RPC Automation: Making Legacy Code Relevant

Andreas Bergen, Yağız Onat Yazır, Hausi A. Müller, Yvonne Coady
Department of Computer Science, University of Victoria, Canada
{andib, onat, hausi, ycoady}@cs.uvic.ca

***Abstract*—Due to the well-known issues with Remote Procedure Calls (RPC), the rather simple idea of modifying legacy applications—that have low spatial locality to the data they need to process—to execute all of their procedures via RPC is not a feasible option. A more realistic and feasible alternative is to provide a self-management mechanism that can dynamically monitor and alter the execution of an existing application by selectively modifying certain procedures to execute remotely when it is necessary to improve spatial locality. In this paper we describe the motivations behind such a self-management mechanism, and outline an initial design. In addition, we introduce our vision for the required profiling component of these applications. As such, we introduce the Automated Legacy system Remote Procedure Call mechanism (ALRPC). It automatically converts existing monolithic *C* applications into a distributed system semi-automatically. Thus automation is a key criterion for successfully competing with existing remote procedure tools for legacy applications and with newer solutions such as SOAP and REST [12], [21]. ALRPC is the core component to convert monolithic applications into distributable self-adaptive RPC systems. The empirical results collected from our initial experiments show that our mechanism's level of automation outperforms existing industry strength tools and improves development time. At the same time our mechanism is able to correctly function with a significant code base and shows acceptable performance in initial tests.**

***Index Terms*—Self-managing system, cloud computing, large data, remote procedure calls**

I. Introduction

As a result of the paradigm and trend shifts in modern computing over the last decade, a number of fields gained special importance and prominence. Cloud computing, remote sensing, social networks, health information systems and scientific computing are a few of these fields. 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 [9].

Despite 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. Performance penalties can be particularly severe in the cases where the spatial locality of application and the data to be processed are not provided by default. Spatial locality here means that applications and their data are separated by few, if any, routing hops and low network latency. Moreover, the performance penalty can be highly variable due to external influences on the network traffic and network connectivity [20]. For instance, scientific data that is recorded in one location is stored presumably in nearby data centers. Researchers around the world requiring access to the data are now faced with the challenge to quickly and efficiently move the data from the data center to their own location for analysis or perform limited analysis remotely. Even when computation is done on machines (worker-nodes) hosted in the same data center, data movement requires bandwidth resources. Identifying and addressing these issues is challenging with traditional models [10]. As a result, it is becoming necessary to consider alternate approaches where pieces of application code are carried to where the data exists.

Traditionally, when an application requires access to data, the data is obtained and moved to the physical machine on which the application resides. This commonly accepted model was altered in the early 1980s by the introduction of Remote Procedure Calls (RPC)—the original ideas date as far back as the mid-1970s [6]. The main purpose of RPC is to provide a mechanism which facilitates remote execution via migration of code—in the form of procedures—to the location where the data exists [6]. The remote execution's results are returned to the caller and the caller's system resumes execution [6], [22].

Key ideas of remote procedure calls have been extensively studied and discussed since the 1980s. In the 1990s, several small companies and large industry leaders alike actively supported development to overcome limitations of these systems [7], [16], [19]. Throughout RPC's existence detractors have been pointing out conceptual problems, technical challenges and performance weaknesses [23]. In summary, RPC is considered complex and cumbersome, requiring additional description languages and code changes [25], [26]. Furthermore, latency, security and legislative restrictions pose further challenges. Due to these issues, the rather simple idea of modifying legacy applications—that have low spatial locality to the data they need to process–to execute all of their procedures via RPC is not a feasible option. A more realistic and feasible alternative is to provide a self-management mechanism that can dynamically monitor and alter the execution of an existing application by selectively modifying local procedure calls to execute remotely when it is necessary to improve spatial locality. Facilitating such a level of self-management requires dynamic monitoring and profiling of metrics such as latency, security and privacy in order to determine when an application will chose to execute remote procedures instead of following its traditional behaviour.

