ALRPC: A Mechanism to Semi-Automatically Refactor Legacy Applications for Deployment in Distributed Environments

by

Andreas Christoph Bergen
B.Sc., University of Victoria, 2011
B.A., University of Victoria, 2007

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Andreas Christoph Bergen, 2013
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

ALRPC: A Mechanism to Semi-Automatically Refactor Legacy Applications for Deployment in Distributed Environments

by

Andreas Christoph Bergen
B.Sc., University of Victoria, 2011
B.A., University of Victoria, 2007

Supervisory Committee

Dr. Y. Coady, Supervisor
(Department of Computer Science)

Dr. R. McGeer, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Y. Coady, Supervisor
(Department of Computer Science)

Dr. R. McGeer, Departmental Member
(Department of Computer Science)

ABSTRACT

Scientific projects, businesses, and individual devices such as smart phones, tablets and embedded devices are collecting and retaining unparalleled and growing amounts of data. Initially, spatial locality of the data (collocation of data and application) cannot be assumed and local resource constraints impact monolithic legacy applications. Resource restrictions and less feasible approaches such as moving large data sets within these paradigms are not feasible for certain legacy applications. As such we have taken a renewed look at Remote Procedure Call mechanisms and designed, built and evaluated a RPC mechanism called *Automated Legacy system Remote Procedure Call generator* (ALRPC). ALRPC allows us to convert monolithic applications into distributed systems by selectively and semi-automatically moving individual functions to different process spaces. This improves spatial locality and resource constraints of critical functions in legacy applications. Empirical results from our initial experiments show that our mechanism's level of automation outperforms existing industry strength tools and its performance is competitive within the scope of this work.

Contents

Supervisory Committee ii

Abstract iii

Table of Contents iv

List of Tables viii

List of Figures x

Acknowledgements xii

Dedication xiii

1 Introduction 1

1.1 Motivating ALRPC and Scope of the Solution it is providing 1

1.1.1 Scope of ALRPC 4

1.2 Overview of previous RPC tools 5

1.2.1 Quick Overview 5

1.2.2 Usability of existing RPC tools 6

1.2.3 RPC tools - history 7

1.2.4 Future of RPC 8

1.3 Thesis Statement 8

1.4 Thesis Outline 9

2 Background of Remote Procedure Calls 10

2.1 Conceptual, Technical Details of Remote Procedure Calls and its History 10

2.1.1 Rise and Fall of RPC 11

2.1.2 Technical implementation of RPC systems 13

2.1.3 Summary 15

2.2 RPCGen . . . 16
2.2.1 RPCGen and External Data Representation (XDR) . . . 16
2.2.2 Remote Procedure Call Language (RPCL) . . . 17
2.2.3 Summary . . . 20
2.3 Sun's RPC . . . 21
2.3.1 History of Sun RPC . . . 21
2.3.2 Overview of Technical Aspects . . . 22
2.3.3 Adoption, Success, and Decline . . . 23
2.3.4 Summary . . . 24
2.4 Recent Decline of RPC's use . . . 24
2.4.1 Rise and Fall of RPC . . . 24
2.4.2 Conceptual Problems . . . 25
2.4.3 Technical Problems . . . 26
2.4.4 Crashes . . . 28
2.4.5 Heterogeneous Systems . . . 28
2.4.6 Performance . . . 29
2.4.6.1 Summary of Critiques of RPC model . . . 29
2.5 Limitations of RPC within Legacy Code . . . 29
2.5.1 Parameter Passing . . . 30
2.5.2 Type Detection . . . 31
2.5.3 Discussion . . . 31

3 ALRPC **32**
3.1 Conceptual Design of ALRPC . . . 32
3.1.1 Usage of ALRPC . . . 33
3.1.2 Modular Building Blocks of ALRPC . . . 36
3.1.3 History of ALRPC . . . 39
3.2 Lexing and Parsing the Input . . . 42
3.3 Code Generation . . . 44
3.3.1 Active Python Modules of ALRPC . . . 44
3.3.1.1 Control and Common Headers: ctags_parser . . . 45
3.3.1.2 Generating local files: local_impl . . . 46
3.3.1.3 Serialization and Deserialization . . . 46
3.3.2 Output Files . . . 47
3.3.3 Compilation and Integration of Generated Files . . . 49

3.4 Challenges in C 50
3.5 ALRPC - Status Quo 52

4 Experiments **54**
4.0.1 ALRPC — RPCGen initial benchmarks layout 55
4.1 RPCGen experiments 57
4.1.1 The tested functions 58
4.1.2 Correctness 60
4.1.3 Manually written lines of Code 60
4.2 ALRPC experiments 61
4.2.1 The tested functions 62
4.2.2 Time measurements 63
4.2.3 Correctness 65
4.2.4 Manually Written Lines of Code 65
4.3 Performance Mircobenchmarks 67
4.4 ALRPC in Action 69
4.4.1 ALRPC and Real World Games 69
4.4.1.1 Introducing LTris 70
4.4.1.2 Choosing the right function 70
4.4.1.3 Using ALRPC with LTris 72
4.4.1.4 LTris challenges for ALRPC 73
4.4.2 ALRPC and large systems — an investigation 74
4.4.2.1 ALRPC and GDAL 74
4.4.2.2 ALRPC and OpenSSL 77
4.4.2.3 Large applications and ALRPC findings 78
4.5 Feasibility of RPC 78

5 Evaluation, Analysis and Comparisons **82**
5.1 Lines of Code Manually Written 82
5.2 Correctness of RPC 86
5.3 Performance Evaluation 87

6 Conclusion and Future Work **92**
6.1 Future of ALRPC 93

A Additional Information **97**

A.1 Monolithic C code — Micro Benchmarks 97
A.2 C code after manual modification due to RPCGen 99
A.3 Function calls for system when using ALRPC 100
A.4 C code for client using "Berkeley" sockets 100
A.5 C code for client using Unix Domain sockets 101
A.6 Lines of code manually changed or added due to use of RPCGen . . . 103
A.7 Statistical Data Monolithic Micro Benchmarks 104
A.8 Statistical Data ALRPC vs. RPCGen (local) 105
A.9 Statistical Data: ALRPC vs RPCGen (separate physical machines) . 105

Bibliography **107**

List of Tables

Table 3.1 Overview of ALRPC's Python module. 37

Table 4.1 Time Measurements for non-RPC functions (time measurement in microseconds). Note that several calls completed faster than the measurement precision. 56

Table 4.2 Time Measurements for RPC functions in RPCGen built system (time measurement in microseconds). Server and client are on same physical machine. 59

Table 4.3 Time Measurements for RPC functions in ALRPC built system (time measurement in micro seconds) using Unix domain sockets. Server and client are therefore on the same physical machine. . . 63

Table 4.4 Time measurements comparing ALRPC system and RPCGen system where server and client are on different machines and network bandwidth is approximately 38Mbits/sec. Measurement unit in microseconds. 64

Table 4.5 Time measurement in seconds for function execution in hybrid and monolithic remote system. 68

Table 4.6 Breakdown of function and parameter types in real systems . . . 80

Table 5.1 Manual change metrics. For simple Function signatures as shown ALRPC requires fewer manual code changes and additions than competing RPC tools. 83

Table A.1 Number of trials which did not result in 0 time measurment . . 104

Table A.2 one_line function statistics, monolithic C application, 1 million trials. 105

Table A.3 10000 trials ALRPC, same physical machine Unix Domain sockets, Measurement in microseconds. 105

Table A.4 10000 trials RPCGen same physical machine. Measurement in microseconds. 105
Table A.5 10000 trials ALRPC different physical machine (38MBit/s). Measurement in microseconds. 105
Table A.6 10000 trials RPCGen different physical machine(38MBit/s). Measurement in microseconds. 106

List of Figures

Figure 1.1 Decision to move data to local server or to use remote function is based on threshold. . . . 3

Figure 2.1 Conceptual Overview of the Processes guiding any RPC use. . . 14
Figure 2.2 Integer specification in XDR Specification RFC4506. . . . 17
Figure 2.3 Enumeration specification in XDR Specification RFC4506. . . 17
Figure 2.4 RPCGen overview [35]. . . . 18
Figure 2.5 Sample RPCL code modified from [2]. . . . 19
Figure 2.6 Segment of Sample C client code to establish connection to server. Modified from [2]. . . . 20
Figure 2.7 Tanenbaum's example on conceptual distinction of client and server. . . . 26

Figure 3.1 Static control flow of ALRPC tool chain. . . . 33
Figure 3.2 Custom Buffer of ALRPC. . . . 35
Figure 3.3 Original crypt function in unistd.h. . . . 36
Figure 3.4 Prototype signature for crypt function in automatically generated marshalling code serialize.h/c. . . . 36
Figure 3.5 Code snippet of serialize.py: This generates standard Code to allow marshalling of parameters on the client side. . . . 38
Figure 3.6 Code snippet of deserialize.py: This generates standard Code to allow unmarshalling of parameters of the type char* (assumed to be String). . . . 38
Figure 3.7 Code snippet of deserialize.py: This generates standard Code and switch statements to direct the function to the correct implementation at the server side. . . . 38
Figure 3.8 Call graph of ALRPC tool. . . . 43
Figure 3.9 Excerpt of ctags output file containing the line describing the *crypt* function. . . . 44

Figure 3.10 Code excerpt of the function *cp_parse_ctags* in Figure 3.8. . . 44
Figure 3.11 Code snippet which creates local serialization function: This allows marshalling of parameters on the client side. 46
Figure 3.12 Code snippet in tagstmp_mm.c: Serialization, client server connection and extraction of return value for function char * crypt is handled here. 48
Figure 3.13 Function call and signature of the redirection call. 49

Figure 4.1 X file containing RPCGen description for 3 functions. 60
Figure 4.2 Return statement in server side code of RPCGen distributed system for the original function *char * one_line(char * string)*. Complexity is high due to dereferencing and casting operations. 61
Figure 4.3 Illustration of monolithic system before and after the manual modification necessary to use ALRPC correctly with code base. 66
Figure 4.4 Sample code of prime number function. This is relatively compute intensive. 68
Figure 4.5 Screenshot of LTris after startup. 70

Figure 5.1 Comparing performance on a function by function level between ALRPC and RPCGen. 88
Figure 5.2 Comparing performance on a function by function level between ALRPC using Berkeley sockets and Unix Domain Sockets for communication. 89

Figure 6.1 Servers where analysis of log files occurs are not the same as where data files are stored. 95

ACKNOWLEDGEMENTS

I would like to thank:

Rick McGeer, Justin Cappos, for their support and feedback throughout,

Chris Matthews, for his mentorship,

Yvonne Coady, for enthusiasm, support, encouragement, patience and always pushing me to new limits.

The hardest thing is to go to sleep at night, when there are so many urgent things needing to be done. A huge gap exists between what we know is possible with todays machines and what we have so far been able to finish.

Donald Knuth

DEDICATION

Just hoping this is useful!

Chapter 1

Introduction

This section situates the work in the context of previous tools and their approaches in the field of Remote Procedure Calls (RPC). A remote procedure call is defined, in the context of this work, to mean that parts of a single application execute in two or more process spaces. Using Birrel's explanation, once "a remote procedure is invoked, the calling environment is suspended, the parameters are passed across the network to the environment where the procedure is to execute" [14]. Following the remote execution the results are returned to the caller and the caller's system resumes execution "as if returning from a simple single-machine call" [14].

1.1 Motivating ALRPC and Scope of the Solution it is providing

In this section we will motivate our mechanism *Automated Legacy system Remote Procedure Call generator* (ALRPC). We will introduce reasons why a previously monolithic application would benefit from a conversion to a distributed system using remote procedure calls. The reasons for using RPC with legacy applications are closely tied to: latency, security and jurisdiction.

The system we propose is intended to address several paradigms which have become prominent in recent years. Smart phones, tablets, and embedded devices have become ubiquitous in our society. Each device is collecting data. Likewise scientific projects and businesses are collecting and retaining unparalleled amounts of data as

well [28].

Moving this data to different physical machines is often required prior to its analysis. This is expensive and time consuming in large data sets. Additionally, further resource or performance restrictions on these physical machines can make analysis of data costly. Subsequently, traditional approaches of moving data is not feasible for certain legacy applications. Thus we propose a renewed look at RPC systems. The RPC tool we develop for this evaluation is called ALRPC. ALRPC and the ability to split off functions into remote processes is a key feature which makes the proposed system feasible. We will set ALRPC apart from other RPC tools by the degree of automation which allows it to easily convert monolithic applications into suitable distributed systems.

Imagine a monolithic legacy application which has a sole purpose of analysing data. Within this system, spatial locality of the data is not given. Obtaining a local copy of this data requires time consuming network communication. At some point this network communication reaches a threshold in terms of time or monetary cost qt which point it is no longer feasible for the system to move the data to the computation server. Profiling of the system detects this change and alters the program's execution to use a remote function whose spatial locality to the data is an improvement for the entire system. The decision making structure and actions of such a system are illustrated in Figure 1.1.

Figure 1.1 illustrates a simple case which we believe is suitable as a common scenario for not only ALRPC but RPC systems in general. In this scenario data files, whether they are log files or any other kind of data, are produced by an application. These data files today can be relatively large and are located on a remote server. In order to analyse these files the analysis software must have access to them. One approach is to move a copy of the data files onto another server and perform the analysis there. This however is costly. Cost is incurred at two levels. First, the movement of data is limited by the transmission rates between the server and the analysis machine. This in turn is dependent on the client machine's network and bandwidth capabilities, both of which can vary greatly [10]. This cost is incurred in the form of latency.

Secondly, this existing approach is also very likely to incur further costs because

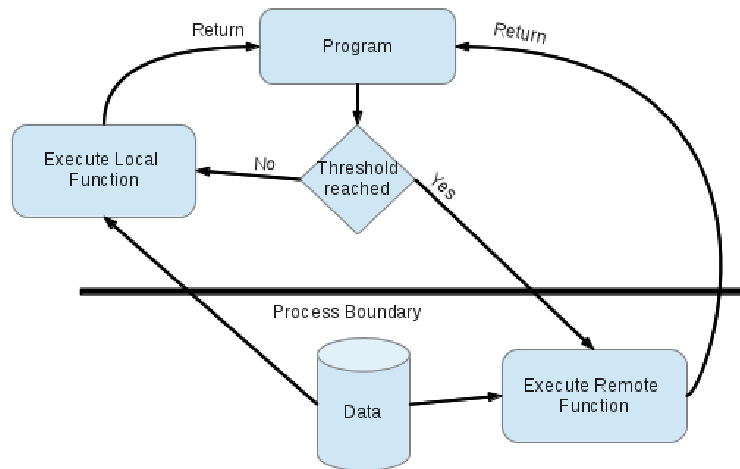

Figure 1.1: Decision to move data to local server or to use remote function is based on threshold.

providers like Amazon's EC2 have payment models where one is charged for moving data as well as for storage requirements [6]. Following this approach would duplicate a log file's storage requirements, at least temporarily, and incur costs from simply moving data to the analysis server.

With ALRPC and any RPC system one could reduce the size of data which has to be moved. Instead of moving the data itself, which can be rather large, a single function can now be moved. Moving a function for remote execution occurs only on an on-demand basis and is application and use case specific. Any single function rarely exceeds a few kilobytes in size, thus reducing the transfer costs in terms of money and time.

However, the question remains: what functions of an application should be moved to a server which is located closer to the data? By identifying candidate functions with the correct criteria one can keep data movement between servers to a minimum while at the same time ensuring that the process of moving functions is as automated as possible.

We have described a scenario related to latency and "big data", however "big data" is only one area where RPC systems may be beneficial. A further advantage of distributed systems such as the one we propose is validation and verification of compu-

tational results. In particular asynchronous execution of local and remote procedures makes this possible. An application can start both a local function call and one or more remote function calls with the same input data. This is useful for two reasons. Firstly it allows the program to verify that the computation was carried out correctly by taking the aggregate of returned answers as the true value. If two or three answers from different sources form a consensus then it is likely that the computation returned the correct results.

The same applies if one wants to ensure that functions use only certified versions of specific library calls stored on remote servers and do not use varying local implementations of it. On the other hand this is also a practical approach for compute intensive operations. The local machine may not be powerful enough to compute the results in a timely manner. This is exasperated when there are large data sets which have to be moved to the local machine. In a distributed system, the application can start local and remote function calls and accept the first returned result as the approved response. Using this approach, an application can utilize superior computational power of remote servers or data centers. This is very similar to map reduce techniques, however our existing tool allows the automatic modification of existing legacy applications to obtain the same leverage [24, 61].

Lastly, jurisdictional obstacles supersede technical challenges. Some data is simply not allowed to leave a given jurisdiction. Yet at the same time access to the data is not prohibited. This scenario is common when dealing with data belonging to governments. Privacy legislation prevents storage of this data in another jurisdiction, yet access to the data is open to the public. An application can be distributed to have the function which accesses the data in the same jurisdiction as the data. At the same time, the bulk of the application can be run from a different jurisdiction.

1.1.1 Scope of ALRPC

In order to accurately evaluate the contribution of this work one must first define the scope of the work and how it is situated with related previous contributions. The thesis explores areas related to RPC systems. As such there are two distinct, yet interconnected, goals.

- An exploration of the limits and feasibility of remote procedure calls in today's world.
- Providing a mechanism for legacy applications to, mostly automatically, convert monolithic applications into a distributable application which uses RPC within the discovered confines.

By drawing on the related work and background of RPC research and then connecting it to the technological realities of today's world this thesis aims to provide the reader with a greater understanding of the current feasibility of RPC. However, in a addition to this approach we are also basing our findings on the experiences encountered in building our own RPC tool for legacy applications in *C*.

For the scope and scale of this work ALRPC is purely a mechanism. As such, its goal is to provide a programmer with an automated mechanism to convert monolithic applications written in *C* into a distributed system using RPC models. Limitations of this tool and the mechanism itself are introduced later to clearly identify the capabilities, strengths and weaknesses of our mechanism's implementation.

Automation is the main distinguishing feature of ALRPC. With ALRPC, unlike with nearly all other RPC systems, maximum automation and autonomy from programmer interaction is desired. We feel this level of automation is necessary to set ALRPC apart from previous tools in an attempt to compete with rivalling technologies.

1.2 Overview of previous RPC tools

This section will only peripherally introduce other key developments and tool in the RPC environment. A detailed background discussion on key RPC tools used with legacy code follows in Chapter 2.

1.2.1 Quick Overview

Factoring existing or new code bases into a distributed framework by use of remote procedure calls has been well understood for many years. Early work in the 1980s produced artefacts directly targeting the use and creation of remote procedure calls [12, 13]. Tools and frameworks, such as SWIG and RPCGen (and others) to name

only a few, have existed since the 1990s. Key ideas of remote procedure calls have been firmly studied and discussed since the 1980s; concretely targeted work for Object Oriented Programming and RPC arises in the 1990s (such as RMI [59]). At the same time in the 1990s, several patent applications from small companies and large industry leaders alike indicate a vested interest in this technology in those times [15, 30, 39].

1.2.2 Usability of existing RPC tools

Refactoring applications in a distributed computation framework is challenging and currently requires either manual refactoring of code, manual verification of the tools' actions, or both. This effort is often increased by requiring that the programmer learns a new description language for the RPC tool of choice. The purpose of the RPC description language is to provide a unique description of both data and functions within the distributed program. From this description, data type and communication requirements, such as buffers and memory can be determined by the RPC tool at run time.

Additionally, in many cases user interaction in the form of source code modification is needed to enable tools to understand how to turn an existing code base into a modular RPC code base. Limitations of the existing tools, in the form of manual source code modifications and the requirement for RPC description languages, are preventing the refactoring of existing code bases. This often prohibits RPC tools to fully leverage the realities of modern computing paradigms.

Significant shifts in the computing paradigm occurred in fields such as cloud computing, remote sensing, social networks, health information systems and scientific computing. A remarkably common concern in these fields centers around the need to process vast amounts of information. Accordingly, with the growth in data sets and individual pieces of information to be processed, applications are facing practical challenges within the confines of today's technological realities [17].

Despite the improvements in network technology, transferring and processing the required amounts of information is still and always will be an issue with respect to practical metrics such as bandwidth and latency. The performance penalty can be particularly severe in the cases where the spatial locality of application and the data

to be processed are not provided by default. Moreover, the performance penalty can be highly variable due to external influences on the network traffic and network connectivity [47].

1.2.3 RPC tools - history

While ALRPC is not a production strength tool, as a mechanism, it tries to join a large number of RPC tools that came before it. Xerox's Courier RPC system is possibly one of the earliest RPC implementations [20,54]. Courier RPC is an integral part of the Xerox Network System (XNS) which aimed at providing general purpose communication mechanisms for distributed systems [22]. Xerox built several RPC suites and tools for which Courier was laying the foundations in terms of using many of Xerox's communication protocols and other standards for routers, network protocols as well as packet switching and forwarding [22,54].

Following Courier, Xerox developed Cedar RPC, which was one of the first attempts to make RPC syntax and semantics close to those of local calls [13,14,54]. Cedar was developed by Birrell and Nelson and, along with the Courier RPC system, these two systems pioneered RPC tools [54].

Another well established tool, which is still used today is Sun RPC. While early systems such as Courier [20] used 3 different layered custom communication protocols, Sun RPC supports the standard UDP and TCP connections. Using these standard protocols, Sun RPC supports multiple types of messages and RPC calls. A closer look at Sun RPC is provided in Section 2.3.

Other companies like HP/Apollo also developed different RPC models. Apollo RPC uses a connectionless-oriented transport layer, which is unlike Courier. Courier requires a virtual circuit as part of Xerox's custom in-house XNS Sequenced Packet Protocol [18, 54]. Following the early developments, improvements in RPC technology were made. Apollo and the Cambridge Mayflower Project RPC provide an improved and rich set of calls to the programmer [54].

MIT Athena Project RPC was developed to study the design and applicability of RPC models under certain restrictions. Stanford Modula/V RPC is generally noted

as improving performance and response time of requests on the server side [54]. The first RPC model to handle orphans (i.e. client dies without notifying the server, this leaves the server orphaned), was the Rajdoot RPC [46, 54].

1.2.4 Future of RPC

Even today, new RPC systems are developed and used. These new systems include Facebook's Thrift, Google's Protocol Buffers, and Cisco's Etch [57]. Yet these modern systems differ from the previously developed tools. The modern RPC systems that today's tool represent are highly specialized protocols and are applied in limited use cases which do not resemble the scenarios that RPC originally set out to address. Their conceptual design, capabilities and use cases are beyond the range of what traditional RPC systems tried to achieve. In the case of Cisco's Etch, their RPC model and tool might even aim at a completely different market segment, that of embedded devices [57].

Approaches similar to the general RPC mechanism emerge in [44] which enforces security policies by separating parts of an application in privileged protection domains. Here the user is also required to provide a security model description in a configuration file. Taking this concept further is COMET [32] which enables a multi-threaded program to dynamically offload threads during runtime to different machines. While these modern concepts are similar to RPC, they are, however, more closely related to process and virtual machine (VM) migration. Additional discussion on alternate approaches, such as service oriented architectures (SOA) is presented in Section 2.1.1.

1.3 Thesis Statement

We show that an automated mechanism to generate a distributed application from a monolithic legacy application written in *C* is possible and that is provides certain benefits over existing mechanisms. At the same time the automated mechanism described here has drawbacks which make automated RPC generation inferior to approaches requiring greater levels of manual programmer involvement. Current RPC models lack the ability to effectively compete with other technologies, mechanisms and approaches in today's world. Despite these concerns, we show that automated RPC mechanisms are beneficial for certain scenarios involving legacy applications.

