SMPL: A Network Architecture
for Collaborative Distributed Services

Carlos A. Rocha
Bachelor of Science in Mechanical Engineering
Universidad de los Andes, September 2002

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of
Master of Science in Media Arts and Sciences at the
Massachusetts Institute of Technology

September 2005

Carlos A. Rocha
Program in Media Arts and Sciences

John Maeda
Associate Professor of Design and Computation
E. Rudge and Nancy Allen Professor of Media Arts and Sciences
MIT Media Laboratory

Dr. Andrew B. Lippman
Program in Media Arts and Sciences
SMPL: A Network Architecture
for Collaborative Distributed Services

by

Carlos A. Rocha

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on August 5, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science in Media Arts and Sciences

Abstract

This thesis proposes a network architecture, called SMPL, for the design and
development of collaboration-oriented, distributed applications over the Internet. The
goal of SMPL is to enable the development of applications that easily integrate the
capabilities of different types of computing resources, software platforms, and data
repositories across the Internet transcending the level of a single device. SMPL
proposes a new abstraction of the Internet as a network composed of services,
resources, and capabilities instead of just machines. The SMPL architecture
distributes resources through a peer-to-peer network of service providers. The design
of SMPL encourages developers to add value to the system by facilitating the creation
of new functionalities based upon compositions of the existing ones.

Thesis Supervisor: John Maeda
Associate Professor of Design and Computation
E. Rudge and Nancy Allen Professor of Media Arts and Sciences
SMPL: A Network Architecture for Collaborative Distributed Services

Carlos A. Rocha

Chris Csikszentmihályi
Muriel Cooper Assistant Professor of Media Arts and Sciences
MIT Media Laboratory
SMPL: A Network Architecture
for Collaborative Distributed Services

Carlos A. Rocha

David P. Reed
Adjunct Professor of Media Arts and Sciences
MIT Media Laboratory
Acknowledgements

This thesis is the product of many collaborations.

First and foremost, I would like to thank my advisor, John Maeda, for his support, intellectual generosity and rigor, and his mentorship. What I take away from this experience is unique and fruitful, had it not been for your generous insight and encouragement. I have learned from your eagerness, energy, and passion, to pursue my endeavors and aspirations without boundaries, and with precision, beauty, and creativity.

My interactions with my thesis readers were inspirational and invaluable. Chris Csikszentmihályi, your knowledgeable comments were invaluable, your endless patience and encouragement has helped me through to the very end. Thank you. David Reed, I am grateful for your input and mentorship.

Thank you John, Chris, and David for helping me realize the opportunity to reflect, improvise, enjoy, and believe.

I have had the privilege to work with past and present PLWers. My sincere and special thanks for your friendship, inspiration, and support. Kelly, Burak, Noah, Mark, Patrick, James Dai, James Seo, Annie, Amber, and Isha, and all the present and past junior varsity, you have inspired my work, you made our collective efforts of staying up through the night and around the clock, enjoyable and worthwhile. Special thanks to the people at hcg@media — I would not have reached the end, had it not been for you guys.

I have also made many friends from around the globe here at the lab. Particularly Ben, thanks for being a champ, really. Claudia, Jose, Arnaud, Hugo, Max, Kelly, Larissa, Carla, Fernanda thanks for caring and sharing your views of the world.

Linda Peterson and Pat Solakoff deserve recognition for their support until the very last second as I hand in this thesis.

Hyun, there are simply not enough words. You were there, through both happy and hard times. I will be there for you too.

Lastly but not least, I want to thank my father and my brother for their unconditional love and support. I wouldn’t be here, if it weren’t for you both. Mom, you made all this possible.

We must move onto new adventures and risks, especially when things are sweet.
A la memoria de mamá.
Contents

1 Introduction 17
  1.1 Motivation ............................................. 19
  1.2 SMPL Overview ........................................ 21
    1.2.1 Contributions .................................... 23
  1.3 Document Overview ................................. 24

2 Background 27
  2.1 Collaborative Distributed Systems .................. 27
    2.1.1 Creating Opportunities ........................... 28
    2.1.2 New Computational Models ....................... 32
    2.1.3 Types of Collaboration ........................... 34
  2.2 Related Technologies ............................. 38
    2.2.1 Network Topologies ............................... 38
    2.2.2 Distributed Architectures ....................... 39
  2.3 Related Projects .................................... 43
    2.3.1 Straum and Hive .................................. 43
    2.3.2 MetaGlue and HyperGlue ......................... 44
    2.3.3 JINI .............................................. 45
    2.3.4 UPnP ............................................. 46
  2.4 Challenges .......................................... 46
  2.5 Summary ............................................. 48

3 Design and Architecture 49
  3.1 Design Principles .................................... 50
  3.2 Architecture .......................................... 53
    3.2.1 Topology ......................................... 54
    3.2.2 Elements ......................................... 57
3.3 Implementation .............................................. 59
  3.3.1 Dispatcher ............................................. 59
  3.3.2 Service Libraries ..................................... 60
  3.3.3 Client Libraries ....................................... 61
3.4 Summary .................................................... 62

4 Scenarios and Analysis ........................................... 63
  4.1 Examples ................................................. 63
    4.1.1 SMPL RFID Reader ................................. 63
    4.1.2 SMPL TCP/IP Speaker ......................... 65
    4.1.3 SMPL Multi-Mouse Service ................. 65
    4.1.4 SMPL Flexible Screen ....................... 67
    4.1.5 SMPL Rendering Service .................... 68
  4.2 Comparison and Analysis ................................. 71
    4.2.1 JINI ............................................. 71
    4.2.2 CORBA ........................................ 73
    4.2.3 Web Services and UPnP ..................... 74
  4.3 Conclusions .............................................. 75
    4.3.1 Strengths ....................................... 75
    4.3.2 Weaknesses ..................................... 76
    4.3.3 Summary ....................................... 76

5 Conclusions and Future Work ................................. 79
  5.1 Conclusions ........................................... 79
  5.2 Lessons ............................................... 80
  5.3 Future Work ........................................... 82
  5.4 Future Directions .................................... 83
# List of Figures

1-1  Example of a mobile phone deconstructed as a group of services. 22

2-1  Google Maps. ............................................ 37
2-2  GeoBloggers. ............................................. 37
2-3  The Croquet project graphical user interface. ............ 37
2-4  Paul Baran’s network topologies. .......................... 38
2-5  Nelson Minar’s network topologies. ......................... 39

3-1  The SMPL network architecture topology. .................. 54
3-2  The SMPL network and associated subnets. ................. 55
3-3  The SMPL network and physical containers. ............... 57

4-1  RFID reader using SMPL technology. ....................... 64
4-2  SMPL audio speaker. ...................................... 65
4-3  SMPL network mouse. .................................... 66
4-4  The SMPL Flexible screen. ................................. 67
4-5  The SMPL Rendering iMac Cluster. ......................... 68
4-6  The SMPL Rendering architecture. ........................ 69
4-7  Treehouse Studio. ......................................... 70
4-8  Handheld device using SMPL Rendering. ................... 70
4-9  Comparison between SMPL and related technologies. .... 71
4-10 WSDL interface for a stock quote service. ............... 77
Chapter 1

Introduction

The widespread adoption of computer networks and devices during the last ten years has helped shape an emerging vision about the future of computing [82]. In this vision, all the computer devices that we use in our daily lives will be able to cooperate with each other. Each device on the planet will be part of a massive network, feeding information of all kinds into a single decentralized system. Some visionaries claim that a new kind of network will be born; one that will combine the processing power of billions of computers [28], and will be enhanced with the special capabilities of all kinds of computing devices.

Even bolder visionaries talk about a future where the convergence of networks of machines, sensors, and actuators will form one unique gigantic machine. Such a machine could possess the conditions necessary for the emergence of a new kind of intelligence. The sensors on the network will create a planetary “sensitive skin” [47], able to “spy on everything from the environment to our highways and bodies” [12]. Some futurists claim that this planet-size computer will surpass the complexity of the human brain [39, 40], “spontaneous computer networks to emerge”, forming a huge “digital creature” [34], or a human-like artificial intelligence [43].

All of these visions, as many others in the past, claim an unparalleled impact on our lives. Such claims are commonly motivated by diverse factors: the lack of scientific skepticism, either the ego or naïveté of their
creators [44], the interest of investors or the sensationalism of the media. And this time is no exception. Beyond the hype and the speculation there is a hard-to-hide fact: soon there will be more computers than people on this planet.

During the next few years, we can expect over 16 billion internet-enabled microprocessors [36] to be embedded into everything from refrigerators, to cars, to weather stations, to watches, and to the clothes we wear. By 2010, three billion internet-enabled cell phones will join the vast diversity of services available on the Internet, which already contains a great part of our recent written knowledge and resources. To accommodate such scales, some important changes will have to be made to the way we design our computing and communication infrastructures. Some are already on the way — the new version of the Internet Protocol, IPv6, can provide 340 undecillion \((3.4 \times 10^{38})\) unique addresses, or over 670 quadrillion \((6.7 \times 10^{17})\) addresses for every square meter of the Earth’s surface.

The right infrastructure could in fact lead to a new kind of network — one in which every node will provide access to not only static information, but also to dynamic processes and abilities such as: sensing, analyzing, and controlling the physical world. Under the appropriate conditions, every new node connected to the network will allow “collateral benefits” [58], enabling new opportunities for collaboration among themselves. All kinds of software, hardware, and data will be able to complement each other, and collectively perform functions beyond the capabilities of any of them alone. As they become part of a whole, the boundaries between them may start to blur. Each part will be enhanced by the functionalities of other parts without being limited by their physical location. Dave Stutz summarizes this idea as, “software above the level of a single device” [72]. The impact of a network of this unprecedented scale is hard to anticipate. However, it is fair to say that it will allow a richer environment for innovation — unexpected new ideas and applications may be developed exploiting a larger set of diverse functionalities and interconnections.
1.1 Motivation

The tendency to integrate computing and communication into our daily lives has led to the convergence of fields as diverse as Distributed Computing, Computer Networking, Software Engineering and Human-Computer Interaction. This convergence is best described by Mark Weiser as, “Ubiquitous Computing.” For Weiser, computation should integrate into the environment, shifting from a “personal computing era, person and machine staring uneasily at each other across the desktop” to a new “age of calm technology, when technology recedes into the background of our lives.” [87] Weiser’s idea is further refined by Michael Dertouzos’ project Oxygen, which believes that computation in the future will be freely available, “like oxygen in the air we breathe” [22]. For Dertouzos, the important change is not that computers will be everywhere, but that they will be able to leverage each other’s capabilities. Larger devices will benefit from the portability of smaller, mobile devices, while smaller devices will be able to use the greater computational power of their larger counterparts. More importantly, this symbiosis brings a new element into play: context. The sensing modalities in one device will be available to others, providing ways for a machine to be “aware” of its location, any nearby devices, and even the identity of the person who is using it. Context-aware devices will be able to adapt and personalize themselves to our presence.

