Architectures of the Third Cloud

Distributed, Mobile, and Pervasive systems design

David Gauthier

B.Sc. in Mathematics, Université du Québec à Montréal, August 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 2009

Author

David Gauthier
Program in Media Arts and Sciences
August 27, 2009

Certified by [illegible]

Andrew B. Lippman
Senior Research Scientist
Program in Media Arts and Sciences

Accepted by [illegible]

Deb Roy
Chair, Departmental Committee on Graduate Students
Program in Media Arts and Sciences

Architectures of the Third Cloud

Distributed, Mobile, and Pervasive systems design

David Gauthier

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 2009

Abstract

In recent years, we have seen the proliferation of ubiquitous computers invading our public and private spaces. While personal computing is unfolding to become mobile activity, it rarely crosses the boundary of our personal devices, using the public interactive infrastructure as a substrate. This thesis develops an approach to interoperability and modular composition in the design of ubiquitous devices and systems. The focus is placed on the relationship between mobile devices and public infrastructure, in particular how a device with access to information about its physical and social context can dynamically configure and extend functionality of its cooperative environment to augment its interactive user experience.

Based on Internet concepts of *connectivity utility* and *resource utility*, we derive the concept of *interaction utility* which we call the Third Cloud. Two complementary systems designs and implementations are presented to support this vision of computing. *Substrate* is an authoring framework and an execution environment intended to provide the necessary language and tools to easily compose self-operable applications capable of dynamically instantiate desired functionality in their proximate environment. The *Amulet* is a discrete portable device able to act on behalf of its user in a multitude of contexts. We evaluate the power and flexibility of these systems by using them in the construction of two applications. In the final chapter, we compare our approach with alternative ways of building such applications and suggest how our work can be extended.

Thesis Supervisor
Andrew B. Lippman
Senior Research Scientist
Program in Media Arts and Sciences

Architectures of the Third Cloud

Distributed, Mobile, and Pervasive Systems Design

David Gauthier

The following people served as readers for this thesis:

Thesis Reader [illegible]

David P. Reed
Adjunct Professor of Media Arts and Sciences
Program in Media Arts and Sciences

Thesis Reader [illegible]

William J. Mitchell
Alexander Dreyfoos Professor of Architecture and Media Arts and Sciences
Program in Media Arts and Sciences

Thesis Reader [illegible]

V. Michael Bove, Jr
Principal Research Scientist
Program in Media Arts and Sciences

Acknowledgements

To my advisers, *Andy Lippman* and *David P. Reed*, I would like to express my deep gratitude for their critical insights and guidance that helped shape the core of this thesis. When I first apply to the Media Laboratory the idea of Living the Future was a bright idea. In retrospective most of this thesis revolves around concepts relating to it. I would also like to thank my readers, *Bill Mitchell* and *Michael Bove*, for their precious time and comments. Working with all four of you has been an extraordinary experience and an amazing privilege.

To *Diana Mainstone* for her precious help and support. You have been my inspiration from the moment I applied to the program, until the very final moment when I submitted this thesis. Your patience has been invaluable, especially through the hard times past away from each other. You are the most amazing partner and accomplice one could ever ask for.

To my lab-mates and friends, *Pol*, *Nadav*, *Fulu*, *Dawei*, *Kwan*, *Sung*, and *Inna* for making the Viral Communications group a fun research program. Special thanks to *Deb Widener* and *Sandy Sener* for their help in making sure that my long order lists would always fit in the budget.

I joined the Media Laboratory because it is a place where thinking unconventionally is the rule. Thank you for giving me the opportunity to be part of such a creative and inspirational family.

"Recordings deal with concepts through which the past is reevaluated, and they concern notions about the future that will ultimately question even the validity of evaluation."

Glen Gould

Contents

List of Figures

1 Introduction

Viral and Ubiquitous Computing

In the last century, development in communication technologies, fabrication processes and electronic miniaturization have dramatically altered the types of objects and environments we can construct and interconnect. In this early 21st century, advances in the development of wireless transceivers and low-power processing units are expanding the limits of where computation can be found and reshaping the ways in which we interact and communicate.

In 1991, Mark Weiser predicted the disappearance of computers into the background of our daily lives freeing us to "use them without thinking and so to focus beyond them on new goals" [Weiser, M. 1991]. Since then, his concept of ubiquitous computing as been one of the most flourishing model of human-computer interaction, leading to the research and commercialization of plethora of new devices and systems which are *now* part of our daily lives. But are they really in the background?

While Weiser's vision of ubiquitous computers has become a reality, his vision of invisible computing and "calm" interaction has not. This is perhaps due to the fact that our current miniaturized devices demand a great deal of attention when operated. These devices are usually designed as standalone units and in many case limited in functionality. Often bound to a single mode of operation and tied to a specific network, they don't easily communicate or interact together. "Even the most powerful notebook computer, with access to a worldwide information network, still focuses attention on a single box" [Weiser, M. 1991]. As more of them populate our daily lives, more of our attention gets distributed over this collection of single boxes. Over time, we have come to assume a fragmented experience of ubiquitous computing.

In parallel to the development of ubiquitous computers, was the evolution of the Internet. Rooted in years of research and development on distributed systems, the focus was placed in building a flexible, interoperable communication infrastructure where computing resources were to be accessed remotely and invisibly through the use of a browser. Very rapidly, new resources and functions were added to the infrastructure and made available to the public. Nowadays, this concept of a *computing utility* has become an invaluable collaboration and social utility for its users. Even the desktop environment is now being reengineered to seamlessly move away from our personal machines to a place in the "cloud".

We believe the same concept of utility ought to be transpose into the ubiquitous computing sphere. In the future, ubiquitous computers will form an *interaction utility* centered on mobile and social communication. The premises of ubiquitous computing are quite different to those of the Internet. "Embodied virtuality" [Weiser, M. 1991] as more implications in physical and social realms than does the transport of bits from end-to-end. Nonetheless, important lessons can be learned from the architecture of the Internet, especially the ones related to flexibility and interoperability.

Research into the architecture of interactive and ubiquitous environments, dubbed *Intelligent Environments* (IE), has been thoroughly conducted at the turn of the century [Brooks, R. A., 1997; Johanson, B. et al., 2002; Rudolph, L., 2001; Sousa, J. P. & Garlan, D., 2002]. Multitudes of human-centered environments featuring interactive displays, multimodal user interfaces, sensors, mobile devices, and so on, have been built around the themes of workspaces or rooms. Although incredibly rich in terms of interactive possibilities, these highly controlled and highly specialized spaces have not been designed for public spaces. In this thesis, we are interested in addressing issues of public and spontaneous access to interactive utilities; much like the Internet did with computing utilities.

Over the last year in MIT's Viral Communications Group, my advisor David P. Reed has come with a name for this emergent field of research – ***The Third Cloud***. Constructed around previous research in distributed systems and the Internet, this new "cloud" is situated at the edge of the network – where interpersonal context matters, mobility reigns, and access to serendipitous interactive contexts are to be negotiated in time and place.

This thesis looks at the topic of interoperability and modular composition in the design of ubiquitous and public interactive systems. The focus is placed on the relationship between mobile devices and public infrastructure, in particular how a socially informed device can dynamically configure its context to augment its interactive user experience. The objective is not to invent new user interfaces per se, but more specifically to identify configuration strategies to spontaneously assemble and hold for a moment all the bits and pieces available in a space needed to perform a given human-centered interaction. In this process, I hope to map possible application areas, to understand what are the limitations and interaction paradigms that may be captures in an authoring framework, and most importantly, inspire future development atop the foundations of the Third Cloud.

Contribution

I am not the first person to attempt to design such a system. After all, the vision of ubiquitous computing has a long history. Instead, I hope to bring a provocative new set of questions and answers about awareness, security, and interoperability by revisiting the fundamentals of distributed, mobile, and pervasive computing. With this in mind, this thesis hopes to provide a contribution in three distinctive ways:

(1) Begin to devise a theoretical framework to discuss public and ubiquitous network configuration strategies involving humans, machines and spaces.
(2) Advance in making accessible to interaction designers and software developers a authoring framework to construct self-configurable, mobile and distributed applications.
(3) Advance in making accessible to the community open source hardware and software components for the development of portable computers.

Thesis overview

Chapter 1: Introduction explains the motivations behind the work presented in the thesis.

Chapter 2: The Third Cloud situates the research area in an historical context and presents a theoretical framework to understand possible implementation strategies.

Chapter 3: Towards a system design presents technological enablers and our design approach in the development of Third Cloud devices, systems and applications.

Chapter 3: Substrate is the authoring framework and execution environment addressing guidelines presented in chapter 3. A high-level language to compose Third Cloud applications is presented, a system-level awareness protocol is devised and a mechanism to dynamically place functionality in a network is explained.

Chapter 4: The Amulet is a portable device personalizing its user in different contexts. The open hardware platform is described and identity based protocols are explained.

Chapter 5: Applications presents two applications drawing upon *Substrate*'s functionality and the *Amulet*'s execution platform. The focus is to demonstrate the flexibility and simplicity of the applications composed with the authoring framework.

Chapter 6: Conclusion and Future work concludes by discussing Substrate's and Amulet's respective implementation's strength and weakness and by discussing future work in the Third Cloud.

2 The Third Cloud

This chapter starts by situating the concept of the Third Cloud in an historical perspective in defining the first and second clouds. A theoretical framework is then presented to bring discussion on the socio-technical strategies and tactics that may be employed to make the Third Cloud vision of computing a reality.

HISTORICAL BACKGROUD

The First Cloud is the early Internet. We refer to this cloud as the one connecting remote machines together to share information. The first cloud is a *connectivity utility.* It presents a scalable communication architecture augmented of generic protocols to deliver information on demand from an addressable machine to a browser.

The Second Cloud represents the evolutionary step towards a computing utility after the First Cloud. We refer to this cloud as the one consisting of resources such as storage, databases, remote functions, specialized software as services, and so forth, made available to perform computing through a browser. In this context, the second cloud is a *resource utility* – embodying concepts of what has been commonly called "cloud computing".

The idea of computing resource utility is not new. In fact, we can date it back to time-sharing systems developed in the 1960s at MIT [Corbató, F. J, et al., 2000] and other institutions where multiple users could access centralized computing resources through remote teletypes. In 1961, John McCarthy suggested that computer time-sharing technology might lead to a future in which computing power and even applications could be sold through the utility business model. Time-sharing systems were offered at the time by Honeywell, IBM, General Electric, and other companies. The business disappeared at the end of the 1970s with the advent of personal computers and the early Internet. These days, the idea is surfacing back with platforms such as Amazon EC2

[Amazon.com, 2009], Azure Services Platform [Microsoft.com, 2009], and Google App Engine [Google code, 2009] to name a few.

Resource virtualization is at the center of the idea of the Second Cloud. Processing, data storage, and hardware are made available to end users as virtual entities which are location independent and accessed concurrently with other users. Resource virtualization enables the dynamic modular composition of computing platforms or applications tailored around specific computing needs. In some extent, the second cloud can be viewed as a computing platform construction toolkit. It gives users agency, which was not the case in the first cloud, in the fact that they can now produce data and construct programs which will be seamlessly hosted in the infrastructure for other users to access or operate. Access to these resource utilities is usually negotiated with a provider (Amazon, Microsoft, Google) prior to their usage.

The Third Cloud is situated at the edge of the First and Second clouds, centered on interactions involving humans, their devices and their immediate environment. Communication in the Third Cloud is enabled by context, both mobile and social. Unlike the two previous clouds, physical context in which computation takes place is of prime importance as it delineates the boundaries of interaction and communication between mobile and local entities. Interacting entities must be collocated in order to cooperate. Communication is also shaped by the social context in which end users are situated. While the second cloud presents applications bound to social aspects of their users, dub social networking applications, the Third Cloud embeds this knowledge at the system level where it can be blended with physical context knowledge. If networking is centered on social relationships, it is crucial that our mobile devices be able to represent ourselves wherever we may roam. Physical and social context are both application invariant as they directly shape forms of communication. Hence Third Cloud platforms are based around *awareness and discovery.* Sensing/signaling presence and activity information are mechanisms providing the necessary means to construct representations of physical and social contexts. We call these adaptive representations *neighborhoods*.

Neighborhoods provide context of interaction involving mobile and local entities. Exploitation of interactive material is performed through cooperative use of resources. In a given neighborhood, an interactive task residing on a user's mobile device may be projected in the environment making use of the infrastructure to carry or amplify the interaction. Interoperability between both mobile and local entities is established through resource and computing virtualization, standardized interfaces and evolvable protocols. Devising Third Cloud systems is not by itself a technical challenge - most of the required technologies are available - but rather a systems architecture challenge.

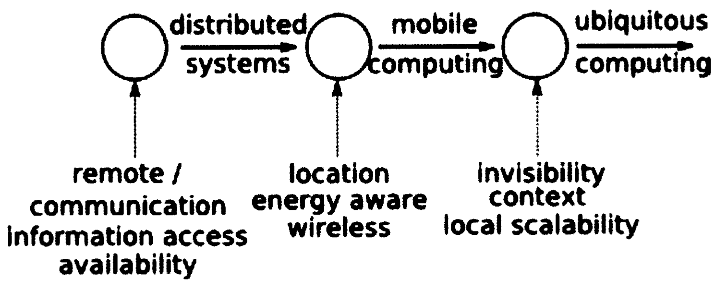

distributed systems
mobile computing
ubiquitous computing
remote / communication information access availability
location energy aware wireless
invisibility context local scalability

Figure 2.1 – Diagram illustrating interrelations between distributed systems, mobile computing, and ubiquitous computing domains.