The goal of this thesis is to answer the following question: **Can an automated mechanism successfully generate a distributed system stemming from a monolithic legacy *C* application that is suitable for today's technological realities of large data and distributed systems**?

1.4 Thesis Outline

Chapter 1 has provided the problem definition and scope that this thesis is situated in. It has also provided a general overview of the content and structure of this work.

Chapter 2 provides the background and related works in the field of RPC system and models. It also provides an overview of two interconnected tools and approaches of RPC systems which will provide the reader with some insight into the implementation and usage of these systems. The hope is that the reader will gain a comparative understanding between RPC systems, their principles and models, and how ALRPC expands on certain specific aspects of them.

Chapter 3 explores the contributions of ALRPC. It also describes the design, implementation and experiences gained from ALRPC. This chapter forms the basis of evaluating ALRPC as a mechanism to transform monolithic applications written in *C* into a distributed system. It will also contribute to determining whether the RPC model is appropriate and feasible in today's world.

Chapter 4 describes the experimental setup and results that were gathered throughout the process. These results will then be used in Chapter 5 to provide an evaluation of ALRPC as a mechanism and the feasibility of the RPC model in today's world. In evaluating ALRPC we chose to directly compete with a modern industrial strength tool which is used for *C* systems: RPCGen [4]. Lastly, chapter 6 concludes this work and presents a motivation and vision for future work.

Chapter 2

Background of Remote Procedure Calls

This chapter introduces the reader to Remote Procedure Calls. We illustrate high-level concepts of RPC as well as interesting technical details of their implementation. To demonstrate these concepts and technical details in existing RPC tools and mechanisms, we showcase RPCGen and SunRPC. Additionally, this chapter also provides an insight into the eventual decline, critiques and unresolved issues with RPC systems.

2.1 Conceptual, Technical Details of Remote Procedure Calls and its History

A remote procedure call is, generally speaking, a mechanism to allow communication between two separate processes: a client and a server. On the client, a call is made to a procedure stub which actually forwards the parameters to a remote server which is listening for incoming connections. At the server, the message is passed on to the remote procedure. At the remote procedure the actual target function is implemented. Finally the results are transfered back to the client.

In an RPC system none of the components in such a model are conceptually making anything other than local procedure calls, with the exception of the stubs. What is even more appealing than this, is the fact that the programmer generally does not have to manually program the communication aspects; they are usually produced

automatically by the compiler or a specialized RPC tool [53].

2.1.1 Rise and Fall of RPC

Remote Procedure Calls (RPC) have existed in the literature since the early 1980s and perhaps as far back as the mid 1970s [14], see Section 1.2.3. The main purpose is, as previously mentioned, to provide a mechanism that facilitates communication across a network between processes written in high level languages. RPC's call model is very similar to that of local processes. The RFC specification for RPC states that in the RPC's caller model "[o]ne thread of control locally winds through two processes" [51].

Remote Procedure Calls have gained prominence starting in the 1980s. During the 1990s, the RPC world strongly moved towards incorporating Object Oriented design and coding principles. But during that time, RPCs' presence was also further reinforced by several patent applications which incorporated RPC techniques and object oriented design strategies. Despite this strong rise of RPC tools and attempts to remain technologically relevant, RPC as a mechanism has gradually declined in popularity in the past decade.

In the 1980s many RPC efforts focused on making the underlying architecture compatible in heterogeneous networks by defining the structure of RPC and the underlying protocols [12]. Other efforts were aiming to ensure that RPC implementations would be similar in use to local calls to facilitate the ease-of-use and adoption of this programming paradigm [13, 14]. Despite the detractors of RPC, such as Tanenbaum and Vinoski [53, 57], RPCs continued to pick up steam during this period. The early 1990s saw an increase in work geared towards improving the efficiency of RPC, building on early analysis from the late 1980s [48]. Certain layers in the architecture of RPC were identified as performance bottlenecks [19], and a general effort to improve the interprocess communication [11] occurred.

This continued progress and increased availability of distributed computational resources, coupled with the ease with which Object Oriented programming could leverage serialization and remote procedures, led to a multitude of patent applications regarding RPC. These patent applications included work for object oriented remote

procedure calls and more often than not focused on specific techniques to increase performance or ease of use [23, 30, 33]; interprocess communication patents in more modern times capture the need to connect processes written in different languages [39], or on heterogeneous devices [15, 23]. The current state of the world includes several tools which grew out of this era of an RPC boom in the 1990s. SWIG and Sun's ONC RPC are just 2 examples [5, 42, 51] and will be discussed in section 2.2 and section 2.3. RPC frameworks were developed to leverage the new paradigms [49], if only in a more abstract way [50].

Legacy systems were slow to adapt to the recent change of cloud computing and distributed systems. Modern languages made the jump quickly and incorporated a simple design structure to facilitate the refactoring or development of applications for the cloud [36]. In fact, many argue that RPC is in decline and is continually being replaced by RESTful architectures simply because they yield adequate results with less burden to the programmer and are not subject to RPC shortcomings [56,57].

Additionally, service-oriented architecture (SOA) is often used to solve spatial distribution of data and the related challenges of self-adaptive systems. SOA approaches also acknowledge that hierarchically structured, stable monolithic systems have moved to distributed federated systems because of changes in technology, user requirements, and legal requirements. SOA is one abstraction used to address this challenge [26]. Adding self-adaptive deployment and configuration models of SOA systems addresses the challenges of networked distributed systems [55]. Similarly Cardinelli proposes self-adaptive models to dynamically react to environment changes to increase a system's dependability [16].

Surpassing the solution space of RPC, today's technology and increasingly complex systems led to discussions of software reconfiguration patterns for dynamic software adaptation in distributed systems; Gomaa describes patterns for transactions where more than one service needs to be updated and coordinated [31]. Other methodologies to integrate design decisions with self-adaptive requirements of the system are also proposed to support goal based approaches which allow system modification at run time [7]. Bencomo proposed a slightly different approach where a more formal methodology is used to describe a solution more closely linked to that of middleware [9].

SOA exposes a service to the application while at the same time not revealing any implementation details. Middleware approaches only replace communication methods of SOA applications and not the conceptual model. RPC for legacy applications competes in many cases with these solutions.

2.1.2 Technical implementation of RPC systems

Since their inception in the 1970s, RPC has followed the same basic concept. Following the same architectural design since the 1970s, RPC design has only undergone minor and incremental changes to remain as technologically relevant as possible. However, the same design still applies; a function or procedure is executed in a separate process space. Often the separate process space is located on a different physical machine. Birrell points out that procedure calls in the same process are well understood and as such RPCs should aim to provide an equally easily understandable mechanism to facilitate procedure calls across network boundaries [14]. The basic conceptual sequence of RPC events can be classified as follows:

1. The client calls the client stub via a local procedure call, pushing parameters to the stack in the normal way.

2. The client stub serializes the parameters into a message. This is called marshalling. Then a system call sends the message.

3. The client's operating system sends the message to the server.

4. The local operating system on the server passes the incoming message to the server stub.

5. The server stub unpacks the message. This step is called unmarshalling.

6. Lastly, the server stub calls the server procedure.

Once these steps are completed, the results from the server procedure are transmitted back to the client in much the same way. Figure 2.1.2 illustrates the process graphically.

While the RFC specification describes the call model with an example using only one server and client process, the specifications do not limit the concurrency model

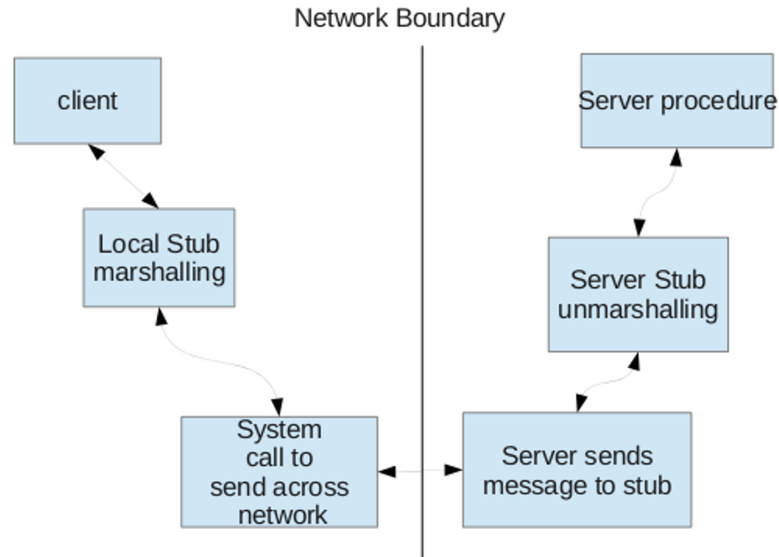

Figure 2.1: Conceptual Overview of the Processes guiding any RPC use.

to prevent multiple client or server threads [51]. For simplicity, the call model has the client caller send a message to the server, which waits for requests; meanwhile the caller blocks (waits) for a response from the server containing the procedure results [51]. The message sent to the server contains information about the procedure parameters, the parameters themselves and other meaningful information for the server process.

The exact details of the marshalling and data structures which are used to contain the packed parameters differ depending on the implementation one uses for RPC. Size and number of parameters which need marshalling can influence the size of the buffer which is sent across the network. However, this again depends on the exact implementation of the communication protocol and techniques used. Protocols and communication standards define sizes of data to be transmitted. If then, the marshalled parameters are smaller than the data segment for a specific protocol, the transmitted size will still be the minimum size of the used protocol.

Bershad classifies three distinct components at *call time* that influence the cross network communication aspect of RPCs. First, the transport protocol. This protocol does not need to be specific to the RPC implementation, a previously established protocol (such as TCP) can be used [12]. Secondly, a call will contain some *control*

information to keep track which state the call or the system is in. This information is included in every transmission that is made between client and server. Lastly, Bershad emphasises that at call time there needs to be a convention for the *data representation*. Especially for heterogeneous devices a common representation ensuring compatibility of the data is crucial. This means that compiler and machine specific differences in alignment and orientation of data and data structures need to be accounted for. However, the compensation for these issues in RPC should be abstracted away from the programmer since RPC should be usable just like a local procedure call [12].

One risk of this is that it may in some cases introduce inefficiencies and undue overhead. However, the overhead is manageable in most cases. Using socket operations and established networking protocols, the size of the message that is passed generally ranges in the kilobyte. Typically a memory page size in Linux is at the very least 4kB in size, though different processors and configurations can produce other results. As such, we chose our message buffer to be 4kB in size. Consequently, the most efficient data transfer occurs if this buffer is fully utilized, since at the receiving end a minimum of 4kB are written out regardless of the buffer being smaller. Subsequently, the message size and agreement on data representation is unique to the implementation of the RPC system.

In order to correctly marshal and unmarshal the parameters of functions a certain level of type information of the parameters is required. The following sections in Chapter 2 discuss this in more detail.

2.1.3 Summary

We presented a high level understanding of the evolution of RPC systems since their inception in the 1970s. The reader is introduced to the rapid rise, improvements and expansions of RPC systems that took place. At the same time the decline of RPC systems and the question regarding their feasibility are also raised.

We also demonstrated a high-level conceptual introduction to RPC systems in general by providing a glimpse into the technical problem that RPC systems are trying to

solve. Our system, ALRPC is situated in the same problem space and thus is trying to compete with other existing RPC tools.

2.2 RPCGen

RPCGen is one of the more popular tools for generating RPC code. Because of its active use, availability and because it covers exactly our target use case of *C* systems it was chosen for a comparative analysis.

On Linux distributions, RPCGen is included in the standard *glibc* package. One of the conveniences of it is that it can be used from the command line. The manual pages [4] demonstrate over a dozen different uses and options for it, making RPCGen a versatile tool with quite a sizable number of customization options. Its main use is that of a compiler-like processor to convert input conforming to a specific description language to C. Several files are generated automatically, while the user is left to develop specific code for both the client and the server components.

RPCGen is formed by combining two components, which are integral in RPCGen's purpose of generating C code:

- External Data Representation (XDR)
- Remote Procedure Call Language (RPCL)

2.2.1 RPCGen and External Data Representation (XDR)

XDR is specified in RFC4506 [27] which, in 2006, replaced the specifications from 1995 [1]. XDR is used to formally describe data types in a manner that is independent of the architecture or platform on which they are used. Within the specifications one finds the precise definitions and representations of most common data types used in C, including: *int* (integers), *enum* (Enumeration), *float* (Floating-Point), *struct* (Structures), etc.. Additionally, one also finds language specifications for notational conventions of XRD, as well as lexical and syntactic notes. The specification documentation makes it explicitly clear that XDR is not a programming language and it can only be used to describe data formats. An important consideration of XDR is that all data types are always stored as a multiple of 32bits (4 bytes) [27]. To

highlight the detail within these specifications lets consider the data type *Integer* and *Enumeration*. Figure 2.2 [27] shows the specification of an XDR integer. Of note here is that the most significant bit is on the left.

The *enum* type for enumerations is also strictly defined with RFC4506 and can be seen in Figure 2.3 [27].

```
                int identifier;

      (MSB)                   (LSB)
      +-------+-------+-------+-------+
      |byte 0 |byte 1 |byte 2 |byte 3 |   INTEGER
      +-------+-------+-------+-------+
      <------------32 bits------------>
```

Figure 2.2: Integer specification in XDR Specification RFC4506.

```
enum { name-identifier = constant, ... } identifier;
```

Figure 2.3: Enumeration specification in XDR Specification RFC4506.

Both examples in Figure 2.2 and Figure 2.3 are representative of the rigour with which XDR is defined. This level of precision in the definitions is required to formally define data and data type representations of XDR. XDR's main purpose within RPC is to allow for a formal description of all standard and custom data types that a user may wish to include in a C program.

2.2.2 Remote Procedure Call Language (RPCL)

Just like XDR is used to describe data and data types, RPCL is used to describe the remote procedure in a meaningful way. RPCL uses XDR declarations of the data within it's description files. RPCL, in contrast to XDR, is a programming language. The syntax of RPCL is designed to be similar to the syntax of C. This similarity is not accidental and so Oracle describes the RPCL as an extension to the XDR language [45] following a C style syntax. Within the RPCL description the programmer

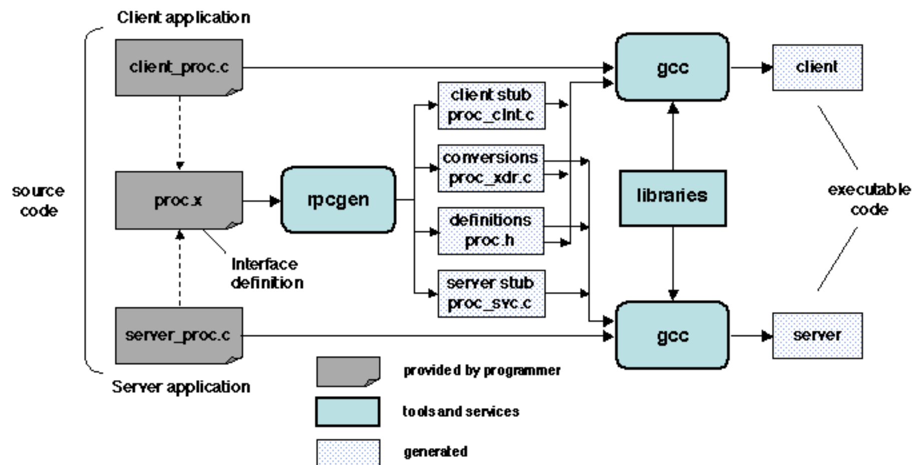

Figure 2.4: RPCGen overview [35].

is required to specify and define data types (constants, enumerations, structures, etc.) as well as function/prototype names and signatures.

The code specified in RPCL is used by RPCGen to generate multiple C programs. To do so, several files are generated automatically: *1*) A header file containing function declarations and variables common to client and server program, *2*) a file containing marshalling functions, *3*) and a file containing stubs with which the client communicates with the server. Figure 2.4 shows the architectural overview of the files and tools used by RPCGen. The same figure also shows the common tools, like GCC, which are also required.

RPCL incorporates data type definitions written in XDR, but it also allows for definitions of functions and programs in a manner similar to XDR yet closely resembling *C* syntax. The complete RPCL file is then read by RPCGen in order to produce the RPC code. The RPCL descriptions are, by convention, saved in an *.x file. In this *.x file, RPCL expects that each procedure is “bound to a name formed by the name of the declared program (but upper cased), followed by an underscore character, and the version number which the referred procedure belongs to” [43]. In effect a C function called *int foo (int x)* would be expressed as *int FOO(int) = 1;*. The only unfamiliarity over C in this example is the ”= 1” at the end of the line. RPCGen allows for multiple versions of any given function or program and uses this notation to mark the version numbers. The declaration of *int foo(int)* would be surrounded by

a similar structure to identify the program and its version in which it resides. Figure 2.5 illustrates this convention of RPCL via the standard example given in nearly all RPCGen tutorials.

Figure 2.5 showcases how a data structure used for linked lists and a function to print the list are represented in RPCL. Enumerations and structures are nearly identical to C, with the exception that the "<>" syntax denotes data of variable length. Program and function names are given in all capital letters and have a version number attached to identify them. By intentional design it is possible to have several identical functions which only differ in their version number. More complicated RPCL setups are discussed in Section 4.

```
enum color {ORANGE, PUCE, TURQUOISE};

struct list {
        string data<>;
        int key;
        color col;
        list *next;
};

program PRINTER {
        version PRINTER_V1 {
                int PRINT_LIST(list) = 1;
        } = 1;
} = 0x2fffffff;
```

Figure 2.5: Sample RPCL code modified from [2].

The default use case of RPCGen is to treat this *.x file as an input parameter to RPCGen. Subsequently it generates four additional files. A flow chart of the architecture is also provided in Figure 2.4. These four automatically generate files are:

1. *.h: C declarations based on the input file.
2. Server code which listens to incoming RPC requests.
3. Marshalling code to serialize the data for transmission.
4. Function stubs which the client can use to communicate with the server.

Whilst these files are generated automatically, the programmer is left to write the code for the function implementation on the server side. Additionally the programmer also has to write the client code to connect to the server and the code to use the RPC functions. Figure 2.6 shows a sample of the client code. As we can see here, RPCGen produces code, and requires the programmer to use existing function calls such as *clnt_create*, which are in the *<rpc/rpc.h>* system library.

```
...
...
    cl = clnt_create(argv[1], PRINTER, PRINTER_V1, "tcp");
    if (cl == NULL) {
        printf("error: could not connect to server.\n");
        return 1;
    }
...
...
```

Figure 2.6: Segment of Sample C client code to establish connection to server. Modified from [2].

Another similar tool which is closely related to RPCGen is SWIG (Simplified Wrapper and Interface Generator). It is often mentioned along side RPCGen when discussing remote procedure calls or legacy applications. SWIG is highly automated and specialized; it allows programmers to connect *C* code with scripting languages platform independently. In this manner, SWIG is used to connect *C* code with a range of other languages automatically and connects the original *C* code with scripts which are run in other process spaces [8]. SWIG is well established to connect *C* code with other scripting languages, but it does not provide the ability to connect *C* code with other *C* code directly.

2.2.3 Summary

RPCGen relies on XDR and RPCL to describe the data and the associated functions. Additionally, it makes use of existing code in the *<rpc/rpc.h>* library. The syntax is similar to that of C and creates the expectation of a relative fast learning curve for using RPCGen. On the other hand, the programmer is required to write RPCL code, including XDR descriptions of data, which are then used to generate new C

code. This does not allow the programmer to simply convert existing C code with RPCGen. Instead, the programmer has to manually write RPCL files, then manually replace the appropriate C code with the automatically generated output from RPCGen. Additionally, the programmer is required to write the code for the client to use the RPC functionality and is also required to write the code for the server which implements the intended functionality of the call. There is no method to automatically turn existing code into a distributed RPC environment.

In this section we have introduced RPCGen. Both the usability, use and technical implementations were presented. RPCGen was chosen as a direct competitor with which ALRPC's capabilities are measured. This choice was made because RPCGen is targeting *C* systems exclusively and it is an industrial strength tool which is still used today.

2.3 Sun's RPC

This section aims to introduce one of the best known tools in the RPC world for *C* systems, namely Sun RPC. The section covers a high level overview of the history of Sun RPC, a technical overview, and concludes with remarks on the rise and fall of Sun RPC.

2.3.1 History of Sun RPC

The history of RPC tools reaches back several decades. This chapter focuses on one of these tools which has gained prominence in dealing with the *C* language. This established RPC tool for applications written in the *C* language is still in existence today. Sun also put forth strong support for standardized RPC protocols, which aided in the directed growth of Sun RPC. Sun RPC is often also referred to Sun ONC or ONC RPC.

Sun RPC was introduced in the early 1980s by Sun Microsystems to facilitate communication between heterogeneous devices in distributed systems. Particularly because Sun RPC has been introduced in 1984 [42] it has undergone a range of developments to keep it technologically relevant as best as possible. Shortly after it's first introduction Sun Microsystems introduced a Remote Procedure Call Protocol Specification as

a proposal for all RPC communication protocols in 1988 [40]. Throughout the years, many have sought to develop optimizations for these protocols including optimizations for Sun RPC [11, 42]. Because of the continued development and optimization efforts Sun RPC can still be found in today's systems. Today, RPC code is found in OpenSolaris, Linux, various portmap deamons, Network File Systems, glibc and kernels can be traced to have originated from Sun RPC [38].

Sun RPC's use has risen rapidly since the 1980s. Its code can be found, in one way or another, in many systems' RPC libraries that exist today. This rise was helped by the propagation of heterogeneous distributed systems, the adoption of protocol standards and distributed computing architectures.

2.3.2 Overview of Technical Aspects

Sun RPC leverages two distinct elements which make it an attractive tool for RPC use cases. First RPCGen, which is a stub generator (described in Section 2.2), is integrated with Sun RPC. Second, it abstracts the burden of network communication away from the programmer.

Because the Sun RPC package also contains RPCGen, it makes use of the interface description language XDR for data representation. Subsequently, it produces client and server stubs, including some of the procedure code for marshalling and unmarshalling [21]. The integration of RPCGen and XDR in Sun RPC serves the purpose of facilitating conversions of parameters and types for transport over networks between heterogeneous distributed systems where the conversion from types declared at the language level to bytes at machine level (and back) must follow some predefined algorithm [41, 43]. For a more complete explanation of RPCGen turn to Section 2.2. By leveraging RPCGen's built in capabilities, Sun RPC can pass arbitrary data structures and data types between processes via conversion to XDR specified models [21]. Most of the time the default implementations are sufficient for generating RPC applications. However, Sun RPC also provides the programmer with access to lower-level controls. Coulouris mentions the following lower-level facilities: testing tools, dynamic memory management in marshalling procedures, broadcast RPC where messages are broadcast to all available services, batching of non blocking calls, call-back by the server to the client, and authentication [21]. These lower-level

facilities are intended to give the programmer greater control over additional aspects of the RPC code if needed.

To efficiently implement the standards described in [40] Sun RPC code consists of a set of microlevels which deal with various aspects of the protocol stack independently [42]. This means that there are modules, which can be considered independent from one another, that for example deal with writing data during the marshalling stage, while another layer exists that controls the reading of data during the unmarshalling stage [42]. Each of these levels provides vectors for independent application specific optimizations. At the most basic level however, Sun RPC is based on the existence of a port mapper which maintains a list of all available programs and their respective processes. [41, 43].

