

# Tracing software evolution history with design goals

Neil A. Ernst and John Mylopoulos  
Software Engineering Lab  
University of Toronto, Canada  
{nernst, jm}@cs.utoronto.ca

**Abstract**—When designing software for evolvability, it is important to understand which particular designs have worked in the past – and which have not. This paper argues that understanding the history of a software innovation is valuable in setting the context for future innovations. There is no formal discipline of software history. While there is an active body of research in IT and innovation management, which seeks to understand how to maximize value from IT spending, this research often ignores the meaningful technological underpinnings of such tools. We suggest that the study of design history should be extended to software artifacts. The paper introduces notions like requirements analysis, technology context, and social context to explain how, and why, certain technologies evolved as they did. We apply these concepts to the history of distributed computing protocols. We conclude with observations drawn from this history that suggest designing software for evolvability must consider the history of similar applications in the requirements analysis.

## I. MOTIVATION

Lehman and Ramil [1] describe the *nounal* view of software evolution as being concerned with “the nature of evolution, its causes ... [and] its impacts ... (p. 35).” One way of looking at software evolution is to examine the low-level artifacts, such as metrics about lines of code, number of modules, or commit logs. What is missing in such studies is a broader look at the the causes of software evolution. Such an examination has two aspects. One, higher-level artifacts need to be considered. Software requirements fulfil this objective. They are high-level expressions of intentions, describing a need in the environment that can be met by instantiation in software. The second aspect is that such a study must be historical, looking at multiple products over a broad sweep of time.

What form should such a study take? One form might be a broad-brush history. For example, Mahoney [2] looks at the entire field of software engineering. Most histories of this type, including many found in the IEEE journal *Annals of the History of Computing*, focus on software engineering in the large, examining the people involved, the decisions made, and the end results. From a software evolution perspective, though, the interesting questions are not “what happened”, but rather, ‘why did design X get chosen and not design Y’?

The motivation for this paper is to demonstrate that an enquiry into the goals a particular software technology tried to satisfy – its high-level requirements – is very helpful to maintainers and implementers of today’s new technologies. It behooves us here to include the well-worn, but nonetheless relevant, quote about history, by George Santayana: “Those who cannot remember the past are condemned to repeat it.”

Studying the change in how technology innovators understand a particular problem captures several things. One is an understanding of the large-scale context for a particular technological problem. The second is a series of lessons in problems faced and challenges overcome. Such problems and challenges may well repeat themselves in the future.

In this paper, our argument is laid out as follows. First, we introduce other work on this topic, and highlight differences. We then introduce the case study we use to argue for conducting disciplined software history. Our case study is a narrative history of distributed computing protocols. This section concludes with our interpretation of the evolution of this technology. We then suggest some lessons the evolution of these requirements provides, and conclude with a call for more such analysis as a way to better understand the context underlying software evolution.

## II. RELATED WORK

There have been a few other efforts to tell the story of specific software technology or systems. Anton and Potts [3] report on a feature-oriented analysis of corporate telephony offerings, a narrative they term functional morphology: the shape of benefits and burdens of a system. They excavate features from past iterations of the system, and use that to argue for their interpretation of the changes in requirements. The principle difference with this paper is their focus on features rather than requirements. They worked with closed-source software which prevented them from delving into the requirements. Similarly, Gall *et al.* [4] describe how they used software releases to gain insight into evolutionary trends. From a product release database, they examine the growth-rate and change in system modules. There is no focus on why changes are made. Spira [5], in the short-lived publication *Iterations*, gave a brief history of the TCP/IP protocol and its creation. This was, again, a history that focused on “what happened” rather than why.

## III. A HISTORY OF DISTRIBUTED COMPUTING PROTOCOLS

To demonstrate the usefulness of an intentional history of software evolution, we present a case study on distributed computing protocols. What we look for is the design decisions, rationale and requirements that motivated each particular implementation or proposal. We can then compare those decisions and rationale with preceding and subsequent work, trying to construct the narrative for the work on distributed computing. Figure 1 shows a rudimentary ‘phylogenetic tree’ for this history.