We can draw important similarities between this Third Cloud concept and the one of ubiquitous computing. The main difference with previous incarnation of ubiquitous environments [Johanson, B. et al., 2002; Rudolph, L., 2001; Sousa, J. P. & Garlan, D., 2002] relates to scope. The Third Cloud looks at bringing cooperative use of interactive resources in the public realm; creating an *interaction utility*. While *Intelligent Environments* (IE) enable the cooperative use of interactive materials, the places in which they were designed to be deployed are private – the office, the house, the living room, etc. In framing ubiquitous computing solely in these environments we lose the quality of a ubiquitous host who can be found anywhere at anytime. In these environments, the context for interaction is well defined, maintained and controlled, while it needs to be spontaneously configured and negotiated in a public setting. This difference begs important systems architecture challenges as public spaces will be in practice less organized, perhaps decentralized and present uneven interactive and computing conditions depending on location.

We approach these challenges in considering *virality* as one of the core interoperability concepts of the Third Cloud. A viral process opportunistically exploits resources of a ubiquitous host. In this thesis, we are particularly interested in identifying the structural elements of the ubiquitous host and the logic that needs to be embedded in the viral process in order to spontaneously migrate to a (interactive) resource's context and consume it in place. In other words, conceive the infrastructure as a type of plastic that one can dynamically shape into forms which are relevant to a certain interaction. Here the infrastructure is not required to be "intelligent" or highly specialized per se, but rather malleable; it is the viral process which figures the context of interaction and injects its own program. The Third Cloud places most of the "intelligence" on mobile devices rather than in the environment. A mobile viral process could potentially scale without boundaries depending on the ubiquity of the hosts. Deployment of these hosts could be based on a contributory participation scheme where you would be required to contribute a host to join the network.

Composition and authoring of viral processes is also of interest in this thesis. We have learned from the Second Cloud that modular composition of resource utilities out of standardized service types enable end users to tailor their computing platform according to their needs. We approach the composition of interaction utilities the same way. By providing the necessary abstractions, one can assemble a viral application which will behave and consume interactive materials in a suitable way.

In placing the Third Cloud in the public realm we need to carefully situate all the social and technical elements involved in its composition, and derive an understanding of the strategies and tactics we may employ to make this vision of computing a reality. In the next section we take a step back from pure technical concerns and present a theoretical framework whose intent is to place Third Cloud concepts in a socio-technical perspective.

THEORETICAL BACKGROUD

A network in the context of the Third Cloud is composed of a collection of *ad hoc* and heterogeneous materials (humans, screens, money, airwaves, institutions, processors, spaces, architectures etc.) which resistance has been overcome. The process of *ordering* this collection of material, from which the network emerges, is crucial; the Third Cloud is nothing other than a patterned network of heterogeneous actors, both social and material.

But how can we refer to a collection of heterogeneous elements as the "Third Cloud". In other words, why are we sometimes aware of the network behind a utility (the web for example) and sometimes it appears as a unique homogeneous thing? Why does the network disappear? Why sometimes it is not the case? Appearance of unity comes from simplification. When a heterogeneous network acts as a single block, it disappears, replaced by the action or the task carried or performed by its actor. All internal complexities (sociological, economical, and technical) of the network are masked and become invisible. This simplification process is what I previously called ordering but may be understand as *figuring* [Elias, 1978], or more specifically *configuring*. In the Third Cloud, configuration is a process or an effect, rather than something that can be achieved once and for all. Configuration is not free-standing, like in the Second Cloud, but a site of struggle, a relational effect which needs to be performed, maintained, negotiated, and mediated in place and time by all actors involved in the network. There is no such thing as a single Third cloud configuration in a given space at a given time; there is a plurality of possible configurations which need to be negotiated.

Humans and machines makers have their own preferences. They present resistance to any imposed ordering and are liable to break down, or make off on their own. Therefore, the object of engineering Third Cloud systems is to describe local process of patterning and devise mechanisms to orchestrate configuration. This, then, is the core of this thesis' approach: a concern on how the Third Cloud mobilizes, juxtaposes and holds together the bits and pieces out of which it is composed; how can we prevent those bits and pieces from following their own inclinations and making off; how to manage, as a result, and conceal for a time the process of configuring so an heterogeneous set of bits and pieces turn into something that passes as a single *ad hoc* computing substrate capable of hosting interactions and end user experiences.

STRATEGIES OF CONFIGURATION

But what can we say about configuring and the methods of overcoming resistance? How the Third Cloud, which can be borrowed, displaced, dissolved, rebuilt, scaled and profited from, can generate the effect of agency and organization? What are the tactics and strategies? I approach this task empirically and present qualitative strategies of configuration which directly influence the stability and resiliency of an *ad hoc* and heterogeneous network. These strategies are not mutually exclusive but rather provide a grid of analysis which can be used to examine related work and drive some of the Third Cloud systems design.

1. **Durability: configuring through time**

   Some materials composing the Third Cloud are more durable than others and so maintain their relational patterns for longer. Interaction initiated by an end user with a stranger in the street may not last very long. But interactions embodied in a building, as an infrastructure, will probably last longer. Thus a good ordering and configuring strategy ought to embed a set of relations in durable materials (infrastructure, friendship, membership, etc.).

   However, caution needs to be taken here. Durability itself is a relational effect, not something given in nature of things. It is in fact related to context. Durable materials may find other use; mediate their effect when located in a new network of relations. For example, I may have control over a set of computing resources residing on the 4th floor of the Media Lab in the weekends, but not on week days, when staff and faculty are present on the floor. My agency over this set of resources is somehow variable through time. Thus negotiation and mediation are central to the process of configuring.

2. **Mobility: configuring through space**

   Mobility is about configuring through space. In particular, it is about ways of reconfiguring a given organization in different contexts. The Third Cloud is centered on human mobility. Therefore it must be elastic and agile enough to translate itself wherever these actors may roam or elect domicile. A good configuration strategy would be to embed relations in mobile and pervasive materials so configuration itself may become a type of immutable mobile [LATOUR B.,1987]. Mobile in the sense that it can be transported from place to place. Immutable in the sense that the thing which is transported keeps its integrity, does not loose its features, and can be re-instantiate at anytime. A virus is good example of immutable mobile. Thus the viral process discussed in the previous section is a good mobility strategy.

3. **Representation: configuring through anticipation**

   Configuration is more effective if it anticipates the responses and reactions of the materials to be configured. Here we resist the functionalism and technological determinism which tend to characterize usual distributed system building. The Third Cloud is about computing in context, not in a vacuum, and takes into account all economical, operational and sociological aspects emerging from the interaction of humans, spaces and machines.

   Humans are very much inclined to anticipate the outcome of a given situation. For example, before going to the NYU library to write this paragraph I have anticipated that I would not be able to get in as an MIT student, and consequently not be able to use their resources to do my work. I was exactly right. MIT students are not allowed in the NYU library since no institutional MIT-NYU agreement has been established which would allow me to use their facilities. Why was I right? How could I predict the outcome of this relational circumstance? The answer is simple: through calculation based on representation. Having traveled a fair amount of distance from Boston to New York and being identified as an MIT graduate student lead me to anticipate this rather uncomfortable situation.

Calculation is a set of relations on its own right. But calculation can only operate on some sort of material representation, which is a relational effect. Thus anticipation has to be based on a system of representation. Under appropriate relational circumstances, this system will in fact have important calculation consequences on how a Third Cloud network takes shape. A great example of anticipation in the current networking literature is the revisit and transformation of the "end-to-end" argument [Saltzer et al. 1988] into a "trust-to-trust" one [Clark, D. D, 2009]. Based on identity and authentication (representation), trust (calculation) relationships are established between all actors involved in a given Internet protocol, leading to a more robust network based on anticipation and trust.

4. **Scope**

   The issue of scope in the configuration process also has to be addressed. Although we have stressed the view that the Third Cloud is local, it is possible to envision configurations that ramify through and reproduce themselves in a variety of network instances and locations.

What might such strategies look like? How have they influenced system design in the past? How will they in the future? It is important to note that this approach of theorizing about networks, in a relational and process-oriented manner, is also used in sociology [Latour B., 1988]. In a radical spirit, it is in our best interest to include such theory in the design of systems for the human and socially centered Third Cloud. Treat different materials – people, machines, "ideas" and all the rest – as interactional effects from which social and interactive networks emerge, rather than primitive causes.

3 Towards a systems design

TECHNOLOGICAL BACKGROUND

We shall now look at some of the existing and emerging technologies that could be used to implement Third Cloud systems and applications. Our focus here is to present technologies which could be employed for (1) communication, (2) dynamic configuration and (3) awareness. Some of these technologies are based on existing standards and benefit from a wide adoption on a plethora of end user devices and computer systems. They are also made available on the market as electronic components for the design of custom circuitry. It is now possible to design and develop, as you will see later in the case of the Amulet, low-power embedded devices featuring a combination of these components without requiring substantial investment. As mentioned earlier, the challenge of constructing Third Cloud systems and devices is an architectural one rather than purely technical.

Wireless communication

Device ubiquity could not be achieved without the use of wireless transceivers. There are a large number of these technologies and standards already deployed in current devices and environments. The most commonly available are Bluetooth [Bluetooth SIG], short-range low-power radio transceiver, and Wifi [IEEE 802.11], mid-range high-bandwidth radio transceiver. Wifi presents some advantages over Bluetooth since modern operating systems offer a more flexible set of interfaces to access "low-level" features of the communication stack. Other standards exist such as ZigBee [IEEE ZigBee], IrDA [IrDa], UWB [UWB], and WiMax [WiMAX Forum], but do not benefit from a wide deployment as does Bluetooth and Wifi.

Virtual Machine

In order to dynamically configure the Third Cloud to produce a distributed execution environment, we must abstract single machines and devices. Virtual machine (VM) technologies are especially designed to perform machine abstraction by logically decoupling the execution environment from the machine where it takes place. Untrusted code may be sandboxed by the execution environment, giving mediated access to hardware. With virtual machines, it is also possible to make decisions at run-time on where, how and when a program will execute, rather then at development time.

Virtual machines were invented by IBM as a development tool in the 1960s, to allow two development teams to share a single computer – *computing utility*, and to emulate machines that did not yet exist. The control program responsible for efficient hosting of multiple instances of virtual machines is known as the virtual machine monitor (VMM). According to an early definition [Popek and Goldberg [1974]] "a virtual machine is taken to be an efficient, isolated duplicate of the real machine", with the following three properties:

Efficiency: Unprivileged instructions (e.g. not affecting global machine state) are executed natively on the underlying hardware.
Resource control: A program executing in a VM cannot allocate system resources for itself without going through the VMM.
Equivalence: With the exception of timing and resource allocation, a program executing in a VM should perform just as it would on real hardware.

Types of Virtual Machines

There is two main types of virtual machines: hardware virtual machines and application virtual machines:

(1) Hardware virtual machines

Hardware virtual machines are implemented in two ways:

1. **Hosted VMM.** The VMM is hosted by an operating system (OS). It provides an homogeneous software layer between guests VM and host OS. This type of emulation is performed by providing specific instruction set architecture (ISA) or by directly translating binaries. The VMM does not include device drivers since it operates through a guest OS. This type of virtualization is used on desktops machines where multiple instances of isolated OSes (Linux, Windows, etc) can seamlessly run on a MAC or PC.

2. **Native VMM**. The VMM provides a "kernel" for all other OS to access hardware. This type of virtualization is more powerful (faster execution) and secure than the previous approach, though VMMs must provide their own device drivers in order to operate. This VMM configuration is interesting for embedded system because these have in many cases real-time requirements which are met with this type of "bare-metal" virtualization. A specialized real-time OS (RTOS) can co-existing with an "application" OS such as Linux on the same machine running directly on the emulated hardware without requiring mediation of another OS.

(2) Application virtual machines

Application virtual machines somehow resemble the first category of hardware virtual machines. A VM is hosted on a given OS and provides a run-time environment where programs are executed. Source code is usually written in a specific high-level language (Java, Python, C#, etc.) and compiled to an "intermediate" format composed of synthetic instructions known by the target run-time environment (JRE, Python Interpreter, .NET interpreter, etc). Each virtual machine presents a different instruction set from which programs are compiled to. At run-time, these instructions are decoded, some are translated into native OS commands, and executed accordingly. This "intermediate" representation of programs permits the abstraction of the underlying OS and enables applications to be portable and mobile across a range of systems for which a run-time environment exist.

Virtual Machine Mobility

This leads us to consider VM and code mobility as a paradigm in the construction systems and applications for the Third Cloud. Here we refer to mobility as the ability a running piece of software has to dynamically and physically relocate itself at run-time in a given network of machines. Hardware and application virtual machines provide necessary mechanisms to construct such system. By decoupling execution environment and underlying machines, the migration of executing software from machine to machine is straight forward. However, an important distinction must be made between both virtual machine technologies in regard to what type of software can be moved.

From a hardware virtual machine perspective the input program of a VM is the binary code of an entire OS. The VMM is nothing more than a machine code translator. If a VM as to move to a new host, the kernel, perhaps the file system, and all applications of the OS as to move as well. Consequently the migration process involves a large amount of data which has to be marshaled, transmitted and re-instantiated at run-time, leading to important *downtime*. Here we refer to *downtime* as the time between moments when a VM suspends its operation and when it resumes them on a remote host. Moreover, the VM has usually no mechanism to determine the "high-level" nature and structure of the binary data it manipulates. In other words, it doesn't know about the internal logic (program) and data structure (file) manipulated by the guest OS. In fact, this type of migration is the most *coarse-grain* type of VM mobility.