Another refinement of Weiser’s vision comes from the field of Human-Computer Interfaces. Terry Winograd describes it [90] as an evolution “towards a different architecture, focused on multiple users in an interaction space rather than focusing on systems as a network of processors and devices.” The consideration of the human factor is an important contribution to this vision. Computers and Networking provide a remarkable platform for human-to-human collaboration. The success of the World Wide Web is a living proof of this platform.

Projects such as Wikipedia and examples of open source software have already shown the possibility of massive direct collaboration over the Internet. The recent increase in the influence of weblogs over public
opinion may be a consequence of a whole new network of ideas that flow back and forth, generating new discussions, making new connections, leaving trails of knowledge, preferences and affinities at every step. In the words of Tim O’Reilly [58]:

“In some ways, you can say that what Internet is enabling is not just networking of computers, but networking of people, with all that implies. As the network becomes more ubiquitous, it becomes clearer and clearer that who it connects is as important as what it connects.”

Furthermore, these technologies are transforming the way in which we collaborate. By connecting to and taking an active role in the way we interact with them, they open a space for new and exciting capabilities. Technologies such as Collaborative Filtering (Flickr), Recommender Systems (Amazon), Online Auction Sites (eBay), and Smart Search Engines (Google), showcase a variety of methods in which the users are augmenting the value of the network by simply using them. The Internet is no longer just a network of machines. It embeds everyday, networks of people. It has become, “a web of clusters of people and things” [14].

As these transformations unfold, we are left with many questions. What kind of system will allow us to make better use of this prospect? What abstractions are necessary to understand and develop these systems? What architectures will allow the development of applications in this larger environment? How can we create a reliable environment out of such diverse range of components? How can we foster participation? What implications will this have in our lives?

To use the Internet more effectively in solving complex problems, we need advanced collaborative systems. We need better tools that foster, engage, and support large-scale collaboration among different groups of people — bootstrapping “our collective capabilities to develop, integrate and apply knowledge” [23]. Such tools should allow groups of users to operate seamlessly and simultaneously from different platforms, creating, sharing, and manipulating different types of media and information.
1.2 SMPL Overview

As a first step, we can start by constructing a compelling Collaborative Distributed System [59]. We propose a new network architecture called SMPL that improves the collaborative experiences between people using the Internet. The goal of the design of SMPL is to lower the entry barrier in the creation of active processes on the Internet. Instead of looking at this new Internet as a large parallel computer, SMPL looks at it as a collective of functionalities and resources that can complement and extend each other. In the SMPL architecture, the network nodes do not represent individual machines. Instead they represent the individual functionalities and capabilities of the hardware, software and data components connected to the network. This facilitates the creation of new functionalities by compositing existing ones.

The underlying intent with SMPL is to promote connectivity at all levels. In this context, connectivity is an innovation catalyst that allows new and unexpected solutions to rise by mixing, compositing, and grouping the components already present in the system. The notion of connectivity used in this thesis framework is appropriately defined by Bob Frankston: “Connectivity is about enabling opportunities for new kinds of relationships between devices as well as people.” [29]

To achieve this goal, we create a service abstraction within the context of the SMPL architecture, which captures the fundamental or more useful capabilities of a computing device either in software form or hardware form. These services are best described as modular functionalities that can be remotely invoked across a network, such as the Internet. SMPL then provides a common framework in which these functionalities can be used and interconnected, allowing the creation of new capabilities by the aggregation or remixing the existing ones. SMPL allows computing devices to search, query and use the functionalities of all other devices connected to the system, without being bound to their physical containers.

SMPL places a strong emphasis on the capabilities of its components. It allows a wide range of interoperability scales, ranging from networks of
simple electronic devices to Internet-scaled distributed projects. SMPL is designed as an architecture that encourages participation by providing developers with mechanisms that allow the easy integration of new services across different platforms. Finally, the development of SMPL will fill a gap that will allow the integration of semantic functionalities into the Internet itself [15, 49] — it will facilitate the automatic composition of services [50, 51, 26] and functions [24]. From a generous point of view, SMPL can be viewed as a first step towards an “Internet Operating System” [57] made of “small pieces loosely joined” [86].

In general, SMPL provides a unified framework for the easy development of distributed software, in which ideally every newly added application increases the value of the system. SMPL facilitates the integration of multiple computing artifacts at the software and hardware level. This integration promotes connectivity and the enabling of new opportunities to develop solutions.

Consider a mobile phone. We can identify just a few of the discrete functional components in its hardware: a microphone, a speaker, a keypad, and possibly a display screen, and a camera. We can also identify distinctive functionalities in its software (i.e. audio encoding) and in the data structures it can manipulate (i.e. phonebook). Instead of being represented as a whole, the telephone joins the network as a group of services, each one representing one of those individual functionalities (see Figure 1-1). Additionally, it is even possible for the telephone to present a service of itself, representing the combined functionalities of all the
sub-services from which it is made. The telephone is deconstructed — it stops being a single functional entity and instead becomes a collection of nodes or services. This requires some principles, that guide the extensibility and purpose of the service deconstruction. We will discuss this topic and provide detailed examples in the remaining chapters of this document.

Finally, the design of SMPL tries to be flexible enough to encourage the participation of many different platforms and computing environments. The SMPL implementation presented in this document uses common networking protocols such as TCP/IP and HTTP to accomplish its goals. Nevertheless it also leaves room for using other technologies that are more appropriate to the capabilities of a particular device, application, or network conditions. New protocols can be added as new services, providing an easy way to integrate emerging technologies in the architecture. In fact, most of the key components of the SMPL architecture, such as discovery, search and administration are also implemented as services.

1.2.1 Contributions

The contribution of this thesis can be summarized as a group of organizing principles, a network architecture and a software implementation that allows the creation, discovery and composition of network services across the Internet. Specifically, the following points outline the principal contributions:

- **SMPL** is a network architecture made of three basic elements: connections, dispatchers, and services. The architecture topology is a hybrid decentralized-centralized model, formed by a peer-to-peer network of service dispatchers. These dispatchers perform the roles of both service providers and directories.

- A *network service* abstraction captures the functionalities of different types of hardware, software and data artifacts. SMPL provides a way to describe each service, as well as an ontological structure that allows the search of particular services across the network.
• A software implementation of the SMPL architecture provides scalable service dispatchers that support multiple concurrent connections. Each dispatcher is capable of both synchronous and asynchronous communications between the associated services. The service dispatcher model also provides a mechanism for supporting multiple application protocols, enabling SMPL services to be called from different software and hardware configurations.

• A client library facilitates the development of applications that use SMPL services. The same library can be used to develop independent SMPL services.

• A group of applications and scenarios has been implemented using SMPL. These examples provide a series of arguments that allow a comparison between SMPL and a group of similar technologies and projects.

1.3 Document Overview

Chapter 2 provides a brief introduction to Collaborative Distributed Systems, contextualizing SMPL in the appearance of recent developments in collaborative applications at the Internet level. It continues with a brief introduction of key technical concepts necessary to frame SMPL in a larger context and compares it to related projects. Finally, it summarizes a series of closely related projects.

Chapter 3 starts with the fundamental principles behind the SMPL architecture, followed by a description of the design, its elements, and design rationale. The chapter finishes with an overview of the SMPL architecture implementation showcasing the most relevant mechanisms that make SMPL work.

Chapter 4 describes a series of projects implemented at the Physical Language Workshop research group that make use of the SMPL architecture. Finally, the chapter outlines the lessons learned from the development process of the examples, and makes a comparison of SMPL against related projects.
To finalize, Chapter 5 concludes the thesis document with recommendations of future changes, improvements, and implications of building Collaborative Distributed Systems.
2.1 Collaborative Distributed Systems

The Internet has evolved tremendously from its infancy as an academic research project. One of the most popular Internet applications, the World Wide Web, has reached a scale with consequences completely unexpected by its creators [86]. The Web is a good example of a Collaborative Distributed System — a system that coordinates multiple computers to support the collaborative efforts of groups of people. The development of such systems has brought together the knowledge and expertise of several research disciplines. Distributed Computation is concerned with the methods by which multiple machines can perform large computations in an orchestrated manner [75, 1], while the field of Computer Networks [74] focuses on the methods of communication among two or more computers [5]. As we will discuss below, the synergy between these two fields has generated a number of new computational models that have drawn the attention of the computer science research community during the last ten years. Finally, the field of Human-Computer Interaction [91] has brought the recognition of human factors [9, 55] in the design of distributed systems.
2.1.1 Creating Opportunities

As discussed previously, one of the salient benefits of a distributed system is the removal of the physical boundaries between machines. The widespread adoption of network-ready devices capable of numerous input and output modalities (cameras, displays, microphones, sensors, actuators, etc), allows the creation of new kinds of distributed systems that bring the collaborative platform closer to the physical world. The scale and variety that are formed by these systems create larger opportunities for innovation. Simply put, they provide the developers of distributed systems with a larger set of components and more importantly, a bigger number of ways in which they can be interconnected and combined. Historically there have being three important factors that have motivated technological innovation during the last twenty years:

1. Moore’s Law.
2. The ubiquity of networking technologies.
3. Architectures of participation.

The presence of these factors has promoted the creation of new computational models [67] and has sustained the development of a multitude of collaborative applications \(^1\) that have had an impact on the way people conduct their business, communicate to their families and friends, and consume media and information. Likewise, they have also sparked new research fields and initiatives such as: sensor networks [8], paintable computers [18], smart dust [83], and viral communications [45], among many others.

Moore’s Law

Gordon E. Moore’s observation regarding the doubling of the number of transistors on integrated circuits every eighteen months, has become the most common ways to explain the rapid progress and diversification of computational technologies during the last thirty years. Its implications

\(^1\)For example: email, Usenet, IRC, the World Wide Web, video-conferencing.
are not limited to computer processing power. They also extend to storage capacity (both hard drive and memory), power consumption, physical size, and manufacturing costs. Nowadays, a small microcontroller costing just a few dollars is just as capable as a desktop computer from 1985. This makes it possible for embedding computation into all kinds of objects at a very low cost.

**Networking Technologies**

Since the 70s, the use of large-scale networks has become common in our daily lives. The most prevalent examples are the cellular phone network and the Internet. If Moore’s law made computation cheaper and more affordable, it is the advancement in network technologies that opens up the exciting possibilities: the power to exploit the synergy afforded by billions of interconnected devices — thousands of devices per person on the planet. Not only will devices be everywhere in all sizes and forms, they will also be able to communicate with each other.

There are two relevant research efforts in this area. The first is the development of lightweight networking technologies, appropriate for low-end microprocessors. Projects such as Internet 0 [42], developed at MIT, aims to provide simple yet powerful technologies for “networking for very large and distributed systems – everything from a distributed sensor data-gathering network to the light switches and light bulbs of a house”. One of the goals of Internet 0 is to eliminate the requirement of central servers allowing devices to form a self-organizing network, and remove some of the computational requirements associated with Ethernet technologies. In contrast, the Ninja Project [33] at U.C. Berkeley proposes a different approach: an architecture to allow devices to communicate at an Internet scale. Instead of forcing devices to accommodate a communication standard, the Ninja architecture presents the use of intermediate active proxies\(^2\) which can transform data types, adapt protocols, or filter information.