Muller also describes the optimization efforts for RPC calls in general and Sun RPC specifically [42]. Muller, while focussing on automatic optimization of Sun RPC, points out that most optimizations of Sun RPC tend to focus on specific aspects of the protocol stack due to the design of Sun RPC which is based on independent mico-levels [42].

2.3.3 Adoption, Success, and Decline

Despite the maturity of the Sun RPC project (it has existed since 1984 [42]), RPC in general is in decline recently (see Section 2.4). Creation of standards and integration of RPC protocols in a variety of systems has facilitated the adoption of Sun RPC and its underlying principles in the world of distributed computing. However, despite various optimizations and the widespread use, RPC and Sun RPC are in decline. Other alternatives provide a sufficient solution to similar problem domains with less effort and risk [57]. New mechanisms are replacing RPC as the dominant form of inter-process communication in a distributed heterogeneous network. Further, the changing landscape of programming languages added to the demise of RPC for legacy systems. Object Oriented Programming (OOP) gained wide-spread popularity in the 1990s and brought with it the advancement of RPC protocols and tools which would facilitate OOP in distributed systems. CORBA is one such example which does include bindings to C (with additional caveats), but instead focussed on including bindings to a number of newer programming languages [58]. Moreover, a greater

concern seems to be the issue of security. Exposing communications to the Internet or other open network traffic is always associated with risks. Neither RPCGen, nor Sun RPC, have provisions in terms of security. Many even argue that Sun RPC and generally most RPC tools and protocols should never be exposed to the Internet at all [34].

2.3.4 Summary

We have shown that Sun RPC has existed for quite some time. Throughout its existence it has undergone developments and changes to stay technologically relevant and to improve its appeal as an RPC tool. Thus, despite detractors' views regarding the confusion of XDR and RPCL, Sun RPC remains "one of the simplest model among all the RPC implementations" [54]. However, RPC systems have been in decline recently as we have seen.

2.4 Recent Decline of RPC's use

In this section we discuss some of the persistent critiques which plague the RPC model. Scepticism and critiques of the RPC model have existed since the early stages of development of RPC applications and tools. Most of these critiques from the 1980s still apply to the RPC model despite the continued improvements that were made to it. We shall highlight the early critiques identified by Tanenbaum in 1987 [53], while relating their relevance to recent criticism of the existing RPC tools and model. A discussion of specific limitations of the ALRPC system introduced in this thesis will follow in Section 2.5.

2.4.1 Rise and Fall of RPC

Tanenbaum presents one of the early critiques of the RPC model. The unsurmountable weaknesses of RPC are of both conceptual as well as technical nature. Tanenbaum describes several issues with RPC as "unpleasant aspects" [53], all of which lead him to advise against using RPC as a general interprocess communication model. At the same time Tanenbaum, however, conceeded that RPC works satisfactorily in use cases that require interaction between a client and a file system. On the other hand, in Tanenbaum's view, RPC is not advisable as a general model, since general models

should not ”require programmers to restrict themselves to a subset of the chosen language” [53]. This view is shared by Vinoski in that even ”the earliest, buggiest RPC framework of the time was good enough for the small scale system of the day” [57]. In fact Tanenbaum describes a hypothetical test for generalizability of models. In this model, two programmers write interconnected procedures independently from one another with the assumption that the system will run in a local environment. However, when the components are combined, both will run in different environments. A general RPC model should be able to deal with this situation since the premise of RPC is an application layer abstraction to hide the network [60]. According to Tanenbaum, the RPC model fails this test. Things will go wrong because of the attempt to make everything look like a local call; transparency is lost when many problems are solved manually based on a programmer's understanding of which functions are local and which are remote [53]. Ultimately, this takes away the benefits of RPC and illustrates that the RPC model is not generalizable.

Additionally, there are several concrete problems which persist in RPC to the present day.

1. Conceptual problems with the model itself
2. Technical problems with the implementation
3. Client or Server crashes
4. Heterogeneous system problems
5. Performance

2.4.2 Conceptual Problems

Tanenbaum highlights a range of valid conceptual problems with the RPC model. The first example is on the diffuse boundaries between establishing which component is to act as the server and which is to act as client. Figure 2.7 allows us to identify several client and server components. The components are *sort*, *infile*, *uniq*, *wc* and *outfile*. Linux allows for piping the output of one program as the input to another. This ability now creates a conceptual model where a component is both client (when forwarding output to the next element in the pipe) and server (when receiving information in the form of input form the previous component in the pipe). The same

conceptual setup can occur in RPC, creating semantic and conceptual problems of identifying which components are the server and which are the client.

Servers can send unexpected messages back to the client. These unexpected messages might be to notify the caller of a runaway process in the server. However, the client must now be configured and programmed to handle these unexpected messages.

```
sort < infile | uniq | wc -l > outfile
```

Figure 2.7: Tanenbaum's example on conceptual distinction of client and server.

Tanenbaum also makes the observation that a potentially disastrous problem exists in regard to data in the RPC model. Consider that the server process generates some data from a real-time experiment. This data had been requested by the client. However, when the server sends the message containing the data back to the client, it does not know that the client has received the message. How long should the server hold on to the irreplaceable data? One solution would be to keep the data until an acknowledgement has been received from the client, or to resend it if it didn't process the acknowledgement. Yet this solution is imperfect. No scenario exists that would guarantee the arrival or sending of either the data nor any number of acknowledgements [53]. In other networked communications this problem exists as well, however it only occurs at the end of the communication. In RPC this issue exists at every single procedure call to the server.

A further conceptual challenge is the issue of multicast messages originating from the client. Tanenbaum points out that this is purely a software issue confined to the RPC model when one client wants to send messages to multiple servers. Hardware devices support multicast and broadcast messages, while the software code of RPC does not. Here each call goes out to specifically one server [53], though as we have seen in section 2.3, some of the later versions of RPC do allow broadcast messages for some type of procedure calls [21].

2.4.3 Technical Problems

Technical problems are the most prominent of the problems with RPC. Tanenbaum identifies three broad classifications of technical problems [53]. This section explores

these limitations within RPC systems. ALRPC, and many other existing RPC tools, are facing a number of these challenges to this day.

The first technical problem that Tanenbaum points out is parameter marshalling. Strongly typed languages do not pose any problems in identifying the number, size and type of a function's parameters. However, non-type safe languages like *C* make this impossible. For example **char[]* is ambiguous and semantically not distinguishable from a pointer to a single *char* or an array of strings. Further, the function *printf* can take multiple parameters each of a different type and size, making it near impossible to deal with in an RPC environment.

Secondly, parameter passing poses a problem. Any value parameters do not impact the system, however, reference parameters cause problems. First, one has to be able to identity the parameter to be a reference type, which poses challenges. Ultimately, however, pointers require one of two expensive strategies. One could only pass the pointer and request the data when it is used on the server. This approach, however, destroys the paradigm set up in the RPC model where everything is to be treated as a local call and that the compiler does not know that it is dealing with an RPC system. RMI and most Object Oriented languages when used in conjunction with RPC have overcome this limitation, because specific measures were taken to identify and address this challenge [59]. In non Object Oriented languages, however, one can often serialize the entire data structure that the pointer references. This approach is very expensive and often impossible in non-type safe languages like *C*. Consider a union of different types. Here it is impossible to determine which actual structure the pointer is referencing.

Likewise global variables used inside functions cause problems. Tanenbaum poses the question of how globals are supposed to be handled in remote procedures [53]. There are no satisfactory answers to this question and one has to resolve to accept that functions which contain global variables are simply not suitable to be remote procedures.

Ultimately, these technical challenges have not been overcome even today. RPC systems remain fundamentally flawed as a generalizable computing model because it ignores the realities by attempting to make the network appear to be just another

part of the local environment [56].

2.4.4 Crashes

In a networked environment there are three components which can unexpectedly fail: the client, the server, or the connection. RPC forces the programmer to introduce exception handling for these types of failures for any remote call. This now introduces additional code and error checking and exception handling into code which in a local environment would not have the possibility to fail at all. This issue persists to the present day as Vinoski makes clear [56].

Additionally, idempotent and non-idempotent function calls pose challenges to the system as well. Remote non-idempotent operations are impossible to detect if the server crashes. Consider updating a file via a remote procedure. If the server crashes, the client will not know about the failure if it wasn't expecting an acknowledgement. However, even if this acknowledgment was required, errors are introduced if the server crashes after updating the file, but before having notified the client. Timeout waits and re-doing operations now would introduce errors [53].

The loss of state also poses problems. Unrecoverable errors occur if a server crashes that held state information on the number of open files. Even a timely restart before the next client's request will not recover the lost information. Today, other non-RPC systems are able to deal with this issue through adequate replication, but RPC itself does not handle these cases [53].

A comparatively minor problem is the creation of server orphans. Servers wait for messages from clients. If a client crashes the server has lost its purpose but continues to exist [53].

2.4.5 Heterogeneous Systems

While the existence and communication of heterogeneous machines is an every day occurrence, the way this communication is designed within RPC poses challenges. Primarily the byte order, structure alignment and type sizes need to be codified and determined. Some RPC implementations, as we have seen in section 2.2, implement such a description [27]. At the same time, different implementations have been proven

to be impossible to create an accurate mapping of the data to its representation in a message [37, 56]. Assuming heterogeneity was abandoned to some extent with the introduction of RMI as it was determined that it is too restrictive and actually causes several of the problems with RPC [59]. RMI for Java for example assumed that at both ends of the network connection JVMs would handle the call.

2.4.6 Performance

Lastly, performance is considered one of the "unpleasant aspects" of RPC systems [53]. First of all there is no parallelism because either, or both, the client or server are idle while waiting for one to send the next message. Additionally, there is no streaming of large results possible because a message has to be fully assembled before it can be sent [53]. This is especially burdensome when one has to send extremely large data segments [56]. Most disturbing, however, it seems for Tanenbaum is the fact that programmers write inefficient RPC code [53]. Often short functions or procedures exist which are called numerous times. If one were to execute these types of functions in a remote process, severe performance hits would occur in addition to being liable to all the aforementioned critiques of the RPC model.

2.4.6.1 Summary of Critiques of RPC model

Critiques of the RPC model have existed for nearly 30 years. Nearly all of the issues with the RPC model still exist in today's implementations. Many alternate approaches are less complex than RPC, while still providing the same results. RESTful services for example are message focussed and considered less expensive, simpler and easier to implement [57]. The improvements of RPC which aimed at addressing the shortcomings, many of which were identified by [53], have led to a number of RPC systems: Sun RPC, Apollo NCS, DCE, CORBA, J2EE, SOAP, etc. [57]. In the end, RPC is complicated and overshoots the needs of many of its potential customers while at the same time giving competing technologies, like REST, an avenue to provide a service that is "good enough" [57].

2.5 Limitations of RPC within Legacy Code

This section brings together the general limitations of RPC systems and highlights the limitations which persist in ALRPC. Most of the limitations of ALRPC are carried

forward from Section 2.4 due to the fact that ALRPC tries to adhere closely to the RPC model; ALRPC also targets a non-type safe programming language, and; ALRPC attempts to automate the RPC generation as much as possible. Subsequently, ALRPC has limitations in the following areas *a*) parameter passing, and *b*) type detection.

2.5.1 Parameter Passing

Firstly, parameter passing identified by [53] and discussed in Section 2.4 poses a problem when the parameter is a pointer. In fact, J. Waldo points out that RPC systems were never designed to easily handle anything but primitive data types and composites of primitive data types [59]. With this in mind, general identification of pointers poses a problem and ALRPC makes several unsafe assumption when encountering certain types.

The identifier **char* in *C* could reference a single *char* primitive type, or it could point to the beginning of a string. ALRPC assumes that **char* is always the beginning of a proper string. Evidently this assumption excludes a lot of cases.

Additionally, identifiers connected with the *'*'* character are always assumed to be pointers. We have seen above that pointers in *C* can be ambiguous. Two strategies are proposed by [53] to deal with pointers. ALRPC's approach is to make a copy of the data structure by value when the actual parameter is a reference parameter. For example a **double* is dereferenced and its value is placed in the message buffer. This transfers the data to the server rather than just the pointer. However, Tanenbaum [53] also pointed out that this approach has only limited suitability to be generally applicable. Furthermore, pointer chains (e.g. linked lists) are not dealt with in ALRPC. This limitation is due to the ambiguous nature of *C* and the non-type safe semantics of the language. Correctly identifying structure internal pointers automatically based on prototype definitions proved impossible. The only information given to ALRPC is the prototype signature in a header file which does not contain information about the internal representation of the structure.

2.5.2 Type Detection

Type detection is the ability to determine the type of a parameter. As we have seen previously, *C* is not type safe. Consequently, detection of types in an automatic fashion is not generalizable when ALRPC has to work with the constraint of inferring and determining types solely from the procedure signature.

The possibility of passing *void* pointers prevents ALRPC from making any inferences about the actual parameter type. *Void* pointers and types could refer to any type and require programmer knowledge to cast the type correctly in the procedure body. None of this information is available in the procedure signature, which is the core constraint of the information that is available for ALRPC to analyse.

2.5.3 Discussion

Despite the limitations described previously of ALRPC, we believe that ALRPC provides benefits which outweigh its inferiorities compared to other tools such as Sun RPC or CORBA. Sun RPC is discussed in Section 2.3 and requires the programmer to learn and produce an almost entirely new language and description language. As such it is contrary to ALRPC's attempt to provide as much automation as possible. Likewise, CORBA is not suitable for legacy applications written in *C* either. CORBA is not object oriented and requires the programmer to manually emulate object oriented features in C code, making it a rather tedious type of work [58].

Additionally, we believe that ALRPC also addresses the issue of marshalling *C* code. Once marshalling is achieved, an application can integrate with any transport mechanism, whether that is REST, ALRPC or any other technology. Marshalling of *C* function parameters in an automated manner is key to using either of these transport mechanisms, each of which has their own benefits. For this thesis we focus on RPC as a transport mechanism.

Chapter 3

ALRPC

This chapter discusses the design, implementation and use case experience with ALRPC. The chapter is organized to first present the conceptual design of ALRPC which closely follows that of other RPC tools. Following this, we present a look at the parsing and code generation capabilities of our tool. We conclude this section with a discussion of the existing limitations of both the mechanism and the existing implementation of our tool.

3.1 Conceptual Design of ALRPC

In this section we discuss the high level overview of the software tool ALRPC, its design, architecture, implementation and usage.

ALRPC is a RPC mechanism which can be used as a command line tool for Linux. It adheres to the RPC models described in Chapters 1 and 2. Following this approach, ALRPC generates, stubs for client and server, as well as communication procedures automatically. The goal of ALRPC is to automate the entire process as much as possible. Automation is the primary goal. As such, the intended use case is to provide ALRPC with existing monolithic application code which is then transformed automatically into a distributable system. ALRPC's input is kept simple. Generally a programmer is only required to provide a header file which includes the procedures one wishes to execute remotely as an input parameter to ALRPC. From this input alone, ALRPC generates client procedure stubs, networking code, server procedure stubs, marshalling and unmarshalling code automatically.

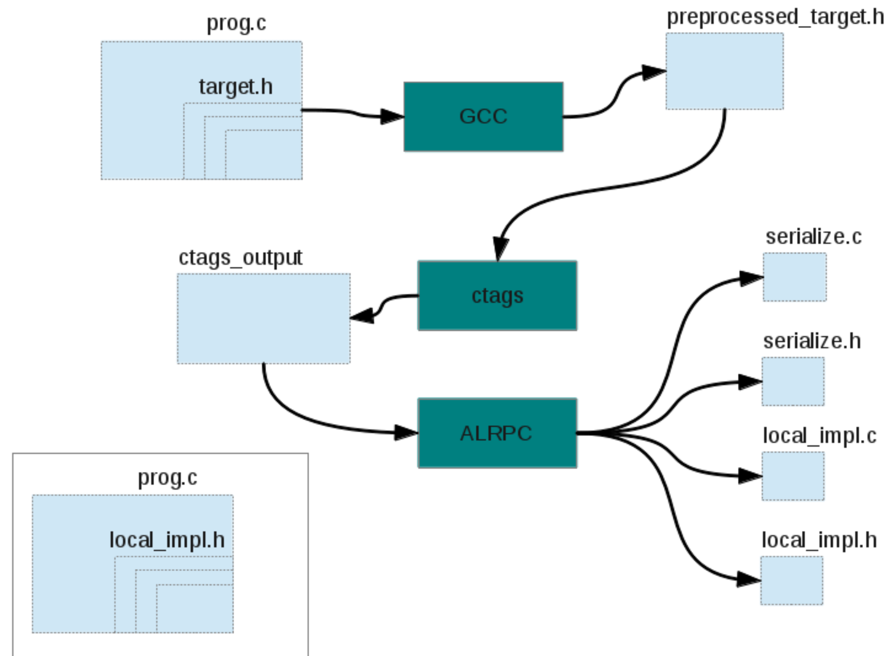

Figure 3.1: Static control flow of ALRPC tool chain.

The purpose of this header file is to allow ALRPC to perform a static analysis of the given source code to determine the function signatures. This analysis is necessary to determine the data types and names of the procedure parameters, as well as the names of the procedures themselves. Figure 3.1 illustrates the process, actors, input and output files involved when using the entire tool chain centred around the ALRPC mechanism.

3.1.1 Usage of ALRPC

We will see through the remainder of Chapter 3 that the ALRPC tool operates at multiple levels of code lexing, comprehension and generation. Therefore there are various commands required to fully use the ALRPC tool.

To invoke the tool chain of the ALRPC mechanism a simple shell script is required. This shell script in turn invokes the appropriate tools and keeps track of the input and output files. The initial command is ***alrpc.sh target.h***. The parameter *target.h* is the header file within the application which is destined to be executed remotely.

Note that *target.h* can contain recursive headers which are then also prepared for execution in the remote server process. At first, as can be seen in Figure 3.1. The input file is then passed through the GNU Compiler Collection (GCC) for *C* preprocessing. GCC is invoked on *target.h* with the following options to fully pre-process the input file: *-E*, *-fpreprocessed*, and *-dD*. Option *-E* instructs the compiler to stop after the preprocessing stage. Combining *-E* and *-fpreprocessed* ensures that full preprocessing occurs correctly [29]. Lastly, *-dD* ignores predefined macros and outputs the *#define* statements in addition to the regularly produced preprocessing output.

Other common Linux tools like *sed* and *tr* are used to sanitise the output files for further processing. These steps include replacing \t (tab characters) and remaining # characters.

Ctags, also a Linux command line tool, is used to generate tags for source code. In our use case we are producing tags for the preprocessed *target.h*. This file is denoted as *preprocessed_target.h* in Figure 3.1. Options for ctags are *–c-types=+p* which ensure that functions and prototype definitions are included, while *-x* instructs ctags to produce a human readable output file [3]. Ctags' output is then written to the system. This output file functions as input for the actual ALRPC tool which generates the *C* code (denoted ALRPC in Figure 3.1). An in-depth description of the functionality, design and actual activities of the tool which is represented by the "ALRPC" box in Figure 3.1 is provided in Section 3.1.2.

Note that the file *local_impl.h* is replacing the *target.h* in the figure (and in the actual file system). The newly generated *local_impl.h* contains the method stubs which are redirecting to *serialize.h* in order to marshal and send the messages to the server. Four new files are automatically generated directly by the ALPRC tool. Generation of these files is possible because of the analysis method described above. The new files are two pairs, each containing a **.c* source code file and a **.h* header file. The header *serialize.h* contains the *C* prototype declarations for each marshalling function. *Serialize.c* then contains the actual implementations of each function. This file is generated automatically and each marshalling function is customized to deal specifically with the number and type of parameters that the original function contains.

The files *local_impl.c/h* contain the prototypes and implementations for the func-

tions which are destined to be executed on a remote server. Subsequently, if one were to turn the function described in Figure 3.3 into a remote procedure the following steps during execution would occur.

The caller invokes the procedure just as if it was a local call to the *crypt* function. In fact the caller is not aware, syntactically nor semantically that the call to *crypt* is anything other than a local call. Through including *local_impl.h* the originally included *target.h* is replaced. Now all calls to remote procedures are redirected to the automatically generated implementations which handle marshalling and networking. Within *local_impl.c* the function *crypt* (as seen in Figure 3.3) parameters are simply forwarded to the marshalling function along with additional information to match the marshalling function's signature (Figure 3.4). Depending on the static analysis and the source code, this signature can vary slightly. If there is no reason to include options, which is determined at the static analysis described above then neither the marshalling function nor the calling function will require this additional parameter. These steps closely adhere to the general RPC model developed in the early stages of RPC inception in the 1980s. Thus, ALRPC can claim to be truly in the realm of RPC tools.

The actual marshalling of the parameters follows the same basic algorithm for all cases. All transmitted information is stored and sent in a message buffer. This buffer contains data and meta data. The structure of the buffer definition is illustrated in Figure 3.2.

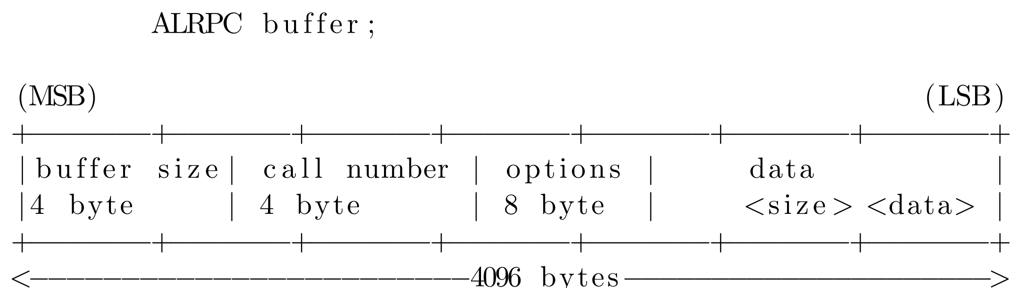

Figure 3.2: Custom Buffer of ALRPC.

The first item in the buffer indicates the size of the buffer. Because the minimum size for a page in Linux is usually 4k, the buffer size is chosen to be 4k as well. The second

item in the buffer is a unique call number which is assigned to each function in order to distinguish it from other remote procedure calls. This is a data element which is of known size. The next two locations are available for additional options and information. In the current implementation of ALRPC these two fields are not used. Before the data segment begins, the buffer contains a field to specify how many parameters this buffer holds. The next segment of the buffer is the actual data section. For each parameter first the size of the parameter is given. Then the actual parameter is placed in the buffer. The receiving unmarshalling function extracts these parameters from left to right. Each remote procedure is uniquely identified by its call number and as such a list of its parameters is known at the server side. The unmarshalling function assigns the data of each parameter to its own list of parameters in a predetermined the order (left to right both in buffer and function signature). Once the data has been extracted from the buffer at the server, the actual function is called with another local call. The result of this call on the server is then placed at the beginning of the buffer in the format of [size of data][data]. Once the return value has been placed in the buffer it is communicated back to the client via the network connection.

```
char *crypt(const char * key, const char * salt)
```

Figure 3.3: Original crypt function in unistd.h.

```
char *serialize_crypt(int num_of_args, const char * _key,\
        int size_arg1, const char * _salt, int size_arg2,\
        int call_num, int options)
```

Figure 3.4: Prototype signature for crypt function in automatically generated marshalling code serialize.h/c.