Migration is becoming a popular feature of virtual machine systems and is heavily used in data centers and grid computing settings [Foster et al., 2002] to manage clusters of machines. It is primarily used for load-balancing processing tasks between machines or as grid computing job containers [Figueiredo et al. 2003, Ruth et al. 2006]. There is a huge amount of research in this field, but the technologies worth noting are VMWare Vmotion [Nelson et al., 2005], XenoServers [Hand et al., 2003], Shirako [Grit et al., 2006] and NomadBIOS [Hansen, and Jul, 2004]. These systems are interesting but not related to our research context.

However, *Internet Suspend/Resume* [Satyanarayanan and Kozuch. (2005)] is another example of system using entire VM migration. The focus here is to support mobile computing. Inspired by the suspend/resume functionality of laptops, this mobility strategy employs VM migration to relocate a complete personal desktop environment wherever a user may be located in the world. A end user may suspend is work machine while at office, travel back home and then seamlessly resume its work environment from his personal machine. In fact, the suspended VM migrates to an intermediate ISM server where it is stored and made accessible through the Internet. When the user wants to resume his environment the new host system will fetch the marshaled VM on the ISR server and re-instantiate its state locally.

The authors advertise here a "carry-nothing" type of mobility in an "infrastructure-rich" environment, where computing terminals are pervasive and made available to the public. However, this vision of mobility is somehow reductive. End users now greatly depend on the availability of computing terminals and performance of their respective internet connections to retrieve entire VM state from the ISM server.

Interestingly, in an attempt to overcome these limitations, researchers have recently proposed a "carry-everything" approach with projects such as SoulPad and Horatio [Smaldone et al. , 2008]. Here the suspended VM is cached on a portable device - USB memory or Smartphone. Some mechanisms have been devised to reduce the amount of data needed to resume a suspended VM [Kozuch et al., 2004]. Although proposing a new paradigm of mobility, the focus is on migrating single desktop images from host to host rather than distributing functionality over a heterogeneous set of machines. In general these strategies' performances suffer from the coarse-grain nature of hardware virtual machines migration (the lowest downtime observed is 45 seconds).

On the other hand, application virtual machines provide, by nature, a finer-grain control over software it interprets. The input of an application virtual machine is not homogeneous machine-code but rather an "intermediate" representation of code – dub byte-code - tailored around a target run-time environment or interpreter. A run-time environment must first reconstruct the logic and data structure of a byte-code program before it can interpret or optimize its content. The optimization phase, which is used to dynamically translate byte-code into machine code, is called just-in-time (JIT) compilation. A more detailed description of the Java loading process is presented in the next section. The point here is that application virtual machine gives a *fine-grain* control over software that can be moved in a network. Rather than operating at the OS level, mobility can be achieved at the code level, and thus providing richer semantics and finer granularity to construct mobile distributed systems.

Applications can programmatically externalize their state and logic to a form which may be later recreated. Mobility is achieved by moving this externalized data and byte-code between machines at run-time. The externalized data format can be optimized to lower transmission cost, leading to better efficiency in terms of downtime and amount of data to transmit. Object mobility is performed in a similar fashion. An object encapsulates code and data as a single structured unit. Mobility can be achieved by moving serialized objects between hosts. However, if one or more threads are executing code inside and object, and thus keeping parts of its state in CPU registers or on the stack, migration must be delayed in order to write back this pending state into the object's activation record. Hence, a well designed object mobility systems will provide mechanism to overcome this problem by performing migration after a program as reach a certain *safe point* in its execution. Code mobility is further discussed in the next section.

Code mobility strategies

This section describes in more details the salient features of code mobility. First, we present two categories of mobility, (1) strong mobility and (2) weak mobility. Second, we describe three strategies that may be employed to perform code mobility in a network, (1) Remote Evaluation (REV), (2) Code on Demand (COD) and (3) Mobile Agent. These categories and strategies are not new [Carzaniga, 1997]. In fact, as we will see in this section, some are heavily used in our current computing systems and responsible for some of the richest Second Cloud interactions presented through web browsers.

Strong mobility - a thread of execution is suspended, migrated and resumed in a remote execution environment. The entire code and execution state is externalized by the thread and recreated at the new location.

Weak mobility – a thread of execution can bind at run-time code which resides in a different location. Code is migrated to the local execution environment and dynamically linked to the thread of execution.

Based on these two categories, here are three strategies which may be implemented to offer mechanisms supporting code mobility in a system:

Remote Evaluation

This paradigm involves both an execution environment *A* , where a thread *T* need to execute a piece of code but *A* lacks the necessary resources to perform the operation properly, and a remote execution environment *B* which posses the needed resources. Thus *T* will send the piece of code to *B* to be evaluated against its own resource. Results will then be transferred back to the originating thread *T.*

This mobile code paradigm is not to be confused by the classic client-server paradigm. In the latter, only functions established prior to the execution of client code on the server side are made available through a given protocol such as RPC [Thurlow R., 2009]. In the former, a client (*T*) can transfer the function itself to the server (*B*) which has no a priori knowledge of the code to be executed. This distinction is somehow very important. The flexibility of recreating executable functions on the fly enables the environment *B* to only expose small, fixed and generic set of interfaces on the network which can be applied to multiple purposes. This enables true *ad hoc* interoperability, as opposed to requiring each party (*T* and *B)* to have prior knowledge of every domain-specific interactions they may be required to perform together. In fact this aspect of code mobility is not only specific to the Remote Evaluation paradigm but also applies to all interaction involving mobile code.

Code on Demand

Involves both an execution environment *A,* which posses a given resource but where a thread *T* lacks the code and logic to properly operate it, and a second execution environment *B* who has the related piece of code. Here *T* requests code to *B* who will, in turn, transmit it back to *T*. Upon reception, *T* will link the given code to its runtime environment *A* and operate the resource accordingly. This paradigm is similar to Remote Evaluation in the sense that no execution state needs to be migrated between *A* and *B*, therefore execution is confined to a single environment. These two paradigms are good example of weak mobility.

Code on Demand is by far the most widely used mobile code paradigm. The Java platform is a great example of a technology presenting code on demand as a central component of its design. The Java run-time environment (JRE) [Lindholm and Yellin, 1996] presents an extensible dynamic class loader who loads Java classes (byte-code) on-demand into the Java Virtual Machine (JVM) [Lindholm and Yellin, 1996]. It enables application to dynamically extend functionality at run-time, rather than development time, through dynamically linked modules. The loading process consist of (1) locating these modules, (2) reading their content (3) extracting the contained classes, representing units of code in the form of structured byte-code, and (4) loading these classes in the local JVM. The Java class loader can be extended in a way that enables a JRE to locate modules over a network. A Java application usually consists of module references (e.g. *import* statement) which are resolved at run-time. Python presents similar functionalities.

Code on demand is also heavily deployed in Second Cloud web technologies delivering rich-internet-application (RIA) to the browser. Javascript [Flanagan and Ferguson, 2002] is a scripting language whose execution environment is tightly bound to HTML documents and their rendering through a browser. Javascript code is usually embedded inside the HTML document or may be located on a remote server referenced through an URL. A Javascript interpreter, commonly called a Javascript engine [Google, 2008], is embedded into the web browser and provides access to specific run-time information provided by the browser (e.g. mouse events, windows size, etc.) which can be linked at run-time through the scripts. When an HTML document gets rendered, its Javascript source code gets resolved and interpreted. Over the last few years, this language as gain an enormous amount of interest from web developers and as becomes the de-facto web language for rich-internet-applications.

One of the main reasons for such success can be explained by the fact that code on demand is used to load balance servers and clients. In other words, by offloading processing to its clients, a server is freed of the costly task of evaluating every single client requests. Moreover, by distributing computational loads away from servers, this paradigm has strong scaling capabilities.

Mobile Agent

Involves both an execution environment *A,* where thread *T* needs to execute a piece of code but lacks the necessary resources to perform the operation properly, and a remote execution environment *B* which posses the needed resources locally. In this paradigm, *T* migrates its entire state and code to *B* and resume execution there, hence accessing the resource locally.

The mobile agent paradigm is fairly different from the two previous ones. An active computational unit is suspended, migrated and resumed. All necessary execution state, code and data is externalized at run-time, transmitted over the network and re-created on a remote execution environment. Here the originating execution environment is completely offloaded of any computation. This paradigm is an example of strong mobility.

In this section we presented Virtual Machine technologies which can be used to instantiate functionality at run-time on a given host. We found that application virtual machines presents a finer grain control over functionality that can be moved in a network in contrast to the most coarse grain nature of hardware virtual machine re-localization. We finally have identified three strategies for moving virtual code in a network and related these strategies some of the Second Cloud systems architecture.

Awareness & Processing technologies

Central to the Third Cloud is the concept of awareness. In this section we are interested in technologies providing functionality to retrieve contextual information from the platform itself or the environment where it operates. Configuration should dynamically react to physical and social context in an optimal and task oriented way.

Sensing has always been an integral part of any computer system involving direct interaction with humans. After all, a keyboard is simply an array of pressure sensors transmitting commands to an operating system, and a mouse is nothing more than just a 2d linear-motion-sensors. Microphones, digital cameras, image processing software and so forth, are all part of an increasingly sophisticated set of sensors that have been added to our computing platforms over time. This increase can be explained by the performance gains of modern microprocessors that are better suited to decode and process real-time sensor data.

Mobile computers, by the very fact they can be moved, change their context of use. This salient property potentially lets software developers design applications and systems that are easier to use. For example, a device could present data which is customized to the current context while hiding commands which are no longer relevant. This type of automatic configuration is called a *context-aware* operation. Developers can differentiate and improve mobile systems over traditional desktop through the use of context as a central part of their application and systems design.

We can divide contextual data into two categories: (1) local platform information - platform sensing, (2) information from cooperating infrastructure - infrastructure sensing.

Platform sensing

Platform sensing is the ability of a system to obtain context from sensors embedded on the device, measuring parameters that do not require additional infrastructure support to interpret. Acceleration, pressure, magnetic field angle, sound level, light level are all examples of information which can be retrieved from sensors embedded on a local platform.

Most of these sensors imply a measure of an analog physical quantity. Any measured value must be recorder in order to be processed. Although analog recording and processing techniques exist, the most flexible recording and processing tool is the computer. An analog-to-digital converter (ADC) interface must therefore be used to convert analog potential difference into a binary number. From a processor's perspective, analog-to-digital conversion becomes a new type of problem: measuring the time it takes for a signal to transition from a logic zero to a logic one. This is now becoming straightforward, with the development of modern microcontrollers based on clock rates ranging from 10 to 300 MHz, derived from high-accuracy crystal oscillators. Typical solution to sensing on a digital platform requires a sensor, an analog interface circuitry, ADC-microcontroller, and connection to a microprocessor's I/O port.

Standardization of serial interconnect buses, such as Inter-Integrated Circuit (I2C) [NXP Semiconductor, 2007] and Serial Peripheral Interface (SPI), has also played an important role in the design of modern sensor interfaces. Although not designed as sensors buses originally, they turned out to be a convenient way for sensor manufacturers to interface physical sensors, analog circuitry and ADC with a standardized interconnect bus. Microcontrollers and System-on-Chips (SoC) usually include hardware support for these standards. Moreover, operating system kernels such as Linux are now providing support for these standards, and thus standard libraries can be used to perform basic communication with the sensor bus, greatly reducing development time. Adding sensing capabilities to mobile computing platforms has never been easier, with accuracy and cost no longer the limiting factor.

Infrastructure sensing

Infrastructure sensing is the ability of a system to obtain context information from other cooperating systems in the environment. We may call this type of information - infrastructural information. Good examples of infrastructural information are location and presence. Both types of information are typically derived from sensing reference points in the environment; RFID tags and infrared transceivers may be used to locate and identify someone in a building, a GPS receiver relies on signal strength transmitted from satellites in orbit around the planet to derive location, a 802.11 wireless adapter relies on broadcast signals from access points (AP) to determine if it can access a network. Two important points regarding infrastructure sensing need to be addressed as part of our Third Cloud system design endeavor: (1) privacy and (2) latency and power consumption

(1) Privacy

While infrastructure sensing technologies provide very useful contextual information to a user, they also raise some important privacy questions which need to be addressed, especially when identity of users may be compromised. Infrastructural information can easily be used to track users and their identities. After all, the cooperating infrastructure is also able to sense users, and thus able to compile sensitive information about them.

Most of the aforementioned sensing techniques use radio frequency technologies as awareness and discovery medium. We know from [Saponas et al., 2007] that RFID present serious privacy and tracking issues. A more generalized privacy concern is raised by [Greenstein et al.,2007] regarding current wireless networks standards (IEEE 802.11, Bluetooth, Zigbee, WiMax). The authors argue that the use of persistent identifiers, such as names and addresses, in probes and beacons to discover services and in data packets are leaking sensible presence information about devices and their network access history. These low level identifiers raise privacy concerns since they can easily be mapped to actual user identities. Recent work in [Greenstein et al. ,2008] propose to eliminate such persistent identifiers in both IEEE 802.11 service discovery and data packets by obfuscating them with a symmetric cryptographic key. Such key needs to be known a piori by both communicating entities in order to decrypt the packets. In [Pang et al., 2007] they present a solution to the creation and exchange of keys between unknown peers. Although incredibly valuable in general, their solution to the creation and exchange of keys between unknown peers is not conclusive enough and still imply strong trust assumption between actual users.

(2) Latency and power consumption

Infrastructure sensing also includes mechanisms by which a device can discover resources and services provided by other devices and systems in proximity. Such mechanisms are known as *service discovery protocols.* Universal Plug-n-Play (UPnP) [Jeronimo and Weat, 2003] and ZeroConf [Williams, 2002] being the two principal protocols.