The second effort is in Wireless Networking, which enables applications

\(^2\)Examples of active proxies include desktop computers, wireless hubs, network gateways, and custom transformational proxies
the use of two powerful characteristics: mobility and physical proximity. When combined, they could allow applications to better accommodate our lifestyles. At the same time, they reinforce the issue of context in a distributed system. Wireless networking provides better possibilities to mediate the creation of device networks in an ad-hoc, local, and self-organizing manner. Traditionally most of the research in this area has been concerned with increasing the range and bandwidth of the wireless connection. Equally important is the recent effort in the development of embedded networks [62] represented by the emergence of communication standards like ZigBee [6]. Such technologies are aimed at applications with low data rates and low-power consumption, enabling wireless networking in a variety of low-end microprocessors.

It is worth mentioning that there are also a number of interesting proposals for the redesign of the Internet infrastructure, such as the work of Braden et al. [17], which aims to identify and improve over its current limitations. Other related advances are based on the experimentation with new networking paradigms, such as Active Networks [77] proposed by Tennenhouse and Wetherall.

Architectures of Participation

The term “Architectures of Participation” was first mentioned in an essay by Tim O’Reilly titled “Open Source Paradigm Shift” [57]. It describes the “nature of systems that are designed for user contribution.” The importance of such systems is that they add a third element to a distributed system, namely, people. If networking allows the cooperation between machines, it is the design around architectures of participation that makes a distributed system truly collaborative by facilitating the cooperation among humans, and between humans and machines.

A system developed around an architecture of participation provides an incentive that encourages a potential user to join or participate in the system. Usually this also means that the system provides a low-entry barrier that makes it easy to add a component into the system. As a result, the tendency of such systems is to grow, as the incentive they offer
is more valuable as more users join. Such phenomena are called the network effects and are the subject of study of a field called applied network theory [84, 85, 12]. This field involves not only the study of computer networks, but also of networks made of very different components: human relationships, transportation systems, nerve cells, and electrical grids. The increase in the value of a network is often illustrated with the following example: a single fax machine is useless, but its value increases with the total number of fax machines, because the total number of people with whom you may send and receive documents increases [5]. The value of such network is said to increase to the square of the number of elements in the network. However, if we consider the diversity among the elements in the network, the value of the network is said to increase exponentially, because of the possible number of sub-groups that can be created among the network participants. This observation is known as Reed’s Law [63]. We use Reed’s Law to justify the effort to include a greater number of computing platforms, services and devices under the same system, allowing for an exponentially increasing number of possible combinations in which these elements can be grouped to innovate or generate new functionalities.

In the context of distributed systems, there are many different mechanisms that allow for the creation of architectures of participation. One of them is the use of standards, which involves the agreement on the communication protocols, interfaces, and data types shared among the elements of a system. An example of a standard is the HTTP protocol and the HTML document specification used in the World Wide Web application. A second mechanism is the provision guidelines that directly affect the architecture resolution. These guidelines usually refer to principles that allow the creation of modular architectures in which different components can be added, replaced, and combined easily. An example of this can be seen in the use of pipes in the UNIX operating system [41], which encourage UNIX programs to be written as small modules designed to cooperate with each other by reading and writing.

---

3 This observation is known as Mecalf’s Law, because it is attributed to Robert Metcalfe, creator of ethernet.

4 It is worth noticing that a recent paper by Odlyzko et al. [56] argues that both Metcalfe and Reed’s law suffer from overestimation, justifying that both arguments are erroneous since they assume that all the connections in the network are equally valuable.
ASCII streams. A final mechanism is through the definition of higher-level design principles. Rather than solving the details of the architecture problem, it provides more general structural parameters. A well-known example is the End-to-End argument [68], which guides the placement of functionality within a system to a level closer to the application that requires it. This principle has served as one of the central design principles of the Internet Protocol (IP): the transport layer is kept free of higher-level functionalities such as encryption, security, and identification, among others. The growth of the Internet has proven the value of the end-to-end principle in the design of systems that are open and foster innovation.

2.1.2 New Computational Models

The three factors discussed in the previous section have played an important role in the commoditization of computing technologies. During the 1990s, computers and networking became common in all sorts of human activities. Nowadays it is possible to find a personal computer in most homes, and mobile phones in many pockets. As mentioned previously, this has opened up many possibilities. In the field of computer science, it allowed for the experimentation with computational models that in the past were unfeasible or too expensive to construct. The formulation of some of these models is not entirely new. However, it is the new possibility to experiment with them that has made them the subject of intense studying and research during the last ten years. Some of these models [67] are: “massive data set computations”\(^5\), “processors with multiple instruction units”\(^6\) and “peer-to-peer networks”. It is the latter model that has recently drawn the attention, for reasons we will discuss below.

---

\(^5\) A massive data set refers to a data set that is considerably bigger than the memory of the computer processing it. Such data has become increasingly common due to the pervasiveness of interconnected devices that can capture great amounts of all kinds of data.

\(^6\) The field of multi-processor systems have been studied for a longer time than the other ones mentioned here, however, it is the scale at which these systems can be build today that has open new paths of exploration.
Peer-to-Peer Networking

The term Peer-to-Peer (P2P) refers to networks in which the nodes are equivalent and that no nodes have a greater degree of control over the others. Recently the term has taken to a more broader definition: it is commonly used to refer to networks in which there is little evidence of a central controlling authority, yet a mechanism is provided that allows the nodes to cooperate in the execution of a distributed application. Another common characteristic of these kind of networks is that it is possible for its nodes to dynamically join or leave the network or group without significant impact on the system[67]. This is a very desirable quality since to a certain degree it makes the network resilient to the failures of its individual components. Another characteristic is that each node usually only has information about its neighbors, and it is unaware of the total size of the network. In contrast to the classical client-server network model, in a P2P architecture each node acts as both a client and a server — it consumes as well as it provides.

One of the first successful P2P Internet applications is Usenet. Usenet is to a great degree a distributed data storage application. The content of the Usenet newsgroups is stored among many servers across the Internet in a replicated form. To replicate the data, each server communicates only a set of its “neighbors” and synchronizes its content by passing back and forth information about the addition or deletion of Usenet messages. The recent interest in P2P technology is commonly associated with another distributed storage application. The demise of Napster exposed the fundamental vulnerability of centralized systems: when the center of command fails, the whole system collapses. The interest in creating less vulnerable systems led to the development of genuine P2P applications such as Gnutella, Kazaa and Freenet. Computers scientist have developed P2P systems aimed at understanding and improving the efficiency, scalability, and fault-tolerance of the networks. One of the most popular advances in this area is the development of an improved mechanism to find and store content named Distributed Hash Tables (DHT) [11]. Examples of DHT can be found in systems such a Chord [71], Pastry [66] and Kademlia [48]. Another popular application of P2P networks is
collaborative distribution of content. Applications like BitTorrent make it possible to distribute large amounts of information at a lower cost, by spreading the bandwidth among the group of clients instead of placing the entire load to the server. There has also been interest in developing platforms that facilitate the creation of P2P applications, such as Sun Microsystems’ JXTA [32], and the Peers project, in development at the MIT Media Laboratory. Finally, there has also been some progress towards creating a platform for testing large peer-to-peer applications under real conditions. PlanetLab [61] is such system: a test-bed for distributed applications consisting of over 500 machines distributed around 25 countries.

The importance of P2P applications in the context of SMPL is that they offer us a good example of an architecture of participation, as well as valuable lessons in the design of resistant, dynamic, and collaborative networks.

2.1.3 Types of Collaboration

In this section, we will describe a list of axes which will help us to frame the role of SMPL in the context of collaborative systems. These axes serve to establish different dimensions in which collaboration can vary inside a distributed system. Before we proceed, it is important that we define the term *collaboration*. Usually, collaboration is understood as a joint effort by a group of people\(^7\) to achieve a common goal. For the purpose of this thesis, collaboration also refers to the participation of a person in a distributed system resulting in the enrichment of such system.

**Closed - Open**

Closed collaboration relates to the interaction around shared data that can only be edited or changed by its owner, but can normally be read and accessed by all the participants in the system. In contrast, open collaboration means that the data (or its source) is not only available, but it can also be changed and edited by anyone. An example of a closed

\(^7\)Or machines.
systems were the first online reference systems, such as the encyclopedia Britannica, which could edited exclusively by its developers. The appearance of Wikis, bulletin boards and dynamic WebPages allowed content to be changed by its users.

An interesting fact is that the WWW was designed from its conception to behave almost as a Wiki [14]. The original Web Browser created by Tim Berners-Lee had an edit function that allowed one to edit the content of the page that was viewed at the given moment. This functionality was not included by other browser vendors, and eventually became obsolete. The same idea was later rescued by the creation of Wikis. The biggest Wiki effort to date is Wikipedia [5], an online encyclopedia that allows anyone to add new content or to modify any existing page.

Another important open architecture of participation is the Open Source software movement, responsible for very popular software products such as Linux and the Apache Web Server.

Direct - Indirect

Direct collaboration implies systems by which the cooperation between participants relates directly to the goal of the system. This occurs in most of the collaborative systems previously mentioned, as well as in weblogs. A good example of indirect collaboration can be found in Google, which uses the hyper-links found in WebPages to index, value and sort the content of the Internet. When the creator of a WebPage adds a hyper-link, he is indirectly informing Google about the perceived value of the destination of the link. Other examples include collaborative filtering services, such as Flickr\(^8\), and del.icio.us\(^9\), and automatic recommender systems such as the one used by Amazon. Finally, indirect systems can also be used for document retrieval, as proposed by Fields [27].

\(^{8}\)http://www.flickr.com
\(^{9}\)http://del.icio.us
Active - Passive

In addition to direct collaboration, active collaboration requires the participant to continuously contribute information into the system. The participant is aware at all times that he is contributing. Generally, in passive systems, the participant adds value to the system by simply being part of it. An example of such system is BitTorrent, which redistributes the bandwidth cost of downloading a file between the people downloading it, instead of putting it all on the server hosting the data. The difference between indirect and active collaboration is that the former relies on external mechanisms and processes to filter the participation, while later, the benefit from participating is a property inherent in the system.

Synchronous - Asynchronous

Synchronous applies to collaboration systems that work in real or almost real time. Examples of such systems are instant messaging, video conferencing and voice over IP. Correspondingly, asynchronous systems work in longer time intervals, and do not depend on timed interactions — as observed in email, Wikis, and online forums.

Static - Dynamic

Static collaboration means that the collaboration takes places mostly around fixed pieces of information. On the other hand, dynamic collaboration implies the sharing of processes, which often take the form of queries, requests or application programming interfaces (APIs). Examples of static systems include email, the File Transfer Protocol (FTP), Napster, and static WWW pages.