In this paper we describe the motivations behind such a self-management mechanism for legacy applications, and outline an initial design. We focus exclusively on legacy applications

in C with available source code. While our mechanism is designed for legacy applications faced with issues of security, latency and jurisdictional restrictions, our evaluation is focused solely on latency. In addition, we discuss our vision for the required profiling component of these applications. While latency, security, privacy and performance are important aspects of these systems, this paper focuses specifically on the static analysis mechanism. As such, we introduce our Automated Legacy system Remote Procedure Call mechanism (ALRPC). It automatically converts existing monolithic *C* applications into a distributed RPC system with minimal programmer interference. Automation is critical to successfully compete with existing remote procedure tools for legacy applications as well as to be able to compete with newer solutions such as SOAP and REST [12], [21]. ALRPC is the core component to convert monolithic applications into distributable self-adaptive RPC systems. The empirical results collected from our initial experiments show that our mechanism's level of automation outperforms existing industry strength tools and improves development time. At the same time our mechanism is able to correctly function with a significant code base and shows acceptable performance in initial tests.

The remainder of the paper is organized as follows: We present a problem description in Section II to motivate the need for our approach. Then Section III situates our contribution among related works. Sections IV and V will showcase an overview of the system and some experimental results, before Sections VI and VII elaborate on the current limitations of the system and conclude the paper.

II. Problem Description

Scientific projects, businesses and individual devices such as smart phones, tablets and embedded devices are collecting and retaining unparalleled amounts of data [15]. 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 [5]. This cost is incurred in the form of latency.

Secondly, this existing approach is also very likely to incur further costs because providers such as Amazon's EC2 have payment models where one is charged for moving data as well as for storage requirements [2]. Following this approach would duplicate a data file's storage requirements, at least temporarily, and incur costs from simply moving data to the analysis server. At some point this network communication reaches a threshold in terms of time or monetary cost which makes it no longer feasible for the system to move the data to the computation server. For instance, consider geographical information systems' (GIS) shared libraries (e.g. Geospatial Data Abstraction Library (GDAL)) and the common use case of GIS data files. These files are approximately 250MB in size. 250MB of data, in specific cases, 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.

As an alternative, any RPC system reduces the size of data which has to be moved. Instead of moving 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 fiscal and temporal terms. However, the question remains: what functions of an application should be moved to the server which is located closer to the data? By identifying candidate functions using appropriate criteria one can keep data movement between servers to a minimum while ensuring that the process of executing functions remotely is as automated as possible. Existing state of the art RPC tools for legacy applications require large manual overhead and often refactoring of existing code bases. In addition to this overhead, programmers are often required to produce description language files for these tools. Thus programmers chose other approaches instead of RPC solutions [23], [25], [26].

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. Yet, 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, remote procedures are not a panacea [23]. Many argue that remote procedures should never be used on networks which are exposed to outside traffic [18]. 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 for doing so would defeat the purpose of the function itself because 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 procedures, it is not implied that these functions actually should be turned into a remote procedure.

A further advantage of distributed systems is validation and verification of computational 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. On the other hand this is 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 exacerbated when there are large data sets which have to be moved to the local machine. In our distributed system, the application can start local and remote function calls and accept the first returned result as the approved response. Thereby an application can utilize superior computational power of remote servers of data centers. This is similar to *map reduce* approaches, however our existing tool allows the automatic modification of existing legacy applications to obtain the same leverage [13], [27].

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. In other words, the reasons why a previously monolithic application would benefit from a dynamic conversion to a distributed system using remote procedure calls are closely tied to: (1) latency, (2) security, and (3) jurisdiction.

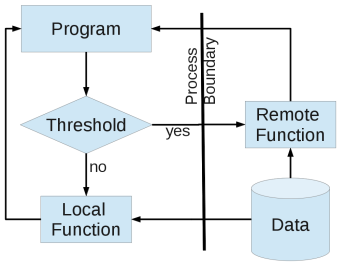

Fig. 1. Decision to move data to local server or to use remote function is based on threshold (Figure grounded in GDAL use case)

These issues point to the need for a self-management mechanism that can equip a legacy application with the ability to dynamically monitor, profile and alter its execution by selectively modifying certain procedures to execute remotely when it is necessary or suitable to improve spatial locality. The required self-management mechanism should capture the changes related to the three issues mentioned above, and decide whether it is necessary to alter the program's execution to use a remote function. Figure 1 illustrates the decision making structures and actions of such a system. In this paper, we propose and outline such a self-management mechanism, and focus on the automation of converting existing applications into distributable RPC systems.