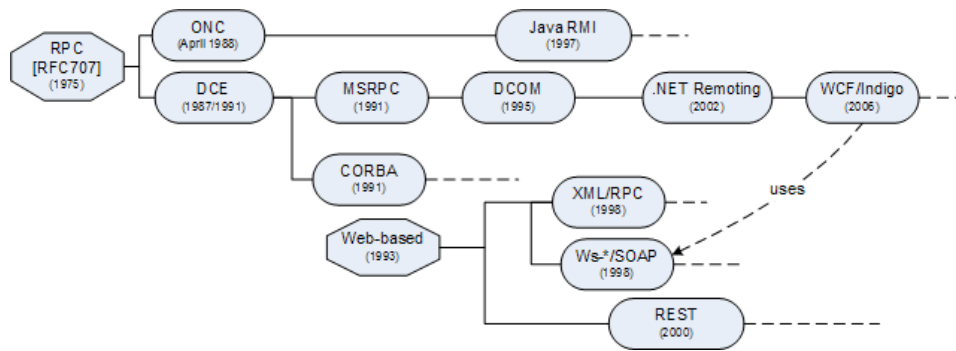


Fig. 1. Phylogeny of distributed computing protocols

### A. Telling disciplined software history

The aim of disciplined history is a plausible explanation of past events [6]. Plausible explanations use sources (evidence) to build an argument about why events occurred. Historians traditionally have used primary sources such as letters, diaries, and newspapers from the past. For telling the story of a software technology, that is, its evolutionary record, the primary sources such artifacts as code, mailing lists, and release-specific documentation, such as READMEs. From these sources, a historian constructs a narrative, providing facts and interpretation.

For this look at the intentionality behind specific technologies, our primary artifacts are the published specifications and proposals for each protocol. These capture, in the words of the innovators, the reasons behind the new proposal. We also draw on histories of the software industry at specific dates to reconstruct the wider context behind each decision. Finally, we draw on the details in the specifications to establish the *de facto* intentions of the protocol. One gap in our reconstruction is a consideration of how the specific protocol evolved – for example, how CORBA changed from version 1.0 to version 2.0. We leave these detailed analyses for future work. We also omit analysis of the actual implementations of the protocols, in favour of a look at the intentions, rather than the reality. Needless to say, how well a specification is implemented has a strong effect on future changes to it and competing specifications.

### B. Definition of distributed computing

Computer programs typically consist of data and definitions for operations thereon. The operations are typically defined as procedures, functions, or methods, and define the steps – the algorithm – for accomplishing the desired output state. Programs can also be seen as resource manipulators - from a start state to an end state. *Distributed computing* is the ability to manipulate resources on disconnected, heterogeneous systems. Such capabilities were one of the reasons computers were networked in the first place. Particularly in the early days of computing, resources like bandwidth or processing power were scarce.

If one is to distribute computation – for example, an algorithm for calculating taxes owed – there are many questions

that must be answered. Are integers big-endian or little-endian? Are we using an ASCII character set? How does System A acknowledge receipt of System B’s request? These questions are answered by creating a protocol. Protocols are standardized descriptions of resources and processes designed to maximize interoperability.

This section describes the protocols that have been proposed over the years for doing distributed computing. For each protocol, the original descriptions of the protocol were used to reconstruct the high-level goals the particular protocol was intended to address.

### C. Early years

The development of the ARPAnet in the late 1960s was the start of the networking age. A few years later, local- and wide-area networking standards, such as IBM’s SNA and Ethernet, made it possible for co-located machines to communicate. Operating systems had earlier developed the notion of inter-process communication, where threads could exchange data, and such notions paved the way for distributed computing.

The initial impetus for a distributed computing protocol can be traced to such programs as FTP and Telnet. Telnet allowed users to run programs on another system [7]. In 1975, recognizing that a number of these ARPA-based processes duplicated functionality, James White proposed the *remote procedure call* [8]. What White argued for was the ability for machines to do the same thing as humans did with Telnet. He critiqued Telnet, and related programs, for missing out on higher-order functionality:

“The syntax and semantics of these interchanges, however, vary from one system to another and are unregulated by the protocol; the user and server processes simply shuttle characters between the human user and the target system [8, p.3].”

Before his proposal, machine-to-machine communication had to be handled low-level, with the communication functions being written anew.

White listed eight requirements for a replacement protocol:

- Allow arbitrary named commands to be invoked on the remote process;
- Report on the outcome of such an invocation;
- Permit arbitrary numbers of parameters;

- Represent a variety of types;
- Allow for the process to return results;
- Eliminate the user/server distinction;
- Allow for concurrent execution;
- Suppress any response if asked.