3.1.2 Modular Building Blocks of ALRPC

ALRPC is written in Python, but also contains some shell scripts. This section describes the modularity of the ALRPC tool and in particular it's Python component. ALRPC's Python component is organized in a modular fashion. There exist currently seven separate Python files.

Table 3.1: Overview of ALRPC's Python module.

File Name	**LOC**	**Purpose**
ctags_parser	205	Basic Building Blocks
defs	10	Contains common definitions and constants
helpers	49	Common functions (e.g. I/O)
deserialize	137	Generates Code for deserialization at server
local_impl	144	Generates local method stubs
serialize	121	Generates code to serialize at client
write_ir_xml	43	Generates xml intermediate representation

Table 3.1 lists the currently seven Python files which compose the Python module of ALRPC. Also shown here is the source code line count for each submodule. The task of ALRPC's Python module is to parse *C* source files and generate code and stubs for the target application. Client and Sever code is also produced which handles marshalling and unmarshalling.

The file *ctags_parser* contains code to control the program flow. It directs the order of lexing, parsing and code generation. Further, *C* files are equipped with common headers via write operations directed from this file.

The file *defs* simply contains constants which are important to regulate the interplay of the Python files of ALRPC itself. It contains definitions for parameter lists, output paths and some naming conventions for generated files.

Helpers is a central location for I/O functions which are used by all other files. Common operations such as debugging, opening and closing of files are grouped together in this one Python file. Subsequently, all other files in this module import *helpers* in order to make use of its functionality.

The file *serialize* generates code required on the client side to allow marshalling of parameters.

Figure 3.5 is a segment in *serialize*. This code segment automatically writes the generic code required to serialize the function's message size and call number into a buffer on the client side. Most of the automatically generated code used to fill the buffer is nearly generic, only differing in the names and sometimes size or type of

```
        # msg_size
        ser_code += '\tmemcpy(&buffer[nbytes], &my_s_size, \
                sizeof(my_s_size));\n'
        ser_code += '\tnbytes += sizeof(my_s_size);\n'
        # call_num
        ser_code += '\tmemcpy(&buffer[nbytes], &call_num, \
                sizeof(call_num));\n'
        ser_code += '\tnbytes += sizeof(call_num);\n'
```

Figure 3.5: Code snippet of serialize.py: This generates standard Code to allow marshalling of parameters on the client side.

parameters.

Deserialize is the counterpart of *serialize* at the server side. Figure 3.6 showcases a small code snippet which is used to populate the servers return message. The return message is sent back to the client in a buffer of specified size. This message only contains the return value since the client is already aware of its length and type. The identifier *func* is appended with the *C* code in string form because at the end of the generation phase, this string is then written into the server's *C* file. *Deserialize*, just like *serialize* build up strings which are then written in one I/O operation to the file system.

```
func += "\n\treply->msg_size = strlen(ret_val)" + \
                                "+ 1;\n\treply->num_of_args = 1;\n"
func += "\tmemcpy(&reply->data[0], ret_val, reply->msg_size + 1);\n\n"
```

Figure 3.6: Code snippet of deserialize.py: This generates standard Code to allow unmarshalling of parameters of the type char* (assumed to be String).

```
switch_stmt += "\t\tcase " + \
                CALL_NUM_DICT[item[SIGNATURE].split("(")[0]] + ":\n"
switch_stmt += "\t\t\t return (void *) " + func_name +"(msg);\n"
```

Figure 3.7: Code snippet of deserialize.py: This generates standard Code and switch statements to direct the function to the correct implementation at the server side.

Figure 3.7 is a segment of code which generates the dispatching of functions at the server side. Here a switch statement is built up which uses a function's call number (passed in via the buffer message) to then call the appropriate function at the server

side. The entire buffer is passed to the correct function. There the actual customized unmarshalling takes place before the call is carried out.

The file *local_impl* generates the stubs which are used to redirect the local calls. Calls from the existing monolithic application are redirected to the client code via the use of these stubs.

Lastly, the file *write_ir_xml* is intended to produce a human or application readable representation of the mechanisms steps and processes. Via this xml file, the intention is to allow the programmer to modify the programmatically inferred steps. Then the xml can be used as alternate input to the ALRPC tool. Currently, this module is integrated in the ALRPC tool chain, but is not used due to incomplete features. However, the concept and mechanism is determined to be viable since information read in can be represented in xml format. Similarly, the conversion of xml information into data that is useful for ALRPC is conceptually doable. Since this step was considered trivial in concept, but challenging in time and implementation it was delayed. It is also not a crucial aspect to demonstrate the viability of the ALRPC mechanism.

3.1.3 History of ALRPC

ALRPC began as a simple mechanism to extract prototype signatures from *C* source code. The currently existing Python modules were present in a single file in much simpler forms. The development process was incremental. At first only simple prototypes were recognized. Once recognition of desired prototypes became a stable feature the code base was expanded to allow the parsing of these prototypes into useful data structures. Protoype names, return types, parameter types and names were all parsed and stored. The next step then was to use this information to automatically generate code which would redirect the original function calls to our own implementations, which later would become the client side.

A first milestone was automatically analysing and generating code to allow a complete round trip pass from original code at the client side to the server. For this, the code base was required to expand on the original success of redirecting the original calls by also automatically generating compilable *C* code which would serialize all function parameters.

Challenges for this part were the correct identification of parameter types and allocation of memory inside the message buffer. Like many of the software engineering challenges of ALRPC the correct solution to this problem was found by first identifying common patterns and then providing code solutions which easily switch between patterns to generate the appropriate *C* code.

Once this task was accomplished and messages were successfully passed to a server located in a different process space we began work to automatically deserialize the message at the server. This presented several challenges again because here an additional level of redirection was present. Before function calls could be made, the server first needed to determine which function call it was in order to ensure the correct unmarshalling of parameters. Due to the standardised buffer structure for ALRPC this resulted in a single *C* operation to offset memory access into a buffer. Once the program determines the correct server function at run time, it passes the buffer to an in-between-function to begin with the task of unmarshalling the buffer. The buffer and its contents need to be unmarshalled before making a call to the actual function itself is possible.

In order to guarantee a complete pass from application via client to server and back, both client and server needed the dual capability of marshalling and unmarshalling the message buffer correctly. The server side needs of course the ability to unmarshall the buffer. But it also needs the ability to serialize the return value and any additional information if so required. Similarly, the client needs to marshal the parameters before they are sent to the server. But further it needs the ability to unmarshall the return message correctly. A complete first pass going full circle on a primitive data type was the first milestone.

The next task involved code fixes to allow for seamless generalizability of all existing components. The first milestone focussed on a single primitive type. Now the code was modified to succeed with all primitive *C* types (i.e. int, char, unsigned int, Bool, etc.).

This proved labour intensive, though most time was devoted to design and testing at this stage. Eventually, milestone two was reached and ALRPC could automatically

convert monolithic applications into a distributed application with the restriction of only converting prototypes of primitive data into RPCs.

Still keeping ALRPC's Python modules, the actual work horse of the ALRPC tool, in a monolithic file, we proceeded to extend its functionality to more complicated data structures. A common data structure was the String, or char* in *C*. For simplicity and because a safe method of determining this type accurately does not exist, we assert that any *char ** encountered is always a String.

Strings pose a challenge which differentiates them from primitive types. A string's length can be arbitrarily long. As such ALRPC needs to be able to dynamically accommodate writing different sizes of data to the buffer. For Strings we make use of the *strlen* function to determine the String's length. This however, relies on the assumption that a *char ** is indeed a string.

A challenge similar to that of strings is pointers of any type. Passing pointers to the server is not useful for the successful execution of a program. Instead when pointers are encountered the data itself must be passed. ALRPC is able to handle at most two levels of pointer redirections. Because of the semantics of C, anytime the star (*) operator is encountered, ALRPC asserts that it is dealing with a pointer. *Char ** builds the exception to this case, where any *char ** type is asserted to be a String, rather than a pointer to a simple *char*.

Already it is becoming evident that the decisions made and the language semantics of C, combined with the desire to provide maximum automation, created certain limitations for the final ALRPC tool.

After these aforementioned milestones were completed and pointers were successfully handled we began a rewrite and modularization of the Python modules of ALRPC. This was done in order to adhere to software engineering principles. Other reasons for the refactoring was to increase maintainability, extensibility and allow for easier identification of bugs. Subsequently, the Python modules of ALRPC are those introduced in Table 3.1.

3.2 Lexing and Parsing the Input

This section talks about the steps needed to parse C input files as well as the output from other program comprehension tools like ctags.

First we are operating under the restriction of only being given the prototype signature. This restriction exists because a full semantic and syntactic analysis of source code is not feasible. Subsequently, this restriction introduces certain limitations in terms of information available and the assumptions we can make about each procedure that is scanned.

We are using output from *ctags*, a Linux tool which can be run form the console. We use ctags with the *–c-types=+p* option to extract *C* language prototype declarations from the source code. Additionally, we use ctags' *-x* option to produce human readable output [3].

Figure 3.9 shows one line in the ctags output file that pertains to the previously mentioned *crypt* function. This output contains all the information available from the prototype declaration. First, the name of the function is given, in this case *crypt*. Because ctags also identifies structures and structure members, this line also informs the user that it is in fact a prototype. The next pieces of information, *30* and *../output/tmp2.h* are line numbers and file identifiers in which the prototype is found. The file name here is due to the fact that the shell script which controls the tool chains moves and renames various pieces of information to the file *../output/tmp2.h* at some point before the ctags analysis occurs. Automatic renaming during the tool's run is done to avoid loss of information and prevent naming conflicts among the files. The last segment of the line is the most interesting part for this analysis. Here the return type is identified along with the parameters of the function. These pieces of information are extracted in the ALRPC function called *cp_parse_ctags* (Figure 3.8). Extraction occurs from the ctags output file to a Python data structure in ALRPC. The data structure is a list of lists.

Once a simple transformation from the ctags text output file has occurred to a Python data structure representing a list of lists, this abstract data type is passed on to a different function in ALRPC which processes each prototype information in order

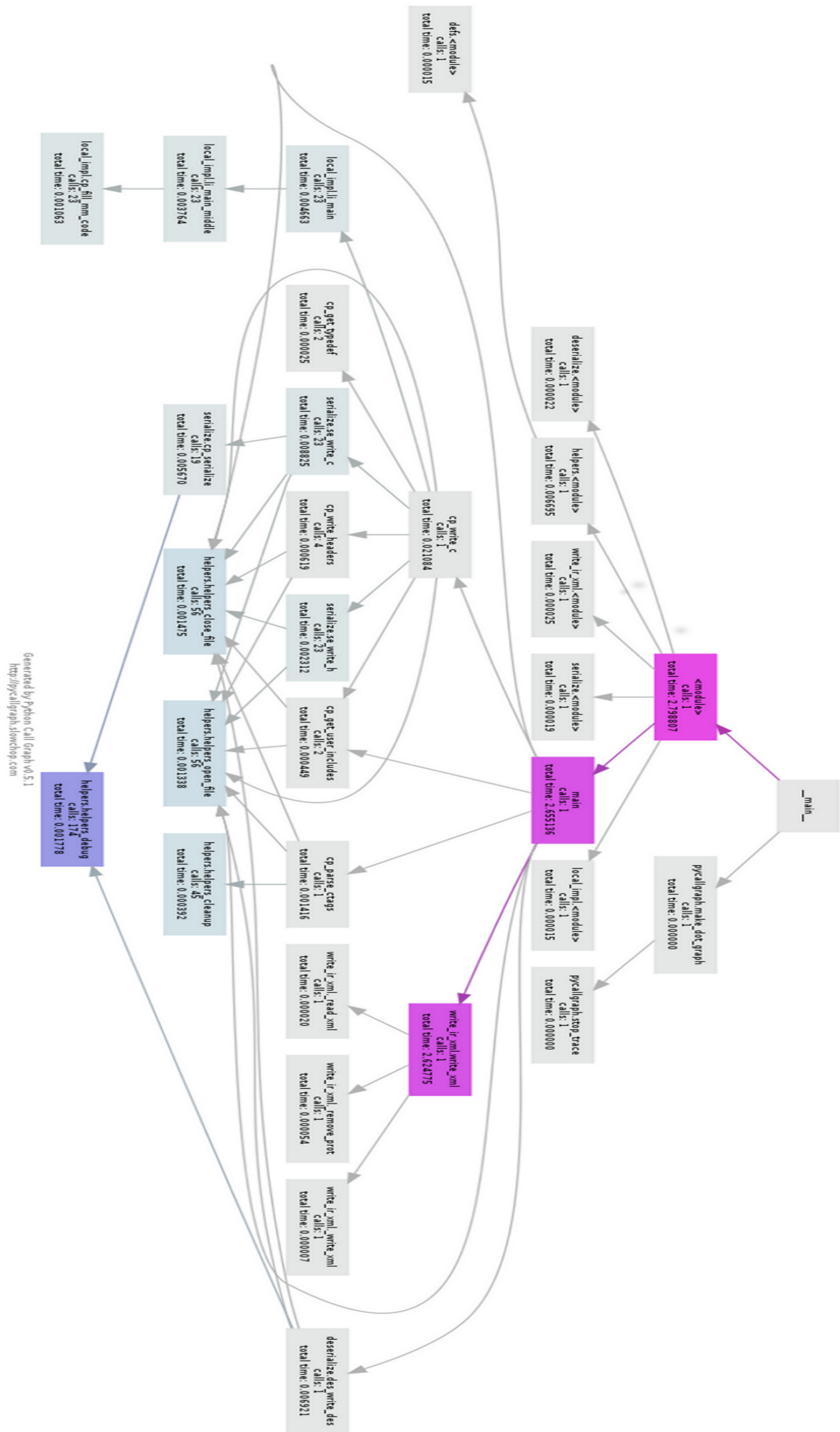

Figure 3.8: Call graph of ALRPC tool.

to automatically generate *C* code (see Section 3.3). Figure 3.9 contains the *C* type string. The same information format is used by ctags for all other data types. Likewise, ALPRC deals with the information in the same way. All prototype information from ctags output is parsed into a list of lists Python abstract data type within the ALRPC tool in the *cp_parse_ctags* function in the *ctags_parser.py* file (Figure 3.8).

```
crypt prototype 30 ../output/tmp2.h extern char \
        \*crypt (const char *__key, const char __salt)
```

Figure 3.9: Excerpt of ctags output file containing the line describing the *crypt* function.

```
for line in f_ctags:
        tmp = line.split(None, 4)
        tmp = helpers.helpers_cleanup(tmp)
        lol.append(tmp)
```

Figure 3.10: Code excerpt of the function *cp_parse_ctags* in Figure 3.8.

The code segment for parsing text input into a data structure in Python is relatively simple. Figure 3.10 represents the loop which parses the input ctags input file line by line and places the result in a data structure which is a list of lists.

3.3 Code Generation

3.3.1 Active Python Modules of ALRPC

This section seeks to answer the questions of which component in ALRPC is active in the code generation; what information are available from the previous component or stage of ALRPC?

Of the Python modules of ALRPC there are four components active: *a*) ctags_parser; *b*) local_impl; *c*) deserialize; and *d*) serialize . The additional files listed in Table 3.1 are active to the extent of fulfilling auxiliary roles in the program execution.

3.3.1.1 Control and Common Headers: ctags_parser

The main task of this module is to control the information flow of the code generation. For this, the *ctags* output needs to be consumed and code must be generated. Headers and type definitions from the original file are often required to exist in the modified local code, client code and the server code. Therefore a central tasks is to write common headers and type definitions from original files into the generated files. Additionally this module also replaces specific headers in the original file in order to intercept function calls from the original code.

Before all this can occur, this module parses *ctags*' output and stores it into a data structure. Now any information relevant to ALRPC is stored entirely within the Python program and can be accessed and manipulated.

This Python module of ALRPC is also tasked with controlling the program flow of the other modules. Important steps such as writing out xml or initiating code generation for client and server occur here. After the parsing of the *ctags* file is complete, creating a modified local implementation and serialization (client) is the next task. Only after this task complete is the deserialization code generated. While the order could theoretically be arbitrary, ALRPC propagates information obtained (or inferred) at earlier stages of generating serialization code to the segments which are tasked to generate the deserialization code. This process saves compute time since information does not need to be recomputed. It is made possible because serialization and deserialization are closely related and require very similar information. The kind of information which is passed forward from serialization to deserialization includes type and name information for parameters of given functions.

Any write operation to propagate generated code onto the file system is kept to a minimum. Writes and files system accesses are delayed as long as possible. Therefore, when write operations occur, a large string, representing the generated C code is written out at once. 205 lines of Python code achieve this inside the ALRPC's Python module.

```
tmp += "serialize_" + autogen_info[0].split()[-1]+"("
```

Figure 3.11: Code snippet which creates local serialization function: This allows marshalling of parameters on the client side.

3.3.1.2 Generating local files: local_impl

Only 144 lines of Python code generate the functions necessary to connect the client and server of the final distributed application product. Essentially two code transformations are performed here. First function calls are made which handle the marshalling of parameters. Second, the parameter list must be appropriately transformed to ensure that pointer arguments are passed correctly via their values.

The first aspect of this is a simple call generation. The new call which will marshal the arguments is identified by prepending it with the identifier "serialize_". Figure 3.11 illustrates this simple one line process. As a visual marker, we can clearly identify the opening bracket at the end of the code in the figure. There is a special case for this step as well. If the return value includes a pointer then the pointer is removed since it is irrelevant for the serialization of its parameters. The second aspect of this module is to correctly pass the arguments to the actual serialization function. Pointers are the main challenge. Whenever a pointer is identified through simple string analysis then the variable's value is passed to the marshalling function instead of the pointer. This ensures that the actual value pointed at is placed in the message buffer rather than the pointer's value which would hold no meaning at the server side. While this essentially alters the program's implementation, these changes are contained beyond the level of function calls which the original code is interacting with. Original function calls are still allowed to remain unaltered. This is important to keep the required manual code changes to a minimum.

After finishing execution of this module, a complete function signature in string format, which can then be written out to the file system, is returned.

3.3.1.3 Serialization and Deserialization

Both these modules are closely related. The only purpose of marshalling function parameters is to correctly place them in a predefined buffer structure which can then be sent across the network to a recipient. Marshalling is done in the serialization

module. Because information flows from client to server and vice versa, both modules contain serialization as well and deserialization capabilities.

The serialization module generates code which effects marshalling of parameters. To do so it is given the parsed and stripped information from the *ctags* output from the previous stage. We have seen in Figures 3.6 and 3.7 the marshalling and unmarshalling code in the deserialization module. The equivalent code segments in the serialization module are nearly identical. The automatically generated code must ensure that the message buffer is filled in accordance to the specifications described in Section 3.1.1 and illustrated schematically in Figure 3.2.

The main challenge here is to ensure that the automatically generated code can correctly handle the storage of parameter values into the buffer. This is challenging when the original parameter is a type of pointer. The module is able to dynamically detect this case via a simple string analysis of the original information. Once such a case is detected, the automatically generated code reflects this information and produces code which instead of simply copying the parameter itself into the buffer, extracts the value which the pointer referenced. Then this value is extracted directly and placed in the buffer. Special care is then taken to ensure that the server side and the actual implementation of the original function call are correctly executed.

3.3.2 Output Files

As a result of the automatic generation of *C* code, several output files are generated. These files are required to convert a monolithic application into a distributed application using RPC.

Code which actually handles the serialization is placed in two code and header files named *tagstmp_mm.c* and *tagstmp_mm.h*. The result of ALRPC's Python module for serialization is written out to this file. Subsequently it contains the instructions to serialize the desired function calls, passes the information to the client, receives the return value from the client and then extracts the return value from the return message. When looking at the actual content of the file one notices that all function names are prepended with the string *serialize* as one would expect. Further the re-

turn value from the servers message is extracted via a *memcopy* system call.

```
char *serialize_crypt(int num_of_args, int arg1_size, const char *__key,\
 int arg2_size, const char *__salt, int call_num){
        memset(&buffer[0], '\0', BUF_SIZE);
        int my_s_size;
        my_s_size = BUF_SIZE;
        int nbytes;
        nbytes = 0;
        memcpy(&buffer[nbytes], &my_s_size, sizeof(my_s_size));
        nbytes += sizeof(my_s_size);
        memcpy(&buffer[nbytes], &call_num, sizeof(call_num));
        nbytes += sizeof(call_num);
        nbytes += 4;
        nbytes += 4;
        memcpy(&buffer[nbytes], &num_of_args, sizeof(num_of_args));
        nbytes += sizeof(num_of_args);
        memcpy(&buffer[nbytes], &arg1_size, sizeof(arg1_size));
        nbytes += sizeof(arg1_size);
        strcpy(&buffer[nbytes], __key);
        nbytes += sizeof(__key);
        memcpy(&buffer[nbytes], &arg2_size, sizeof(arg2_size));
        nbytes += sizeof(arg2_size);
        strcpy(&buffer[nbytes], __salt);
        nbytes += sizeof(__salt);

        int rc = cli_connect_buffer(buffer);

        int ret_s;
        char * ret_v;
        memcpy(&ret_s, &buffer[0], sizeof(int));
        ret_v = (char *) &buffer[20];
return ret_v;
}
```

Figure 3.12: Code snippet in tagstmp_mm.c: Serialization, client server connection and extraction of return value for function char * crypt is handled here.

Less spectacular is *tmp.h*. This file simply contains all the prototype declarations which are now to be used as remote calls by the original application.

The second last file which is generated is the *deserializer.c* and *deserializer.h* pair. These files contain the server side code. Once the message buffer is transfered to the server, the server connection passes the message buffer to *deserializer*. Then the call number is extracted from the buffer. Based on the call number a simple switch

statement calls the appropriate deserialization function. Within this deserialization, a call is made to the actual implementation of the function on the server side. A return value from this particular call is captured and stored in a predefined location of the buffer. A reference to this buffer is then returned via the servers response message.

Lastly, the *local_impl.c* and *local_impl.h* remain. The header file contains all the prototype definitions which are selected for conversion into remote calls. This header file is then included in the original application. The *C* code file contains a simple redirect to the serialization code. Figure 3.13 illustrates the function body for the original *crypt* call, while Figure 3.12 illustrates the client side of the same function.

```
serialize_crypt(2, strlen(__key), __key, strlen(__salt), __salt, 15);
```

Figure 3.13: Function call and signature of the redirection call.

Notably the function identifier contains the prepended string *serialize*. Another particular parameter which is not part of the original call is the first parameter. In Figure 3.13 the first parameter of the function is the number 2. This is automatically generated and signifies that the number of parameters of the original function is exactly 2. The other novel parameter is the last one. Here we see the number 15. This number signifies the call number of the function within the ALRPC system and is actually placed in the message buffer during serialization. During code generation the Python modules of ALRPC keep track of the number of functions which have been dealt with. In this use case it turns out that the *crypt* function was the 15th which was handled in a single run. At the server side this call number parameter is then extracted and used to select the correct switch statement for deserialization and execution at the server.

3.3.3 Compilation and Integration of Generated Files

The newly created files must now be integrated with the distributed system. For this the ALRPC mechanism makes use of the standard *gcc* compiler. However, this step is fairly straight forward if the automated code generation did not produce mistakes. Common errors early in the development phase were type mismatches between caller and callee of functions.

All files are linked and compiled with the original code, except those relevant for the server. The deserialization files, *C* code and header files, are compiled with the server. For this task, simple *make* files are used to coordinate the compilation of both server and client side code.

After this step, the only connection between the client side and the server is a single interface message passing connection. Only the message buffer can pass between the client and server; no other connection is established to ensure the execution of the remote functions.