III. Related Work

Service-oriented architecture (SOA) is often used to solve spatial distribution of data and the related challenges of self-adaptive systems described above. 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 [14]. Adding self-adaptive deployment and configuration models of SOA systems addresses the challenges of networked distributed systems [24]. Similarly Cardinelli proposes self-adaptive models to dynamically react to environment changes to increase a system's dependability [8].

Increasingly complex systems led to discussions of software reconfiguration patterns for dynamic software adaptation; Gomaa describes patterns for transactions where more than one service needs to be updated and coordinated [17]. 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 [3]. Bencomo proposed a slightly different approach where a more formal methodology is used to describe a solution more closely linked to that of middleware [4].

SOA exposes a service to the application while not revealing any implementation details. Middleware approaches only replace communication methods of SOA applications and not the conceptual model. Our approach provides the ability to control the communication method, the self-adaptive nature of the application and the exact implementation of the remote procedure while at the same time overcoming challenges common to SOA and middleware approaches.

Finally, our proposed solution differs from approaches which target mobile applications and use complete thread migration—such as CloneCloud [11]—by using RPC mechanisms to redirect a call to a different execution environment instead of migrating threads to a point of execution.

IV. Proposed Solution

This section introduces each of the architecture components required to build a system which dynamically reconfigures itself from a monolithic into a distributed application.

A. Static Analysis

We developed a proof of concept implementation which enables automated generation of distributed RPC systems. Our tool, Automated Legacy code Remote Procedure Call (ALRPC), is able to semi-automatically convert a monolithic application into a distributed system. ALRPC, like any RPC mechanism abstracts the burden of programming networking code. With ALRPC a programmer is able to automatically extract the desired functions from the program and convert the system into one using remote procedure calls. This conversion follows a static analysis of the application's code base prior to run time.

ALRPC generates, code stubs for client and server, as well as communication procedures automatically. The differentiating feature of ALRPC compared to other Remote Procedure Call tools written for *C* applications, such as RPCGen [1], is its degree of automation. Automation is the primary goal.

As such, the intended use case is to provide ALRPC with an existing monolithic application to transform it automatically into a distributed system. Generally a programmer is only required to provide ALRPC with a header file that includes the procedures one wishes to execute remotely. From this input alone, ALRPC generates client procedure stubs, networking code, server procedure stubs, marshalling and unmarshalling code automatically. The final system at this stage does not differ significantly from other RPC systems, barring the automation which generated it.

The purpose of the header file is to allow ALRPC to perform a static analysis of the given source code to determine function signatures. This analysis is necessary to identify data types and names of the procedures' parameters, as well as the names of the procedures themselves. These pieces of information are procured and further used by existing Linux tools and ALRPC itself. Figure 2 illustrates the process, actors, input and output files involved when using the entire tool chain centered around the ALRPC mechanism. A *target* header file containing the definitions of the functions which one desires to execute remotely is preprocessed by *GCC*. Then the preprocessed file is parsed by *ctags* to extract the function signatures. These signatures are in turn provided to *Python* modules of ALRPC which analyse them and automatically produce *C* code for a fully functional distributed system which has the same purpose as the original monolithic system.

Special care is taken to ensure that the selected functions are suitable for remote execution. Criteria include data restrictions (legal or size), function parameter types (not all types are acceptable) and calling frequency.