White's list of requirements give some insight into the problems he was trying to solve. For example, he wanted to remove any constraints on remote method invocation, to make remote procedures 'feel' like local calls. His proposed type system was designed to deal with heterogeneity, so that remote systems would be able to talk to a local machine in the same syntax. He also makes mention of the notions of synchronicity and blocking, i.e., what to do with long-running or complex calls. Finally, his proposal emphasizes the need for a "simple, rigidly specified command language". This goal would eliminate the profusion of heterogeneous programming languages which he saw as an obstacle to further progress in distributed computing. All of these requirements would come to inform distributed computing protocols for the next 15 years.

#### D. RPC - Remote procedure call

The first substantial work on creating an implementable standard for distributed computing was that of Birrell and Nelson [9] in 1984, working on behalf of Xerox. The Xerox proposal drew on White's work with the following different emphases. Like White, they highlighted the requirements that the proposal make distributed resource sharing indistinguishable from local method calls, down to emulating program semantics - i.e., "the semantics of remote procedure calls should be as close as possible to those of local (single-machine) procedure calls [9, p. 43]". They focused on 'clean and simple semantics'. Complex semantics would make debugging more difficult, and development tools like IDEs and distributed debuggers were in their infancy. Efficiency and latency were also concerns, given the slower network speeds of the day. Finally, the proposal emphasizes generality in the interface, allowing, like White, arbitrary commands and parameters. The goal was to allow the protocol to be as flexible as necessary.

Language constructs dictated elements of the distributed computing paradigm: the use of the Mesa language - a modular, imperative language - suggested the use of a remote procedure call paradigm over a message-passing one, as noted in the protocol. As well, exception signals were generated that would appear indistinguishable in Mesa from local exceptions. The architecture chosen for the RPC proposal was that of 'stubs and skeletons'. A remote procedure would generate a stub for the local program to call. The RPC layer would handle procedure calls to the stub, marshaling and serializing them over the network, where the skeleton would hand off to the remote procedure. The local stub served as the interface description, a contract the local caller could rely on. The stub and skeleton mechanism is fairly rigid in terms of evolvability; for any improvement on the remote site, an entire new stub needs to be generated locally.

1) *ONC*: Following the Birrell and Nelson paper in 1984, several software companies began to develop RPC implementations. The software industry was undergoing a tremendous

growth period [10]. Two standards enjoyed primacy. These were DCE, discussed in the section following, and Open Network Connectivity [11], proposed by Sun Microsystems in 1988, and the basis for their still-popular Network File Sharing (NFS). While following most of the requirements listed in [9], Sun aimed for transport independence, so programs could be reused to some extent. They explicitly added state to the remote procedure by adding a transaction ID that served to identify each call uniquely. Overhead was also important, so the protocol ignores reliability, allowing the transport to handle that. There was also a mention of authentication, perhaps the first realization that distributed procedures created several security risks. Finally, the specification gave a detailed definition of call semantics - such as at-most-once, at-least-once (execute the client's invocation at most once), and idempotent (no changes no matter how many calls).

2) *DCE*: While Sun's ONC was gaining traction, other vendors and institutions, such as Cambridge, HP, and MIT, developed RPC implementations according to their interpretation of the issues. Eventually this gave rise to a standardization process, under the aegis of the Open Software Foundation, called DCE, the Distributed Computing Environment [12], in 1991. The larger context was the work of Sun and AT&T on Unix, which threatened to dominate the efforts of the other vendors.

Like earlier efforts, DCE was designed with heterogeneous, distributed networks in mind. Unlike Sun's effort, DCE was explicitly designed to work across multiple operating systems. DCE was more than just RPC, however. In order to compete with NFS, the DCE proposal included a distributed file system, time server, and concurrent programming support. New design requirements were added as well. For the first time, a distributed computing standard recognized the notion of service-orientation, an architectural style that separates business rules from application logic. DCE provided rudimentary support for this by recognizing that a procedure's invoker may not necessarily be identical to its consumer. DCE also explicitly supported bindings for different languages, a recognition of the organizational structure of OSF.

#### E. CORBA

Perhaps one of the most influential distributed computing protocols, or rather, suite of protocols, was CORBA. CORBA is an object-based distributed computing specification. Rather than remote procedures, self-contained components are transferred by middleware mediators called Object Request Brokers. A product of an industry standards group called the Object Management Group (OMG; also responsible for UML), the CORBA specification was first proposed in 1991, but this standard release is generally thought to be of poor quality; for example, it did not define how to communicate between different object brokers [13]. By late 1996, major inconsistencies were cleared up and the second version released [14], [15].