Although effective in a wired infrastructure these protocols were not designed especially for wireless *ad hoc* networks. They are IP-based (layer-3) and thus rely on mechanisms to establish IP addressing, either through a Dynamic Host Configuration Protocol (DHCP) server or by relying on IPV4 or IPV6 link local address. In a wireless *ad hoc* network, this discovery process is suboptimal. It affects both discovery time and energy consumption of battery operated devices. This problematic is clearly presented by [Sud et al., 2008]. To speed up the discovery process, they propose to integrate discovery mechanisms at the link-layer (layer-2) rather than at the transport-layer (layer-3). By devising a layer-2 beaconing scheme, they clearly show an improvement of speed in the order of 50x over current layer-3 solutions.

But these protocols are primarily designed and optimized to *obtain* resource or service information on a network. The information itself is treated as a second citizen as it only contains a representation of the presence of a resource.

In UPnP the simple service discovery protocol (SSDP) [Goland et al., 1999] is employed to find services on a network. SSDP discovery packets only contain minimal information about services they represent; an identifier and a pointer to its description. Actual information about the resource must be fetch, by performing an HTTP GET, from the pointer (URL) provided in the discovery packet. Here, discovery and awareness processes are separated. First discover the presence of a host, and then get informed about its capabilities. This mechanism is not spontaneous and often leads to a lengthy process. Other discovery protocols such as SLP [Guttman E. , 1999], Bluetooth SDP[Bluetooth SIG, 2009] and Jini [Waldo, J. 2000] present the same inefficiency.

While the system presented in [Sud S. et al., 2008] gives us highly relevant insights on engineering a connectionless (layer-2) resource discovery protocol which is 50x faster than simple UPnP and Bluetooth SDP, it lacks flexibility in representing resource in a way that could assist our mediation process. In their beacon scheme, a resources or services representation is reduce to a minimal bit in a four bytes bitmap, informing only of its presence on a given host.

Decentralized Authentication

By nature, wireless technologies give devices the possibility of directly interacting with each other without being thigh to common infrastructure. Basic *ad hoc* interactions between unknown devices are common practice in the wireless world. These basic interactions can be limited to discovery protocols while more advanced interactions, such as access to a resource, necessitate authorization. Authorization is the process of validating that an authenticated subject as the authority to perform a certain action.

Although authentication is conceptually based on identity, material used in the authentication process is usually based on knowledge; that is something an end-user knows such as a password or a personal identification number. Unlike other domains requiring authentication, such as access to a building or alcohol where authentication and identity is based on device ownership - an ID card, identity is rarely solely mediated by our machines, it usually involves a user in the loop providing that magic piece of knowledge. In the context of the Third Cloud, this authentication process could easily become impracticable. Authentication based on device ownership seems to be a better approach.

In our context, authentication should be performed in a decentralized fashion. Access to centralized server or infrastructure must not be required. It must also be medium agnostic that is not relying on a particular medium or technology providing IDs, such as addresses. It must provide its own distributed naming scheme. Two main system design and implementation are presented here: SPKI/SDSI and Unmanaged Internet

Architecture (UIA).

SPKI/SDSI

Simple public key infrastructure / Simple Distributed Security (SPKI/SDSI) [D. Clarke et al., 2001] is a public key infrastructure providing locally-scoped naming and grouping mechanisms bound to cryptographic keys forming a decentralized and composable namespace. SPKI/SDSI associates public cryptographic keys with human readable identifiers known and managed by a user. To access a protected resource, a client must prove to the server it has the right credentials. The proof is done by means of a *certificate chain* leading to trusted users (principals), where the client's key gets resolved to be member of the same resource's group or has been delegated by another principal part of the resource's group.

Unmanaged Internet Architecture

The Unmanaged Internet Architecture (UIA) [Ford, B et al., 2006] is an architecture aimed at providing ease-of-use secure connectivity among one's personal devices, using the idea of persistent personal names. Similar to SPKI/SDSI, a decentralized namespace is constructed out of locally generated names bound to cryptographic targets without the need of a central authority (CA). In UIA a distinction is made between users, devices and keys. While SPKI/SDSI directly binds principals' identities and keys, requiring users (principals) to manage their keys across devices, UIA instead assign a single key to each device a user may own. UIA devices form user identities out of cooperating groups of personal devices, which the user builds, like in the SDSI case, through simple device introduction and merge. UIA constructs an encrypted overlay network between introduced devices sharing a public/private key pair. This network is use to communicate application information but also as a control channel to manage distributed keys. It provides a gossip and replication protocol to manage naming and group state across devices – introduction, merge and revocation key management is therefore "simplified" and is invisible to users.

Our Design Approach

In this section we describe the formative elements for the design of Third Cloud systems and devices. Based on the previously presented technology enablers, our focus is to establish attributes composing a coherent set of mechanisms relating to configuration strategies presented in section 3.1: (1) *Durability*, (2) *Mobility* and (3) *Representation*.

Discovery and Awareness

Ability of spontaneously retrieve information from the context in which computation takes place is central to our systems design endeavor. Our design goal is for devices to dynamically *mediate* their mode of operation to take full advantage of the interactive resources in proximity. This mediation process as to be *ad hoc* and spontaneous.

We are using the word *mediation* here to express a part of anticipation in the process of configuration. Prior to re-organize its operations, a device should be able to get into relation with its context, establish a set of possible configurations and put them into relation based on calculations specific to the interactive task which must be performed. It may thereafter automatically elect an optimal network configuration by anticipating that its constituent will execute the task with a high degree of fidelity. As mentioned earlier, calculation must be rooted in a system of representation. Although mechanism must be devised to *obtain* information from local device usage and cooperating systems in the environment, a system of representation must also be established making this contextual information valuable to the mediation process. Discovery and Awareness is thus a *Representation* strategy.

Current discovery protocols were presented in previous *Awareness & Processing technologies* section. We argued that their focus is solely to optimize obtainment of resource information in a network, treating actual information as an after thought. Information often only contains resource presence without any rich description. By not defining proper mechanisms to establish a rich contextual system of representation, these protocols rely on end users to compare competing resources. Hence anticipation is always and solely performed by the end user. In the Third Cloud, we wish to provide the necessary spontaneous information for a system to put these resources in relation. In doing so, it will be able estimate the outcome of configuring itself according to a certain pattern. We surely do not want to eliminate the end user from the anticipation equation but we believe Third Cloud devices should at least be able to estimate, based on a set of discriminants, possible patterns of configuration.

The Third Cloud is meant to create a public interaction utility; devices should thereof not only be aware of available resources in a space but also be informed about the socio-economics attributes underlying the usage of the interaction utility. After all, the Third Cloud can be viewed as a site of struggle where trust must be established on the fly and where transactions must be negotiated on the spot. In an attempt to capture these qualities, a system of representation should be based on:

- **Estimable resource description** – Applications should present structural components that could be used to estimate a set of discovered resources by providing a metric. The awareness and discovery engine could then order this set based on a given application requirement. The underlying protocol should be flexible enough to have resources described by a rich set of attributes and provide references to related estimators.

- **System overload metric** – Resources are cooperatively shared among peers. Systems hosting resources should dynamically signal the level of concurrent processes present in the resource's context. For example, my device may anticipate to not displaying my personal content on a screen if shared with other users.

- **Transactional information** – Resources are usually bound to an ownership structure. Control and access over these must be granted according to an agreement between the provider and the client. Pricing, accountability terms and leasing structure are all quantifiable information that could be made available on the fly part of the discovery protocol.

- **Identity** – Resource anonymity and confidentiality should be supported by the discovery protocol. It should provide mechanisms to control the release of attributes (identity of the advertiser, resource description, etc.) on a shared network. Only peers having the right credentials should be able to reconstruct this information. Identity and Authentication are described in more details in the following section.

These principles can form the base of a rich system of representation conferring devices the ability to anticipate self-configuration outcomes. An awareness protocol should be devised integrating the aforementioned principles while drawing form concepts and mechanisms of the connectionless (layer-2) discovery protocol presented in [Sud S. et al., 2008].

Identity and Authentication

Identity can be the base of a strong system of representation. As an effect, durable relations based on trust between identifiable entities may be established. Identity and Authentication are both *Representation* and *Durability* strategies. They are tightly coupled to the aforementioned discovery and awareness process, in the sense that access to a discovered protected resource may be granted only after the requester as been authenticated. It is important to remember that all operations should be decentralized. Access to a central authority must not be required.

Following are the design principles we which to address part of our system implementation:

- *Identity Management* – A device should be able to represent its user in different contexts. When located in a new network of relations one's identity may need mediating to adapt to the new situation. Hence, Third Cloud devices should be able to manage multiple identity representations without requiring users to intervene in the process.

- *Decentralized grouping* – A device should have the ability to locally generate keys and identifiers to create group with other devices on the fly. These groups could form new identities to be managed by the device. Devices should be able to manage group identities without requiring user input.

- *Authenticity* – A device should be able to prove its identity upon challenge. A device should be able to challenge other devices whenever needed. Strong decentralized authentication mechanism need to be devised supporting custom access control policies.

- *Anonymity* – Devices should be able to hide their identity while communicating with each other. In a more general statement, a device should be able to control release of attributes (IP addresses, list of resources, etc.) when transmitting data over a wireless channel.

- *Confidentiality* – Devices should be able to establish a secure channel between each other preventing other listening devices from eavesdropping on their private communication. Content should be obfuscated in a way that only sender and receiver are able to retrieve meaning.

Drawing upon previously presented decentralized authentication and security technologies [D. Clarke et al., 2001; Ford, B et al., 2006; Greenstein et al. 2007], an identity and authentication middleware needs to be devised, that is a system-level software layer sitting between network interfaces and user applications, based on these design principles.

Platform and Code Virtualization

The Third Cloud is a space where an interaction or a task residing on a user's personal mobile device can be opportunistically loaded and executed into the environment making use of private or public devices to carry and amplify the interaction or task to be performed. In the previous sections we have covered system design requirements enabling the mobile device to spontaneously discover and access public or private resources in its environment. The question here is how will personal and public devices interoperate? This simple question begs important architectural implications. Clearly, the two entities must have some shared agreement on some type of interface one export which is know to the other party. There must be *a priori* agreement on the form (syntax) and function (semantics) of communication. The question that is at the core of this thesis is: at what level should this agreement take place in order to have an application residing on a mobile device shape its cooperative surrounding to support its own operation in an *ad hoc* fashion? There are a number of common approaches one might undertake to implement interoperability between a set of networked machines.

One approach is to agree on one or multiple *domain dependent* interfaces. These are typically protocols that are bound to an application or to a specific domain of functionality presented by a device or resource, for example RFB [RealVNC , 2009] and RPC [Thurlow R., 2009]. The obvious problem with such interfaces is there is a numerous collection of them and it is difficult to guarantee their availability in a given environment.

Another approach is to agree on the use of *domain independent* interfaces. That is an interface in which a rich set of interactions and functions are *funneled* into a single protocol. The web uses this approach, HTML over HTTP being an example. There are no different protocols to render a Gmail application page or a Hulu application page even if they present very different functionality. HTML achieves interoperability through standardization. It is nonetheless a highly generic and open interface. Generally, the promise of interoperability on the web is predicated on there being a small and fixed set of protocols and data types know to all entities, which are generic enough to be easily adapted to new needs.

In order to support scalable *ad hoc* interoperability between heterogeneous machines, a small set of generic domain-independent interfaces must be agreed upon, rather than requiring them to present a multitude of domain-specific protocols. These generic interfaces should be multipurpose by providing funneling of functions that could be transported over the network and be placed wherever needed.

Virtual machine technology becomes here an important part of the equation since it provides that level of dynamism and mobility while presenting secure mechanism to sandbox execution of unknown functions, as discussed in the previous *Virtual Machine* section. Application virtual machines are more suited than hardware virtual machine in establishing light-weight interoperability between a set of heterogeneous machines because it offers a finer-grain control over the specific behavior it may place and move. Hence, agreement of functionality which needs to be in a network is determine at run-time rather than development time. This degree of indirection makes the process of configuring interoperability between machines scalable and invites virality.

By establishing a small set of generic protocols able of transporting functionality from one host to another, we are shifting focus from how communication needs to be constructed to *how* functionality extension should be performed and *where* these functions should be placed. We presented three code mobility paradigms in the previous section: Remote Invocation (REV), Code on Demand (COD) and Mobile Agents (MA). These can obviously answer the *how* question. Underneath the *where* questions begs an even more interesting architectural question: should the infrastructure contain most of the "high-level" knowledge on how to operate resources or this knowledge should reside on the mobile device which will, in turn, opportunistically inject it into the infrastructure?

In this thesis we argue for the latter. The former approach is generally used in *Intelligent Environment* (IE), such as MIT's project Oxygen Intelligent Room [Rudolph, L., 2001], Standford's iRoom [Johanson, B. et al., 2002] and CMU's Project Aurora [Sousa, J. P. &

Garlan, D., 2002], where interaction of the application running on a mobile device and the functions embedded in the environment are tightly integrated and controlled. The Intelligent Room is composed of a multitude of resources, ranging from devices to software components, which need to interoperate. The Room has a communication layer which consists of individual software agents. Each agent is tied to a specific resource, abstracting away its "low-level" interface. Agents need to speak to one another to control and coordinate tasks. *Metaglue* [Coen M. et al., 1999] was implemented as a system enabling communication between agents. *Metaglue* agents can communicate by requesting a *stub interface* for another agent, and calling methods directly to the (perhaps remote) agent on that stub. This process of acquiring stubs connected to agents is completely mediated by the central infrastructure which performs book keeping in the form of a catalog of all deployed agents and their related stub interfaces deployed in the Room. Much like Java's RMI [Lindholm and Yellin, 1996], the stub code can be downloaded from the centralized catalog according to the COD paradigm.

