http://citeseer.ist.psu.edu/393991.html* 1.2.1

Williams, T. and Kelley, C.: 2003, GNU Plot Homepage.
URL: *http://www.gnuplot.info* 6.3.2

Wireshark: 2006, Wireshark: The World's Most Popular Network Protocol Analyzer.
URL: *http://www.wireshark.org* 8

Xie, Q., Stewart, R., Stillman, M., Tüxen, M. and Silverton, A.: 2006, Endpoint Handlespace Redundancy Protocol (ENRP), *Internet-Draft Version 13*, IETF, RSerPool Working Group. draft-ietf-rserpool-enrp-13.txt, work in progress.

URL: *http://www.watersprings.org/pub/id/draft-ietf-rserpool-enrp-13.txt* 3.3, 2, 3.7.1.4, 3.7.2, 3.10, 3.10.2.1, 3.10.2.3, 4, 3.10.3, 3.10.5

Xie, Q. and Yarrol, L.: 2004, RSerPool Redundancy-model Policy, *Technical Report Version 02*, IETF, Individual Submission. draft-xie-rserpool-redundancy-model-02.txt.
URL: *http://www.watersprings.org/pub/id/draft-xie-rserpool-redundancy-model-02.txt* 3.12.4

Zhang, Y.: 2004, *Distributed Computing mit Reliable Server Pooling*, Master's thesis, Universität Essen, Institut für Experimentelle Mathematik. 3.6.5, 8.3

Index

Curriculum Vitae

Name:	Thomas Dreibholz
29.09.1976	born in Bergneustadt, Germany
08/1983 - 09/1987	Student at the Grundschule Bielstein (Primary School), Germany
09/1987 - 06/1993	Student at the Realschule Bielstein (Junior High School), Germany http://www.gm.nw.schule.de/~rswiehl
08/1993 - 07/1996	Student at the Gymnasium Wiehl (High School), Germany http://www.gm.nw.schule.de/~gymwiehl
08/1996 - 04/2001	Student of Computer Science at the University of Bonn, Germany http://www.uni-bonn.de
since 10/1998	Vordiplom (Bachelor's Degree) of Computer Science at the University of Bonn, Germany
since 04/2001	Diplom (Master's Degree) of Computer Science at the University of Bonn, Germany
2000	Research Assistant in the Computer Networking Technology Group at the Department of Computer Science IV of the University of Bonn, Germany http://web.informatik.uni-bonn.de/IV/
since 05/2001	Research Associate and Ph.D. Student in the Computer Networking Technology Group at the Institute for Experimental Mathematics of the University of Duisburg-Essen in Essen, Germany http://tdrwww.exp-math.uni-essen.de
since 2002	Cisco™ Certified Network Associate (CCNA) and Cisco™ Certified Academy Instructor (CCAI) at the Cisco™ Networking Academy of the University of Duisburg-Essen in Essen, Germany http://cna.uni-essen.de