CORBA was an enormous suite of standards with widespread industry involvement. A key goal, common to RPC technologies as well, was the separation of interface from implementation. Although such encapsulation was done in

RPC as well, the wider context was the rising use of the object-oriented programming paradigm [10]. Key to this concept was the Interface Definition Language (IDL). The IDL provides language-independent declarations that are then mapped into a programming language to create the object stubs. Clients invoked only those operations that a remote object exposed through its IDL interface. A second innovation was the introduction of multiple distributed systems. The ORB one used could also communicate with other ORBs through a protocol called the Internet InterORB Protocol (IIOP). IIOP defines the protocol by which messages are passed. Finally, CORBA made reference to scaling requirements, e.g., load balancing. This is one of the first mentions of this issue in distributed computing.

#### F. Java: RMI and EJBs

In late 1995 Sun Microsystems released Java 1.0. The Java platform soon nurtured a large community of developers. The initial support for distributed computing in the Java platform came from the bundled `java.rmi` library. RMI stands for Remote Method Invocation, and as the name suggests, is an RPC-like remote object protocol [16]. The real power of RMI came from the way it tightly integrated the distributed object model into the Java programming language. The other important features of RMI, other than its excellent integration, are its simplicity and efficiency.

By 1998 developers of complex distributed systems were running into difficulty. As applications grew in size, the facilities in existing architectures, like CORBA or RMI, seemed inadequate for handling complexity. Sun's solution was the release of the Enterprise JavaBeans (EJB) platform [17]. This release heralded the development of application servers, where a central server delivers applications to client machines. The central server contains the application and business logic, which addresses (some of) the issues of complexity. Contemporaneously, Sun and Oracle were developing the notion of the network computer.

The EJB specification emphasized three benefits:

- an underlying framework, initially using CORBA middleware, that abstracted issues related to transport;
- separation of business logic from application logic, such as user-facing interfaces;
- portability and reusability from a component model.

These were not new ideas, but again, were well-integrated for a large, pre-existing market. By this point, as listed in the specification, Sun was beginning to see the value in 'web services'.

#### G. Microsoft: DCOM

Microsoft's competing technology for CORBA, Java, and RPC was an extension of the DCE-RPC standard they called DCOM, for Distributed Component Object Model. Like Java, DCOM saw widespread adoption due to its tight integration with the large installed base of Microsoft operating systems and server tools. The DCOM specs [18], [19] mention several design concerns, including the importance of garbage collection and memory management; security via access control

lists; ease of deployment; and the ability to version remote objects: "With COM and DCOM, clients can dynamically query the functionality of the component."

The 1990s ended with a focus on the notion of SOA, or service-oriented architecture. This was the idea of separating business logic from application logic, and establishing service contracts via interface definitions. SOA systems were loosely coupled and focused on the notion of reusability and encapsulation. The next decade, up until the present day, has seen this mindset gain primacy.

#### H. The web years

In 1990, Tim Berners-Lee proposed the World Wide Web [20], based on what would become Internet standards: HTTP for interactions [21] and URIs for resource identification [22]. Distributed computing technologies leveraged these new opportunities – for example, that all computers soon had a Web browser, that HTTP-based servers were ubiquitous – to develop new protocols and technologies. A related standard, XML [23], allowed for a ubiquitous and extensible format for data exchange. "What distinguishes the modern Web from other middleware is the way in which it uses HTTP as a network-based Application Programming Interface (API) [24, §6.5.1]." This allowed protocols to abstract away the network requirements and focus on the semantics of the distributed services.

1) *Web services:* As XML was being standardized, there was ongoing work on SOAP, the Simple Object Access Protocol. Due to delays, and in the interests of creating a usable, light-weight RPC mechanism free from the influence of Microsoft, XML-RPC [25] was released as a specification. It used XML, which permitted introspection, was extremely simple (2 pages), and quickly gained support. The data representations chosen were fairly primitive, however, and better support for data types was needed.