Three things worth noting here:

(1) Functionality of neither the client nor the infrastructure is extended, only the means of communicating to the agent are acquired on the fly by the client.
(2) The client application is not mobile; it solely stays on the client's device.
(3) It is not the goal of such platform to provide a set of modular composable service types that can be assembled by end-users in pre-defined ways, and therefore it remains the responsibility of developers to create the applications with which users will interact.

The same arguments hold in the case of Stanford's iRoom Event Heap [Johanson, B. and Fox, A. 2002] and CMU's Aurora [Sousa, J. P. & Garlan, D., 2002]

In contrast, our approach is to dynamically create a network of virtual machines nodes, composed of proximate devices, and move executing processes into these nodes. Rather than passing function parameters among nodes, we migrate processes. Clearly, infrastructure should have all the "low-level" knowledge to operate local resource, such as device drivers. Functionality should be exported through a set of APIs – which is currently case with standard Java or Python. The "high-level" abstraction of these – classes using them to extend functionality – should reside on a mobile device and moved to the desired resource context whenever needed by an application. *Dynamic binding* can thus be used at run-time to connect "high-level" code to "low-level" implementation. This way, interaction or a task residing on a user's personal mobile device will effectively be migrated into the environment extending its functionality on the fly, rather than explicitly using references to remote "high-level" objects already deployed in the environment, to carry the task or interaction. This distinction is important. In some ways, our approach uses a mix of Remote Invocation (REV) and Mobile Agent paradigms rather than solely relying on Code on Demand (COD) as does *Metaglue.* Moreover, since the mobile device initiating interaction with its environment posses the necessary code to instantiate desired functionality on a given host, there is no need to rely on a central "catalog" to provide such knowledge. It can happen on a peer-to-peer base and thus present a high degree of extensibility and scalability.

Virtualization is an answer to the *Mobility* strategy. Through the virtualization of code and computing context, a viral application on a mobile device is able to instantiate functionality in its environment; functionality which was not known to the environment prior to the interaction. Guidelines concerning the authoring of such application are described in the next section.

Authoring framework

The Third Cloud is composed of an ad hoc heterogeneous set of materials whose resistance has been overcome by an ordering process. We called this process configuration and derived that the Third Cloud is simply a patterned network of heterogeneous material. Configuring is the process of simplifying the complexity of the underlying patterned network, so it may appear as a unified block while hosting a task. In the previous sections we presented attributes and design guidelines answering the need to devise configuration strategies such as *Durability*, *Mobility* and *Representation.*

These building blocks could form the base of our system but still need assembling. How can we orchestrate configuration and express a task and interaction – in a meaningful way?

Interesting concepts regarding serendipitous discovery of resources and the assembly of patterned networks are present in [Want R. et al., 2008]. They present a dynamic authoring framework intended to enable a mobile device to spontaneously assemble a logical computer from a set of wireless components available in proximity. Their focus is to improve connection set-up of wireless devices by providing a user interface (UI) enabling the aggregation of remote services recquired to configure a user's platform. The system abstracts away devices properties by advertising services on the wireless network. These services are being presented to a user through the UI helping him to navigate and configure his network. A handful of domain dependent interfaces such as VNC [RealVNC, 2009] are being advertised as services.

Although quite similar to our endeavor of assembling a computing substrate out of wirelessly connected entities, our respective approaches greatly diverge. They are mainly focused on abstracting connections between machines supporting a fixed set of well known services to help configuration of a wireless network, while we are focused on spontaneously extending functionality of machines by abstracting computation needed to perform an interactive task. In our approach, the task is made explicit and drives the underlying need for a patterned network to take form. In their scheme, task and configuration is disjoint. They mention this task oriented modular composition strategy but do not propose any solutions.

As explained in the previous section, most of the knowledge on how to operate needed resources should be placed in mobile device's applications. In doing so, we guarantee they can self-operate in a multitude of contexts by pollinating as needed their environment with functionality. Hence, they should be able to self-configure their network. In our scheme, task and configuration are partially blurred into the same thing. Therefore, composing configuration as to be related to the authoring of applications. Task and configuration process should be part of the same programmable unit capable of self-operating in multiple contexts.

An authoring framework should be devised combining both application and configuration assembly. It should be tied to *Awareness and Discovery* mechanisms and base interactions with other cooperating system on *Identity and Authentication* concepts. It should allow a high degree of expressiveness and flexibility to match diverse application design models, while being able to incorporate a wide variety of context awareness.

4 Substrate

In the last chapter we presented our design approach intended to support spontaneous configuration of cooperative devices in a Third Cloud network. This configuration was oriented towards the execution of a task or interaction introduced by a user on its mobile device. We blurred the boundary between task and configuration and derived that a authoring framework ought to be devised supporting the authoring of self-configurable applications.

In an effort to address this issue I have developed *Substrate*, both an authoring framework and execution environment intended to abstract low-level aspects of distributed and context-aware software engineering. It is intended to present the necessary paradigms and tools to support an interaction design practice while providing an environment for software development. It combines a computing model tailored to support the expression of context based logic and a high level language enabling composer to easily script applications and generate optimized executable code. In addition, *Substrate*'s execution environment enables software to migrate from machine to machine informed by a resource discovery mechanism, enabling serendipitous interoperability between cooperating machines. It integrates (1) *modular composition*, (2) *discovery and awareness* and (3) *virtualization* elements presented as formative attributes of Third Cloud systems.

Substrate is still at an early prototyping stage, but its development has so far provided insightful information about the functionality necessary to compose context-aware applications able to opportunistically span execution on a set of heterogeneous and proximate machines. Functionality is iteratively added to the framework as new needs are discovered developing applications.

In this chapter I will discuss the design principles and motivations behind the development of *Substrate,* its computing model, the language in which applications and configurations are composed and transformed into executable code and its distributed execution environment.

Design Principles

Authoring framework

The Third Cloud represents a space of interaction where people using their mobile devices are able to inject human-centered interactive tasks in their surrounding infrastructure. While providing system and software engineering support for the construction of these tasks, *Substrate* authoring framework should also propose abstractions and tools to support an interactive design process. Interaction design is the discipline of defining the behavior of products and systems that a user can interact with. Therefore the framework should present the following attributes:

- Unify low-level software engineering and high-level application design
- Have ability to model a wide variety of complex contexts and configurations
- Assist dialog between interaction designer and the emergent application
- Facilitate interaction design transfer to technologists
- Allow an iterative design process

Awareness and discovery protocol

The protocol should seamlessly integrate with the authoring environment. An application composer should be able to describe a resource which needs to be discovered in a simple manner. Resource semantics need to be strongly defined. Control flow of an application should also be based on the discovery process.

The protocol needs to integrate design guidelines explained in the *Discovery and Awareness* section of chapter 3.

Computing Model

Central to *Substrate's* modular composition paradigm is an *automaton* computing model giving an application quality of a self-operating machine [Arbib, M. A.,1969; Ginsburg, S., 1982]. This automaton model gives us a powerful abstraction to describe and understand programs in a formal way.

A finite state automaton is composed of a finite set of states linked together by transitions. Given an input event, the automaton will transition from one state to the other following the dynamics of its description. More formally, a finite state automaton is composed of a 5-tuple $(Q, \Sigma, T, q0, F)$ consisting of:

- a finite set of states Q
- a finite set of input symbols Σ
- a transition function $T: Q \times \Sigma \rightarrow Q$
- a start state $q0$
- a set of final states $F \subseteq Q$

In our current model, a transition $T: Q \times \Sigma \rightarrow Q$ is governed by a set of conditions. Input symbols can therefore, embedded in a condition, represent actions or event that can be either true or false. Therefore our finite state automaton can be defined as non-deterministic; that is for a given state, there can be zero, one or multiple transitions for a given input symbol.

The rationale behind our decision to impose a finite state automaton computing model in *Substrate* is three fold:

(1) A finite state automaton's logic can be represented as a state diagram. This way of representing computation is denotative and expressive enough for a non-computer scientist to clearly understand a system's or application's behavior and control flow. A finite state automaton and its related state diagram gives the

right semantics to compose, alter, and analyze self-operating software.

(2) An application can be modeled by defining states in which output actions occur and transitions between states triggered by a *context-change*. In a way, applications written with this model have the possibility to be *context-reactive,* where control flow is governed by context changes. Transitions and conditions can have as input information from an awareness engine (sensing and discovery). An automaton model gives all the necessary tools to encapsulate all the necessary dynamics an application needs to reflect its immediate context.

(3) We can easily describe a self-migratory machine capable of relocating itself in a network according to the automaton's configuration. States can represent machine location and transitions migration checkpoints. Conditions can receive as input information from a discovery engine, and thus trigger migration upon detection of a given resource or service present on a network. An automaton could even clone itself (or part of itself) and relocate it on a remote site.

Core implementation of the automaton model was written in Python using the object-oriented paradigm. Base classes such as *States*, *Transitions*, and *Conditions* can be derived into new classes, providing specific functionality or attributes to these basic building blocks, which can be instantiated in different locations in a network. A state machine engine part of *Substrate*'s execution environment, discussed in details in the next section, uses polymorphism mechanisms to operates these new objects. Base classes present basic finite state automaton functionality such as: on state entry actions, on state exit action, on state suspended action, on state resumed action, on transition action and test condition action. These all provide placeholders for subclasses to extend functionality tied to the automaton's self-operation.

In structuring *Substrate*'s code base development this way, we isolate extendable functionality at specific points, while keeping the control flow under the govern of the automaton. For example, awareness and discovery mechanisms only have to subclass a *Condition* or a *Transition* in order to seamlessly integrate our model. Migration mechanisms will derive from a *Transition*, while stable and more durable interaction with a user may inherit from a *State*. The process of augmenting core functionality is thoroughly automated and described in details in the following section.

Authoring Environment

In an effort to provide semantics to compose self-configurable viral application based on *Substrate*'s automaton computing model, we devised an authoring framework consisting of a formal scripting language, a suite of code generators, and a build system, providing all necessary tools to rapidly assemble applications and extend core functionality of the base engine. It conceals both application scripting and library development in a unified framework.

Substrate's authoring framework is conceptually separated into two layers: (1) a descriptive layer and (2) a constructive layer. Both applications and libraries are expressed and constructed using mechanisms related to these two layers. This separation enables abstraction of the semantics used to describe a self-operating program from its actual implementation into an executable and visual format. A detailed description of our implementation is provided in Appendix A.

The descriptive layer constructs an intermediate representation of an application or library which is *implementation agnostic*; that is, not depending on any implementation details such as programming language (C++, SVG, Ada, etc.) or physical machine architecture. It provides mechanisms to bind library definitions with application scripts, enabling application composers to easily and quickly assemble functionality based on a set of specialized libraries.

The constructive layer transforms intermediate representations into executable code and visual representation. It is tightly tied to specific implementation details such as programming language and physical machine architecture. By using code generation as its central mechanism, our constructive layer is able to produce a suit of application script validators and code-generators based on custom library definitions.

The key proposition of *Substrate*'s authoring framework is that it can be configured and extended to match the diverse design models of application composition, while being able to incorporate a wide variety of context awareness and code mobility, anytime during the prototyping of applications.

Execution Environment

Substrate's execution environment consists of two main components: (1) Awareness and Discovery protocol and (2) Execution migration system. Both of these components augment the functionalities of applications written with *Substrate*'s authoring environment. Specific libraries were constructed to integrate awareness and discovery into the automaton computing model. The automaton is also used by the migration execution system to move threads of execution from local machines to cooperative hosts at run-time.

Awareness and discovery protocol

Awareness and discovery is a central part of the Third Cloud architecture. The focus of the Awareness and discovery protocol is on situations were a mobile device needs to query a space about resources made available by cooperative nodes. Mobile device and cooperative nodes need to be in the same physical area – one hop away from each other. Two approaches are possible to communicate discovery information from the nodes to the mobile device on a wireless medium. Respectively using:

(1) Beacons – are periodic transmissions used to communicate resource information on a given wireless medium. Each node pulses information while the mobile device listens for incoming data on the same wireless channel. If it hears a beacon of interest if can initiate a connection with the node. This scheme presents some advantages in terms of energy consumption since a node can

sleep between pulses and only listen to incoming connections for a short period of time after transmission of the beacon.

(2) Probes – are request sent by the mobile device to the collection of nodes petitioning their resources. Upon reception of such requests, the nodes can reply back to the node directly with their respective resource information. In this scheme, nodes need to actively listen to possible incoming probe requests – perhaps wasting more energy than the previous beacon scheme.

Our current implementation of the awareness and discovery protocol uses a beaconing scheme to inform an environment of the presence of shared resources. An in depth description of our beacon implementation is present in Appendix A

The information shared in the protocol is based on the guidelines presented in the *Discovery and Awareness* section of chapter 3. More precisely it conveys the following:

- Resource name hash – generated by *Substrate*'s authoring environment linking resource to code semantic (class name).
- Resource description – defines the resource in a format understandable to the resource estimator.
- Resource estimator name hash – generated by *Substrate*'s authoring environment linking resource estimator to code semantic (class name).
- Overload metric – custom field, represents resource's overload level.
- Transactional information – custom field, represent pricing in our experiments.
- Callback address - address of Substrate's migration execution system.
- Valid period – period of time in which the beacon is valid.