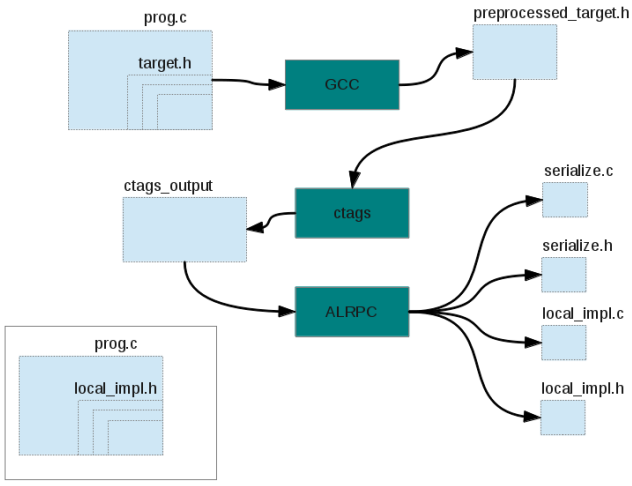

Fig. 2. Static control flow of ALRPC tool chain

B. Self-Adaptive Dynamic Component

ALRPC is the mechanism which enables our proposed system to be created with minimal and simple programmer interaction from a static analysis of the application's code base. Through it we are able to turn a monolithic application into a distributed system with far more ease than with any competing RPC tool for legacy systems. But in order for the distributed system to function efficiently we require an additional component: the profiler. The profiler monitors all metrics for threshold values which decide on using local or remote function calls. Metrics for such a profiler are centered around data models, latency and legal requirements. The profiler will, application specific, detect the latency of file transfers for example. Additionally, data models can also be used in the profiler to determine when data cannot be moved to the application due to legal constraints. In both cases the profiler could trigger the execution of a remote procedure.

Specifically, once a latency threshold is reached, the application will no longer attempt to transfer the data to the application, instead a switch forces the application to use a remote function. The intended benefit here is that the time for invoking a remote function and the function completion time are less in sum, despite the networking component, than a transfer of the data.

Keeping a cache of metrics allows for intelligent decision making processes. Once a decision is made to switch modes to remote procedures, periodic updates of this cache are required to ensure network and process conditions are still warranting this switch rather than a return to the original model.

V. Evaluation

In this section we present performance micro-benchmarks and measurements regarding automation of generating RPC systems with ALRPC.

A. Experimental Setup

A need for a customised, application specific solution to determine when to use local or remote function calls is purely dependent on the network capabilities of each client machine. A client machine requires access to the data and the performance of this operation is solely dependent on the client's network capabilities [5]. Table I illustrates that even for a 50MB file the GET times vary greatly dependent on the network capabilities of the client and on spatial locality of the data. In this paper we focus on a mechanism which enables us to convert a monolithic application into a self-adaptive distributed system. A primary criteria of this mechanism is automation to far greater levels than competing RPC tools for applications written in *C*. Our experiments target aspects

TABLE I
GET times in seconds for Geographically Dispersed Nodes accessing 50MB file on AWS S3 account hosted on US West Coast [5]

Site	Min	Max	Mean	Median
UVic	17	532	53	26
UW	3	110	15	12
UKY	45	76	48	47
Germany	159	179	169	168
Brazil	213	613	344	294
Taiwan	224	230	226	226
France	161	4243	950	998
EC2	1.7	67	10	4.7

of ALRPC to determine its competitiveness in regard to performance of the final system and the degree of automation when converting the system.

In the Geospatial Data Abstraction Library (GDAL) we find functions with signatures such as *int GDALCheckVersion (int nVersionMajor, int nVersionMinor, const char * pszCallingComponentName)*, *int CPL_DLL CPL_STDCALL GDALInvGeoTransform(double *padfGeoTransformIn, double *padfInvGeoTransformOut)* and *int GDALWriteWorldFile (const char *, const char *, double *)*. Figure 3 shows measurements comparing ALRPC implementations with RPCGen implementations of the same system. Here functions *f1*, *f2* and *f3* represent a subset of real systems functions with which ALRPC can currently deal with. Function *f1* has the signature *int f1(int x)*, *f2* has *int f2(int x, int y)*, and *f3* represents *char * f3 (char * s)* which can be mapped to GDAL functions.