The SOAP standard [26] (Simple Object Access Protocol) was driven by Microsoft as a way to address the growing influence of Sun's EJB activities. Up to that point, Microsoft's distributed computing solutions (DCOM) were Microsoft-specific. SOAP was a means to communicate with any other heterogeneous computing platforms, as long as that platform could parse XML. One of SOAP's design goals was flexibility. As one of the designers mentions, "we looked at the existing serialization formats ... and RPC protocols ... and tried to hit the sweet spot that would satisfy the 80% case elegantly but could be bent to adapt to the remaining 20% case."<sup>1</sup> To this end, a number of features common to, for example, DCOM, were omitted, such as garbage collection. To define interfaces – what endpoints expect to receive as a message, and can return as a result, WSDL was introduced [27]. SOAP also took advantage of existing work on XML datatypes, in the form of XML Schema.

The design goal for SOAP was to be fairly neutral architecturally. SOAP can be used in a number of different distributed computing architectures, such as message passing and remote

<sup>1</sup><http://webservices.xml.com/pub/a/ws/2001/04/04/soap.html>

procedure calls: “SOAP is fundamentally a stateless, one-way message exchange paradigm [26].”

To define some more complex features on top of SOAP, such as transaction support, reliability, security, and so on, the ongoing WS-\* process [28] is defining more standards. Web services are SOA-style interactions over the web, usually using SOAP and WSDL.

One challenge for web services is discovery and chaining. Semantic Web Services [29] are a suite of ontology-based tools that aim to add machine-interpretable semantics to services. The goal is to automatically perform service composition using “generic, high-level procedures”. To date this remains largely a research activity.

2) *REST*: An alternate approach to SOAP’s message passing paradigm is REST, REpresentational State Transfer. Defined in 2000 by Roy Fielding [24], REST is “a small set of simple and ubiquitous interfaces to all participating software agents. Only generic semantics are encoded at the interfaces. The interfaces should be universally available for all providers and consumers.” The idea is to move away from encoding the interface in an object, and use the fairly universal HTTP actions: PUT/GET/DELETE/POST, among others. This supports generality, and has the benefit that the effects of each action are universally understood; e.g., GET is an idempotent operation (according to the HTTP specification). The complexity in the interaction is handled by the resource state: “[unlike SOA], for a client to invoke a REST service, it must understand only that service’s specific data contract: the interface contract is uniform for all services.” [30]. Fielding’s thesis mentions several design goals for REST, induced by its stateless nature. These are visibility: the entire nature of a request (invocation) is embedded in the message; scalability on the server, as all state is client-side; and reliability, because partial failures can be dealt with.

There are concerns about security, but the advantage to REST – and its fundamental difference with the other technologies we’ve looked at – is that it completely abjures the interface contract and transport mechanisms. These are managed by the underlying protocol, typically HTTP. Security, for example, can be handled by the HTTP-Authentication standards. REST’s only interest is in the transfer of application state.

With REST, we conclude our survey of distributed computing technologies. We have shown how, for each technology, the designers focused on particular requirements in order to address challenges they identified. In the following section, we present our interpretation of this narrative as it relates to the wider goals of managing software evolution.

### *I. Interpretation*

This section takes the specific details from each of these distributed computing technologies and ties them together in a coherent narrative. What is common to all such distributed computing models is the primary distributed computing requirements set, identified in *REC707* and unchanged since. This is because the underlying challenges and opportunities remain similar. The ability to access remote services and to

abstract implementation details are compelling. Dealing with challenges of heterogeneity and message format remain.

What we see, then, is not major revisions in this feature set – all technologies, for example, abstract implementation details of the remote service – but incremental updates as new problems with existing implementations are identified. And as the context changes – for example, as the World Wide Web, HTTP, and URLs gain massive and widespread acceptance – the protocols evolve to leverage that context. With Birrell and Nelson’s work [9], the concept of remote procedures was shown to be feasible. However, once implemented, certain challenges arose. These included security, scaling, and transparency. With the rise of OOP, and the industry dominance of certain players such as Microsoft and Sun, pressure grew for smaller vendors to embrace cross-vendor standards to compete. These forces led to the development of CORBA, DCOM, and RMI. The limits to these technologies became more apparent as systems grew in size. That by the mid-1990s most servers used common transport standards, namely HTTP, and common interchange format (XML), supported the development of XML-based standards like SOAP and XML-RPC. Today we are witnessing the evolutionary competition of the web services standards with the goals of REST, that scale, generality and abstraction are best accomplished by focusing solely on resources identified by URIs. Time will tell which set of technologies is best able to satisfy the fairly consistent high-level goals of distributed computing.