Examples of dynamic collaboration include relational databases, distributed computation projects like SETI@Home\textsuperscript{10}, dynamic WebPages, and Web Services technologies, which will be discussed later in this chapter. Google Maps\textsuperscript{11} is one recent example of what dynamic services can do. Google Maps offers an API that developers can use to query an

\textsuperscript{10}\url{http://setiathome.ssl.berkeley.edu/}
\textsuperscript{11}\url{http://maps.google.com}
online version of the maps, and seamlessly merge the results in combination with other geo-location based services. After the API was made public, amateur developers have created a diverse range of projects. The web site GeoBloggers\textsuperscript{12} mixes the maps with the location-based information embedded in some of the photos at Flickr, a digital-photo sharing web site. Similarly, the web site IncidentLog\textsuperscript{13} presents a real-time map of emergencies happening in major U.S. cities.

The most advanced collaborative distributed architectures posses characteristics of all different types as we just described. For example: the Croquet Project \cite{croquet} focuses on providing software and an architecture platform to support \textit{synchronous} multi-user collaborations inside a shared three-dimensional virtual environment. Croquet makes extensive use of the Squeak programming system; it makes the creation of virtual \textit{dynamic} objects possible, which can be shared across many machines connected to

\textsuperscript{12}http://geobloggers.com
\textsuperscript{13}http://incidentlog.com
the system. The synchronization and communication between objects is
done by an advanced subsystem named TeaTime. Croquet aims towards
active and direct collaboration, providing the equivalent of a broadband
conferencing system built on top of a three dimensional user interface (see
Figure 2-3) and a peer-to-peer network architecture.

2.2 Related Technologies

In this section we will introduce a series of technical concepts, ideas and
definitions that will lay the fundation for a discussion about the design
and implementation of SMPL during this thesis document.

2.2.1 Network Topologies

Network topology refers to the set of patterns in which a network can be
organized. They are an important consideration in the design of
distributed systems, since each topology has different characteristics and
unique advantages and disadvantages.

One of the first classifications of such topologies dates back to 1964, in
the work of Paul Baran at the Rand Corporation. Baran is generally
recognized as one of the conceivers of the Internet by the invention of
digital package switching, one of the key technologies behind our modern
computer networks. While working at Rand, Baran was assigned to the
design of a telecommunication scheme that could survive a nuclear
attack. In one of his reports [13], Baran describes three possible network configurations: centralized, decentralized and distributed. For Baran, a distributed network was one that presented a mesh like structure. Baran followed the classification by an assertion of the vulnerability of centralized networks, and argued for the design of distributed networks. This observation had a profound impact on how the Internet was designed.

![Network Topologies](image)

Figure 2-5: Nelson Minar's network topologies.

Years later, the term *distributed* is commonly used in reference to any system in which certain processes or applications involve more than one machine. Nelson Minar recently made an informal but broader classification of distributed systems topologies[53]. What Minar calls a *centralized + decentralized* topology is particularly relevant to SMPL. In such topology most of the nodes in the network have a centralized relationship to a “super node”. In turn, the super nodes connect to other super nodes in a decentralized way. Minar notes that this kind of topology presents great scalability, is fault tolerant and has a partially coherent structure. Finally, it is noted that this kind of topology is difficult to manage and presents the same security problems of other decentralized systems.

### 2.2.2 Distributed Architectures

The interest in distributed systems has led to the establishment of a group of distributed applications that can be classified according to their
primary use. Below we will mention the salient types and will provide a short description of the abstractions and mechanisms they employ. This sets a precedence that we will use later as a comparison point between existing distributed architectures and SMPL.

**Computation Oriented Architectures**

These architectures allow the use of processing resources of many separate computers connected by a network to solve large-scale computational problems. This type of distributed computation is generally considered to be a part of the general *parallel processing* problem. This refers to the division, adaptation and execution of the same computational problem on multiple processors in order to obtain faster results. They were originally formulated after the appearance of computers equipped with multiple processors; it was later extended to multiple computers after the establishment of network technologies.

With the rise of the Internet, it was then possible to use the computing power of millions of interconnected computers to work on problems that were previously intractable because of the cost of dedicated supercomputers. The SETI@Home and Folding@Home\textsuperscript{14} projects are good examples of computational oriented distributed architectures that work over the Internet. They operate by distributing chunks of data from a central database across many personal computers. Once each chunk analyzed, the results are sent back to project servers.

The main protocols used in distributed computing are MPI (Message Passing Interface) and RPC (Remote Procedure Call). MPI is a library of routines that allow different parallel processing techniques to be executed in various programming languages. It is very common in high performance computing environments. As the name indicates, a RPC allows a computer program running on one computer to initiate the execution of a routine hosted on a different machine.

A RPC is initiated by a client by sending a request message (which includes the name and arguments of the request) to a server which

\textsuperscript{14}http://folding.stanford.edu/
triggers the execution of a certain procedure. Once the server finishes processing the request, it sends a message to the client with the results. Common RPC protocols use XML or binary formats. XML-RPC is a simple RPC encoded in XML which later gave origin to a more complex format called SOAP, which is part of the Web Service standards. XML formats usually have higher bandwidth and processing power requirements since they are verbose and require complex interpreters. In contrast, binary protocols are designed to be compact and easy to parse, but not human-readable. An example of a binary RPC format is the Hessian protocol. Hessian is also well suited to send pure binary data (such as images) without using attachments.

**Storage Oriented Architectures**

These architectures allow the storing of data on a network by its distribution across many machines. Distributed storage systems can either be based on client-server or peer-to-peer architecture. In the client-server case, the data is stored on a server to be accessed by clients. Examples of this type of systems are: NFS (Network File System), AFS (Andrew File System), and SMB (Server Message Block).

Peer-to-peer storage systems such as Gnutella, Freenet, and CFS (Cooperative File System) provide massive storage capabilities without the need of centralized servers. They commonly operate by dividing data into blocks and storing them on a number of peer network nodes. In some cases, the users can reciprocate and allow other user to use their computer as a storage node.

**Object Oriented Architectures**

Distributed objects are software modules that work together but reside in different computers. These modules are called objects, and follow the same definition from object-oriented programming (OOP). Computer programs in OOP are made of self-contained objects composed of variables and related methods instead of just procedures. An object also has an associated state, which changes according to the interactions of
other objects or procedures. In distributed objects architecture, objects are instantiated by the client and for most purposes look like normal local objects. However, their functionality and state is maintained at a remote host. The mechanisms that make this possible are usually very complex. The main distributed objects platforms are CORBA, created by OMG, DCOM (Distributed Component Object Model) by Microsoft, and the Java RMI (Remote Method Invocation) by Sun Microsystems. A new emergent competitor is ICE (Internet Communication Engine) [35] developed by ZeroC, a company founded by prior developers of CORBA technologies. ICE promises a simpler and more functional platform for the development of distributed objects.

These architectures also make use of an Interface Description Languages (IDL): a platform-independent machine-readable description of the methods available for each object. The IDL is used to create platform specific representations of each distributed objects by providing a concise mapping between standard data types, and the data types specific of each platform.

**Service Oriented Architectures**

Like the objects in distributed object architectures, services in service-oriented architectures are also software modules that can perform a group or related operations. However, one of the distinctive differences is that unlike objects, services have no notion of state. Another difference is that services are not instantiated on the client. Instead they offer a connection, which the client can use to pass service requests. This allows the service architecture to be significantly simpler than the object-oriented architectures, comprising loosely joined and more interoperable applications. It is worth noticing that the request messages send from clients to the server differ from standard RPC [80]. Service requests are *document-messages* that are *processed* by the corresponding service.

The most popular Service Oriented Architecture is the Web Services initiative, moderated by the World Wide Web Consorsium (W3C). The
initiative defines a whole group of protocols that allow the transport and interpretation of messages, as well as the description and discovery of services. All of the protocols specified by the W3C use an XML format. Web Services have recently taken the interest of many business applications. Despite a significant adoption by various vendors [78] they have not achieved the ubiquitous status promulgated during their initial hype. To this date, Distributed Object Architectures offer a more reliable platform, especially in terms of the reliability and maturity necessary in some corporate applications.

A smaller set of technologies associated with service discovery are Zeroconf by Apple, and the Service Location Protocol, an IETF RFC. Both allow clients to query and find all services connected to their corresponding local networks. SLP also allows services to present specific attributes — in the case of a SLP printer, the attributes can be the formats supported, its location or if it can print in color or not.

### 2.3 Related Projects

In this section we will briefly describe a series of projects that relate to SMPL. As we just saw, the body of related technologies can be quite large. For each technology, we may be able to find several projects that showcase a special collaborative application or improve over a certain limitation. For the purposes of the analysis and comparison of the SMPL architecture that will be made in chapter 4, we selected just a few closely related initiatives. The criterion of selection is the relevance of the projects in their ability to integrate functionalities across different kind of devices connected in a large-scale network.

#### 2.3.1 Straum and Hive

Proposed by Nelson Minar [52, 54], Straum, and its successor Hive, model decentralized applications as ecologies of distributed software agents. Implemented in Java, the systems were built on the principles of autonomous, proactive software agents that seek to discover available
services and collaborate with other agents to achieve specific tasks. Hive proposed a combination of shadows, representing local resources, running trusted code, and agents that fulfill dynamic, networked roles. The approach addressed the balance between interdependent, slow synchronous communication and lossy, difficult to scale asynchronous message passing.

During Hives development, small applications were implemented to test the system. Music player and kitchen helper examples using several agents on a few machines demonstrated a flexible infrastructure with self monitoring to detect and reset agents, and ease of resource discovery to adapt to the addition of new hardware. Scaling organisation and mobility in large distributed systems proved challenging. The project offered the promise of large networks of task oriented agents and a toolkit of accessible hardware resources, that could be built into new applications: “Hive allows anyone to connect those agents together and build new distributed applications. From the experience of the Web, we know the power of such synergy.”

2.3.2 MetaGlue and HyperGlue

Active architectures allow the transfer of executable code between different computers or devices on the network. Systems like MetaGlue and HyperGlue [19, 60] allow software agents to move freely between machines in the context of intelligent environments. Metaglue is an extension to the Java programming language that addressed many of the needs for distributed, parallel computing in Intelligent Environments. It provides the computational glue necessary to connect distributed modular systems. By extending the Java language with an Agent class, Metaglue gives great flexibility for people who want to implement agents. Metaglue is able to change an agent’s attributes even when it is running, allowing for real time interaction. In addition, those agents can specify what requirements are needed before they can run. For example, an agent can specify which computer it must run on, what type of hardware it needs, and what amount of memory is necessary. If the agent does not start on a machine that meets its requirements, Metaglue moves it to
another machine on the network that can meet its requirements. Agents maintain and store state by saving and retrieving their fields from the internal Metaglue database.

Hyperglue is an extension of Metaglue for use in larger networks. While Metaglue was used for smaller networks, consisting of the various intelligent components of the Intelligent Room, Hyperglue was made to integrate multiple rooms, dealing with the scalability problems of Metaglue. Hyperglue also focused on maintaining privacy, and personal preference in the rooms, which were not accommodated for in the communication protocol that Metaglue used. Rather than using a centralized catalog, which made separating room and agent specific information difficult, Hyperglue uses local and remote resource managers, which are capable of contacting each other for resources and allocating resources. These resource managers can also decide which resources to make available to others at any given time.