Resource representations are tightly coupled with *Substrate* authoring environment's classes and namespace in order to provide the necessary interoperable semantics between mobile devices and infrastructure nodes. A node needs not to posses the entire authoring environment locally per se but rather a subset of the generated classes representing resources to be advertised on the wireless medium. As matter of fact, a

Resource class has been added to *Substrate*'s authoring environment. A *Resource* is a base class – similar to *State, Transition* and *Condition* - which can be extended by library developers to represent actual resources. When auto generating code for a *Resource* subclass*, Substrate*'s authoring environment generates four key features (1) an integer *hash* based on the class name, (2) *compare* function stub used to compare a pair of resources, (3) a *serialize* function stub used to generate of description of the resource, and (4) a *deserialize* function stub used to construct a resource from a description. These classes can be minimal, that is only containing minimal information about an available "low-level" library driving a given resource. For example, a *Substrate* library developer could write a resource class to signal that a given platform as access to a Wiimote [Nintendo.com] through Python bindings. This class would not require any functionality; only signifies through its class name (*Resource name hash*) that code using a Wiimote through Python can dynamically bound the local platform to consume the resource.

Thus an actual node connected to a Wiimote, and setup so it can be accessed through Python, could then advertise this resource description. In turn, a mobile device entering the environment where the node is situated would be spontaneously informed about the presence of this type of resource. If the resource is required to execute an interactive task, then mobile device could initiate code migration to the node's context by using the given callback address in the beacon - migration is explained in the following *Execution Migration System* section.

Resource class names are converted to hash integer datum using simple non-cryptographic hash functions (in our implementation java.lang.String.hashCode) . The requirement of having collision free hash function is relaxed here since *Susbstrate* already provide a namespace preventing class name collisions.

Awareness and Discovery Engine

A system wide *Awareness and Discovery Engine* constructs and retrieves discovery beacons and acts as a mediator between applications and the network interface. It keeps track of all the available local and remote resources and provides the means to advertise resources on the network. More precisely, it presents methods to dynamically publish/unpublish beacons and update their content, methods to query if a given resource is available on the network and can signal an observer when a new resource is discovered.

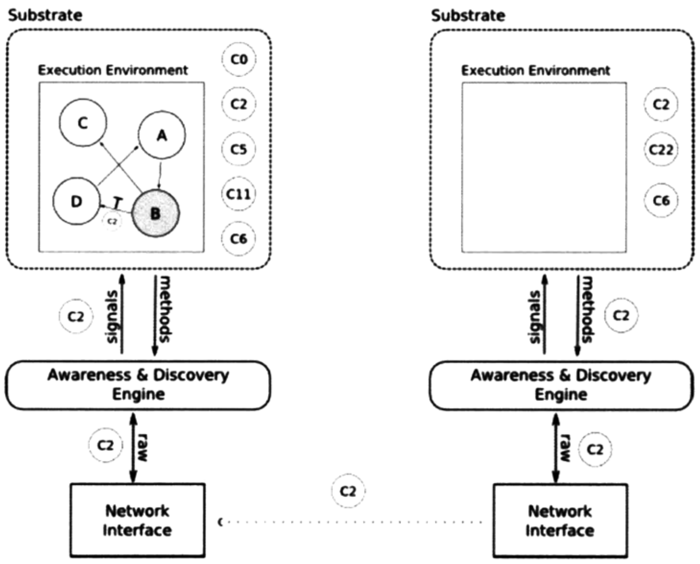


Figure 4.2 – Diagram illustrating Discovery and Awareness process

Local resources are advertised through the *Awareness and Discovery Engine.* Node wishing to make a resource available on the network, calls the *publish* method of its local engine and provides all resource's information as arguments. Local resources are removed from the engine's local resource table by invoking the engine's *remove* function with a valid resource name. Advertisement of local resources on the wireless network is done periodically by constructing beacon frames of all the local resource present in the local resource table at that time, and transmitting them through the network interface. An in depth description of our engine's implementation is present in Appendix A.

As part of *Substrate*'s default authoring libraries we provided the necessary software components to construct finite state automata based on the *Awareness and Discovery Engine* functionality. Special *Condition*s are made available to connect applications with *signal*s in order to trigger an event when a resource becomes available (unavailable). *Transition*s are specialized to observe the presence of a given resource and initiate a migration upon discovery. Upon initialization, the awareness engine can also automatically publish a set of local resources provided in a special configuration file.

Negotiation Protocol

Rather than relying solely on the end user to navigate and compare interactive resources utilities available in its environment [Want, R. et al., 2008], we devised a negotiation protocol enabling an application to externalize its preferences and a mechanism to automatically elect a host presenting a "best" match. This way, network patterning and configuration relates to the application's composition and task to be performed.

Substrate's authoring environment gives application composers the possibility of defining and referencing a *Resource* preference in their application script (see *figure 4-3*). This preference definition is used to compare competing discovered resources in order to infer a best match. Resource preference type and discovered resources types (or class) should match in order to be compared properly. Type matching is enforced by the *Transition* component which implements both the negotiation protocol and the discovery process. A *Transition* only observes *Resource* types which it can compare to its preference's *Resource* type. As illustrated in *figure 4-3*, our specialized transitions are constructed by providing a reference to a resource preference. In turn, the transition periodically requests a list of discovered resources from the *Awareness and Discovery Engine* containing the same *Resource Name Hash* as its preference. Here we periodically requests a list rather than connecting to a signal since we want to compare from a set of competing descriptions as opposed of getting discreetly inform upon new arrivals.

```
<ListOfScreens>
   <Screen name="Big" resX="2560" resY ="1600"/>
   <Screen name="Small" resX="320" resY ="480"/>
</ListOfScreens>
...
<TransitionMigrateOnResource name="ToDisplay" nextstate="Display" resourceRef="Big"/>

<TransitionMigrateOnResource name="ToDisplay" nextstate="Display" resourceRef="Small"/>
```

Figure 4-3 Application code illustrating resource referencing in TransitionMigrateOnResource.

We decided to implement negotiation at the *Transition* level, in our automaton model, since discovery of resource constitutes a context change in the environment from the perspective of a mobile device. Consequently, a sensed context change is translated as a state transition in a mobile application. Therefore the base *Transition* was extended to support both negotiation and discovery. Negotiation is performed on the device seeking to find a resource matching a given preference.

Algorithm

The negotiation algorithm selects the 'best' resource variant by a process of elimination. It is somehow similar to the concept of *Transparent Content Negotiation in HTTP* [Holtman, K. & Mutz, A., 1998].

(1) Requests a list of discovered resources from the *Awareness and Discovery Engine* containing the same *Resource Name Hash* as the preference. For each entry in the list construct a *Resource* from the provided *Resource description* using the preference's *deserialize* function. If the list is empty proceed to step 3.

(2) Apply each of the following tests in order. Any variants not selected at each test are discarded. After each test, if only one variant remains, select it as the best match. If more than one variant remains, move on to the next test.

    (1) Select the variants producing the smallest result from comparing with the preference.
    (2) Select the variants with the smallest transactional information.
    (3) Select the variants with the smallest overload metric.
    (4) Select the first variant of those remaining.

(3) End.

The algorithm is periodically executed until it finds a "best" match. We decided to order discriminator tests in this order - (1) description match, (2) transactional information, (3) overload metric - but we can easily fiddle with their order to accommodate different negotiation styles.

Execution Migration Process

A migration of functionality usually occurs when a context change is sensed in the environment. This change may be initiated by the user himself (pressing a button) or by discovering a resource context in the vicinity. As we mentioned earlier, a context change in the environment is translated into a state change in our automaton model. A *Transition* is the elements representing state change, linking *current state* and *next state*.

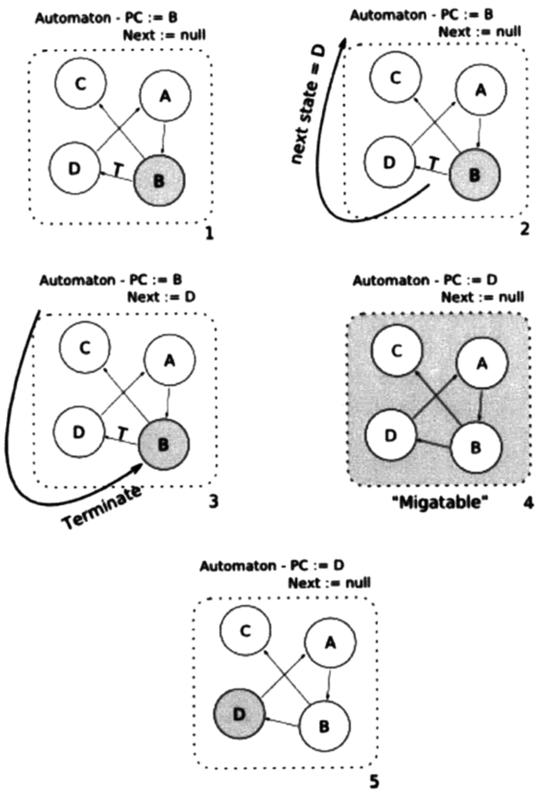


Figure 4.4 – Diagram illustrating automaton migration process

When a given *Transition*'s conditions are met, it signals the automaton about the need to perform state change by providing it with the *next state* identifier. The automaton then terminates the *current state* and switches control to the *next state* (see *figure 4-4*). The automaton's state identifier acts as a sort of program counter (PC). Control switch is the moment where execution thread relocation occurs, since it provides a *safe-point* to perform migration as explained in the *Virtual Machine* section of chapter 3. At this moment of the automaton's execution, all threads initiated by the *current state* are terminated and only a single thread of execution remains - the one on the automaton. Having the PC and execution control, the automaton can be suspended, marshaled, and sent over the network to a remote host, keeping all its state information in the process. On the new host, it can be instantiated, resumed, and control can be granted to the *next state* identified by the PC.

A *Transition* can be fired by a set of related *Condition*s or directly by itself. An example of self-initiated transition was described in the last section where specialized transitions were devised to discover resources and elect a host presenting the "best" match in regard to a resource preference. We provide specialized *Transitions* part of *Substrate*'s default components supporting both cases, condition initiated or self-initiated transitions.

Discussion

In this chapter we presented *Substrate*. Relating to design approaches highlighted in chapter 3, *Substrate* considers the three design points of *Authoring, Discovery and Awareness*, and *Platform and Code Virtualization*.

An automaton computing model was proposed to be at the core of these three mechanisms. The automaton model provides the means to construct self-operating machines capable of capturing and reacting to context changes in the environment. It also presents powerful semantics to express and understand computation modular composition. A state diagram representation can be derived from a formal automaton description, giving an evocative vocabulary for non-programmers to clearly understand a system's or application's behavior and control flow. Self-migratory machines can be described using this computing model, framing code virtualization and migration in a self-operable structure.

Substrate's authoring framework was constructed as a composition platform featuring an extensible "high-level" language based on the automaton computing model. Armed with a suit of code generators, the authoring framework unifies library software engineering and application scripting. By featuring a set of extensible classes, the framework's code base can be augmented with functionality that can automatically be made available to application composers through "high-level" language extension. More detailed examples of application scripting and functionality extension are presented in chapter 6.

The goal of the framework is to construct a set of modular and composable service types that can be assembled by end-users in pre-defined ways, minimizing the responsibility of the software developer to solely create the application with which the user interact. We approached this goal by proposing a modular scripting language. We understand that end-users may not be able to script applications themselves, but we argue that the scripting language brings us half-way there. In fact, the current language could easily be used by interaction designers, with a minimal knowledge of web programming, to prototype complex distributed applications. Interestingly, we could use state diagrams to generate applications. In the current framework's implementation, these blueprint diagrams are generated from scripts which are edited by hand in a text editor. In an effort to address end-user composition, future development of *Substrate*'s authoring environment could rather present state diagrams as an editing tool and generate a script according to the visual representation of an application.

The *Awareness and Discovery* protocol presented in this chapter was devised to feature an extensible and representation rich vehicle for resources advertisement and discovery. Entrenched within *Substrate*'s semantically rich authoring framework, resource descriptions can be evaluated and ordered according to a preference. An application can externalize this preference and a negotiation protocol will elect the "best" resource made available in a given environment. Our current algorithm is somehow very simple and does not involve negotiation between the advertiser and petitioner. Future development of the protocol could implement such interaction. Discovery behaviors were easily integrated into the automaton model by providing special *Transitions* capturing discovery-based context changes.

In basing our implementation on the Python programming language (see Appendix A) we partially answered the design point of *Platform and Code Virtualization*. However, as mentioned in the introduction of this thesis we are interested in the *viral process* as interoperable mobile entity. We were not simply interested in moving bare code from machines to machines but rather moving self-operable thread of executions. This is the reason why we based our computing model on an automaton in the first place. In this chapter we presented the mechanism that migrates an automaton from a context to another. Functionality may be extended in an environment by injecting automata code in its fabric. As explained in the authoring framework section, by deriving a subclass of a *State* one can add desired functionality to an automaton which can get installed on a cooperative machine, informed by the discovery process, using the aforementioned automaton migration mechanism.

5 The Amulet

The last chapter we focused on the development of a system considering design guidelines relating to *Authoring framework, Awareness and discovery*, and *Platform and Code Virtualization*. We showed that *Substrate* provided the necessary mechanisms answering these guidelines. However, in chapter 3 we also derived that *Identity and Authentication* needed to be part of our Third Cloud systems design endeavors.

In this chapter we present the design and implementation a custom portable device part of the Third Cloud ecosystem. The Amulet is a personal device capable of acting on behalf of its owner in a multitude of contexts, thus addressing the guideline of *Identity and Authentication.* It is also the natural habitat of Substrate, and provides us with a fully open source embedded systems prototyping platform.

In this chapter I will discuss the design principles and motivations behind the development of the Amulet, its hardware and software architecture, and the systems and protocols involved in the *Identity and Authentication* mechanisms supported by the device.

Design Principles

Here are the driving design principles for the development of the portable device:

- *Authentication* - The key feature of the Amulet is that owners always have it with them. The device is able to prove their identity to other systems.

- *Interoperability* – Presents high degree of interoperability with other computing systems and offers the possibility to use multiple radio interfaces.