Some of the lessons suggested by this study for designing for software evolvability (in the large):

- 1) Vendor lock-in is a blessing and a curse. In the case of DCOM, tight integration with the dominant platform of the day made development easier. At the same time, extending applications beyond that platform was essentially impossible. Open standards processes can lead to ‘design-by-committee’ syndrome, but can also greatly increase adoption.
- 2) Successful protocols work best by allowing the developer to focus on what matters to their application. Such generality and encapsulation become more and more successful as the underlying technologies, such as HTTP, are themselves standardized and propagated.
- 3) Service orientation, the separation of business logic and application code, and the ability to separate invoker from consumer are important mechanisms for handling application complexity and promoting reusability.
- 4) Treating remote resources as though they were local is misleading. There are certain properties of local resources, such as latency, that are very difficult to ignore. One of CORBA’s problems was its insistence on this principle. Quality of service properties are important to consider. Web services specifications, such as WS-ReliableMessaging, are attempts to address this.

What does the future hold? Based on this historical study some educated guesses suggest themselves. For one, the notion of distributed computing is so useful that it will likely have a growing influence on software engineering. Even today, many users access remote services for everyday computing tasks,

such as checking email. With mobile computing playing a larger and larger role, this will likely expand. Evolvability in distributed computing protocols will continue the process of stepping away from lower-level details like transport and security, focusing instead on separation of concerns and sound application design.

#### IV. CONCLUSIONS

As this workshop's website states, "the ability to evolve software over time to meet the changing needs of its stakeholders is one of the principal challenges currently facing software engineering". In this light, what did the study teach us? What does the evolution of distributed computing protocols have to say about the practices of software engineering? This paper aimed to "deepen insight into the types of activities, methods and tools required, identify and determine those likely to be beneficial, when and how they should be used and how they relate to one another [1, p.35]." Conducting a longer-term analysis on the evolution of design requirements for distributed computing has provided insight into all of this.

In the future, we would like to work towards a comprehensive theory of requirements evolution. Our preliminary work has focused on the notion of context and constraints, taking cues from Cognitive Work Analysis [31] and work on design evolution [32]. As this paper has demonstrated, considering the wider context for design decisions is essential in answering 'why' questions - and for designing better products in the future.

#### V. ACKNOWLEDGEMENTS

We would like to thank Ashleigh Androssoff for her insights into the discipline of history.