2.3.3 JINI

Suns Microsystems JINI Technology [10] is designed to provide highly adaptable distributed services oriented to the needs of electronic devices. JINI uses a common run-time environment, the Java Virtual Machine. Emphasis is placed on run-time organization and adaptation, with portable code and service-centric philosophy. The approach aims to provide lookup services for dynamic information on resource availability within a “federation” of autonomous agents. Hive shares many of the features of JINI, including RMI distributed objects and mobile code. The need for temporary agent collaboration is maintained by the concept of leasing.

Applications using JINI have demonstrated flexibility to removal and addition of functional components to environments such as mobile phones, without destabilizing other elements of the system. Challenges include self-configuration, inter-operability and standardization.
2.3.4 UPnP

Industrial standards like UPNP [7] provide a set of rules that foment the connectivity among stand-alone devices and personal computers. The goal of this initiative is to allow different consumer electronic devices to be controlled from a centralized hub, or media centers. A standard method for describing and monitoring device capabilities is established. UpnP addresses challenges of device announcement and discovery, and automatic naming/addressing and network configuration. It is low-level, device-centric standard facing challenges of competing standards of name resolution and resource discovery, and corporate maneuvering.

2.4 Challenges

“Progress imposes not only new possibilities for the future but new restrictions.”
Norbert Wiener [88]

To clarify the differences between a local and a distributed application, we have to consider the primary challenges faced by each one. Those differences are often overlooked, especially by architectures that try to hide the distributed nature of the application by providing abstractions that make the system look as one monolithic giant local system. Waldo et al. [81] expose this elegantly — the environment of distributed applications is very different from the one of a local system, and because of that, the differences should be taken seriously and not removed from the development process. The following are some considerations that concern the design of distributed systems.

**Scalability** indicates the capability of a system to perform its function efficiently as more participants join the network. A scalable system is that which capacity increases as more participants join in.

**Concurrency** is the name given to the problem that arises when many participants in a system try to access the same resources simultaneously. It is a very common problem in a distributed environment, and is usually solved by different techniques such as scheduling, queuing, and time-sharing.
Limited Resources implies that the different parts of a distributed architecture, such as protocols and data types, take account the capabilities of the machines that participate in the system. Some of these considerations are from processing speed, memory, and latency.

Composition relates to the development of mechanisms that allow a participant in a distributed system to use the capabilities offered by other participants, composing a new functionality. This also involves the creation of dependencies, an important consideration when stability and flexibility are the primary objectives of a distributed design.

Discovery and Search refers to the problem that a participant faces when looking for a desired capability in the system. It involves the mechanisms to perform the search, as well as the methodologies in which the participants should describe themselves to others.

Firewalls/NAT showcase one of the many small technical hurdles that distributed systems face when they are deployed in real-world networks such as the Internet. The problem that firewalls and NATs cause is that they make it impossible (or at least very hard) for an outsider to establish a connection with a participant that is inside a Local Area Network. This prevents many peer-to-peer applications from running properly.

Relevance signifies the struggle of any emerging technologies to keep up with its own promises. Many technologies claim simplicity and ease of use, only to become heavy and bloated after becoming mainstream.

This list can be further extended to include some of the problems that both local and distributed systems face: reliability, security management, trust, complexity, and deployment. The future challenge of distributed systems is to be autonomous and automated; in the meantime, we can attempt to provide tools that tackle the inherent complex nature of a distributed system.
2.5 Summary

This chapter presented an introduction to the field of distributed systems, and their evolution as a collaborative platform. We presented three major factors that shaped the evolution of the field, as well as introduced key components and ideas that shaped the development of SMPL. Finally, we enumerated a series of related projects, as well as the challenges they face. This will serve us to compare and analyze SMPL in the following chapters.
Chapter 3

Design and Architecture

As processors get faster and smaller, and networking technologies become more massive, we are presented with new opportunities for building more diverse and larger distributed systems. However, the problem that arises in integrating all the diverse platforms, devices, protocols and specifications that grow out of such an environment. This thesis proposes SMPL, a network architecture that provides a unified framework for the easy development of collaborative distributed applications that integrate the capabilities of all the different hardware, software and data components connected to a network.

The underlying goal of SMPL is to promote connectivity as a catalyst for innovation. Connectivity allows new and unexpected applications to be created by combining and compositing different functionalities. SMPL is a collaborative system by allowing, on a first stage, the collaboration among developers in the creation of active processes (called services) that can be found and reused by being connected to the network. At a second level, SMPL will allow the creation of better collaborative applications aimed at a more general audiences, enabling opportunities for new kinds of relationships between devices and people.

To achieve these goals, the SMPL architecture provides various elements: first a series of design principles that allow the deconstruction of existing software and hardware artifacts into smaller functional pieces that can be composed together to form new functionalities. Second, SMPL defines a
network topology made of a small set of components (services, connections and dispatchers) and mechanisms aimed to integrate and leverage different computer platforms. The final element is a software implementation that allows a gradual exposure to the architectural elements. SMPL provides a simple core functionality and the mechanisms to extend it. This translates into a best-effort approach for service delivery and helps to improve the connectivity between platforms with higher capabilities to leverage the expanded capabilities it can use from the SMPL platform. In this chapter we will present all these elements, their design rationale and a discussion of their implications.

As a final note, it is worth mentioning that SMPL does not try to address all the difficulties present in the design of distributed systems. At its best, SMPL provides an aggregated first-step approach to most of the problems that have already been solved in more mature and heterogeneous distributed systems platforms. Many issues such as security, failure-tolerance, versioning, and transaction reliability are not dealt with directly in the design of SMPL. However, the design provides the extensibility necessary to address those issues in the future.

### 3.1 Design Principles

The SMPL architecture proposes a methodology for extending the current model of distributed systems (Networks of Computers) to a revised model that matches more closely with the SMPL design goals (Networks of Services). SMPL looks at the network as a collective of functionalities and resources that can complement and extend each other. In the SMPL architecture, the network nodes do not represent individual machines. Instead they represent the individual functionalities and capabilities of the hardware, software and data components connected to the network. As discussed, SMPL is a collaborative platform as it provides the facilities to allow the creation of new functionalities by compositing existing functional nodes exposed to the network. To achieve this transformation, we define a two step procedure: functionality deconstruction followed by functionality composition.
**Functionality deconstruction** Functionality deconstruction refers to the identification and breakdown of the functionalities exposed to a user by a specific piece of hardware, software or data. In SMPL the minimum unit of decomposition is a function of related capabilities that should be exposed to be used efficiently by another module. These minimum units are called *services*. We will have a more detailed look at what services mean for the SMPL architecture later in this chapter. As a quick introduction, we can informally define a service as anything that can be done as a result of a message sent from one piece of software to another, which may or may not involve human direction. Moreover, a service is anything that starts as software, but the result doesn’t necessarily stay in software\(^1\). The granularity of the deconstruction should be guided by a set of criteria defining how services can be integrated or composed with others.

**Functionality composition** Functionality composition is the ability to create new functionalities by the composition, chaining, or integration of already existing functionalities represented by services. To facilitate their composition, services should present a way to be recognized and identified. Such identification is done in two steps. The first one is by *categorization*, which determines the family of functionalities to which the service belongs. The second step is the *definition of attributes*, which determines the specific abilities of services according to the category where they belong. For example, in the case of a networked speaker, the speaker would be of the category ”audio display,” whereas the attributes would refer to the kind of audio files that could be displayed or other technical parameters. For composition to work well, services should be able to be discovered in the network.

As a result, the computer is deconstructed and transformed from a monolithic object to a collection of exposed functionalities glued together by the network. In SMPL, services do not explicitly care about their origin, they are not bound to a particular physical object or context. This

\(^1\)As is the case of services that result in an interaction with the physical world such as mechanical actuators, printers, or even human intervention. A service for example may trigger a request ticket, which instead of being attended by a machine is attended by a human.
enables applications that use SMPL to be above the level of a single
device, getting “a large part of their value from software that resides
elsewhere.” [57].

The two principles discussed above require the design of services that can
be reused by being interconnected with other services. This premise
allows us to define a few guidelines about the granularity of the
deconstruction which determines the size or scope of a service. In general
terms, the delimitation of the scope of a service follows the same criteria
established in most software engineering practices, specially in Object
Oriented Programming. Note that there are significant differences
between an object and a service, as we will mention later in this chapter.
An object has data structures (state) and is associated with a thing; a
service can be as thought of as more of an action.

In the process of deconstruction, a question can arise regarding the extent
of granularity to decompose towards. In other words, how finely can you
subdivide a service into subservices until it no longer becomes
meaningful. The scope of a service is what minimally defines the set of
self-contained related functions that operate over a certain type of data.
For instance, in the case of a file storage system, the scope would define
the limitations of reading/writing data. This is complemented by the
need of a categorization that allows the composition of services. As we
will discuss later this categorization is structured by an ontology, that
defines a service as an action that operates over certain type of data. This
ontology currently spans media and actions over media (e.g. display,
input, conversion). Finally, as a rule of thumb, it is good to consider that,
at least in the context of SMPL, services as designed to be used by other
developers should provide just the right set of functions that allow
simple, easy access to the functions that the service provides, promoting
its reusability in different scenarios.

As an example, a video projector, a computer monitor, or the canvas in a
software program can all expose a similar functionality by means of a
DisplayImage service. This service provides just the right level of
granularity to allow all the devices to be used equally by external
applications. The only difference between the three services, is that each
one may have different service attributes than the others (i.e. different supported image formats, different size and resolution). These attributes can be read by clients or other services. This is important, since it allows the search of services based on some required criteria: for example, an image processing application can search and use a `ImageFormatConvert` service to read an unknown image format.

Function deconstruction and function composition are fundamental concepts within SMPL. Combined, they facilitate the development of new functionalities which, at the same time, contribute to the overall value of the system. The purpose of these elements is to support the concepts and abstractions necessary for developers and end users to understand, create and use a richer next generation of Internet applications.

### 3.2 Architecture

SMPL tries to achieve the goal of a highly diverse group of devices and services at the architectural level. It provides an accessible, clear and powerful infrastructure that allow an application to discover and connect to services in network. This matches the current trend of high-end computation research as moving from one single core computer to multiple machines that allow the powerful exploitation of parallelism.

A key design goal for SMPL is scalability of services and devices from a small network scale to an Internet-wide scale. Usually this class of scalability is achieved by current approaches to peer-to-peer networks which demand that all nodes maintain multiple connections and therefore a proportionally high computational demand. Instead, in the SMPL architecture, we have a peer-to-peer network of intermediaries, called dispatchers, realized by machines that are more powerful as they are able to maintain concurrency and multiple connections as a small part of their overall respective computational loads. We know that due to Moore’s law, we can foresee that smaller nodes will become more powerful and become powerful enough to support SMPL-class dispatcher services. As computers become increasingly faster, there will be always a bottom
layer: cheap low-power devices which will be produced in bigger numbers. What SMPL enables, is to constantly reach that bottom layer, and leverage their mobility and higher quantities, with the processing power of machines capable of supporting dispatchers.