3.4 Pointer Chains and other Challenges with C

Because of the semantic nature of the *C* language, there are several issues which are challenging. Some of these challenges were overcome by ALRPC, while others remain.

Pointers are a huge challenge in *C*. This is emphasized by the original goal of ALRPC to maintain maximum possible automation when converting monolithic applications into a distributed program using RPC. Why are pointers so challenging?

First, there is no unique way of identifying a pointer in *C*. While the existence of the star (*) operator can be a hint, there is no requirement for such a convention. Conversely, it is possible in *C* to have the star operator but have it not signify a pointer at all.

In short, *C* language semantics makes it impossible to automatically identify all cases of pointers correctly. We address this by asserting that the existence of the star operator is a requirement for the identifier to be a pointer. Currently ALRPC can successfully deal with two levels of pointer indirection.

Pointers of primitive types are challenging, but manageable. Once the primitive type is identified, we are able to know its size. The next step is to locate the star operator, if it exists in this case for any particular identifier. Once a star operator has been found ALRPC treats it differently for serialization. In order to pass the correct information to the server, ALRPC now follows the pointer to the referenced value and deals with it directly. Subsequently, the referenced value and not the pointer is

placed in the message buffer.

Strings are handled in a similar manner. Any *char ** is given only one meaning, that of a pointer to the beginning of a string. *C* of course allows the *char ** to be more than a sting pointer, but for simplicity and determinism we assert that the *char ** is a pointer to the beginning of a string.

Once a string is identified, copying it into the buffer requires the same pieces of information as copying primitive types. We require the location and size of the data we wish to copy. The location is given by the initial pointer value. The length is identified via the system call *strlen*.

Using our approach clearly ensures that we are able to distinguish between reference and value parameters. On the other hand it severely limits the *C* code which ALRPC is able to successfully parse and analyze. We accept this limitation in order to provide automation for the cases with which ALRPC is able to successfully deal with.

While we are able to handle several challenging pointer operations in this proof of concept tool and mechanism, several unresolved challenges remain. Pointer chains are impossible to deal with successfully during automated RPC generation. This is because *C* semantics lack the information required to accurately deal with this case. While solutions have been previously identified [53]. Two solutions for pointer chains are evident. Firstly one can recursively follow the entire pointer chains and copy every single element into the buffer. This, however, is space and time intensive. The space intensive part is also contrary to the idea of RPC, which emphasizes small data elements to be passed [56]. The other alternative is to only copy elements as they are needed. This then requires secondary communication between the server and client to request the needed information in an ad hoc fashion. This options was not chosen either because it is also contrary to the principles of RPC [53]. Additionally, the analysis of prototype signatures and the resulting semantics make it extremely challenging to identify complex data structures which would require such an approach. We will see later that the simple cases, for which this proof of concept was developed, dominate and thus the decision to not focus on complex cases was correct.

Lastly, global variables are not dealt with at all. Again, *ctags* produced prototype

signature information which contains absolutely no information about global identifiers used inside these functions. A function signature does not contain information about local or global variables in *C*, only parameters and return type are identified in the signature. Without extending the original task and problem space, any remote procedure call will fail if it is requiring a global variable. However, this failure will ideally occur at compilation time and can then be corrected by not selecting the affected function into a remote call.

3.5 ALRPC - Status Quo

In this section we have introduced ALRPC. ALRPC is a tool and a mechanism. The tool itself makes use of a variety of existing Linux tools. Static analysis of source code is performed via ALRPC while using and coordinating existing Linux tools. Shell scripts, *ctags*, *gcc* and the Python programming language interact to make ALRPC work.

The goal of automation was maintained by making several assumptions when obtaining information from parsing *C* source code. These assumptions and assertions with the given information are necessary in order to maintain high levels of the tool's automation despite the ambiguous semantics of *C*. While these assumptions enabled us to move forward with automatically converting *C* applications into distributed RPC applications, they also introduce several limitations (which will be described in a later section).

The largest aspect of ALRPC is its Python module. These modules gather information from other Linux tools' output. Additionally, ALRPC's Python modules are also tasked with automatically generating *C* code for all aspects of a functioning RPC system: local function stubs; client code; and server code. Existing Linux tools are used to compile the generated code with the existing source code of the original application. The other aspect in which existing Linux tools are used is the parsing and organization of source code. We use *ctags* to parse *C* source code and extract prototype signatures for analysis.

With ALRPC we are able to automatically convert a monolithic *C* application into a distributed application where parts of it are executed in different process spaces.

We are able to extract specific functions from the original application. Calls to these extracted functions are then redirected to the client modules and passed to the server. At the server the actual implementation occurs. Once the return message is received back at the caller, execution continues.

The final system of ALRPC does adhere to the general RPC principles. The programmer and compiler do not have to be aware of the fact that the actual implementation of the function call is in a different process space. Only small amounts of data are transmitted. Additionally, automation is kept at a maximum in order to ensure competitiveness with existing technologies.

Chapter 4

Experiments

This chapter describes the experiments which we carried out to evaluate ALRPC. It includes the experiments which were carried out to compare ALRPC with RPCGen, a state of the art existing RPC tool and presents our resulting micro benchmarks. As such this section introduces further ALRPC micro benchmarks, results obtained from using ALRPC with real third party applications, and an investigation into large systems which might potentially benefit from our proposed mechanism ALRPC.

We initially use prototype declarations with signatures representative of some of the capabilities of ALRPC. These functions are then used to test ALRPC and RPCGen in a comparative evaluation using micro benchmarks. This is showcased in Sections 4.1 and 4.2. Then Section 4.3 will introduce micro benchmarks highlighting a performance argument for ALRPC and RPC mechanisms in general. Following this, Section 4.4 will describe our experiments using ALRPC to evaluate its suitability in terms of distributed systems' performance, suitability of individual functions and by doing so apply ALRPC to real world applications. We use ALRPC to split off functions from a third party application to illustrate that ALRPC can indeed be used outside the lab environment. We then provide further indications and micro benchmarks to suggest that ALRPC could be useful in improving legacy *C* applications in a distributed system provided that suitable conditions exist.

The outline of this chapter is as follows: Introduction of the experiments will proceed in two phases. First the RPCGen and ALRPC experiments are described and their results shown. Then performance micobenchmarks for ALRPC are introduced; following this ALRPC is shown in connection with third party applications. An

evaluation of the experimental data is carried out in Chapter 5.

4.0.1 ALRPC — RPCGen initial benchmarks layout

For the RPC experiments involving ALRPC and RPCGen we will present data according to the following categories.

- Listing of functions and their signatures
- Correctness
- Number of additional lines produced manually
- Performance of generating an RPC system
- Performance of the RPC system

First, each section will introduce the functions which are used to conduct the experiments. This includes a description of the functions' signature to demonstrate that it is representative of a function which can be found in real systems. Then the RPC system is evaluated in terms of correctness. The RPC system passes the test if and only if the return value of a function is identical to that of the non-RPC system. The third evaluation criteria is the additional work required to produce an RPC system from a monolithic application. This measurement takes the form of determining the number of lines which have been produced manually. Lastly, the performance evaluation focusses on two aspects. One of these is the time required for each RPC tool to convert a single function call into RPC. Then the performance of the RPC system is evaluated. This last measurement is intended to identify RPC performance compared to non-RPC performance.

Each section also includes an overview of the state of the RPC tool if valuable insights can be gained from doing so. This will include any information regarding unexpected or observed behaviour while using the tools.

To establish a baseline for performance measurements and many of the other metrics, we built a comparable test system as a monolithic application. This application is then turned into a distributed system via the respective tools. Table 4.1 shows some of the function signatures which were used to measure the call completion times for

a non-RPC system. We focus on these functions to illustrate basic features and characteristics of the type of functions that ALRPC can handle. The table is extensible, both in principle and in practice for the other types of function signatures which are within the scope of ALRPC, however we focus on these to allow for more detailed analysis. Complex pointer chains and de-referencing operations are not within the scope of ALRPC and are thus not included in the experiments since they are beyond the scope of this work (see Sections 3.4 and 3.5). Here all functions are located in a monolithic application. Each measurement for each function is listed. To measure the time it takes to complete a function call we use the *gettimeofday* system call which is defined in *sys/time.h* on Linux systems. Further each measurement takes into account the completion of the *assert* system call to verify that the measured call returns the expected value. A measurement result of zero (0) indicates that the call completed faster than the measurement tolerances account for. This means these calls required less than a microsecond to complete which are 6 decimal points of a second. To elimiate unwanted compiler optimizations such as inlining functions, the code for the monolithic application was compiled with *-O0* compiler option and each of the functions was appended with the *gcc* specific attribute *__attribute__((noinline))*. See Appendices A.1 for a detailed listing of the source code and the resulting assembly code. Appendix A.7 also validates in experiments with 100000 and 1000000 trials that the monolithic application's performance for micro benchmarks is exceptional and consistent with the measurements in Table 4.1.

Table 4.1: Time Measurements for non-RPC functions (time measurement in microseconds). Note that several calls completed faster than the measurement precision.

Function Signature	int foo(int x)	int foo_add(int x,int y)	char * one_line(char* string)
Measurement 0	-	-	7
Measurement 1	-	-	5
Measurement 2	-	-	8
Measurement 3	-	-	5
Measurement 4	-	-	12

A detailed description of these functions follows in Section 4.1 and Section 4.2. When evaluating these measurements in Chapter 5 one must keep in mind that these tests did not run on an isolated single-user system. Thus performance variation can be impacted by the state of the system at the time the measurement was taken. However, this impacts all measurements and given a sufficient sample space the effects

should be negated. Additionally each measurement was taken in series, however the order of the functions within each measurement was switched each time to account for certain startup differences that might have impacted the application. Appendix A.7 illustrates results of an exhaustive experiment for these micro benchmarks.

Correctness of the monolithic system is verified by inspection of output values. These values are compared and confirmed with expected values. The monolithic application's values and measurements are taken to be the baseline for the evaluation of the RPC systems. This metric is chosen because the final RPC system, regardless of which tool was used to produce it, should not behave differently than the original system. Output and return values are used to establish correctness of a system.

The metric of number of additional lines of code and the performance in generating an RPC system do not apply to the monolithic application. This metric is intended to measure how much additional work a programmer is required to perform when turning an existing system into a RPC system. Therefore this metric only applies to the RPC systems which are compared.

The performance of the system is established as the baseline with which all RPC systems in this work are compared. Results are measured and recorded in Table 4.1 for the micro benchmarks regarding function calls.

4.1 RPCGen experiments

In this section we will describe the experiments and the data that was collected with our RPCGen system. The experiments focus on performance for individual functions. Further we collect data and establish metrics in terms of automation of processes to generate an RPC system; manual lines of code written in the generation of distributed systems; and we explore the complexity of producing an RPC system with RPCGen.

One problem with RPCGen, as with a lot of other tools like it, is that the correctness of the samples and tutorials is fairly low. To find a tutorial that, in complexity goes beyond the pure basics and is without error, is not without challenge.

4.1.1 The tested functions

In order to evaluate RPCGen with ALRPC and the baseline metrics (see Table 4.1) we need to use the same functions in all our systems. The tested functions which we focus on in this thesis are representative of the range of functions which ALRPC can handle correctly. The first function is *int foo (int x)*. Its purpose is simple: Return the parameter unchanged to the caller. This function posed no problems and did not require any major modifications at first. The automatically produced RPC files forced some slight syntactic changes. These changes are necessary because RPCGen in default mode required rewrites in the function signature at the client side. Instead of the parameters being accepted in the original form, RPCGen's default mode requires the programmer to accept pointers to the parameters at the server side. Altering the code on the server side in the actual implementation of the function was a relatively simple change and was completed by modifying a few lines of code in one file of the implementation on the server side. The modifications were to dereference the parameters and store them in local variables which then allowed us to keep the actual implementation of the function body untouched.

Yet despite these easy to overcome challenges the default mode of RPCGen was quickly abandoned. RPCGen allows only one parameter per function in its default mode. While this is certainly not a "show-stopper", it would require complex workarounds had we decided to continue with the default mode. In order to get around this restriction for the function *int foo_add(int x, int y)* we would have to introduce additional shared data structures and greatly alter the implementation at the server side as well as the calling semantics on the client side. Such changes were unacceptable for a function that takes two integer parameters and returns the result of the arithmetic addition of the two. The required changes were determined to outweigh the importance of the function in complexity. Instead we opted to use RPCGen's *N* flag to address this issue. This option invokes the "new" RPCGen functionality. Using the *N* flag allows the programmer, among other things, to have more than just one parameter per function. But it also required a rewrite of the previously implemented functions. The *N* flag is incompatible with the default operating mode of RPCGen in terms of the server side programming. The *N* flag produces function signatures on the server side not with pointers but rather allows parameters to be passed by value. Thus a rewrite of the previously implemented single parameter func-

tions was required since they expected parameters by reference.

The function *char * one_line(char * string)* also required a rewrite on both server and client side because its signature was altered from the original syntax. RPCGen automatically changes the signature to return a *char ***. While these changes were relatively trivial to implement, they do require knowledge of pointers in *C* and a knowledge of RPCGen's output.

Aside from the aforementioned changes to the code and system, which are important to understand what system the actual measurements were obtained from, there are additional changes to the system with RPCGen. RPCGen modified all function signatures to include a parameter which specified the remote host. This remote host definition was obtained from *rpc/rpc.h*. As a result of this, all functions on the caller and callee side needed to be modified to address this requirement. Additional changes to the name of functions were also required to specify the correct version of each function as it is called on the remote host.

All measurements in Table 4.2 were obtained from a system built with the *N* flag of RPCGen. The relevant source code can be found in Appendix A.2.

Table 4.2: Time Measurements for RPC functions in RPCGen built system (time measurement in microseconds). Server and client are on same physical machine.

Function Signature	int foo(int x)	int foo_add(int x,int y)	char * one_line(char* string)
Measurement 0	90	90	30
Measurement 1	90	20	90
Measurement 2	120	200	80
Measurement 3	30	30	20
Measurement 4	110	100	140

Comparing the performance of these functions one notices that the time for completion across the board is relatively similar. Unlike the baseline measurements in Table 4.1 the RPCGen system's performance does not show one function implementation to be noticeably different than any other in Table 4.2. Noteworthy however is that the resulting system is approximately 5 to 15 times slower in the *char * one_line (char *)* function and significantly slower in the others as well. A more detailed evaluation of these measurements and their impact is provided in Chapter 5. Additional measurements are shown in Appendix A.8.

```
program TESTER {
        version TESTER_V1 {
                int FOO(int) = 3;
                int FOO_ADD(int, int) = 4;
                string ONE_LINE(string) = 5;
        } = 1;
} = 0x2fffffff;
```

Figure 4.1: X file containing RPCGen description for 3 functions.

In this case we were running the server on the same physical machine but in a different process space. Connections were established via the *rpc* system libraries which is the standard method of implementing RPC systems with RPCGen.

4.1.2 Correctness

Correctness of the system is defined as follows: The values returned to the client are identical to those expected and produced by the baseline system. All our tested functions returned the correct results. This is an expected result and does not require further explanation.

4.1.3 Manually written lines of Code

In order to measure the metric for manually written lines of code we examine all the files and required changes to them. Each function required several changes to convert it into a distributed application using RPCGen. Both functions *int foo(int x)* and *int foo_add(int x, int y)* required 9 lines of code to be altered. Five of these lines are additions in the *llist.x* file which defines the RPC system. It is in this "x" file were the program and function versions are defined for the RPC system.

Figure 4.1 contains three definitions for RPC functions which are used in our experiments. One function alone requires at a minimum 5 lines of code in this "x" file. This is the minimum number of lines of code required to specify information regarding the program and function signatures which are used. RPCGen is the tool which uses the entries in the "x" file to automatically generate *C* code which can be used to connect client and server code with each other. Each additional function

adds at least one additional line of code to this ”x” file.

Two additional line changes are required at the client side when calling the remote function such as *foo* and *foo_add*. Here additional parameters are required and the function names actually change to include a prefix identifying the version number of the function. In the end the signature *int foo(int x)* must be altered to *int * foo_1(int x, CLIENT * cl)* (where CLIENT is created via a *clnt_create* system call). Similarly at the server side changes are required to address the change in function naming. Postfixes are added to the function signature automatically by RPCGen to identify the correct callee. Additionally the return value is accessed via a dereferencing operation, resulting in a total of two server side changes. Thus these two simple functions require 9 lines of code changes, neither of which are actually trivial since they modify the interfaces and include syntactic changes resulting in alterations of how the parameters are accessed.

```
return (char **) &ret;
```

Figure 4.2: Return statement in server side code of RPCGen distributed system for the original function *char * one_line(char * string)*. Complexity is high due to dereferencing and casting operations.

The function *char * one_line(char *)* requires additional changes to those named previously for the other functions. This particular function deals with pointers to start with and RPCGen forces the programmer to adapt the system to cope with further modifications of these requirements. This results in additional lines in the client and server side to ensure the correct dereferencing and following of pointers. Figure 4.2 demonstrates the complexity required to produce an error free system for this function by showing the resulting return statement at the server side code. The return value has to be cast to match the function signature and then accessed with the & operator because of the dereferencing that is required to compile the code error free while keeping local changes in the implementation to a minimum.

4.2 ALRPC experiments

The experiments in this section are used to provide a comparison between distributed systems which were built by using ALRPC and RPCGen tools. Additionally, measure-

ments and performance is also compared against the baseline monolithic application. Therefore we are conducting the same experiments as for RPCGen. Measurements are reported here, and evaluation is provided in Chapter 5. ALRPC experiments follow the same procedures and record data for the same metrics are the monolithic application which is used as a baseline.

4.2.1 The tested functions

Functions listed in Table 4.1 are the ones for which time measurements were taken in a monolithic application. We will describe the functions briefly to provide the reader with an understanding of what they achieve and how they have to be modified, if at all, to comply with requirements imposed by ALRPC. While the implementation and integration of functions may require additional work by the programmer, function behaviour should not change when a monolithic system is transformed into a distributed RPC system.

First *int foo(int x)*. This is the simplest use case of a function tested in ALRPC. An integer "x" is passed in as a parameter and returned unchanged. Understandably, this call completes relatively quickly in a monolithic application. This function was one of the early prototypes developed and chosen for testing. It is very simple and requires only the movement of one value across the network. Once this functionality was achieved in our ALRPC system the semantics and mechanisms for nearly all other parameters were identical.

The function *int foo_add(int x, int y)* takes two integer parameters and returns the sum of the two as another integer. This function call too completes faster than the tolerances of the system call used for time measurement allows. Like the *int foo(int x)* function, the purpose is first and foremost to establish the ability of the ALRPC system to move parameters across a network and process boundary. Additionally, this function also was one of the prototypes used to test ALRPC's packing, or serialization or marshalling, mechanisms.

The next function is *char * one_line(char * string)*. This function takes a *char ** pointer. Because of the assumptions made regarding the *C* language semantics, this parameter is assumed to be pointing to the beginning of a string. In this case the

function is also actually dealing with a string. Its purpose is simple. Take an arbitrary string and convert any newline character into a space, effectively turning a string spanning multiple lines into a single line string. This sample function tests ALRPC's ability to deal with *char ** string pointers.

4.2.2 Time measurements

The time measurements of Table 4.1 are considered to be the baseline. These measurements are from calls of a monolithic application run in a single process space. Table 4.3 shows measurements for the same function calls in an RPC system. Here the function calls span multiple process spaces. However, no networking connections are made between different physical machines. We are using Unix domain sockets to establish communication connections between different processes on the same physical machine [52].

Table 4.3: Time Measurements for RPC functions in ALRPC built system (time measurement in micro seconds) using Unix domain sockets. Server and client are therefore on the same physical machine.

Function Signature	int foo(int x)	int foo_add(int x,int y)	char * one_line(char* string)
Measurement 0	190	147	60
Measurement 1	102	48	60
Measurement 2	192	48	90
Measurement 3	148	80	80
Measurement 4	114	392	223

Measurements for Table 4.3 are taken on a non-isolated system. None of these tests occurred in single user mode, which impacts the measured variation. However, the relative comparison between ALRPC system and the non-RPC system is obtained when one compares Figure 4.3 with the results from Figure 4.1. Additionally, server and client are hosted on the same physical machine. Further measurements are shown in Appendix A.8.

Functions where integers are passed as parameters show a significant slowdown. On an non-RPC system the time to complete not only the actual function call but also the system call to assert the return value's correctness are so small that they are not measurable. On the other hand, in the ALRPC system these calls are completed in time frames which are measurable. Somewhat surprisingly the function calls which

required two primitive type parameters to be sent completed in nearly half the time than those calls which only sent one parameter in many cases. These results were independent of the order of the test runs. Results are consistent with preliminary observations where it was determined that the time to completion was independent of whether *int foo(int x)* or *int foo_add(int x, int y)* was run first. The variation therefore is attributed to the other load factors of the system at the time of the experiment. It is however significant that the ALRPC system requires more time to complete the function call than a non-RPC system due to the additional overhead.

For completeness we also performed these same tests with an ALRPC implementation that uses Berkeley sockets for network communication instead of Unix Domain Sockets. The results are presented in the Section 5.3 and evaluated in a comparative analysis.

Table 4.4 shows a comparison between the performance of an ALRPC system using Berkeley socket implementations and a RPCGen system. Here server and client are hosted on separate physical machines and communication speeds are affected by network conditions. We can see from this table that the RPCGen implementation is consistently faster than the comparative ALRPC implementation. Further measurements comparing ALRPC and RPCGen systems on a real network are presented in Appendix A.9.

Table 4.4: Time measurements comparing ALRPC system and RPCGen system where server and client are on different machines and network bandwidth is approximately 38Mbits/sec. Measurement unit in microseconds.

	foo	**foo_add**	**one_line**
ALRPC	6216	5700	6010
RPCGen	1164	1814	1046
ALRPC	5990	7531	6514
RPCGen	1680	1158	7269
ALRPC	5691	3854	6264
RPCGen	1626	755	910
ALRPC	4823	7587	7642
RPCGen	1479	703	1260
ALRPC	6626	7833	5471
RPCGen	1359	1676	1006

In the ALRPC system additional function calls are made and data is copied in and out

of buffers, all of which takes time. The measured slowdown in the *char * one_line(char * string)* function is roughly a factor of 10x. Reasons for this slowdown are most likely the additional overhead produced by the ALRPC system. Further function calls for redirection and data transfers impact performance.

4.2.3 Correctness

Correctness of the systems, whether they are RPC or monolithic applications, is established by comparing returned values with predetermined results. We use the *assert* system call to validate the returned results. Prior to executing these calls the expected value was determined and given as a hard coded parameter to the *assert* function. We are able to establish correctness with this method because non of our systems exhibit nondeterministic or multi threaded characteristics. The *assert* system call used to verify correctness has a small impact on the system's performance. Our own preliminary experiments indicate that the assert call is negligible since no significant changes in performance were observed when the call was removed.

4.2.4 Manually Written Lines of Code

Another parameter which we use to evaluate the success of RPC systems, including the ALRPC system, is the number of lines of code which have to me manually written. We expect ALRPC to perform well in this category especially since one of the original goals of ALRPC was to minimize programmer interaction and produce RPC systems with maximum automation.

This prototype version of ALRPC requires only minimal programmer interference. Functions which are supposed to be turned into remote calls must be provided in a separate header file. The separate header file is a design requirement because the ALPRC tool takes as a parameter the header file which contains all the functions that are going to be turned into remote procedures.