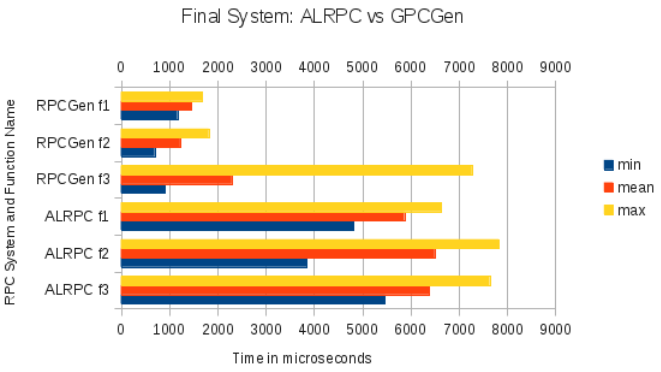

Fig. 3. Comparing ALRPC system and RPCGen system where server and client are on different machines and network bandwidth is approximately 38Mbits/sec. Time shown in microseconds measuring call completion

The measurements represent call completion time of each remote function call. Measurements were obtained over a network with approximately 38Mbits/sec bandwidth (obtained via iperf measurement). For relatively large data files (a few MB) this system may not exhibit a need for remote procedure calls at all. However, we propose a system which can dynamically monitor and profile an application on a case by case basis. The same experiment as in Figure 3 was performed when connecting to the server via a mobile phone network. Here the available bandwidth ranged between 150 and 170Kbits/second. A profiler as we propose it can determine that data transfers of even a few MB require significantly more time than executing a remote function call. This is even true when using a mobile phone network where the remote function call time for each of the tested functions experienced a 100-300 times slowdown compared to the measurements in Figure 3.

The system using ALRPC was built semi-automatically, only requiring cut and paste edits to move function definitions into a separate header file. Using the RPCGen tool required code changes to the application as well as creating additional description language files. While the RPCGen system shows a performance benefit, the manual programmer involvement was significantly larger. ALRPC required cut and paste type changes to the existing code base with the additional effort of including one header file in the source code. RPCGen on the other hand required significant changes such as creating files in a RPC description language. Additionally RPCGen also required code changes to reflect automatically generated function stubs, identifiers and parameter types. Lastly, for the functions tested RPCGen forced us to use it in a mode which is incompatible with some older auto-generated files.

B. Empirical Results

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.

Table II presents the recorded metrics related to manual programmer involvement. 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* application that had been converted from a monolithic application. Automating this process addresses the number of lines of code changed, the complexity of these changes, and the number of files which contain these changes.

TABLE II
MANUAL CHANGE METRICS. FOR SIMPLE FUNCTION SIGNATURES AS SHOWN ALRPC REQUIRES FEWER MANUAL CODE CHANGES AND ADDITIONS THAN COMPETING RPC TOOLS

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)	5	4	3	6
RPCGen	int f2(int, int)	5	3	3	6
RPCGen	char* f3(char *)	5	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

Even for simple test functions ALRPC requires far fewer changes than the competitor RPCGen. Moreover the changes for ALRPC are simple cut and paste operations of moving prototype declarations and headers into new files. RPCGen requires code changes due to automatically generated function stubs which do not agree with the original code base. Unlike RPCGen, ALRPC does not require additional description language files either.

Analysis of Figure 3 shows that there is some slowdown in the resulting system when using ALRPC compared to an industry strength tool such as RPCGen. However this slowdown is, dependant on the use case, acceptable for the gain ALRPC provides in automating the code refactoring from monolithic application to distributed RPC system.

VI. LIMITATIONS

Semantics of the target language *C* and the goal of automation produce several limitations for ALRPC. These limitations

affect the automation of generating a distributed application, they do not impact the viability of our proposed approach.

First ALRPC can only deal with simple function signatures. *Simple* refers to primitive data types, pointers of primitive data types and structures. All cases are limited to one level of pointer redirection only. Semantics of *C* are ambiguous and make it impossible to obtain a unique data type in some situations. To illustrate this complexity, consider the case of *char * s*. Here,"s" is likely to be one of two types. First, "s" could be a pointer to a null terminated string. Secondly, "s" could also be a pointer to a location in memory of size *char*. Even for simple cases like this, *C* semantics are ambiguous.