#### REFERENCES

- [1] M. M. Lehman and J. F. Ramil, "Software evolution - background, theory, practice." *Inf. Process. Lett.*, vol. 88, no. 1-2, pp. 33-44, 2003.
- [2] M. S. Mahoney, "Finding a history for software engineering." *IEEE Annals of the History of Computing*, vol. 26, no. 1, pp. 8-19, 2004.
- [3] A. I. Anton and C. Potts, "Functional paleontology: system evolution as the user sees it." in *International Conference on Software Engineering*, Toronto, Canada, 2001, pp. 421-430.
- [4] H. Gall, M. Jazayeri, R. Klsch, and G. Trausmuth, "Software evolution observations based on product release history." in *International Conference on Software Maintenance*, Bari, Italy, October 1-3 1997, pp. 160-166.
- [5] J. B. Spira, "20 years-one standard: The story of TCP/IP." *Iterations: An Inter-disciplinary Journal of Software History*, vol. 2, pp. 1-3, April 2003.
- [6] J. Dixon and J. W. Alexander, *The Thomson Nelson Guide to Writing in History*. Scarborough, ON: Thomson Nelson, 2006.
- [7] J. Postel and J. Reynolds, "TELNET protocol specification," Internet Engineering Task Force, Tech. Rep. RFC854, 1983. [Online]. Available: <http://tools.ietf.org/html/rfc854>
- [8] J. E. White, "A high-level framework for network-based resource sharing," Internet Engineering Task Force, Tech. Rep. RFC707, March 1975. [Online]. Available: <http://tools.ietf.org/html/rfc707>
- [9] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39-59, 1984.
- [10] W. E. Steinmueller, "The U.S. software industry : an analysis and interpretative history." Maastricht Economic Research Institute on Innovation and Technology, Maastricht : MERIT, Research Memoranda, 1995. [Online]. Available: <http://ideas.repec.org/p/dgr/umamer/1995006.html>
- [11] Sun Microsystems, Inc., "RPC: Remote procedure call." Internet Engineering Task Force, Proposal RFC1050, April 1988. [Online]. Available: <http://www.ietf.org/rfc/rfc1050>
- [12] B. C. Johnson, "A distributed computing environment framework: An OSF perspective," The Open Software Foundation, Tech. Rep. DEV-DCE-TP6-1, June 1991. [Online]. Available: <http://www.opengroup.org/dce/info/papers/>
- [13] D. Chappell, "The trouble with CORBA," *Object News*, May 1998.
- [14] J. Boldt, "The common object request broker: Architecture and specification, v. 2.0," Object Management Group, Specification formal/97-02-25, Jul. 1996. [Online]. Available: <http://www.omg.org/cgi-bin/doc?formal/97-02-25>
- [15] S. Vinoski, "CORBA: Integrating diverse applications within distributed heterogeneous environments," *IEEE Communications Magazine*, vol. 35, no. 2, February 1997.
- [16] Sun Microsystems, Inc., "Java remote method invocation," Sun Microsystems, Inc., Palo Alto, CA, Specification, 1999. [Online]. Available: <http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmiTOC.html>
- [17] V. Matena and M. Hapner, "Enterprise JavaBeans specification, v1.1," Sun Microsystems, Inc., Palo Alto, CA, Tech. Rep. EJB1\_1spec, 1999. [Online]. Available: [http://sdic-esd.sun.com/ESD4/JSCDL/ejb/1.1-fr/ejb1\\_1-spec.pdf](http://sdic-esd.sun.com/ESD4/JSCDL/ejb/1.1-fr/ejb1_1-spec.pdf)
- [18] Microsoft Corporation, "DCOM technical overview," Microsoft Corporation, Redmond, WA, DCOM Technical Article, November 1996. [Online]. Available: [http://msdn2.microsoft.com/en-us/library/ms809340\(d=printer\).aspx](http://msdn2.microsoft.com/en-us/library/ms809340(d=printer).aspx)
- [19] M. Horstmann and M. Kirtland, "DCOM architecture," Microsoft Corporation, Redmond, WA, DCOM Technical Article, 1997. [Online]. Available: [http://msdn2.microsoft.com/en-us/library/ms809311\(d=printer\).aspx](http://msdn2.microsoft.com/en-us/library/ms809311(d=printer).aspx)
- [20] T. Berners-Lee and R. Cailliau, "WorldWideWeb: Proposal for a hypertext project," CERN, Proposal, 1990. [Online]. Available: <http://www.w3.org/Proposal>
- [21] T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext transfer protocol - HTTP/1.0," Internet Engineering Task Force, Informational memo RFC1945, May 1996. [Online]. Available: <http://tools.ietf.org/html/rfc1945>
- [22] T. Berners-Lee, "Universal resource identifiers in WWW," Internet Engineering Task Force, Informational memo RFC1630, June 1994. [Online]. Available: <http://tools.ietf.org/html/rfc1630>
- [23] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, "Extensible markup language (XML) 1.0," World Wide Web Consortium, W3C Recommendation REC-xml-19980210, 1998. [Online]. Available: <http://www.w3.org/TR/1998/REC-xml-19980210>
- [24] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.
- [25] D. Winer, "XML/RPC specification," Userland Software, Tech. Rep., 1999. [Online]. Available: <http://www.xmlrpc.com/spec>
- [26] D. Box, D. Ehnebuske, G. Kakiyaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple object access protocol (SOAP) 1.1," World Wide Web Consortium, W3C Note NOTE-SOAP-20000508, May 2000. [Online]. Available: <http://www.w3.org/TR/soap11/>
- [27] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web services description language (WSDL) 1.1," World Wide Web Consortium, W3C Note, March 2001. [Online]. Available: <http://www.w3.org/TR/wsdl>
- [28] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, "Web services architecture," World Wide Web Consortium, W3C Note NOTE-ws-arch-20040211, February 2004. [Online]. Available: <http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/>
- [29] S. A. McIlraith, T. C. Son, and H. Zeng, "Semantic web services," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 46-53, 2001.
- [30] S. Vinoski, "REST eye for the SOA guy," *IEEE Internet Computing*, vol. 11, no. 1, pp. 82-84, 2007.
- [31] K. J. Vicente, *Cognitive Work Analysis: Toward Safe, Productive, and Healthy Computer-Based Work*. New Jersey: Lawrence Erlbaum Associates, Apr. 1999, published: Paperback. [Online]. Available: <http://www.amazon.de/exec/obidos/ASIN/0805823972>
- [32] S. Brand, *How Buildings Learn: What Happens After They're Built*. Penguin, 1995.