The SMPL architecture is defined by the following elements:

1. Services: Active processes that don’t necessarily terminate.

2. Connections: Mechanisms that allow communication between services and dispatchers. It is how applications talk to the overall SMPL infrastructure.

3. Dispatchers: Service providers and directories that keep track of where services are, and are executed.

### 3.2.1 Topology

![SMPL Network Architecture Topology](image)

**Figure 3-1: The SMPL network architecture topology.**

The SMPL Architectural elements can be grouped together in a hybrid
centralized-decentralized network topology as seen in Figure 3-1. Services follow the more traditional server/client pattern, while the dispatchers form a peer-to-peer network. This topology can be used to describe different distributed computing environments, such as clusters, intelligent rooms, and sensor networks.

Networks can be formed as isolated subnets (See Figure 3-2, that allow for secure arrangements, where certain services and dispatchers can be hidden from a bigger network by an intermediate filtering dispatcher. As mentioned before, there are no restrictions on where an element should be located. A group of services and a dispatcher can all exist inside an individual computer; they can be distributed around a smart building, or connected across greater distances thought the Internet. The same architectural elements can be used in all cases.
Composition

The ability to integrate services depends upon ways to search for the kind of services that are available. An application or service communicates with the dispatcher to achieve one of the following goals:

1. **discovery** What services are available?

2. **categorization** What type of services are they?

3. **finer query** What kind of attributes exist?

All of these services that are discovered are never completely dependable, thus they are designed to always be loosely coupled. Thus the design for how these services interconnect, must be sensitive to the design of error handling. In this manner, a system with potentially incompatible computational resources can potentially connect and succeed in a best case scenario. In a worst case scenario, failure would not result in a complete system failure. A common example in a programming language would be exception handling.

De-localization

In the SMPL architecture, the topology is always exposed. This de-abstraction is an important factor in making a robust system of networked services. By making the structure more explicit, a developer can more gracefully plan for unforeseen situations. Thus although a network of services can be modeled as a local system, it is obvious to the developer that it is composed of non-local elements. However, at the user level a device is seen as a collection of networked services, and the underlying topology is transparent. A service can exist on a machine different from the machine that the user is physically using (See Figure 3-3). For instance, instead of using the microphone attached to one’s computer, it could be using the microphone on the computer next to the user’s computer.
3.2.2 Elements

The core elements of SMPL are services, connections, and dispatchers. By limiting SMPL to just three elements, as compared to a system like CORBA that has 16 basic elements, I believe that SMPL forms a better learning platform for designing complex networked systems. Central to the argument for establishing SMPL is the need for creating simplified architectures for developing complex architectures.

Services

At a glance, a SMPL service is a group of related functionalities, which could potentially belong to the same device, software application or set of data. A service is a self-contained, stateless function which accepts one or more requests and returns one or more responses through a well-defined interface. Services can also perform discrete units of work such as editing and processing a transaction. Services do not depend on the state of
other functions or processes and are always connected using an SMPL connection to an SMPL dispatcher. In the special case when the service is inside the dispatcher the connection becomes self-contained.

**Connections**

A connection is made out of a transport protocol and a payload protocol. Particular protocols have specific requirements. A transfer protocol must sustain a connection between two end points, and also allow bi-directional send and reply. The payload protocol should be agnostic of transfer protocol. As a reference, the current implemented transport protocol is TCP/IP, and the payload protocol is http (used in a special bi-directional fashion).

There are three levels of connections to make clear:

1. *Send a request without a reply.* For instance an input device like a mouse sends out a simple message, without expecting a specific response.

2. *Send a request and get a reply.* An example would be a standard http style server request that operates based on transactions.

3. *Events generated by a service.* In this case a service must be listening for a reply. It is the most computationally intensive type of connection.

Having all of these levels of connectivity support the best-effort approach in that they progressively require more computational resources. SMPL tries to leverage this gradated level of connectivity.

**Dispatchers**

Dispatchers are directories of services and references to services that are not on the same machine. The dispatcher has the role to keep track and advertise the services connected to it, and to allow the communication between these services and the services connected to other dispatchers. The dispatcher communicates to other dispatchers using a common
SMPL transport. Services may optionally use a different transport to communicate to their parent dispatcher. In the latter case, the dispatcher is in charge of doing the translation.

A dispatcher thus offers multiple functionalities:

1. mediating the connection from application to service
2. providing the discovery of services
3. allowing the directory of services
4. enabling the ability to search for services

There are three primary components to a dispatcher. The first is the main processor, or essentially the primary container, which provides the mechanism for discovery. The second component of a dispatcher are a set of handlers that allow different transport protocols to connect to the core. The third component is a set of core services that include the ServiceManager, the ServiceDirectory, the ProtocolManager, and a ConnectionManager.

### 3.3 Implementation

#### 3.3.1 Dispatcher

The SMPL dispatcher was implemented as a standalone Java program, designed as a high-performance TCP/IP server. Currently, the only supported transport protocol is HTTP over TCP/IP. The dispatcher’s principal role is as a “server”, managing connections and distributing service requests coming from the connections. Its secondary role is a service holder, keeping active instances of “local”² services, which include the four core services mentioned previously. Additionally, it keeps references to remote services not hosted in the same machine as the dispatcher and that have an already established connection with the dispatcher.

²SMPL services implemented in Java running in the same machine as the dispatcher.
The server’s main thread manages the TPC/IP connections and distributes the data coming from those connections among a pool of worker threads. The main thread uses multiplexed non-blocking I/O and a simple state machine that allows the thread to listen, open, and maintain several connections simultaneously. Since the non-blocking channels cannot give us information about the data coming from each connection, we delegate that responsibility to a set of working threads which are in charge of decoding the data, and executing the corresponding request using the appropriate service. To assure the multiplexer loop is never blocked, the incoming data from each TCP/IP connection is put in a buffered piped stream associated with each connection. When a stream has data to be processed, it generates a request, which is stored in a concurrent queue, in a scheme similar to a producers-consumers pattern common in concurrent programming. The queue is attended by a pool of worker threads, which assures that the requests are dealt sequentially for each connection. The worker thread will then use the appropriated handler to decode the transport protocol, and make use of the ProtocolManager to find and decode the appropriate payload protocol. Once this is done, the decoded information about the invoked service, method and arguments is used in the ServiceDirectory to find and finally call the correct service.

Additionally, the dispatcher also can be discovered in a local network by either a client or a remote service. The dispatcher responds to Zeroconf requests or to UDP packages broadcasted to a specified port.

### 3.3.2 Service Libraries

The services libraries allow the easy creation of services. These services can reside in the same machine as the dispatcher, or in another by registering them as remote services with a dispatcher hosted somewhere else. In general, services are composed of two parts: a service interface and the service implementation. The service interface works as the IDL of the service, showcasing the methods, data structures and events available from the service.
A simple example service interface looks like this:

```java
package smpl.examples;
public interface Counter {
    public int hit();
    public int count();
}
```

Once the interface is established, it can be realized by a standard Java class that implements the interface and extends the `GenericService` class. The latter provides utilities to deal with concurrency problems — there is a utility to store data in the current worker thread attending the service, and a similar one to store data in the associated connection. Finally, remote services make use of a `RemoveConnection`, which internally shares some of the mechanisms of the main thread in the dispatcher, but does not allow for the use of multiple protocols over the connection, nor allows clients to use the service directly.

### 3.3.3 Client Libraries

The client libraries were designed to provide a fast and easy way to use SMPL services. Normally, those services should be able to be accessed without the library. However, that requires that the developer keeps track of the connection to the SMPL dispatcher, as well as the encoding of the service requests in the appropriate transport and payload protocols. The client libraries simplify the procedure, by providing a proxy object for every connected service. The proxy object takes care of the appropriate encodings. The use of the client libraries is very straightforward:

```java
public static void main(String[] args) {
    Connector connector = Connector.getInstance(SMPL.getDefaultURL());
    Counter counterService = (Counter) connector.getService(Counter.class);
    int h = counterService.hit();
}
```

It is important to remember that the use of these libraries is not required to use SMPL services, and that they can be accessed from different
programming languages, such as Python, provided they support the transport and payload protocols.

3.4 Summary

In this chapter we described the SMPL architecture, its rationale and current implementation. SMPL was not designed to solve all the problems associated with collaborative distributed systems. In fact, its design is focused to provide just the minimum components to allow the construction of distributed applications. The architecture is only made of three elements: services, dispatchers and connections. Dispatchers require more computational resources; by being the central hub for the connectivity of applications and services, they leverage the power of small devices and larger systems. The simplicity of the SMPL architecture allows developers to integrate devices very easily. This enables experimentation, which with time will provide insight on how larger and more robust collaborative distributed systems should be built.
Chapter 4

Scenarios and Analysis

The Physical Language Workshop research group at the MIT Media Laboratory is currently developing an extensive user collaboration effort, named Treehouse Studio. Treehouse Studio is an online community with a suite of design tools and associated services to create and share different types of media. It has been implemented on top of successive versions of the SMPL infrastructure, and its breadth of functionality demonstrates the flexibility of the architecture described in this thesis. SMPL was conceived initially inside Treehouse Studio project.

In this chapter we will present different projects that use SMPL. All the projects were done by different members of the Physical Language Workshop as part of the Treehouse initiative. We will discuss briefly the projects, and provide some insight of the role of SMPL in each one. This will be followed by a concise analysis of how SMPL compares against other similar technologies.

4.1 Examples

4.1.1 SMPL RFID Reader

Radio Frequency Identification (RFID) is a technology that has gained in popularity in recent years as many large retailers have shown it to be effective in large-scale inventory tracking applications. Outside of such
business applications, the technology also offers novel opportunities for computer interaction through familiar and tangible items. While most systems are still designed to be task-specific mostly around inventory tracking, the SMPL RFID Reader makes a general-purpose reader available for use in a wide variety of applications.

One sample application, developed by Noah Fields [27], uses RFID tags embedded in physical photographs to provide a physical bookmark to a more complete photo album. In such an application, a photograph of the Eiffel Tower can be used to access an entire collection of digital photographs of a recent trip to Paris. Since SMPL is accessible through standard web technologies, the photograph is recognizable from any SMPL RFID reader that is attached to the system, anywhere in the world.

The architecture of the reader is quite simple. Rather than sending the tag notifications to a local machine, the hardware module uses onboard networking capabilities to route all communication through a central SMPL service. The additional cost of having networking capabilities is justified by the increase in connection opportunities with other devices and applications that are connected to the SMPL system. In this way, RFID technology is accessible outside of the conventional models that are found in inventory systems and becomes part of a global toolbox of interoperating hardware and software.
4.1.2 SMPL TCP/IP Speaker

A computer usually ships with its own speaker, or else the ability to attach speakers to the computer. A SMPL service was constructed around a simple loudspeaker attached directly to the network using a Rabbit 2000 microcontroller. This service is operated by attaching from an Icecast server and streaming MP3 files to the SMPL service. The benefit of this approach to streaming digital audio is that any number of speakers can be added to the network economically.