This header file must then be included in the original location to ensure seamless compilation. However, this task can be as easy as moving existing *include* statements into a separate header file. Then this target header file is included in the original

application's source code. An illustration of this process can be seen in Figure 4.3. On the left side one sees the original application's source code arrangement with a code file and header files. These header files are included at the top of the source file. Next assume the programmer wishes to convert all functions inside the header files

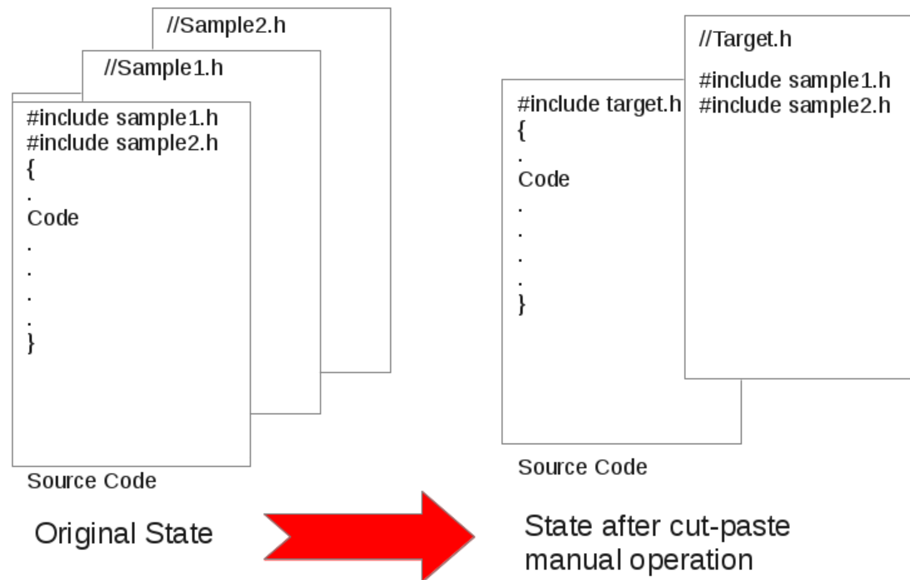

Figure 4.3: Illustration of monolithic system before and after the manual modification necessary to use ALRPC correctly with code base.

sample1.h and *sample2.h* into remote procedures. To achieve this the programmer copies the two lines composing the include statements and places them verbatim in a new header file, here called *target.h*. *Target.h* is then included in the source file and provided as input to ALRPC. These are the only modifications required to instruct ALRPC which functions are converted from local to remote procedures.

Additionally, since ALRPC is not yet at production strength, the actual implementations of the functions must be moved manually to the server side where they can be compiled. Further client and server code to establish connections must be written only once. The actual connection code is reusable since communication occurs via an established standard connection where a buffer is transmitted. Communication is generic, only the extraction and use of the communicated data is unique and automatically produced. Subsequently, the number of lines of code which is manually produced is kept at a minimum.

4.3 Performance Mircobenchmarks

In this section we introduce further micro benchmarks which were gathered to investigate the performance impact on applications when functions are split off. To do this we present two kinds of measurement. The applications we chose are small third party applications which are concerned with solving one specific mathematical problem. This results in a design where a single function is the dominant factor of the application's performance. For our two metrics we first run the entire application on the remote server and measure the execution time of the target function. The second measurement is taken when the application is run on a separate machine with only the target function executing on the remote server. The goal of this experiment is to test the hypothesis that in a hybrid system, the performance of the target function when run on a remote host, is nearly the same as when the entire application is run on the remote host. We expect a small performance hit due to additional networking components and additional function calls.

We focus the evaluation on two small third party applications. One is concerned with detecting prime numbers, while the other is calculating the slope of a function at specific intervals. The compute intensive component of the primes application is shown in Figure 4.4. Computationally this method has a large running time due to the nested loop. The other application implements Euler's method shown in Formula 4.1.

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{4.1}$$

The compute intensive nature of this application is based on the value of h and the size of the interval. The smaller the value for h, the more computations have to be performed before the final result is obtained. Small values of h are desirable to increase the precision and accuracy of the final result of the computation.

The target function for the prime number application has the following signature: *int check(int count, int max_value);*. The function determined the number of prime numbers which are located between 2 and the value of *max_value*. *count* is the counter for the number of prime numbers (normally this variable would not have to be passed in).

Table 4.5 shows time measurements in seconds containing the minimum, maxi-

```
for (candidate=2; candidate<=max_val; candidate++)
{
  max = sqrt(candidate) + 0.001;
  for (n=2; n<=max; n++)
    if (candidate % n == 0)
      break;
  if (n > max)
      count++;
}
```

Figure 4.4: Sample code of prime number function. This is relatively compute intensive.

Table 4.5: Time measurement in seconds for function execution in hybrid and monolithic remote system.

	Hybrid System	Pure Remote System
min	30.7s	30.3s
max	31.8s	31.1s
mean	31.2s	30.7s
std	0.35s	0.21s

mum, mean and standard deviation of our experiments which were conducted over a network with an effective bandwidth of approximately 38MBits/second as determined by iperf measurement. Measurements from Table 4.5 are obtained in the following way: Time stamps are collected immediately before and after the call to the target function. On the hybrid system we see that the calls' performance is slightly worse that of the monolithic system. This can be due to a possible combination of several factors. The hybrid system has several calls for redirection on both the client and server side, plus there is an additional variability in the measurement due to the networking component. However, the networking component can be considered relatively small as was shown in Sections 4.2 and 4.1

The actual measurements show that the target function's performance in a hybrid system is only minimally worse than the functions' performance on a monolithic system. The monolithic system is run on the remote host. Running the monolithic application on the remote host demonstrates that the hybrid system reaches near remote host performance levels. This is useful in the case that the remote host is significantly more powerful than any of the local machines. In such a case, converting the monolithic local application to a hybrid system can yield significant benefits and

even approach the performance of the remote host.

The results of Table 4.5 is confirmed in our second mathematics application. Here the execution time is far greater, in the range of 9 minutes. The difference between hybrid and pure remote host performance becomes even more indistinguishable. The variability is insignificantly small when compared to the total running time of the function. Fractions of a second difference in running time on a total measurement of several minutes becomes an insignificant factor.

Thus we are quite confident that our experiments have confirmed the hypothesis that the performance of a hybrid system can near the performance of a monolithic system run on the remote host at function level granularity. This becomes desirable when particular functions in an application are compute intensive. Moving the compute intensive function to a more powerful remote server can then result in performance improvements.

4.4 ALRPC in Action

We have established in Section 4.2 that ALRPC produces correct results for the scope within which we claim ALRPC's success. Now we describe our experience using ALRPC on real world applications. We experimented with several applications. These applications include a game which is included in the Debian repository, a GIS application's shared library and the OpenSSL shared library on Linux systems.

4.4.1 ALRPC and Real World Games

The purpose of this experiment was straight forward. Move ALRPC out of the realm of purely laboratory self-constructed toy applications and use it with an application that is written and used by a third-party entity. By using a real world application that is included in the Debian repositories we show that ALRPC can be used to convert monolithic applications, such as this game modeled on Tetris, and turn it into a distributed application. Of course for a game such as this we did not expect any performance gains or other benefits which we will introduce in the later sections. However, we are successfully demonstrating that ALRPC can be used on real world

applications and convert them into distributed applications.

4.4.1.1 Introducing LTris

“LTris is a very polished tetris clone, which offers three types of games. The classic mode, a figures mode, where different figures appear every level, and a multiplayer mode. LTris is highly configurable through its menu system” [25]. Its package size is roughly 500kb depending on the particular architecture and its install size is approximately 1200kb. We chose this application because its source code size is manageable yet at the same time it represents a system complex enough to provide real challenges for ALRPC. Additionally the fact that visual inspection of correctness of the ALRPC produced system was possible was seen as a bonus (i.e. the game still works when it is a distributed application).

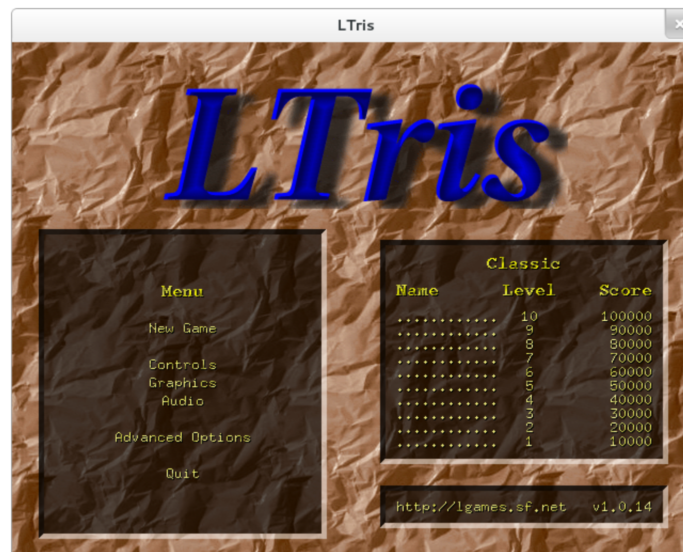

Figure 4.5: Screenshot of LTris after startup.

4.4.1.2 Choosing the right function

Since the purpose of using LTris is to demonstrate that ALRPC can turn a real world application into a distributed system we chose several functions on the merit of highlighting ALRPC’s capabilities. We are not expecting a performance gain, or any other improvement of LTris by converting it into a distributed application. Additionally, as

we will see to be a common theme among the applications which we would like to use ALRPC with, finding suitable functions is extremely challenging. This is partially due to the limitations of the implementation of the mechanism. This is, however, not a shortcoming of the mechanism itself. Many functions have been optimized for monolithic execution over time. Thus complex data types and structures are used which are unfortunately too complex for our implementation of ALRPC.

One issue with LTris is the small number of functions which are suitable for ALRPC. Many functions return *void*; a large number of functions use parameters which are enumerations and structures that contain pointers, something our implementation of ALRPC cannot serialize correctly. Hence these functions are unsuitable for converting into a remote procedure call with our current mechanism. Functions with return type *void* are generally not suitable because they are used to access (read or modify) data which is either a global variable or identified in one of the parameters itself. In either case converting these functions into remote calls is not desired and was thus excluded from the design specifications and capabilities of ALRPC. Ultimately only a small number of functions was found to be suitable. A suitable function is anything that we can use to turn an application into a distributed system.

From the limited number of functions that were identified to be suitable, we chose two to illustrate that ALRPC works with an external application. We focus this experiment on these two because the signatures of the other suitable functions do not differ significantly and thus would not offer additional insights. The calling frequency of the selected functions varies. One of these functions is called rarely (and in some cases we artificially injected calls to this function in order to test the system). The other function we chose is called regularly during game play to calculate the position of every Tetris tile at each step during its descent.

This function *int get_dist(int x1, int y1, int x2, int y2);* returns the distance between two points. It does not appear that this function is called very often. To remedy this, during initial experiments we artificially injected calls to this function in several places to ensure that the remote function works. There is no practical value to this other than testing the resulting distributed system.

However, the second function we chose is frequently called and forms an essential

part of the application's game play experience. For this second function we also needed to cut and paste a struct referred to as *Counter*. The *typedef struct* contains two members *double approach* which is the value used for a smooth counter display during game play and *double value* which is the actual value. The function's signature is "*double counter_get_approach(Counter counter);*". One defining feature of this function is that it takes a structure as a parameter. Simple structures such as this one when used as parameters fall within the scope of ALRPC. Since previous sections haven't highlighted the fact that ALRPC can deal with this type of structures we chose it as an example for the distributed game application.

The next section will describe step by step actions taken to convert the monolithic Linux game LTris into a distributed application with ALRPC. The steps introduced in the following section are applicable regardless of the function type one wishes to convert with ALRPC.

4.4.1.3 Using ALRPC with LTris

The first step of converting the monolithic LTris application to a distributed one was to identify suitable functions as described before. Then two cut and paste operations needed to take place. First the prototype declarations and the *Counter* structure were removed from the header file and pasted, as is, into a separate header file which we created. Then the code actually implementing these functions was cut from the source code file and also placed in a separate source code file. While the header file was then included in the original header with an *#include* statement, the source code was extracted to compile and move it to the server side eventually.

At this point ALRPC can be run with the newly created header file as its input (see Section 3.1). This input file now contains only the structure *Counter* and the two function declarations. As described in Section 3.1, ALRPC now performs various preprocessing steps to extract the return and parameter type information from this header file. Once this information is retrieved the Python modules, which generate *C* code are invoked. The resulting *C* code for this third party application is produced in the exact same manner as the code for the microbenchmarks.

Because ALRPC produces the *C* code according to a previously defined design, the

code is placed in an "output" folder. The LTris build system expects source files to be in specific locations since it too is automated and only possesses limited flexibility to deal with other directories and files. Therefore we now needed to perform additional manual steps.

First we moved the generated files into the "src" directory of LTris. The files moved included the client code files which establishes a connection to the server, the file containing the local stub for redirection to the client and the file containing the serialization. Additionally these source files' header were also moved. These were straight forward copy/move actions from ALRPC's "output" folder to the LTris source code folder to accommodate the requirements LTris' automatic build system. The last remaining manual changes were to remove the relative paths to the "output" folder in the autogenerated files. Previously our tests used the files for compilation directly form the "output" folder and thus ALRPC automatically produced *C* include statements in the headers to point to this output folder. For example a code line of *#include "../../output/header.h"* needed to be changed to *#include "header.h"*.

At this stage the system can be compiled and the server code moved to the remote server. We tested this over fast network connections and only when introducing print statements at the server side were slowdowns noticeable, because they delayed the return of the function. All other functionality of the application was unaffected by this experiment with ALRPC. ALRPC was successfully applied to this application. We chose functions purely on the merit of being suitable for the current implementation of the mechanism. This was done in order to demonstrate that ALRPC can be applied to third party applications which did not originate in our own lab. Thus we established that ALRPC can be used with any application under the condition that this application contains functions which fall within the capabilities of ALRPC.

4.4.1.4 LTris challenges for ALRPC

The primary challenges of using ALRPC with existing systems is twofold. First the implementation of the mechanism faces limitations in regard to the complexity and types of function signatures which can be handled. This complexity is only limited to the type of parameters and return types, not however to the number or order of the parameters. Secondly, existing systems do not readily support use with ALRPC since

their function signatures are often optimized for monolithic executions. Consequently the function signatures are largely unsuitable for the current ALRPC implementation. This is because ALRPC lacks the capability to correctly handle non-basic data types.

LTris in particular uses unsuitable functions which have void return types. These void functions' then are used to modify either its own parameters in some way. Alternatively these particular functions in LTris use globals. Global variables are not detected since ALRPC is working only with the prototype declaration. The declaration does not include information on global variables. Further, complex structures in the function's parameters or its return type are significant challenges for using ALRPC with LTris. Void functions, global variables and complex structures are ideal for LTris as a monolithic application. Unfortunately they limit the number of functions suitable for ALRPC. These are limitations of the implementation rather than the mechanism itself. More importantly, these challenges are not unique to LTris. Existing applications are optimized for monolithic execution and not for RPC suitability, therefore parameters are rarely only of a basic type which is the current requirement of ALRPC.

A minor challenge with LTris and ALRPC was the build system of LTris. Its existing automated build system required source files in specific directories. ALRPC was not custom tailored to accommodate this so manual copying of files needed to take place.

4.4.2 ALRPC and large systems — an investigation

Up to this point our experiments with ALRPC invloved largely micro benchmarks and relatively small applications such as LTris. The small application LTris did not fall into the category of large analysis and compuatational systems for which ALRPC is envisioned to be benficial. This section now introduces some of these large systems where ALRPC is believed to make a positive impact in the future.

4.4.2.1 ALRPC and GDAL

The motivation of applying ALRPC to systems like the Geospatial Data Abstraction Library (GDAL) are resource driven motivations. Scientific projects, businesses and

individual devices such as smart phones, tablets and embedded devices are collecting and retaining unparalleled amounts of data [28]. Initially, spatial locality of the data cannot be assumed. Obtaining a local copy of this data requires time consuming network communication. The movement of data is limited by the transmission rates between the server and the analysis machine. This in turn is dependent on the client machine's network and bandwidth capabilities, both of which can vary greatly [10]. Lets consider the geographical information systems' (GIS) shared libraries (e.g. GDAL) and the common use case of GIS data files which are approximately 250MB in size. Any RPC system reduces the size of data which has to be moved. Instead of moving the data itself, which can be rather large, a single function can now be moved. A single function on the other hand is comparatively small, only a few kilobyte in the worst cases.

GDAL is a large complex system. We did not run ALRPC on GDAL applications due to this complexity. In this section we now present an high level overview of the type of functions which we would ideally like to be able to split off. We also provide some insight into the kind of functions which the current ALRPC mechanism implementation is capable of dealing with in GDAL.

The type of functions which one would like to split off are those which access large data files or those which result in significant computation. These types of functions would benefit from remote execution, under the right circumstances, as the microbenchmarks have indicated. Moving compute heavy functions to remote servers where the performance benefit outweighs the networking costs is a feasible option for RPC. The same applies for functions which move large amounts of data over slow network connections.

The functions located in gdal_alg.h are candidate functions for this purpose. There are 51 functions in this header file. Functions such as

- *int GDALComputeMedianCutPCT (GDALRasterBandH hRed, GDALRasterBandH hGreen, GDALRasterBandH hBlue, int(*pfnIncludePixel)(int, int, void *), int nColors, GDALColorTableH hColorTable, GDALProgressFunc pfnProgress, void *pProgressArg)*
  or

- *PLErr GDALContourGenerate (GDALRasterBandH hBand, double dfContourInterval, double dfContourBase, int nFixedLevelCount, double *padfFixedLevels, int bUseNoData, double dfNoDataValue, void *hLayer, int iIDField, int iElevField, GDALProgressFunc pfnProgress, void *pProgressArg).*

A theme common to nearly all functions in GDAL which one would like to split off is that they contain parameter types which the current implementation of ALRPC is unable to deal with. *void* pointers and complex structures prevent automatic conversions into remote procedures. Without any prior knowledge of the size of the void pointer types any RPC system is unable to correctly marshal these parameters. These two functions highlight the point made earlier that existing systems are optimized for monolithic execution. Passing *void* pointers and structures is efficient in such a scenario, but does not lend itself to an automatic conversion into a distributed application.

Our investigation into GDAL has, however, revealed suitable functions. Here the term suitable means that the current ALRPC implementation is likely capable of dealing with these functions successfully. The following list includes several of the functions likely suitable:

- *int CPL_DLL CPL_STDCALL GDALInvGeoTransform(double *padfGeoTransformIn, double *padfInvGeoTransformOut);*
- *int GDALReadWorldFile (const char *, const char *, double *)*
- *int GDALWriteWorldFile (const char *, const char *, double *)*
- *int GDALCheckVersion (int nVersionMajor, int nVersionMinor, const char *pszCallingComponentName)*
- *const char * GDALDecToDMS (double, const char *, int)*
- *double GDALPackedDMSToDec (double)*
- *double GDALDecToPackedDMS (double)*

These functions are chosen because both their return type and the parameter types are within the current capabilities of ALRPC. The function signatures differ only

in the identifyer names from the function types used in our micro benchmarks and the LTris application. Turning any of these selected functions into a remote call is not expected to provide a performance boost or to improve spatial locality to data. We find that the functions which ALRPC's current implementation is capable of dealing with are not within the realm of achieving the benefits which an RPC could potentially provide when a monolithic application is converted into a distributed one.

4.4.2.2 ALRPC and OpenSSL

OpenSSL is a similarly large shared library as GDAL. The original motivation for OpenSSL was simple: Imagine your version of a shared library is in some way compromised or simply has an incorrect version. However there exists a remote version of this shared library with the correct version. Now one could ship functions which use these shared libraries to the remote host and propagate the results back to the client. This would guarantee that all clients use the same version for a shared library, the one hosted on the remote server.

Yet the very nature of this particular shared library raises several questions. Since OpenSSL deals with security related principles it is counter intuitive to send data over unencrypted network connections to the remote host in order to encrypt the data there for example. Thus, with a shared library such as OpenSSL it was made evident that simply because one can convert functions into remote calls, it is not always desirable to do so.

While there are several functions, for example in the MD4/5 header files, which ALRPC is capable of dealing with, there are also several which are unsuitable from a purely technical standpoint. Reasons for unsuitability are the same for OpenSSL as for GDAL, LTris and any other application we have looked at. Complex structures, void pointers or unsuitable return types are the obstacle. Again, this is an implementation issue unique to our existing implementation of ALRPC.

Expanding this view to the entire shared library directory on the test machine confirms this finding. Many functions are unsuitable for the same reasons. Other functions are however suitable. One such example has been presented earlier in Figure 3.3.

4.4.2.3 Large applications and ALRPC findings

Many applications have been optimized for monolithic scenarios. Splitting these applications into distributable components is difficult. The obstacles are function signatures which are not suitable for ALRPC. These signatures are optimized to deal with a monolithic data model where global variables, pointers to local data and complex structures are the norm. Code and data model are too immovable to accommodate distributed environments and subsequently only a small number of functions are suitable for the current implementation of ALRPC from a technical point of view. These functions are rarely of the type which would improve the applications' performance or data locality.

The GDAL example and to some extent the Tetris example showed that monolithic applications are not written in a way that would allow ALRPC to easily split off truly any function. More often than not, these applications are optimized for monolithic execution and as such the function signatures are complex which often prevents ALRPC from being used. However, if ALRPC is extended to deal with these situations then that would be no problem anymore.

4.5 Feasibility of RPC

This section investigates the pros and cons of RPCGen and ALRPC as they were experienced in this experimental setup. We will focus on the challenges and benefits that each tool made visible to the developer. The purpose of this is to provide a comparison and understanding of the differences and experiences with ALRPC and RPCGen. But, this section also has the purpose of highlighting some common themes of RPC tools and their suitability to today's technological realities.

This section is organized in the following way: First we discuss the overall experience with both RPC tools (RPCGen and ALRPC), then we will focus on the particular experience with each tool individually and highlight perceived strengths and weaknesses.

Both ALRPC and RPCGen produce a distributed system. As mechanisms to convert a monolithic system into a distributed system via RPC, both succeed. The particular details of how this is achieved vary greatly and as a result the details of the

resulting system and their implementations differ. ALRPC is only a proof of concept which investigated a mechanism to increase automation when producing RPC systems. Consequently, the programmer's task is supposed to be kept simple and not involve a lot of work. On the other hand, RPCGen is a production and industrial strength tool which has been developed by multiple programmers and corporate backing. RPCGen is capable of producing complex systems. In order to evaluate these systems in a comparative environment the common denominator was chosen. This was the capability of ALRPC.

First we discuss RPCGen specifically and the experiences that were gathered while developing the system used in our tests. The primary note is that RPCGen requires the use of XDR [27]. This results in the requirement for a programmer to familiarize themselves with the semantic and syntactic peculiarities of the description language used for RPCGen. We have seen that even for small function, such as *int foo(int x)* a disproportionate number of lines of code had to be put into the "x" file to specify the functionality.

Additionally, RPCGen altered the interface or API that one would expect to use in the RPC system. It did so by changing function signatures. These changes were not just limited to names of individual functions, but also extended to changing the types and number of parameters. As a result of these changes the program needed to be altered at the client side and in many cases more significantly at the server side to account for the different parameters. At the server side the strategy used by us was to create appropriate local variables and then convert the input parameters accordingly so that the actual body of the function implementation could remain unchanged and not have to deal with changes to variable access and dereferencing semantics. Of course this also required a change in the return value. However, once this was known about the RPCGen tool, each function required roughly the same modifications and changes in order to compile in the RPC system.