- *Context awareness* – Able to establish context in which its user is situated. This context may be related to the environment, other people close by, and perhaps the purpose of its user.

- *Openness* - Open platform featuring an open source operating system and present the means for software and hardware developers to easily augment system and device functionality. It functions by using open and extensible protocols.

- *Discreetness* – No need to present advanced user interface, it is rather able to parasite interactive resources from its environment. It is not a voice oriented device.

Open Platform

The motivations behind development of the device are diverse. Only few years ago the most common mobile operating systems were Symbian [Symbian.org], Windows CE [Windows.com], and Brew [brew.qualcom.com]. While these offered application programming interfaces and software development kits, they did not propose the level of interoperability with desktop machines we were interested in. Moreover these OS are proprietary, thus access to system code is not straightforward. Then came along open source OS based on a Linux [kernel.org] kernel. It was only a matter of time to see the advent of GNU/Linux [gnu.org; kernel.org] in the mobile OS domain, and more importantly see a standardization of platforms between computer OS and mobile OS (Windows, Darwin and Linux).

However, as mentioned in the design principles, the Amulet is not a phone. It resembles more a mobile single board computer than a cell phone. In this area, fewer products are available; Gumstix' [Gumstix.com] "computer-on-module" being one of the top contenders due to its small form factor. One minor problem with the platform is that it does not provide lithium-ion battery charging. The major problem with Gumstix is that it is not truly open source – as open source hardware. The main computer-on-module schematics are not part of their distribution. Part of the current implantation of the Amulet was the principle of having all parts of the system open – software and hardware.

Our first custom implementation of the Amulet provides us with fully open embedded prototyping platform. Most of the technologies presented in chapter 4 are currently deployed on the device. It presents basic functionality to operate in the Third Cloud. We tailored a custom GNU/Linux distribution and augmented it with Substrate functionality. Ubiquitous device designers wishing to make a custom piece of hardware available to the Third Cloud can use the open source schematics, printed circuit board layout, bill of material, and Linux distribution to augment, modify or directly manufacture the device. Such initiative is somehow at the core of this project.

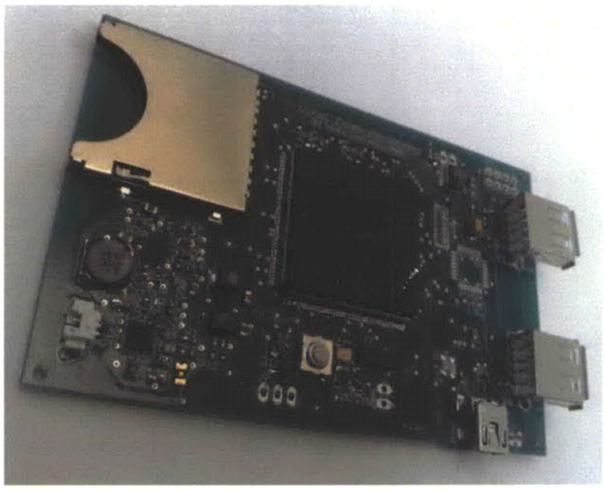


Figure 5.1 – The Amulet version 1.0

Hardware

We chose hardware components that were freely available (e.g. no non disclosure agreement attached). Here is a list of the core features and peripherals:

- ATMEL AT91RM2000 microprocessor (ARM9 core with MMU)
- 32 MB SDRAM
- 8 MB SPI Dataflash
- Lithium-Ion polymer battery charger through USB
- 2 x USB 2.0 Host ports
- 1 x USB 2.0 Device port
- 1 x SD Memory card socket
- 1 x ISO7618 (smart card) interface
- 1 x USART port
- 1 x JTAG interface

A more detailed list of components, schematics, and board layout are available in appendix A.

To address the design principle of *discreetness*, the current version of the Amulet does not present a screen. It rather parasites on other visual interfaces a user may carry with it (mobile device, laptop, etc) using *Substrate*'s code migration mechanism.

We decided to embed USB host ports in order to be able to "plug n play" different radio interfaces on the device. This enabled us to prototype applications based on Bluetooth [Bluetooth SIG], Wifi [IEEE 802.11] and IrDA [IrDa].

Software

The Amulet currently runs a Linux 2.6.27 kernel [kernel.org] and a custom "Viral" OS distribution. OpenEmbedded [openembedded.org] was used to construct the ARM cross-compiler, GNU cross-platform toolchain, librairies and applications.

All necessary software components such as Python and dbus [freedesktop.org] are part of the "Viral" OS distribution in order to deploy *Substrate* on the device. The *Interoperability* and *Context awareness* design principles are met by the *Execution Migration Process* and *Awareness and Discovery Engine* provided by *Substrate.*

Identity and Authentication

Central to the Amulet is the concept of identity and authentication. Users of the device are being "personalized" by it; user's memberships and relationships are recorded, embedded, and mediated through the device. Relating to work done in [Ford, B et al., 2006] on personal devices, we decided to assign a single identity key to each Amulet, rather than abstracting identity from devices and thus requiring the user to inform the Amulet about its identity each time he uses it [D. Clarke et al., 2001]. The Amulet forms user identities out of cooperating groups of Amulets, which the user builds through simple device introduction.

The design and implementation of *Identity and Authentication* mechanisms on the Amulet has been developed in conjunction with a partner company that provided us with a dedicated cryptographic hardware module – in the form factor of a smart card. This module greatly eases the authentication process since it is manufactured with embedded secret knowledge, and thereof solves some of the problems of key exchange and introduction of unknown devices [Pang et al., 2007, Ford, B et al., 2006].

In the next paragraphs, we present three authentication protocols. The Fiat-Shamir Authentication is the basic form of authentication where upon challenge, an Amulet is able to prove it posses the hardware module to another Amulet. The Secure Channel protocol enable a pair of Amulet's to establish an encrypted communication channel between each other. And finally we explain the protocol behind the creation, verification and revocation of group memberships between collections of Amulets.

Fiat-Shamir Authentication

The Fiat-Shamir Identification scheme was introduced in [Fiat A. and Shamir A., 1988]. The authentication used with the smart card is based on an improved version of this scheme, described in [Micali S. and Shamir A., 1990].

The Fiat-Shamir authentication is based on the intractability of factoring – the assumption that it is hard to find the prime factors of a composite number. It is a Zero Knowledge protocol between a *Prover* and a *Verifier*; where the *Prover* has to prove that he knows some secret s, related to a public value v. The protocol use the following parameters.

n	Common modulus, known to both *Verifier* and *Prover*. It is selected as a multiplication of two secret primes p and q, $n = pq$
v	The *Prover's* public key. v is derived by applying a non-secret function – $Id \rightarrow V$ to the smart card ID. This allows the *Verifier* to link the smart card ID to its pubic key when performing the authentication.
s	The key related to v and known only to the *Prover*. It satisfies the following conditions: • It is co-prime to n • $1 \leq s \leq n$-1 • $v * s^2 = 1 \bmod n$

Protocol

This section presents the protocol implemented between the *Verifier* and the *Prover*. In this implementation, the smart card comes already manufactured with the knowledge of *p*, *q*, and generates the secret key *s*.

(1) The *Prover* selects a random number *r*, $1 \leq r \leq n-1$ and sends the *Verifier* $h(x)$, where:
   - $x = r^2 mod\, n$
   - h is a collision free hash function.

(2) The *Verifier* randomly selects a *challenge* bit $e = \{0,1\}$, and sends *e* to the *Prover*.

(3) The *Prover* computes and sends back to the *Verifier* the value $y = rs^e mod\, n$ with:
   - $y = r$, if $e = 0$ (trivial)
   - $y = rs\ mod\ n$, if $e = 1$ (trivial)

(4) To confirm the proof of knowledge the *Verifier* does the following:
   - Rejects the proof if $y = 0$
   - Verifies that $h(x) = h(y^2 v^e)\ mod\ n$, with
     - $x = y^2$, if $e = 0$
     - $x = y^2 v\ mod\ n$, if $e = 1$, since $v = \frac{1}{s^2} mod\ n$

The above protocol is repeated k times so that the probability of being fooled by a cheating *Prover* is $1/2^k$. A legitimate *Prover* with a valid smart card and correct unique key s will always answer all possible challenges correctly.

This simple authentication protocol can be performed on the clear since no secret information is leaked. It is used to verify that a remote communication peer is an Amulet.

Secure Channel

The establishment of a secure channel is intended to secure the communication between two Amulets; more precisely it is intended to establish a secure channel between an Amulet, and the smart card of a second Amulet.

The primary purpose of establishing a secure channel is to prevent an attack in which an unauthorized "intruder" eavesdrops on the communication line between the two Amulets, and can either obtain information for which it is not authorized, or inject information.

The proposed scheme integrates encryption and authentication of the communication between an *Amulet A* and an *Amulet B* based on public cryptography and symmetric keys.

Handshake Protocol

The *Amulet A* shall perform the *"Initial Handshake"* when it first starts "talking" with the *Amulet B* and upon every smart card reset thereafter. The handshake consists of the following stages:

(1) **Secure Channel Initialization** – *Amulet A* sends a *Request for card ID* to Amulet B. *Amulet B* sends back its card ID that Amulet A will consider as a *Partner ID*.

(2) **Secure Channel Configuration Check** – *Amulet A* sends a *Request of Channel Configuration* to *Amulet B*. *Amulet B* sends back information regarding its *RSA Modulus length* (Public Key modulus length).

(3) **Shared Session Key Existence Check** – *Amulet A* and *Amulet B* check if they have already a Session Key. *Amulet A* sends its own card ID to *Amulet B*. It then sends a *Request for card ID*. *Amulet B* sends back its card ID that Amulet A will consider as a *"Partner ID"*. To infer if a Session Key between the two already exists, *Amulet A* requests *"Partner Smart card ID"* from its local smart card. This information represents the Session Key's partner ID the card is configured to. If the *"Partner Smart card ID"* is equal to Amulet B's *"Partner ID"* then a shared session key exists and Amulet A goes to phase 5. If not, a Session Key binding both Amulets must be generated.

(4) **Session Key Generation** – The Session Key is securely passed from *Amulet A* to the *Amulet B*, RSA encrypted using the *Amulet A*'s RSA Modulus and keys, following these steps :

   (1) *Amulet A* sends its RSA modulus to *Amulet B* (Public Key). Upon reception, *Amulet B* starts to calculate the Session Key.

   (2) *Amulet A* periodically "pings" *Amulet B* until calculation is over.

   (3) *Amulet A* then sends a *Request for Session Key* command to *Amulet B*. *Amulet B* sends back a padded block of data RSA encrypted with Amulet A's modulus (Public Key) containing both the signing and encryption Session Keys based on the previously calculated Session Key.

   (4) *Amulet A* creates a hash out of the entire block of data. This has is identified as *AuthData* and will later be used in the incremental Fiat-Shamir authentication protocol.

   (5) The Amulets perform several rounds of Fiat-Shamir authentication algorithm, based on *AuthData.* These rounds shall be performed

without interruption. If the authentication fails, abort the Session Key generation.

(6) *Amulet A* decrypts the block of data with its RSA private key and stores both the signing and encryption Session Keys in its local smart card for later use.

(7) On completion, the *Amulet A* updates its "*Partner Smart Card ID*" value with the current card ID of *Amulet B*.

(5) **Message Key Generation**

(1) *Amulet A* creates a *random seed* and sends it to *Amulet B* encrypted using the Session Key. The random seed will be used to construct the Message keys.

(2) *Amulet A* sends a *Request for Message Key to Amulet B*. Upon reception, *Amulet B* creates a Message Key based on the *Amulet A*'s *random seed* and the previously established Session Key. It then calculates both signing and encryption Message Keys based on the value of the Message Key. In response to the *Request for Message Key, Amulet B* sends back both signing and encryption Message Keys and the original *random seed*, encrypted with the Session Key.

(3) When receiving the response to its *Request for Message Key, Amulet A* does the following:

    (1) Decrypts the data received and validate the Session key signature. If the signature verification fails the Amulet marks the Session Key as non-valid and resets its smart card.

    (2) Compares the random seed with the seed it sent previously. If the seeds do not match, resets its smart card.

(4) On completion, *Amulet A* stores both signing and encryption key in memory and use them do securely communicate with *Amulet B*.

Standard Rijndael block cipher (AES-128) is used for signature and encryption. The *Incremental Hash* function is also a Rijndael-based hash function. The following table summarizes the information shared between that smart card and the Amulet.

Hardware module internal memory	Amulet internal memory
RSA modulus, RSA modulus length, RSA private key, Session keys (signature and encryption), *Partner Smart Card ID*	Message Keys (signature and encryption), Incremental Hash function, *AuthData*

Group management

Groups permit the establishment of membership relation between collections of Amulets. An Amulet can authenticate its membership to other members of the same group. The smart cards embedded on each Amulet are manufactured with pre-configured secret group knowledge (password and identification code).

The following protocols require that a Secure Channel be established between a pair of Amulets and a valid Message Key is available. All communication between Amulets is encrypted and signed.

Joining protocol

Scenario: Amulet A is part of *group G* and wish to add *Amulet B* to the group.

(1) *Amulet A* sends a *Join Group* command with a secret code identifying *group G* to *Amulet B*.

Verifying protocol

Scenario: Amulet B insinuates it is part of *group G* and *Amulet A,* who is effectively part of the group, wants to verify the assumption.