The benefits of using the SMPL architecture are that a SMPL client can connect to any speaker on the network, and thus these speakers need not be placed in a local environment. For instance, one could imagine a painting application that when digital ink is applied is converted to sound spanning many SMPL speakers across the world. This system also demonstrates the low computational requirements of managing a SMPL service as it is powered by a simple Z80-based processing system. The same infrastructure was used in the development of the RFID reader.

4.1.3 SMPL Multi-Mouse Service

SMPL provides the means by which it is easy to add multiple mice to an application. Using our existing framework for application development,
Treehouse Studio, connected to a SMPL service that supports multiple mouse devices demonstrates the flexibility of SMPL as an infrastructure for networked collaborative environments. This application was developed by Mariana Baca.

The SMPL server runs a service that manages network mice. The network mouse device connects a generic two-button mouse to an ethernet connection. It has a Javelin Stamp [3] (a JAVA microcontroller) to boot the mouse and interpret mouse signals through a serial port. It then sends a string containing the mouse ID and the state of the mouse to the Latronix XPort [2] connection. The network mouse service, running on the SMPL server, can communicate through tcp/ip with the Xport. First, the user sends a request to the service to initialize the mouse, using as parameters the IP address of the mouse, and the mouse ID. The service opens a TCP/IP connection with the Xport, and parses the string sent by the Javelin chip. The service holds a hashtable of all the different mice which have been initialized, and reads off the TCP/IP connections for all of them on a timer. Separately, applications can send URL requests to the service to ask for current mouse status for a given mouse ID. The SMPL server allows multiple applications to access a mouse, since the Latronix XPort can only handle one client at a time. The service allows for a standard interface for applications to access mice, and abstracts the IP connection away from the application code, and allows applications to
add mice.

### 4.1.4 SMPL Flexible Screen

![Figure 4-4: The SMPL Flexible screen.](image)

A flexible screen prototype with a bending interface has been created by using an SMPL device service. Developed by Burak Arikan, the purpose of the prototype was to design a screen-paper hybrid device experience. By bending the screen we are able to control the graphical application that uses the SMPL service which reflects the bending sensor. The sensor input is processed by an embedded micro controller. Its serial output is converted into TCP packets, and transmitted to an SMPL server over the network.

SMPL services have been used to create some simple applications using serial input/output that control and sense devices over the network. A single force sensor has been used to control graphics in multiple software, a servo motor has been controlled by mouse clicks in an application, and finally a composition of multiple infrared sensors and knobs used in a single application. In these experiments, SMPL services basically connect to the devices over the network, and the attributes that are reflected in the service are used by the applications.

These experiments are focused on bidirectional connections between hardware and software through SMPL services. SMPL creates new
opportunities by enabling multiple connections between different functions. Thus it defines a new ground for thinking how we experience the networked computational medium.

4.1.5 SMPL Rendering Service

The SMPL Rendering service, developed by Kelly Norton, is a software service that leverages the SMPL framework to connect an arbitrary number of computers that support the compositing and rasterization of network documents. This service is the most technically sophisticated use of SMPL to date. The service is designed to provide rendering support for display devices that are unable to properly render certain graphical formats. Additionally, it can also provide support for application developers that wish to avoid dependency on particular formats or to ensure a level of rendering quality for the network-delivered content. To illustrate typical use, two example applications are described below.

The SMPL Rendering service is currently being used to support a photo album application that runs on a PDA and is able to browse and view a complete collection of full-resolution digital photos. Portable devices, such as PDAs and cell phones, seem an obvious tool for accessing digital media. However, current devices are generally not equipped with enough computing or networking resources to make such scenarios feasible. The SMPL rendering service removes much of the resource demand from the
device by off-loading computational and memory intense tasks to backend servers. When the portable device needs to display thumbnails of photos, for instance, the backend server performs the scaling of the images and delivers just the small-scale image that is displayed on the devices screen. As the user selects and views an individual photo, the server is again transforming the original and sending only what needs to be shown on the devices small screen. It is even possible to view photos at full resolution with such a system as the image that is sent to the device is clipped to the current viewing area.

The SMPL rendering service is a distributed application built on top of the SMPL architecture. The service handles requests from clients to assemble one or more documents into a single raster view, which is composited properly and returned to the client. To exploit opportunities for concurrent processing, the system is broken down into three different node types:

![Diagram of SMPL Rendering architecture]

*Figure 4-6: The SMPL Rendering architecture.*

**R-Dispatchers** Only a single R-dispatcher is needed for each instance of the rendering service. R-dispatchers are responsible for receiving the requests from clients and issuing them to the compositor with the lightest resource load. They also act as a directory service for the system, since they also keep up with the available rasterizers in the system.
Compositors Each request is assigned to a single compositor, that is responsible for issuing further request to one or more rasterizers and then combining the results into a single image according to the details in the clients request. As the name suggests, compositors combine the images returned from the rasterizers and do full alpha compositing to produce the final image to be returned to the dispatcher and finally back to the client.

Rasterizers The rasterizer is responsible for creating a rasterized image from a documents or a sub-section of a document, regardless of the format. The system is currently able to render many formats including most pixel-based formats, PDF and other vector formats that are used in the Treehouse Studio system. Other formats can be easily added through a plugin infrastructure.

Each type of node in the rendering service makes extensive use of SMPL as each of the three node types are implemented as separate SMPL services and all network communication, including inter-node messaging, is done through SMPL protocols.

The implementation of the rendering services using SMPL, provides an important advantage: the service can be reused by external applications and devices. An example of this is the use of the rendering service in the Treehouse Studio project (see Figure 4-7). The rendering service is used to generate on-demand thumbnail images of all the documents created using Treehouse applications. Another example is the use of the SMPL renderer in a handheld application (see Figure 4-8), such as the one developed by Kelly Norton. This application allows the handheld to display Portable
Document Format (PDF) documents that by other means would not even fit in the device memory. This is a clear example of a small device taking advantage of the power of a larger system, and where the larger system is extended by the mobility given by the small device.

4.2 Comparison and Analysis

In this section we will compare the design of SMPL with a group of related technologies. We focus on comparing distributed systems platforms that have parallels to SMPL. It is important to mention that the design of SMPL was based on many of the lessons acquired by observing how other related technologies were designed. Finally, an analysis on SMPL will be drawn from the comparison. Figure 4-9 summarizes this comparison.

<table>
<thead>
<tr>
<th></th>
<th>CORBA</th>
<th>JINI</th>
<th>UPnP</th>
<th>WS</th>
<th>SMPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computational</td>
<td>High</td>
<td>Moderate</td>
<td>Moderate</td>
<td>Moderate</td>
<td>Variable</td>
</tr>
<tr>
<td>Requirements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Handling</td>
<td>Excellent</td>
<td>Good</td>
<td>Implementation</td>
<td>Implementation</td>
<td>Programmer</td>
</tr>
<tr>
<td>Concurrency</td>
<td></td>
<td></td>
<td>Dependent</td>
<td>Dependent</td>
<td>Dependent</td>
</tr>
<tr>
<td>Module Interdependencies</td>
<td>Tightly coupled</td>
<td>Tightly coupled</td>
<td>Loosely coupled</td>
<td>Loosely coupled</td>
<td>Loosely coupled</td>
</tr>
<tr>
<td>Firewall Compatibility</td>
<td>Problematic</td>
<td>Problematic</td>
<td>Not applicable</td>
<td>Resolved</td>
<td>Resolved</td>
</tr>
<tr>
<td>Language Dependency</td>
<td>Bindings available on many platforms</td>
<td>JAVA only</td>
<td>Open</td>
<td>Open</td>
<td>Open</td>
</tr>
<tr>
<td>Protocol Flexibility</td>
<td>Single primary protocol (HiP)</td>
<td>Single primary protocol (RMI)</td>
<td>Open</td>
<td>Open</td>
<td>Open and Custom protocols possible</td>
</tr>
</tbody>
</table>

Figure 4-9: Comparison between SMPL and related technologies.

4.2.1 JINI

Of all the projects mentioned in this section, JINI [10] is perhaps the one that shares the most similarities with SMPL. JINI was from its conception envisioned to bring together all kinds of computer devices. This idea was propelled by the Java’s motto: "write once, run everywhere," which is reflected in the heavy use of the Java Virtual
Machine (JVM) throughout the design on JINI. To be fair, the JINI specification does not truly enforce the use of a specific platform or specification. However, even years after its release there was no available support for any other platform besides Java. A similar situation exists with the protocols that JINI uses to connect its different components. The JINI specification was protocol agnostic, however the initial implementation only supported custom design protocols that were not available in other platforms.

The major difference between SMPL and JINI is the way that services are used. Since JINI depends upon the JVM, it can function by moving java objects through the network. Services are dynamic – the code can move from machine to machine. This enables many possibilities, but at the same time, raises the computational requirements for devices, and thus limiting the number of platforms where JINI can be implemented.

Another significant difference is that JINI services connected to the network are in fact peer-to-peer. The existence of the LookupService (similar to the ServiceDirectory in the SMPL dispatcher) only allows services to find each other, but the connection is handled directly by the devices hosting the services. This again forces devices to include more computational resources to handle the overhead of managing multiple connections. In contrast, the SMPL dispatcher has a much more active role managing the distribution of connections and requests to the appropriate service. A remote service in SMPL only has to manage the connection to a dispatcher. The dispatcher requires computational resources similar to a normal desktop PC – requiring maintenance of the concurrency among services, the connections with clients, and the directory of services among others. SMPL service devices can be extremely light. They only need to support a connection to a dispatcher and provide the functionality they offer. SMPL helps larger, more capable devices to benefit from the portability of smaller mobile devices, and vice-versa.
4.2.2 CORBA

CORBA [79], and associated technologies (e.g. Microsoft’s DCOM, Java RMI, etc.) are distributed objects architectures aimed mostly towards enterprise-class applications. They do not intend to be operational in low-level devices. However, they offer an important point of comparison since they represent the most widely deployed distributed system architectures. Common to all of these technologies is that they try to hide the network from the programming point of view. Remote objects look just like local objects – hiding most of the problems caused by being connected through a computer network [31]. This makes it difficult to determine the point of failure, since many of the different networks components have been abstracted and hidden to the programmer.

In contrast, SMPL tries to make the network more visible to the programmer. The architecture itself exposes the connection elements that comprise the implementation. Programmers instantiate a SMPL connection before they can access services. The status of the connection can be monitored at all times, and network failures are represented by appropriate exceptions that can be traced.

The SMPL client library provides an access to services through proxy objects, which behave exactly like a Java object. This may lead to confusion to the developer, which may treat services just as remote objects. To solve this, we follow the use of common programming conventions, such as the use of the word Service at the end of all proxy object names.

CORBA also uses a highly complex mechanism to provide many features that are missing from SMPL: transactions, session management, security, and robust fault tolerance. These features that are core to the implementation of CORBA mapping demand a specific class of platform. The commercial interest around these technologies has resulted in a larger number of CORBA bindings, available in a large variety of programming environments. A new contender in the distributed objects market is ICE [35], which is designed to provide most of the same functionality as CORBA, but with a simpler design. In design terms, SMPL uses an IDL
4.2.3 Web Services and UPnP

Web Services (WS) have been heavily hyped over the last several years [78]. They are a flexible set of technologies that leverage web standards and protocols to facilitate their ease of adoption. This can be observed in the intense use of XML in all the WS standards. The use of XML as the basis for interconnectivity between computer platforms demands sufficient computational resources to handle XML. Furthermore XML is not compatible with the efficient handling of binary data such as images. For the design of SMPL, the moderate success of WS technologies was taken into account, and the first implementation was done over existing and popular standards (HTTP, XML-RPC). However, SMPL was designed in an extensible manner that allows protocols more appropriate for certain platforms to be easily used by services.