The ALRPC system on the other hand requires the user to essentially cut and paste code into different files. First the programmer is required to cut and paste the include statements which contain functions for RPC systems into a separate header file. This file is used as input for the ALRPC tool as well as for the final system at compile time. Other alterations are kept to a minimum. This results in a high deal of automation.

The down side of this is however that the system is limited in its ability to deal with complicated semantics and syntactic uncertainties inherent to the *C* system. Several assumptions had to be made when ALRPC is running.

Ideally, ALRPC is capable of handling even complex function signatures. An investigation of the 4 real world systems commonly found on today's machines revealed that a large number of functions' parameters and return types fall within the domain of ALRPC. Table 4.6 provides a breakdown of the functions found in the source code of four real life system. These metrics include an analysis of the code of shared libraries in Linux, Firefox, Iceweasel and wget. We see from this table that the majority of functions return simple type parameters. Simple type parameters are defined to be of primitive types or of structures which do not contain pointers. Conversely complex arguments and return types are those which are passed via pointers. Even though ALRPC is able to correctly deal with pointers one level deep, we chose to group all pointers together. *Char ** is again assumed to be what is commonly referred to as a string and uniquely identified in the table. However, most modern systems include combinations of these parameters which highlighted some of ALRPC's implementation weaknesses.

	Shared Libraries	**Iceweasel**	**Firefox**	**wget**
Number of Functions	284126	255	1963702	25886
Arguments containing String	**48349**	**53**	**313929**	**13539**
Simple Arguments	**313453**	**225**	**2015023**	**25550**
Complex arguments	139056	198	812537	10810
Return type String	**1820**	**0**	**3414**	**834**
Return type complex	41086	26	221861	5661
Return type simple	**217664**	**222**	**1579495**	**17603**
Return type void	23556	7	158932	1788

Table 4.6: Breakdown of function and parameter types in real systems

On the other hand, moving functions to the data and not the other way around, can potentially be seen as beneficial. This is especially true if on considers that data sets are only getting larger and performance to move large amounts of data is purely determined by the client's capabilities [10]. Thus, for these reasons function shipping and RPC systems, including ALRPC may be justified.

Section 2.4 includes several of the criticism which any RPC system is faced with. RPCGen certainly is exposed to all of them. ALRPC is likewise prone to most of them. One of the most striking points is the complexity which is involved for the programmer to convert monolithic systems into RPC systems. RPC in general is in decline for these very reasons [53] and is simply outperformed by competing technologies. RPC systems are powerful, but they are also complex and require complex tasks and planning to succeed, while competing technologies like SOAP and REST are sufficient to accomplish the task and are much easier to maintain [56,57]. However, with ALRPC we reduce the manual programmer burden to generate distributed systems using RPC communication mechanisms. Moreover, the contribution of this work is also to identify challenges in marshalling *C* function calls successfully. Once the challenges of *C* semantics are dealt with, either automatically or semi-automatically, the communication mechanism is not limited to RPC anymore and could make use of other technologies such as REST.

The complexity involved in RPCGen to accomplish even the simplest tasks appears to outweigh its benefits. The complexity of ALRPC was far lower for the programmer than then competing RPCGen, however the tool and mechanism is plagued by severe limitations in order to achieve the level of automation currently seen. For our use cases and applications the capabilities of ALRPC may be sufficient, but it is doubtful that they will ever compete with industry strength tools such as RPCGen. Further, security considerations have not been taken into account with ALRPC and an implementation of security standards has an unknown impact in terms of performance and complexity.

Chapter 5

Evaluation, Analysis and Comparisons

In this section we will provide a critical analysis and evaluation of the data obtained in the experiments described in Chapter 4. While doing so we will demonstrate that appropriate research methods have been used. We will also highlight this work's contribution of potentially applying the strengths of RPC systems to the function discovery and identification approach.

The evaluation and analysis will closely address the metrics obtained in Chapter 4. Therefore, this section will first deal with the evaluation of the RPC systems, inlcluding the metric of lines of code, as well as those metrics and benchmarks for correctness and performance. Then we evaluate the proposed combination and integration of RPC systems with the proof of concept work we developed for function identification. We evaluate our approach for function evaluation independently of RPC as well.

5.1 Lines of Code Manually Written

It is very important for us to consider the degree of automation that each RPC system is providing. Automation is one of the main criteria to evaluate our system. Existing RPC systems are considered to be heavy weight and in need of a lot of manual programmer interaction to produce a distributed system and it is this aspect that ALRPC is trying to improve.

Table 5.1 presents the recorded metrics related to manual programmer involvement.

RPC tool name	Function	Lines of RPC description Language	Manual lines of Code changed	automatically generated files	total number of files
RPCGen	int f1 (int x)	9	4	3	6
RPCGen	int f2(int, int)	9	3	3	6
RPCGen	char * f3(char *)	11	5	3	6
ALRPC	int f1(int)	0	1	5	10
ALRPC	int f2(int, int)	0	1	5	10
ALRPC	char * f3(char *)	0	1	5	10

Table 5.1: Manual change metrics. For simple Function signatures as shown ALRPC requires fewer manual code changes and additions than competing RPC tools.

Manual programmer involvement is directly related to one of the main goals of ALRPC: automation of the process of generating a RPC system which is a distributed *C* system that had been converted from a monolithic application. Automation can refer to many things, it includes the number of lines of code changed, the complexity of these changes, and the number of files which contain these changes.

The number of lines of code which needed to be changed is a specific metric which is only concerned with the lines of code which exist before the tool is run but then, as a result of the tool, require alterations. As we will see later this can happen when the tool produces function signatures on either the client or server side which differ from the original system's.

ALRPC as a mechanism is highly automated. The creation of the final RPC system is automated to a large degree. The initial step of analysing the monolithic code's header files, which is the reading in of files, is automated. But this alone does not make ALRPC superior to RPCGen. ALRPC follows the principle of pure RPC systems (see Chapter 2) that neither the programmer nor the compiler are supposed to "know" that they are dealing with a RPC system. As such changes to the function signature are avoided by ALRPC. This then prevents the programmer from having to make changes to these files too, something which is not the case with RPCGen. Ultimately, with ALRPC the programmer does not have to make changes to the files

except in one single place. This change is in the form of adding an "include" statement in the original code's C file.

Including this header file is required because it contains the function signatures and fully expanded headers that the programmer choses to convert into RPC functions. Creating this header file also requires manual involvement, however it is quite trivial in nature. The manual changes that need to be made are: cutting the include statements and putting them in a separate header file, writing the server and client code (these two files are reusable across all applications since all they do is establish a connection and forward the message to the initial RPC component on the server side which is produced fully automatically). This also includes 2 header files bringing the total to four files so far which need to be written manually. Another cut and paste operation is moving the actual implementation into the appropriate stand alone file. The fifth file which needs to be manually produced is also the result of a cut and paste operation again. The actual implementation needs to be put into a separate file. Completing this task requires the programmer to remove the appropriate lines of code and place them in a separate file which can be moved to the server side of the distributed system. This step, too can be achieved with a simple cut and paste operation.

A more complex involvement between programmer and code is required when dealing with RPCGen. RPCGen changes the function signature and requires the programmer to propagate these changes to the client and the server. This results not only in changing the calls themselves, but also changing the implementations on server and client side because the parameters are now different types (see Appendix A.6). Consequently, calls on the client side and the setup for these calls needs to be changed. Further the implementation on the server side needs to be changed and simple cut and paste operations are no longer sufficient. In our simple cases these changes can be effected in a few lines of code and they can be made quite close to where the function call is made, thereby keeping the changes quite contained. However if these calls are made frequently then it may become slightly less attractive as a feature. Ultimately, the number of changes required for even relative simple cases and function calls is enormous. RPCGen introduces additional coding challenges to the programmer in exchange for automatically producing the part of the system that handles network communication. This is a trade-off and depending on circumstances may be justified.

Only two additional manual changes are required in our current implementation to control the network communication implementation. Switching the system from Berkeley sockets to Unix Domain Sockets and vice versa can be done with a simple edit in two *make* file. One *make* file for the server and one for the client control which implementation is used and compiled.

Overall the changes in ALRPC are simple cut and paste operations, no actual changes are required beyond the one line change to include the new header file which contains the prototype declarations which will be redirected via RPC. On the other hand RPCGen requires the user to make complicated implementation changes in multiple places. Any time there is a change required it introduces the potential for errors and increases development time. Even for simple use cases as we have seen them, these manual changes are substantial.

RPCGen also has the description language *RPCL* which requires lines of code to be written into a separate file. For our simple cases, a minimum of five lines of code are required to produce just one function in an RPC system. ALRPC does not have a description language, one simply extracts all the header files which include the declarations of the functions one wants to turn into RPC functions. Then, with ALRPC one simply points the tool at this file and an automated process takes over. This automated process generates *C* code and produces the complete RPC system.

As such, when comparing ALRPC with RPCGen in terms of the available automation when producing an RPC system both have advantages over the other in certain areas. ALRPC produces more files, though two of them are directly related to the code files and are just simple headers. These code files are reusable for all other systems which follow since they contain standard networking code in *C* and forward a buffer which is in the standard ALRPC format (see Figure 3.2). One advantage of ALRPC is that it does not require any changes to the implementation. Function signatures are directly adopted and made to work in the RPC system in an automated way. On the other hand RPCGen can deal with more complicated cases and has a clear advantage in this respect. RPCGen is able to deal with a wide variety of *C* systems since the description language has more clearly defined semantics than *C* itself (i.e. the description language has a string type which can be of variable length, ALRPC cannot

clearly identify a string and relies on assumptions which limits the uses). This gives RPCGen an advantage in the complicated cases, but ALRPC has the clear advantage in terms of automation and the complexity of changes required to produce a working stable system. ALRPC does not require complicated changes to the implementation. RPCGen, on the other hand, forces the programmer to modify code in a number of places; many of these changes are not trivial either. Thus, automation as a design goal of ALRPC is successfully achieved, though there are limitations as a consequence of this high level of automation.

We observed similar results in terms of automation when using ALRPC with third party applications. In the case of LTris and the microbenchmarks of math applications semi automation was achieved. Simple cut and paste operations dominated the required user involvement.

5.2 Correctness of RPC

For any system to be considered viable it is expected to demonstrate the correct behaviour. That means the system must perform the tasks as expected and produce the expected results. In our evaluation we consider the RPC system which is produced by the tool ALRPC and compare it with the industry strength tool RPCGen. For both ALRPC and RPCGen systems correctness is defined as producing identical output to that of the non-RPC system which is used as a baseline.

When the system is ready, after the required modifications are made (see Section 5.1), RPCGen produces the correct output for a broad range of potential conditions. RPCGen is able to produce correct output for the *C* language.

ALRPC on the other hand has certain functional restrictions. ALRPC is only able to deal with primitive data types, simple structures which do not contain pointers within them, one level of dereferencing of primitive data types and simple structures. This excludes a fair number of potential parameter types available in the *C* language. Considering the limitations, ALRPC is correct within the constraints of not exceeding beyond those restrictions.

Consequently, the evaluation of both systems is limited to the common set of ca-

pabilities of both systems. While our tests remain within those restrictions for the functionality of our RPC systems in this test, both final RPC systems produce the correct results.

Therefore both final systems fulfill the minimum condition for any system to be considered and investigated: a system must produce the correct output. RPCGen does produce the correct output after all the modifications are made which are forced on the programmer by the automation steps. ALRPC is correct for the cases with which it is able to deal. This was confirmed via micro benchmarks on our own code, as well as a third party applications in the form of a game and micro benchmarks using small third party applications.

5.3 Performance Evaluation

Performance evaluation is based on how much time the completion of each function call requires in the final RPC system. We establish micorbenchmarks where we are directly comparing ALRPC with RPCGen and use the measurements of the monolithic application as a baseline. Comparing ALRPC and RPCGen is necessary to gain an understanding of the viability of our mechanism. We also demonstrated via our experiments that an ALRPC produced system is achieving performance result near those of the remote server. If the automation, which is a key feature of the mechanism, is not able to produce a system that is competitive with an existing tools' results then the merits of ALRPC are questionable.

First we consider the micro benchmarks comparing ALRPC with RPCGen. Measurements for the monolithic system are introduced in Table 4.1, those of the ALRPC system are in Table 4.3 and Table 4.2 contains the measurements for the RPCGen system. As anticipated, ALRPC performs generally worse than the system created by RPCGen. Figure 5.1 illustrates this. Both the best and worst measurements of ALRPC are larger than those of the RPCGen system. However, ALRPC can be considered to be a competitor to RPCGen purely on a performance basis. While the general timings of ALRPC are worse than those of RPCGen, an industry strength tool, the measurements do overlap. That is, Figure 5.1 shows clearly that some cases of the ALRPC benchmarks fall within the range of those of the RPCGen system. This is confirmed, by all but the most simple function call. The bar representing the

data point for each functions' mean includes a standard deviation range. Standard deviations of these function calls overlap except in the simplest case of function *foo*.

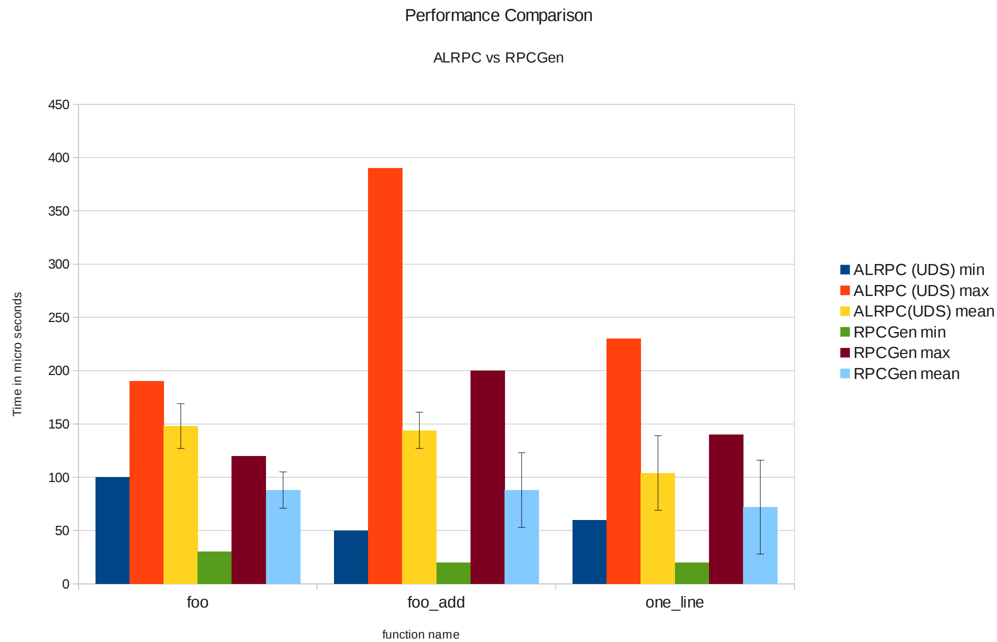

Figure 5.1: Comparing performance on a function by function level between ALRPC and RPCGen.

Overall however, ALRPC is, as expected, slower than RPCGen when client and server are on separate physical machines. Appendix A.9 confirms in an experimental setup with 10000 runs that over a real network ALRPC is approximately 3-8 times slower than a distributed application which was generated by RPCGen. The reasons for this are manyfold. Primarily we believe that ALRPC performs worse than the RPCGen system because ALRPC has not been optimized. Neither the automatically generated code is optimized, nor are there hooks for dynamic software optimizations. That is there is no code within the generated system that is able to dynamically detect whether certain optimizations or shortcuts are possible on a function by function basis. There is a lot of redirection, on both client and server side, because every call has to go through the same lengthy call sequence to be decoded successfully. Another alternative to improve performance could relate to copying values into the

buffer. Currently the copying is done on a step by step basis for every parameter which needs to be marshalled. This implementation could be replaced with more efficient vector copy operations where multiple values are moved as one operation. Other optimizations include varying the number of parameters which are sent across functions. This optimization would potentially speed up the intra-process communication as currently some functions have as many as seven parameters which is not optimal and forces parameter passing not via fast registers.

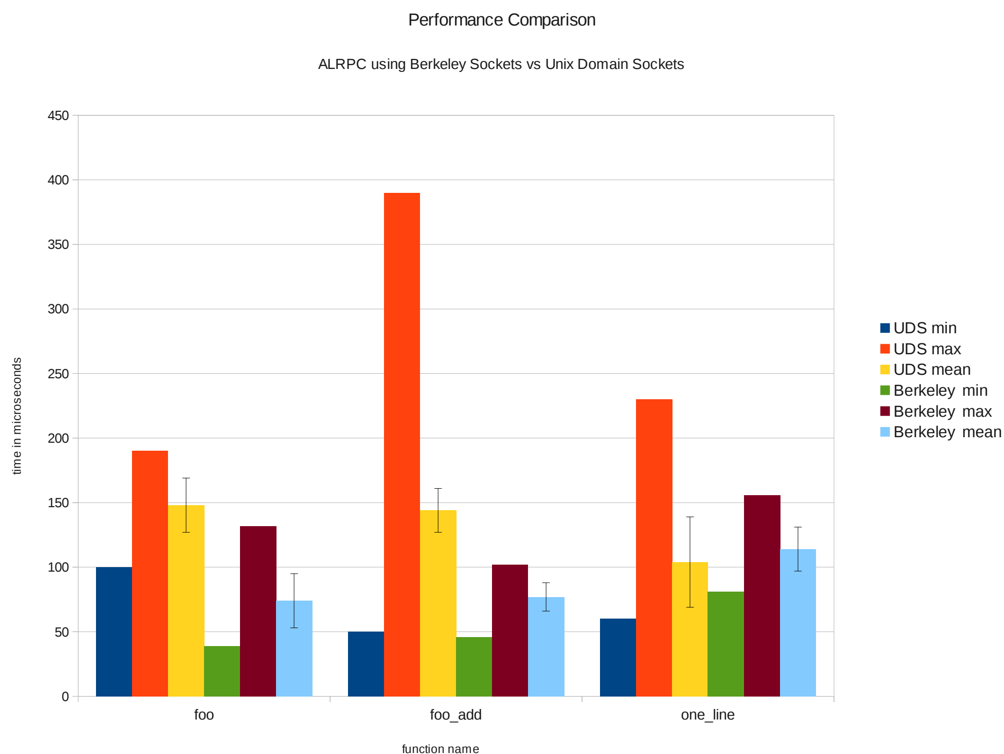

Figure 5.2: Comparing performance on a function by function level between ALRPC using Berkeley sockets and Unix Domain Sockets for communication.

Figure 5.2 illustrates the ALRPC system's performance using different implementations for the network communication. Using Berkeley sockets has resulted in a, on average, better performance than when ALRPC is using Unix domain sockets. However, the measured performance results are overlapping in their minimum, mean and maximum times for the same functions. Here we also see that the overlap and proximity of the standard deviation bars in the graph is very close. This means that no clear statement about the superiority in terms of performance of one system over the other

can be made. These graphs were produced with the use of 5 experimental runs, or data points. UDS sockets are expected to perform better than Berkeley type sockets when communication on the same physical machine. This observation is corroborated in Appendix A.8 where the same experiment was performed with 10000 trials in order to gain data which lends itself to statistical observation. Subsequently we are unable to draw definitive conclusions regarding one implementation being faster than another. Using Berkeley sockets is required when running this system over a real network connecting server and client on two separate physical machines. The results in Figure 5.2 are also promising as they are an indication that any performance drawback is not significantly impacted by the underlying network communication implementation. Comparing the Berkeley socket implementation of ALRPC with the system produced by RPCGen shows that performance of the two systems is roughly the same.

Using small third party applications concerned with solving mathematical questions we were able to determine that the hybrid system, one where performance dominating functions are shipped to a remote server, perform roughly the same as if the entire application had been executed on the remote server. This supports the conclusion that the performance of these functions is largely impacted by the remote host's capabilities and the networking component plays a minor role. Using RPC in these situations becomes feasible when

$$P_{old} > P_{new} + N \tag{5.1}$$

where P is the performance on an old monolithic system and on remote host system respectively; N is the performance of the networking component of such a system. Splitting functions off the application is feasible for performance if the network connection plus the execution time on the remote host is less than then execution time of the function on the original host. We have shown that the networking component is relatively small to ship the parameters to the remote function and the overall performance measurement is dominated by the performance of the remote host for the function.

Despite the clear advantage that RPCGen has, such as years of development, industrial backing and sponsors, large development teams, its performance advantage

is not necessarily significant compared to ALRPC. All in all, ALRPC is considered to be a viable competitor in terms of performance to RPCGen.

Neither system is able to perform anywhere near as fast as the monolithic baseline system. This result is also expected since the compiler has a lot more possibilities to optimize when all functions are in the same process space. Optimizations such as inlining and loop unrolling and more cannot be done in RPC systems. Additionally, the baseline system does not have the additional performance burden of the RPC systems in the form of function call redirections. These redirections introduce overhead not only because there are additional function calls which require access to the stack; but also because these additional functions have a greater number of parameters there is even more potential for performance slowdowns when these parameters are passed to functions not via registers.

Chapter 6

Conclusion and Future Work

Throughout this work we have introduced ALRPC, a mechanism for the automated generation of RPC systems written in the *C* programming language, in order to answer the following research question: *Can an automated mechanism successfully generate a distributed system stemming from a monolithic legacy C application that is suitable for today's technological realities of large data and distributed systems.*

ALRPC is a mechanism that is far more automated than the existing tools. This automation is a key feature to attempt to provide an attractive RPC tool which can compete with the much simpler to use mechanisms like SOAP and REST. Automation for ALRPC comes in the form of simple programmer tasks. The programmer is only required to perform simple cut and paste operations when using the ALRPC tool to generate a distributed system. This automation succeeds in providing a system which presents, to both the programmer and the compiler, a system which is not identifiable immediately as one that is using RPC. As a result of this achievement the programmer is neither required to write programs via a description language, nor is the programmer required to modify the existing code base to match APIs and function signatures that were automatically created by the tool. Minimal programmer interaction was successfully achieved with ALRPC.

ALRPC is however more limited in its capabilites as a result of the automation. While tools such as RPCGen are powerful, ALRPC is limited in its use. RPCGen achieves this great applicability by requiring the programmer to write a RPC system program description in a different language which can be read in by the tool itself. This description language is semantically stronger than that of the target system *C*.

Consequently, the description language and file are used to generate a custom tailored RPC system. However, RPCGen forces major rewrites even for simple remote procedures. This produces a large amount of overhead in terms of programmer involvement and introduces the potential for errors. While RPCGen is largely automated, it does require substantial programmer involvement to produce a working RPC system.

ALRPC is largely automated but suffers from limited cases where it can be used in. In order to succeed in this large degree of automation compromises were made. These compromises resulted in allowing the ALRPC tool to make several assumptions about the semantics of the code and function signatures it processed to convert them into and RPC system. Largely, these assumptions follow safe and good programming principles. However, there are no guarantees that these assumptions hold in all cases, thus these cases must be classyfied as weaknesses of the system.