Despite these limitations we find that there is still a large body of suitable functions. Table III shows an analysis of functions in source code of different origins. The table separates the return types and parameters. It shows the percentage for each category with which ALRPC is able to deal with automatically. ALRPC is able to deal with a large body of functions in real world systems. Whether these functions *should* be converted to remote calls is not answered by this table.

TABLE III
PERCENTAGE OF FUNCTIONS AND PARAMETERS WHICH ARE OF A TYPE ALRPC COULD DEAL WITH

	Shared Libs	**Iceweasel**	**Firefox**	**wget**
Acceptable Argument type	**72.2%**	**58.4%**	**74.1%**	**78.3%**
Acceptable Return type	**77.2%**	**87.0%**	**80.6%**	**71.2%**

With ALRPC we can now develop self-adaptive systems that monitor, profile and alter an application's behaviour to switch between local and remote functions. The motivations for such a system are latency, large data files, legal restrictions on the movement of data and security concerns. Such a system can turn a monolithic application into a distributable system with limited manual programmer involvement.

VII. CONCLUSION

We have shown a need for applications to dynamically switch between local and remote function call implementations. We have made this argument on latency requirements. Security concerns and legislative requirements regarding the data's location were not considered in this paper, but these concerns can outweigh any consideration for technical benefits as illustrated above. We have also shown that a dynamic profiling capability needs to be implemented on a case by case basis. Further, it is the capability of semi-automatically converting monolithic applications into distributed systems that makes these considerations feasible in the future. Through ALRPC manual changes to the code base are fewer and simpler than with competing tools such as RPCGen. ALRPC is the mechanism that allows us to semi-automatically convert existing monolithic applications into a distributable system. Once this is achieved profiling of the application allows the system to become truly self-adaptive based on application and scenario specific metrics. ALRPC outperforms existing RPC tools in this task due to its high degree of automation.

REFERENCES

[1] RPCGen manual pages, 2012.
[2] Amazon. AWS Pricing, 2012.
[3] 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.
[4] 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.
[5] 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.
[6] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.
[7] R. Brandle, D. Goodliffe, D. Keith, R. Robinette, R. Sizemore, G. Smithwick, and A. Zappavigna. Remote procedure calls in heterogeneous systems, June 8 1993. US Patent 5,218,699.
[8] 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.
[9] 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.
[10] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of network and computer applications*, 23(3):187–200, 2000.
[11] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314, 2011.
[12] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the web services web: an introduction to soap, wsdl, and uddi. *Internet Computing, IEEE*, 6(2):86–93, 2002.
[13] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[14] 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.
[15] genevaassociation.org. Risk management n 47 / may 2010, 2010.
[16] A. Goldsmith, D. Goldsmith, and C. Pettus. Object-oriented remote procedure call networking system, Feb. 13 1996. US Patent 5,491,800.
[17] 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.
[18] IBM. Ibm internet security systems: Sunrpc, 2013.
[19] E. McManus. Inter-process communication using different programming languages, Oct. 28 2008. US Patent 7,444,619.
[20] V. Paxson. End-to-end internet packet dynamics. In *ACM SIGCOMM Computer Communication Review*, volume 27, pages 139–152. ACM, 1997.
[21] L. Richardson and S. Ruby. *RESTful web services*. O'Reilly Media, Incorporated, 2007.
[22] R. Srinivasan. Rpc: Remote procedure call protocol specification version 2. 1995.
[23] A. Tanenbaum and R. van Renesse. *A critique of the remote procedure call paradigm*. Vrije Universiteit, Subfaculteit Wiskunde en Informatica, 1987.
[24] S. van der Burg and E. Dolstra. A self-adaptive deployment framework for service-oriented systems. In *Software Engineering for Adaptive and Self-Managing Systems*, pages 208–217. ACM, 2011.
[25] S. Vinoski. Rpc under fire. *Internet Computing, IEEE*, 9(5):93–95, 2005.
[26] S. Vinoski. Rpc and rest: dilemma, disruption, and displacement. *Internet Computing, IEEE*, 12(5):92–95, 2008.
[27] T. White. *Hadoop: The definitive guide*. O'Reilly Media, 2012.