WS — by design — were not intended to be used in consumer devices. UPnP [7], however, has taken a broader look at WS standards and places them in the context of consumer electronic devices; once again the use of XML technologies increases the computational requirements. Similar to JINI, every device connected through UPnP acts as a server and a client. The justification for this redundancy is Moore’s Law – that computational power will rapidly catch up to the higher computational requirements. It may well be true that the processing power will continue to decrease, but the complexity and difficulty of programming these systems will not decrease. Compare a WS interface in WSDL [38] (see Figure 4.3.3) to the same interface in SMPL:

```java
public interface StockQuoteService {
    float GetLastStockPrice(String StockSymbol);
};
```

A key difference between UPnP and SMPL is at the abstraction level of a service. UPnP promotes the standardization of service interfaces at the device level. The current standards point towards groups of functionalities, such as the capabilities of a VCR. UPnP exposes those
capabilities in a monolithic block, much like a remote controller might be described. SMPL encourages separating functionality into smaller blocks that can be easily reused and accessed to create new functionalities. For example, the speakers of a stereo may be used as the output from a telephone or a TV.

### 4.3 Conclusions

In this section we will derived a list of strengths and weaknesses in the design and implementation of SMPL based on the the experience of building the SMPL applications described in the previous chapter, and the comparison between SMPL and related technologies done in the previous section.

#### 4.3.1 Strengths

The most salient strength of SMPL is that the simplicity of the architecture and the associated programming libraries facilitate significantly the development of new services and the integration of different devices into the network. This allows the rapid prototyping of distributed applications, which eventually could help to gain insight into the problems and implications of constructing massive distributed systems.

A second strength is in the structure that dispatchers impose in the system. This allows to move most of the computational resources to the most powerful machines, while allowing smaller low-end devices to access the capabilities of all the services in the network. Another advantage of the dispatcher configuration is the ability for applications to traverse firewalls and NATs.

The third strength of SMPL, is its ability to use multiple protocols transport and payload protocols. If the current implementation only support one transport protocol (HTTP), the variety of payload protocols allowed for performance and development benefits in some of the applications described previously.
4.3.2 Weaknesses

The principal weakness of SMPL is that it lacks some of the most popular characteristics of related commercial technologies. Particularly, SMPL does not possess any security or validation mechanisms, which could open the door for many potential problems, as well as undermine the interest of SMPL as an industrial or commercial platform. However, we believe that by dropping that specific characteristic, the design is made simpler providing better ground for experimentation.

The second weakness is in the implementation. By using an object oriented language, we created an ambiguous abstraction: services are being represented by objects. This requires special care from the programmer, who should know about the differences between the assumptions made by the object and the service abstractions.

4.3.3 Summary

In summary, like all systems, SMPL has its own set of strengths and weaknesses. By highlighting them, we set the next steps in the development of SMPL. The design of a distributed system without security in its basic mechanisms may seem naive in this incredibly complex world of today. However, this reduction makes it easier to compose interesting systems out of a larger set of parts. We believe that there is always a benefit to designing for simplicity, and that this will clearly play out in future developments.
Figure 4-10: WSDL interface for a stock quote service.
Chapter 5

Conclusions and Future Work

“The tar pit of software engineering will continue to be sticky for a long time to come. One can expect the human race to continue attempting systems just within or just beyond our reach; and software systems are perhaps the most intricate and complex of man’s handiworks. The management of this complex craft will demand our best use of new languages and systems, our best adaptation of proven engineering management methods, liberal doses of common sense, and a God-given humility to recognize our fallibility and limitations.”

Frederick Brooks, Jr. [37]

5.1 Conclusions

This thesis presents SMPL, a network architecture designed to promote the integration of data, devices, and processes into collaborative efforts among people. SMPL targets this goal by providing an accessible, clear and simple architecture that emphasizes a generation of new functionalities composed of existing resources. The architecture is designed around the abstraction of functionalities as Internet Services that communicate between each other in a hybrid centralized-decentralized network topology. A central theme in the design of SMPL is to provide just the basic functionality for developing a distributed application. More of the complicated issues such as security and versioning were ignored so to open room for the experimentation of simple models and mechanisms. This chapter reviews the lessons from this experiment. Based on those conclusions we will then present possible
next steps for the development and implementation of SMPL. Finally, we will proceed to describe future directions for systems like SMPL.

5.2 Lessons

The SMPL architecture facilitates the participation of developers — by working under the assumptions that successful technology that scales is more about than just decentralization. It is also about how easily technologies can be appropriated by the developer community, and in the long term, by the general public. It is important for the design of a successful system to be easy to understand, develop and maintain. SMPL reflects this assumption: very few elements are involved in the architecture, and they interact in a simple predictable manner. This proved to be very useful, as it was easy for other developers to pick up the concepts and start developing SMPL applications and services with the provided tools.

Another example of the importance of a low-entry level for systems like SMPL is the relative success of the Query protocol. The protocol was initially developed for debugging purposes. It allows the invocation of SMPL services using only a URL which made possible the invocation of services from a web browser. Due to its simplicity, it became the most popular option for the invocation of SMPL services from small devices, like the SMPL network mouse or the SMPL flexible screen.

At the same time, the gain in efficiency facilitated by the use of the Hessian protocol in the SMPL Renderer, showed the importance of providing flexibility in the system. In the case of SMPL, such flexibility did not come from the adoption of a unique standard, but from the provision of mechanisms to use the payload protocol that is better suited for the task or the device involved. We believe that this is an important consideration in the future development of collaborative distributed systems - instead of desperately trying to design the perfect standard to suit all needs, more work should be done in designing systems that allow an application to choose the methods that better matches its needs.
The use of HTTP as our first transport protocol proved to be problematic. HTTP was not really designed to take full advantage of one of the most important characteristics of TCP/IP: bidirectionality. HTTP was designed with web pages in mind: there is no need for a permanent connection since web pages take some time to read, and the transactions always come in the form of a request and a reply. However, HTTP has become the preferred method of transport for many new distributed applications, as evidenced by its use in UPnP and Web Services. The justification for this is that almost every programming platform provides support for HTTP. We used HTTP based on that justification but found many difficulties while trying to use the protocol, especially for asynchronous event notification.

We feel that the increase in complexity created by using HTTP asynchronously could have been compensated better by developing a simpler transport protocol — one that is better tuned to the nature of a distributed system. Based on some experimentation undertaken in that direction, we found that the development of such protocol should not be taken lightly, as it requires a good amount of experimentation. Again, we think it is important to leave open the option for the use of different transport protocols.

Finally, we try to find a balance between a simple architectural model while exposing the practical challenges of a networked application. We think that making the connection element explicit in the architecture and implementation is necessary. We also believe that the service abstraction helps reinforce the concept that the application is made of component parts that are distant, and so, require a unique approach. Exposing the network constraints contradicts to a certain extent our intention to make a very simple-to-use system. The success of the C language can be credited to a similar design philosophy of what you see is what you get (WYSIWYG) from the perspective of the programmer.

To make a reliable application or service, the developer should not only be familiar with the SMPL architecture, but also with the concepts and problems inherent in a distributed system. Furthermore, the tools

---

1A very early version of a SMPL protocol was developed, based mostly on BEEP, the Block Extensible Exchange Protocol [65].
provided by the programming languages add to the confusion: to facilitate the interaction with SMPL services, we created a way to use proxy objects. A developer is more likely to be familiar with the concepts from Object Oriented Programming than with the ones from Distributed Applications Programming. This leads the developer to assume that the object is in fact local, free from the problems associated with network connections. The solution for this problem is not clear, but it is not too much of a stretch to say that it may involve evolving the abstractions in our programming languages. As Andrew S. Tanenbaum is generally quoted: “Distributed systems need radically different software than centralized systems do.”[5]

5.3 Future Work

SMPL was not designed to be the silver bullet for distributed systems. It was designed to provide the basic functionality required by a distributed application, as well as to offer a low-barrier entry level for development. This however does not mean that some functionalities like security, reliable transactions and versioning should not be explored. They are fundamental for an architecture like SMPL to be successful in real world applications. Our belief however, is that the next step is to experiment with trying to layer those functionalities on top of the existing SMPL system, instead of adding them as fundamental parts of the design. At this stage it is hard to tell if this is feasible or not. At the same time it is an important effort; keeping the architecture simple removes the need for computational resources, which may limit the number of devices that can use SMPL.

Another immediate step for SMPL, is to test how well it deals with composition. The applications portrayed in this thesis did not really make full use of the composition facilities of SMPL. It is not yet clear how well SMPL will deal with dependencies or how well it can recover from failures. This is a vital road to explore, and one that will most likely necessitate big changes in the overall design.

Some minor next steps involve the completion of the SMPL protocol and
the addition for support of other transport protocols like the Simple Mail Transfer Protocol (SMTP), and BlueTooth for wireless devices. Although minor, these contributions could help test SMPL in a more diverse environment. Finally, it is very important to assess the performance of an SMPL application across the Internet. This could provide great information about what is necessary to tune and improve in the SMPL implementation.

5.4 Future Directions

Besides the improvements in the design and implementation of SMPL mentioned in the previous section, we think that the most interesting opportunities for exploration arise from closing the gap between architectures like SMPL and the end users. The important questions that remain, address how these systems are used and manipulated by people, leaving aside the role of developers. As a first instance, this road could start with the construction of tools that show and make explicit to a user the network of functionalities available by a system like SMPL. A further step could be the creation of applications that would enable a user to compose services in a simple manner. However, such applications are not so interesting without first understanding the real relevance of providing the user with the ability to access and compose functionalities using all the computer devices around them. The question that remains is if this system will ultimately be useful, and allow the creation of truly collaborative structures. We think the Treehouse Studio initiative could provide such opportunity for experimentation, and help bring more insight about the significance of these technologies in contexts of education and creative expression.

Another road open for exploration is in the development of better tools for the creation of distributed systems. As mentioned previously, the current paradigms seem to be insufficient to capture the nuances and problematic nature of distributed services environments. Systems like SMPL could also be used as glue between many other existing technologies. Finally, this could also lead to the development of parallel
technologies that will allow systems like SMPL to provide more autonomous functionalities. A system based on SMPL could start to merge with projects like the Semantic Web [15], which could allow the automatic discovery and composition of functionalities on the network. This would lead to better distributed networks, that could heal and recover from failures. More robust and reliable systems with less interdependencies, will open up room for even greater innovations.
Bibliography


[40] Kevin Kelly. We are the web. *Wired Magazine*, 13(08), August 2005.


[58] Tim O'Reilly. The Network Really is the Computer.


88