We have shown that ALRPC, as an implementation of an automated mechanism can successfully generate a distributed system stemming from a monolithic legacy *C* application. We have demonstrated this via micro benchmarks to illustrate that ALRPC produces a system that is comparable in performance to state of the art solutions like RPCGen. Further, we have shown that ALRPC requires less manual involvement of the programmer to generate an RPC system. We have applied ALRPC to a monolithic application and converted it semi-automatically into a distributed application while maintaining its correctness and performance characteristics. As such ALRPC is successful in generating a distributed system from a monolithic system. Yet, due to the limitations of our implementation, the size and scope of large distributed systems (GIS/GDAL) we were unable to demonstrate ALRPC directly in these specific scenarios. At the same time, we have, however, shown that ALRPC as a mechanism can be applied to some aspects of these large systems and that a large number of other functions can be suitable to an extended implementation of ALRPC.

6.1 Future of ALRPC

Immediate future work of ALRPC includes expanding its scope and usability. This means that ALRPC needs to be expanded to be able to deal with complex function signatures in *C*. Most likely this will also mean a major shift of the initial scope of ALRPC's static analysis component. Currently the static analysis component is

only concerned with function signatures. This is often inadequate when dealing with complex data types. As such the static analysis module will need to be expanded and an analysis of more than just the function signature is warranted. Further the limitation of only dealing with *C* is rather restrictive. Chosing a different static analysis approach may also allow us to expand the range of target programming languages.

Beyond the immediate technical challenges, for future work we propose a system which alters an application's characteristic behaviour dynamically in a self-managing fashion. In other words this system re-configures itself from a monolithic application to a distributed system. This would mean that an application becomes self-adaptive and intelligently determines whether to use its traditional monolithic approach where, for example, data is moved to the application, or whether it uses its distributed features and invokes remote procedure calls. Such a decision would be based on dynamic profiling metrics which determine the feasibility and benefits of either approach.

We believe that multiple use cases for systems which could benefit from ALRPC exist today. Consider GDAL and the common use case of GIS data files. These files in our projects are approximately 250MB in size. 250MB of data represent information for a tile which is 25km X 25km and stored with lossless compression in 5 bands of light. Mapping any sizable geographic region with this method results in large amounts of stored data. To move this data is costly and prohibitive. Further, GDAL is a legacy application written in *C* and is still heavily used in production environments.

Obtaining access to data is not the sole reason for converting an existing monolithic application into a distributed system using remote procedures. Another scenario is centred around log files. Servers always produce log files, whether they originate from applications or the kernel itself. These log files are usually moved to a central server for storage or analysis. The logging functionality which is gathering the information is required to be situated on the actual server where it performs logging. However, the function which writes the information onto disk in a log file can be a remote procedure. This would allow all servers to automatically write their log files to a central server, pre-empting the need for data movement later and making the log collection scripts obsolete.

However, as Tanenbaum points out remote procedures are not ideal for all situations [53]. Many argue that remote procedures should never be used on networks which are exposed to outside traffic [34]. For example, there are many functions in the *OpenSSL* shared libraries which are suitable for conversion into remote procedures from a technical standpoint. Yet, the reasoning behind doing so would defeat the purpose of the function itself: sending data over an unsecured network to encrypt the data is counter intuitive. In other words, simply because an analysis of a system suggests that functions can be converted into remote procedure, it is not implied that these functions actually should be turned into a remote procedure.

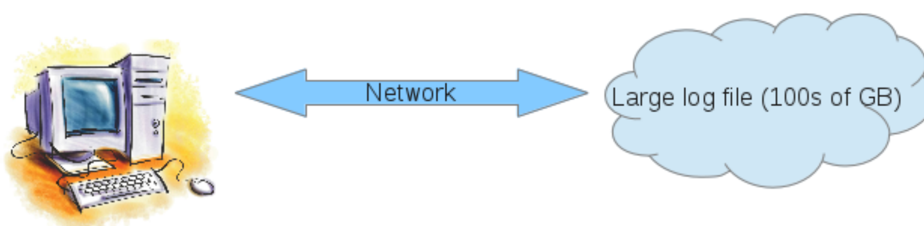

Figure 6.1: Servers where analysis of log files occurs are not the same as where data files are stored.

A further avenue for ALRPC to pursue in connection with *C* is the integration in Integrated Development Environments (IDEs) and so called wizard functionality. Through ALRPC much of the marshalling can be automated. However, there are aspects of *C* semantics which are ambiguous. Combining automation of ALRPC with programmer guidance and wizard functionality of IDEs has the potential of significantly facilitating the refactoring of monolithic legacy applications into distributed systems semi-automatically.

Additionally, as alluded to earlier other concerns are jurisdiction and validation. While data movement is often constrained by jurisdiction of the data owner, access to the data is less limited. Remote procedures are ideal to allow analysis of these types of data sets (often government collected data sets). When moving only individual functions to the jurisdiction of the data, one improves spatial locality of the function and data while at the same time retaining the ability to run the remainder of the application in a separate jurisdiction all together. Further, the ability to validate computational results can also be increased with remote procedure calls. Imagine one suspects that the local version of a shared library is corrupted or out

of date. We envision that one would use ALRPC to convert this monolithic application into a distributed one. Now the distributed application can ship functions to a remote server where the correctness of the shared library is validated and guaranteed.

Improving ALRPC so that it is able to be fully used in one of these scenarios is the driving force behind the future work. Data sets continue to grow, existing alternatives lack some of the power which the legacy systems had, but refactoring legacy applications manually is not an option. Automating remote procedure call generation for legacy systems further, we hope to provide a viable alternative to the status quo.

Appendix A

Additional Information

A.1 Monolithic C code — Micro Benchmarks

```
#include <stdio.h>
#include <sys/time.h>
#include <assert.h>
#include <string.h>
#define TRUE 1
#define FALSE 0
#define DEBUG FALSE


__attribute__ ((noinline)) char *  one_line(char * string) {
   int i;
   i = 0;
   while (string[i] != '\0') {

      if (string[i] == '\n') {
         string[i] = ' ';
      }
      i++;
   }
   return string;
}

__attribute__ ((noinline)) int foo(int x){
   return x;
}
```

```
__attribute__ ((noinline)) int foo_add(int x, int y) {
   return x+y;
}
int main () {
   struct timeval tv_start;
   struct timeval tv_end;
   int x, y;
   char z;
   x = 300;
   y = 300;
   z = 6;

   int num_of_tests = 0;
   int num_of_tests_t = 3;//11;

   gettimeofday(&tv_start, NULL);
   assert(x == foo(x));
   gettimeofday(&tv_end, NULL);

   /*print the time stamps*/
   printf("time: %ld.%06ld, %ld.%06ld\n", \
      (long int) tv_start.tv_sec, (long int) \
      tv_start.tv_usec, (long int) tv_end.tv_sec, \
      (long int) tv_end.tv_usec);
   num_of_tests++;
   printf("passed: int foo(int x)\n");

   gettimeofday(&tv_start, NULL);
   assert(foo_add(x,y) == x+y);
   gettimeofday(&tv_end, NULL);
   /*print the time stamps*/
   printf("time: %ld.%06ld, %ld.%06ld\n", \
      (long int) tv_start.tv_sec, (long int) \
      tv_start.tv_usec, (long int) tv_end.tv_sec, \
      (long int) tv_end.tv_usec);
   num_of_tests++;
   printf("passed: int foo_add(int x, int y)\n");

   char  s[] = "hello this is the\nworld";
   gettimeofday(&tv_start, NULL);
```

```
    assert(strcmp("hello this is the world", one_line(s))==0);
    gettimeofday(&tv_end, NULL);

    /*print the time stamps*/
    printf("\ntime: %ld.%06ld, %ld.%06ld\n",\
        (long int) tv_start.tv_sec, (long int) \
        tv_start.tv_usec, (long int) tv_end.tv_sec,\
        (long int) tv_end.tv_usec);
    num_of_tests++;
    printf("passed: char * one_line(char *)\n");

    printf("\n\nPassed %d / %d\n", num_of_tests, num_of_tests_t);
    return 0;
}
```

A.2 C code after manual modification due to RPC-Gen

Relevant code diff for micro benchmarks of application when using RPCGen.

```
CLIENT *cl;
.
.
.
gettimeofday(&tv_start, NULL);
result = foo_1(x, cl);
gettimeofday(&tv_end, NULL);
.
.
.
gettimeofday(&tv_start, NULL);
result = foo_add_1(x, y, cl);
gettimeofday(&tv_end, NULL);
.
.
.
gettimeofday(&tv_start, NULL);
char **s2 = one_line_1(s, cl);
gettimeofday(&tv_end, NULL);
```

A.3 Function calls for system when using ALRPC

Relevant code diff for micro benchmarks of application when using RPCGen.

```
gettimeofday(&tv_start, NULL);
result = foo(x);
gettimeofday(&tv_end, NULL);
.
.
.
gettimeofday(&tv_start, NULL);
result = foo_add(x, y);
gettimeofday(&tv_end, NULL);
.
.
.
gettimeofday(&tv_start, NULL);
assert(strcmp("hello this is the world",\
   one_line("hello this is the\nworld"))==0);
gettimeofday(&tv_end, NULL);
```

A.4 C code for client using "Berkeley" sockets

Relevant code diff for micro benchmarks of application when using RPCGen.

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>

int MSG_SIZE = 4096;


int cli_connect_buffer(void * buffer) {
        int sockfd = 0, n = 0;
    char recvBuff[MSG_SIZE];
```

```
    char sendBuff[MSG_SIZE];
         sprintf(sendBuff, "hello test string");
    struct sockaddr_in serv_addr;

    memset(recvBuff, '\0',sizeof(recvBuff));
    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Error : Could not create socket \n");
        return 1;
    }

    memset(&serv_addr, '0', sizeof(serv_addr));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(5000);

    if(inet_pton(AF_INET, "142.104.69.191", &serv_addr.sin_addr)<=0)
//    if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0)
    {
        printf("\n inet_pton error occured\n");
        return 1;
    }

    if( connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {
       printf("\n Error : Connect Failed \n");
       return 1;
    }
         write(sockfd, buffer, MSG_SIZE);

         read(sockfd, buffer, MSG_SIZE);

        close(sockfd);
    return 0;
}
```

A.5 C code for client using Unix Domain sockets

Relevant code diff for micro benchmarks of application when using RPCGen.

```
#include <stdio.h>
#include <sys/socket.h>
```

```
#include <sys/un.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <limits.h>
#include <assert.h>
#include "uds_helper.h"
#include "uds_client.h"


int UNIX_PATH_MAX = PATH_MAX;

int MSG_SIZE = 4096;

/* int cli_connect_buffer(char * buffer)
 * Receives a buffer of size MSG_SIZE and forwards it to the server
 * Arguments:
 *      buffer  char * containing a serialized version of the actual function call
 * Result:
 *      returns an int currently, should change this to return void * so that the
 *              caller can just return it without problem. The return value should be
 *              specified somewhere in the buffer probably.
 */
int cli_connect_buffer (void *buffer) {

        /* overlaying struct message over buffer */
        message *msg;
        msg = (message *) buffer;

        /* preparing unix domain socket (uds) */
        struct sockaddr_un address;
        int socket_fd, nbytes;

        socket_fd = socket (PF_UNIX, SOCK_STREAM, 0);
        if (socket_fd < 0) {
                printf ("socket() failed\n");
                return EXIT_FAILURE;
        }
        memset (&address, 0, sizeof (struct sockaddr_un));

        /* populating the struct for the socket */
        address.sun_family = AF_UNIX;
```

```
        snprintf (address.sun_path + 1, sizeof(address.sun_path) - 2, SOCK_PATH);

        /* make the connection */
        if (connect (socket_fd,
                        (struct sockaddr *) &address,
                sizeof (struct sockaddr_un)) != 0)
        {
                printf ("connect() failed\n");
                return EXIT_FAILURE;
        }


        /* write to the socket and pass the buffer on to the server */
        int rc = write (socket_fd, msg, MSG_SIZE);
        assert(rc != -1);

        /* read the server's response */
        nbytes = read (socket_fd, buffer, MSG_SIZE);

        assert(nbytes != -1);

        /*      close the socket's filedescriptor handle        */
        close (socket_fd);

        /* I should return a void pointer so the caller can figure out what to do
         * but this should come from the buffer somewhere
         */
         return EXIT_SUCCESS;

}
```

A.6 Lines of code manually changed or added due to use of RPCGen

At a minimum 5 lines of code are added for the first function, a minimum of 1 additional line is added for every further function:

```
program PRINTER {
        version PRINTER_V1 {
                int FOO(int) = 1;
                int FOO_ADD(int, int) = 2;
```

```
            string ONE_LINE(string) = 3;
        } = 1;
} = 0x2fffffff;
```

Each function requires the added client structure:

```
CLIENT *cl;
```

Each function call needs to be appended with the _[version number] and an additional parameter for the network connection.

```
foo_1(x, cl);
result = foo_add_1(x, y, cl);
char **s2 = one_line_1(s, cl);
```

The function signature is further changed by altering the return type with one further pointer, resulting in a type change of the return variable.

```
int *result;
char **s2....
```

The return type on the server side is forced to be passed by address, requiring one line of code to change.

```
int result;
```

Passing an argument of type string requires two additional lines of code to change as well.

```
static char * ret = string;
return (char **) &ret;
```

A.7 Statistical Data Monolithic Micro Benchmarks

Table A.1: Number of trials which did not result in 0 time measurment

	100000 trials	1000000 trials
foo	0	0
foo_add	2791	29301
one_line	8215	55397

.

Table A.2: one_line function statistics, monolithic C application, 1 million trials.

min	0
max	39
mean	0.0595
std	0.2582

Table A.3: 10000 trials ALRPC, same physical machine Unix Domain sockets, Measurement in microseconds.

	foo	add	one_line
min	11	12	11
max	24818	1114	117
mean	17.17	14.59	17.5
std	248.2	23.28	14.45

Table A.4: 10000 trials RPCGen same physical machine. Measurement in microseconds.

	foo	add	one_line
min	16	15	20
max	18159	18225	17617
mean	51.77	41.81	51.18
std	726.7	567.7	564.2

A.8 Statistical Data ALRPC vs. RPCGen (local)

A.9 Statistical Data: ALRPC vs RPCGen (separate physical machines)

Table A.5: 10000 trials ALRPC different physical machine (38MBit/s). Measurement in microseconds.

	foo	add	one_line
min	4021	4429	7247
max	214899	108806	15793
mean	10306	10247	9760
std	3747	4787	2059

Table A.6: 10000 trials RPCGen different physical machine(38MBit/s). Measurement in microseconds.

	foo	add	one_line
min	667	674	733
max	97553	111127	59389
mean	1546	1470	1638
std	2551	2257	1593

Bibliography

[1] XDR: External Data Representation Standard. http://tools.ietf.org/html/rfc1832/, 1995. [Online; accessed 19-11-2012].

[2] Writing Remote Procedure Calls (RPC) in C. http://www.cprogramming.com/tutorial/rpc/remote_procedure_call_start.html, 2010. [Online; accessed 20-11-2012].

[3] CTAGS Manual Pages. http://linux.die.net/man/1/ctags, 2012. [Online; accessed 10-11-2012].

[4] RPCGen Manual Pages. http://man.he.net/man1/rpcgen, 2012. [Online; accessed 03-12-2012].

[5] SWIG Manual Pages. http://www.manpagez.com/man/1/swig/, 2012. [Online; 10-12-2012].

[6] Amazon. AWS Pricing. http://aws.amazon.com/ec2/pricing/, 2012. [Online; accessed 27-10-2012].

[7] L. Baresi and L. Pasquale. Adaptive goals for self-adaptive service compositions. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 353–360. IEEE, 2010.

[8] D.M. Beazley et al. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk workshop*, pages 129–139, 1996.

[9] N. Bencomo, P. Sawyer, G. Blair, and P. Grace. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In *2nd International Workshop on Dynamic Software Product Lines (DSPL 2008), Limerick, Ireland*, volume 38, page 40, 2008.

[10] A. Bergen, Y. Coady, and R. McGeer. Client bandwidth: The forgotten metric of online storage providers. In *Communications, Computers and Signal Processing (PacRim), 2011 IEEE Pacific Rim Conference on*, pages 543–548. IEEE, 2011.

[11] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems (TOCS)*, 8(1):37–55, 1990.

[12] B.N. Bershad, D.T. Ching, E.D. Lazowska, J. Sanislo, and M. Schwartz. A remote procedure call facility for interconnecting heterogeneous computer systems. *Software Engineering, IEEE Transactions on*, (8):880–894, 1987.

[13] A.D. Birrell and B.J. Nelson. *Implementing remote procedure calls*, volume 17. ACM, 1983.

[14] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[15] R.T. Brandle, D.L. Goodliffe, D.E. Keith, R.A. Robinette, R.C. Sizemore, G.J. Smithwick, and A.J. Zappavigna. Remote procedure calls in heterogeneous systems, June 8 1993. US Patent 5,218,699.

[16] V. Cardellini, E. Casalicchio, V. Grassi, F. Lo Presti, and R. Mirandola. Towards self-adaptation for dependable service-oriented systems. *Architecting Dependable Systems VI*, pages 24–48, 2009.

[17] A. Chervenak, E. Deelman, C. Kesselman, B. Allcock, I. Foster, V. Nefedova, J. Lee, A. Sim, A. Shoshani, B. Drach, et al. High-performance remote access to climate simulation data: a challenge problem for data grid technologies. *Parallel Computing*, 29(10):1335–1356, 2003.

[18] G. Chesson. XTP/PE overview. In *Local Computer Networks, 1988., Proceedings of the 13th Conference on*, pages 292–296. IEEE, 1988.

[19] D.D. Clark and D.L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM SIGCOMM Computer Communication Review*, volume 20, pages 200–208. ACM, 1990.

[20] Xerox Corporation. *Courier: The Remote Procedure Call Protocol*. Xerox system integration standard. Xerox Corporation, 1982.

[21] G.F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design*. Addison-Wesley Longman, 1994.

[22] Y.K. Dalal. Use of multiple networks in the xerox network system. *Computer*, 15(10):82–92, 1982.

[23] T.J. Davidson and M.T. Kelley. Method and system for implementing remote procedure calls in a distributed computer system, April 26 1994. US Patent 5,307,490.

[24] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[25] Debian. Package: ltris (1.0.14-1), 2012.

[26] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3):313–341, 2008.

[27] M Eisler. XDR: External Data Representation Standard. `http://tools.ietf.org/html/rfc4506`, 2006. [Online; accessed 04-11-2013].

[28] genevaassociation.org. Risk management n 47 / may 2010. `www.genevaassociation.org/PDF/Risk_Management/GA2010-RM47.pdf`, 2010. [Online; accessed 05-01-2013].

[29] GNU. GNU GCC Preprocessor Commands. `http://gcc.gnu.org/onlinedocs/gcc/Preprocessor-Options.html#Preprocessor-Options`. [Online; accessed 15-09-2012].

[30] A.M. Goldsmith, D.B. Goldsmith, and C.E. Pettus. Object-oriented remote procedure call networking system, February 13 1996. US Patent 5,491,800.

[31] H. Gomaa and K. Hashimoto. Dynamic self-adaptation for distributed service-oriented transactions. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, pages 11–20. IEEE, 2012.

[32] M.S. Gordon, D.A. Jamshidi, S. Mahlke, Z.M. Mao, and X. Chen. COMET: Code Offload by Migrating Execution Transparently. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 93–106. USENIX Association, 2012.

[33] R.D. Hill, A.S. Williams, R.G. Atkinson, T. Corbett, P. Leach, S.J. Chan, A.A. Mitchell, E.K. Jung, and C.H. Wittenberg. Method and system for network marshalling of interface pointers for remote procedure calls, April 23 1996. US Patent 5,511,197.

[34] IBM. IBM Internet Security Systems: SunRPC. `http://www.iss.net/security_center/advice/Services/SunRPC/default.htm`, 2013. [Online; accessed 24-10-2012].

[35] S. Krakowiak. Middleware architecture with patterns and frameworks. 2007.

[36] S.S. Laurent, J. Johnston, and E. Dumbill. *Programming web services with XML-RPC*. O'Reilly Media, 2001.

[37] S. Loughran and E. Smith. Rethinking the Java SOAP stack. In *IEEE International Conference on Web Services (ICWS)*, volume 5, page 2005, 2005.

[38] LWN.net. Sun RPC code to be relicensed. `http://lwn.net/Articles/319648/`, 2009. [Online; accessed 27-11-2012].

[39] E. McManus. Inter-process communication using different programming languages, October 28 2008. US Patent 7,444,619.

[40] Sun Microsystems. RPC: Remote Procedure Call Specification. RFC 1050. `http://tools.ietf.org/html/rfc1050`, 1988. [Online; accessed 05-01-2013].

[41] P.D. Mosses, P.D. Mosses, J. Palsberg, P.D. Mosses, P.D. Mosses, M. Bidoit, and C. Choppy. The Sun RPC Language Semantics. In *Proceedings of PANEL'92, XVIII Latin-American Conference*, volume 655, pages 329–343. Universidad de Las Palmas de Gran Canaria.

[42] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on*, pages 240–249. IEEE, 1998.

[43] M. Musicante. The Sun RPC language semantics. *DAIMI PB*, 21(419), 2002.

[44] B. Niu and G. Tan. Enforcing user-space privilege separation with declarative architectures. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pages 9–20. ACM, 2012.

[45] Oracle. RPCL by Oracle. `http://docs.oracle.com/cd/E19963-01/html/821-1671/rpcproto-24229.html`, 2010. [Online; accessed 12-12-2012].

[46] F. Panzieri and S.K. Shrivastava. Rajdoot: a remote procedure call mechanism supporting orphan detection and killing. *Software Engineering, IEEE Transactions on*, 14(1):30–37, 1988.

[47] V. Paxson. End-to-end internet packet dynamics. In *ACM SIGCOMM Computer Communication Review*, volume 27, pages 139–152. ACM, 1997.

[48] M. Schroeder and M. Burrows. *Performance of Firefly RPC*, volume 23. ACM, 1989.

[49] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. Overview of GridRPC: A remote procedure call API for grid computing. *Grid ComputingGRID 2002*, pages 274–278, 2002.

[50] P. Soule. Introduction. *Autonomics Development: A Domain-Specific Aspect Language Approach*, pages 1–6, 2010.

[51] R Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. `http://www.ietf.org/rfc/rfc1831.txt`, 1995. [Online; accessed 11-10-2012].

[52] Thomas Stover. Demystifying unix domain sockets. `http://www.thomasstover.com/uds.html`, 2011. [Online; accessed 08-09-2012].

[53] A.S. Tanenbaum and R. van Renesse. *A critique of the remote procedure call paradigm*. Vrije Universiteit, Subfaculteit Wiskunde en Informatica, 1987.

[54] B.H. Tay and A.L. Ananda. A survey of remote procedure calls. *ACM SIGOPS Operating Systems Review*, 24(3):68–79, 1990.

[55] S. van der Burg and E. Dolstra. A self-adaptive deployment framework for service-oriented systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 208–217. ACM, 2011.

[56] S. Vinoski. RPC under fire. *Internet Computing, IEEE*, 9(5):93–95, 2005.

[57] S. Vinoski. RPC and REST: dilemma, disruption, and displacement. *Internet Computing, IEEE*, 12(5):92–95, 2008.

[58] Steve Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, 1997.

[59] J. Waldo. Remote procedure calls and java remote method invocation. *Concurrency, IEEE*, 6(3):5–7, 1998.

[60] JE White. RFC 707, 1975.

[61] T. White. *Hadoop: The definitive guide*. O'Reilly Media, 2012.