(1) *Amulet A* sends a *Group Membership Check* command to *Amulet B* with a secret code identifying *group G.*
(2) *Amulet A* sends a *Verify Card Password* command to *Amulet B.* Upon reception, *Amulet B* retrieves the password from its local smart card, encrypts the result and sends it back to *Amulet A.*
(3) *Amulet A* decrypts the response and verifies the signature. If signature is not valid, *Amulet B* is not part of *group G.*
(4) *Amulet A* verifies that the decrypted password is effectively the one of *group G.* If the password is not valid, *Amulet B* is not part of *group G.* If the password is valid, *Amulet B* is effectively part of *group G.*

Removing protocol

Scenario: Amulet A and *Amulet B* are part of *group G. Amulet A* needs to remove *Amulet B* from the group.

(1) *Amulet A* sends a *Remove from Group* command with a secret code identifying *group G* to *Amulet B.*

Authentication Engine and API

In all the protocols mentioned above, the sequence of operations involving the Amulet's embedded smart card is important. The card keeps state of the commands sent to it and thus the steps described in the protocol must be performed in sequence. Concurrent access to the smart card peripheral must thereof be mediated by a system wide interface.

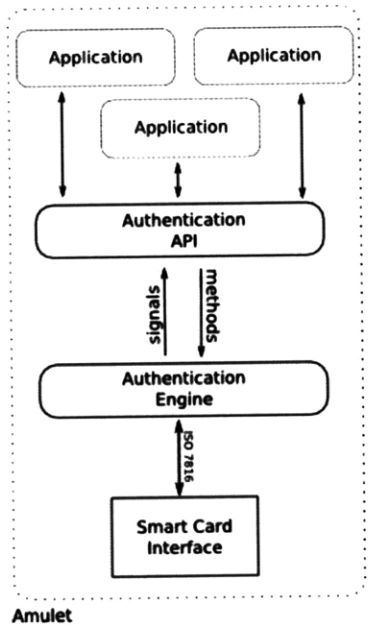


Figure 5.2 – System diagram of the Authentication Engine and API

The Amulet base system provides such interface called the *Authentication Engine*. It is responsible for interfacing with the smart card reader hardware peripheral and encapsulates all the "low-level" card commands needed to perform a specific protocol. Atop of the engine is an Application Programming Interface (API) implementing the Fiat-Shamir Authentication, Secure Channel, and Group Management protocols. Concurrent access to the card is mediated through the *Authentication Engine* where a single instance of the engine is encapsulated in a system wide *dbus* [freedesktop.org] deamon. If a given protocol is under development, access to the card is denied to other applications until the protocol terminates.

Discussion

In this chapter we introduced the Amulet, an open source embedded systems prototyping platform having for design guideline *Identity and Authentication*.

In terms of hardware, the device is not by itself novel. It rather presents the core basic functionality a device should have in order to operate in the Third Cloud. Access to open source hardware design files and bill of material is especially intended to bootstrap the design and implementation of new ubiquitous Third Cloud devices. New designs may be based on the current architecture or may not, the point is to have them open sourced so users can augment, modify, and perhaps distribute emergent hardware devices.

The Amulet achieves interoperability with other computer systems by presenting a GNU/Linux operating system augmented of *Substrate* programs and systems, and by featuring USB ports where multiple standard "plug n play" radio interfaces may be connected to the platform. Using *Substrate*, code and programs are migrated from the embedded platform to more powerful machines and richer interactive systems in the environment, making use of their resources on the fly. Context is captured by the device within *Substrate's Awareness and Discovery* system. Future development of the device may include embedded sensors to bring local platform sensing into play in the establishment of context.

Central to the Amulet is a cryptographic hardware module providing a toolbox for the implementation of authentication protocols and encrypted communication channels. We presented a first zero-knowledge protocol intended to verify that a communicating peer effectively posses the cryptographic module. The verifier itself does not need to posses such modules. It only challenges on the fly a pretending Amulet to prove itself and can infer, through mathematical manipulation of answers given by the Amulet, if it is a trustworthy device. This simple protocol answers the need of *authenticity* while keeping a degree of *anonymity*, in the sense that the Amulet is not directly identified by this zero-knowledge protocol but rather challenged to give proof of being of the Amulet "type".

On the other hand, a *Secure Channel* can only be established between a pair of Amulets. A *Session Key* must first be produced in order to create *Message Keys* used to encrypt and sign messages between communicating peers. This *Secure Channel* protocol relies on one of the Amulet's public key to securely exchange symmetric *Session* and

Message Keys. When generated, these keys can be used for subsequent communications involving the same pair of Amulets. This protocol answers the need for *confidentiality* in our *Identity and Authentication* design guidelines.

Amulets can also form groups between each other. Thereof, a user can forge itself an identity by creating a new group or simply by getting introduced to a preexisting group of Amulets. The fact that we are using a cryptographic hardware module manufactured with preexisting groups in memory eases the implementation of such mechanism; however it is limited since it does not scale. Future development need to take this scaling limit in consideration. Also, groups are only used to authenticate memberships and do not provide cryptographic targets. We can use *Message Keys* generated between a pair of Amulets as groups' cryptographic targets but they were not devised to be used in this scenario.

In general our efforts here have been on implementing systems drivers to communicate with the cryptographic hardware module atop of which we build the protocols mentioned above. Further development of the *Identity and Authentication* system would dedicate efforts on abstracting these core mechanisms away by devising and implementing a key and identity management system presenting a richer semantic to describe, present and manage social relations entrenched within the Amulet. The work presented here could be the foundation for such endeavor.

6 Applications

In this chapter we present two applications that were constructed with Substrate's authoring environment atop the *Awareness and Discovery Engine* and *Execution Migration Process*. The point is to show the flexibility of *Substrate*, the simplicity of its scripting language and the visual feedback produced by the engine. A first application featuring a Notepad capable of migrating from the Amulet to a discovered display in the environment is presented. A second application consists of a face detection search performed in a room where cameras are made available. Search is modeled around Google's map/reduce algorithm.

Notepad

This application is intended to be a simple example of the interaction functionality migration involving the Amulet and interaction resources in its environment. A notepad is deployed on the Amulet, capable of migrating to proximate display resources upon discovery. The following table shows the few lines a *Substrate* script needed to construct the application.

```
<Application name="Viral Notepad Application" author="Daviid Gauthier" xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xsi:noNamespaceSchemaLocation='../generators/viral_framework.xsd'>
  <Interfaces>
    <ServiceDefault/>
    <ServiceNotepad/>
  </Interfaces>
  <Resources>
    <ListOfScreens>
      <Screen name="aScreen" resX="2560" resY ="1600"/>
    </ListOfScreens>
  </Resources>
  <Machines>
    <StateMachineDefault name="Application Automaton" resetOnResume="false" suspendOnLeave="false" maxPlays="10">
      <ListOfConditions>
        <ConditionAlways/>
      </ListOfConditions>
      <ListOfStates>
        <StateDefault name="AmuletState">
          <ListOfTransitions>
            <TransitionMigrateOnResource name="ToNotepadState" nextstate="NotepadState" resourceRef="aScreen"/>
          </ListOfTransitions>
        </StateDefault>
        <StateNotepad name="NotepadState" title="Zapman's Notepad">
          <ListOfTransitions>
            <TransitionMigrationDefault nextstate="AmuletState" name="toAmuletState" ip="00:XX:E7:XX:D2:XX">
              <ListOfConditions>
                <ConditionMigrateNotepad/>
              </ListOfConditions>
            </TransitionMigrationDefault>
            <TransitionDefault nextstate="End" name="toEnd">
              <ListOfConditions>
                <ConditionExitNotepad/>
              </ListOfConditions>
            </TransitionDefault>
          </ListOfTransitions>
        </StateNotepad>
        <StateAppShouldEnd name="End"/>
      </ListOfStates>
    </StateMachineDefault>
  </Machines>
</Application>
```

We developed a Notepad library, referenced by `<ServiceNotepad/>`, containing *StateNotepad, ConditionMigrateNotepad,* and *ConditionExitNotepad. StateNotepad* is derived from *Substrate*'s base *State* class. It overwrites the base class' *OnEnter* method where Python code creating the Notepad window and displaying the current text is placed. *OnLeave* is also overwritten to capture the input text and serialize it before migrating back to the Amulet. The Notepad widow also presents a menu where a user can close the application or signal the Notepad to migrate back to its Amulet. These menu items raise events that are captured by both *ConditionExitNotepad* and *ConditionMigrateNotepad* respectively.

Following is the state diagram based on the "Viral Notepad Application" script, generated by the authoring environment. It illustrates the three states described in the script and the transitions liking them together. This diagram gives the application composer a "high-fidelity" visual feedback of the application automaton being generated in (Python) code.

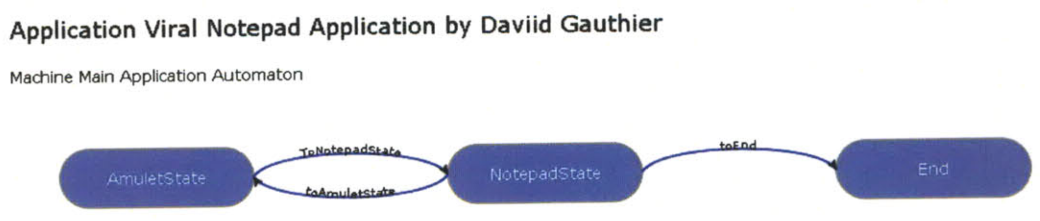

Application Viral Notepad Application by Daviid Gauthier
Machine Main Application Automaton
AmuletState
ToNotepadState
toAmuletState
NotepadState
toEnd
End

Figure 6.1 – Generated state diagram of the Notepad application

Notepad is a simple application illustrating the basics functionality of *Substrate*'s automaton migration. It introduces concepts of *State* and *Condition* functionality extension and effective scripting. A more developed application is presented in the next section.

Map/Reduce Face Detection

This next application is performing a search based on faces detected in a room. In this scenario, cameras attached to cooperating machines are discovered and face detection code is migrated to their respective contexts in parallel, consuming the camera resources locally and using the processing power of the underlying machines. The search itself returns a picture featuring the highest number of faces from the collection of pictures taken from each machines.

The algorithm to perform the search is based on Google's Map/Reduce algorithm [Dean, J. and Ghemawat, S., 2004]. Map/Reduce was designed to execute a parallel search in larger clusters of commodity machines. A programmer wishing to perform search on the cluster writes two functions:

(1) *Map* function - analyzing data on a single machine and generating a set of key/value pair based on the analysis algorithm.

(2) *Reduce* function - having as input the set of key/value pair generated by the map function and reducing it to a smaller size set, generally reducing it to a single pair.

The Map/Reduce system is responsible of sending the custom *Map* function code (REV) to each machines in the cluster in order to be executed in parallel. It then links the results generated by the *Map* functions to the *Reduce* function which, in turn, reduces all the intermediate results to a single search result.

We implemented a similar parallel Map/Reduce algorithm using *Substrate* authoring environment. Two specific states are implemented: *StateMap* and *StateReduce.* These are embedded in two different automata executing concurrently. Here is the script for the Map automaton:

```xml
<StateMachineDefault name="Map" resetOnResume="false" suspendOnLeave="false" maxPlays="10">
  <ListOfConditions>
    <ConditionAlways/>
  </ListOfConditions>
  <ListOfStates>

    <StateDefault name="MigrateMaps">
     <ListOfTransitions>
      <TransitionMigrateOnAllResources name="ToMap" nextstate="Map" resourceRef="aCamera"/>
     </ListOfTransitions>
    </StateDefault>

    <StateMap name="Map" reducerAddr="aaa.bbb.ccc.ddd" reducerPort="8188">
      <ListOfTransitions>
        <TransitionDefault name="ToEnd" nextstate="End">
          <ListOfConditions>
            <ConditionReducer/>
          </ListOfConditions>
        </TransitionDefault>
      </ListOfTransitions>
      <OnEnter>
        <PrintState message="TakeSnapAndAnalyze OnEnter"/>
        <SnapAndAnalyze numberSnaps="10"/>
      </OnEnter>
    </StateMap>

    <StateAppShouldEnd name="End"/>

  </ListOfStates>
</StateMachineDefault>
```

The first state named *MigrateMaps* is responsible to send an automaton to each discovered machines featuring a camera. The same copy of the automaton is sent to each machine and the script is executed in parallel. The *MigrateMaps* 'transition named *TransitionMigrateOnAllResources* is responsible for this parallel migration.

When the automaton has been migrated to the remote context, it enters the *StateMap.* This *State* provides an *OnEnter* function hook for custom code to be scripted in. Here <SnapAndAnalyze numberSnaps="10"/> is taking ten pictures using the camera, analyzing the content with a face detection algorithm, and producing a *file name/number of faces* pair for each pictures. The generated set of *file name/number of faces* pairs is then handed back to *StateMap* to store the result in a global variable. This *StateMap's OnEnter* function is similar to Google's *Map* function in the sense that it can be customized with code as long as the function emits back key/value pairs to the Map system in order to be stored and reduced later by the *Reduce* function.

After *OnEnter* function completion, *StateMap* signals back to the Reducer, still residing on the originating platform, that it has terminated its analysis. The mapping automaton then halts execution and wait for the reducer to arrive before ending the process. This halt is represented by the `<ConditionReducer/>` attached to the *StateMap*'s single transition.

At the other end, the *Reduce* automaton is waiting for signals from the *Map* automata to start execution. This halt is represented by the `<ConditionMapSignals callbackPort="8188"/>`, attached to the automaton in the script bellow:

```
<StateMachineDefault name="Reduce" resetOnResume="false" suspendOnLeave="false" maxPlays="10">
  <ListOfConditions>
    <ConditionMapSignals callbackPort="8188"/>
  </ListOfConditions>
  <ListOfStates>

    <StateDefault name="MigrateToMaps">
      <ListOfTransitions>
        <TransitionMigrateToMapper name="ToReduceResults" nextstate="ReduceResults"/>
      </ListOfTransitions>
    </StateDefault>

    <StateReduce name="ReduceResults">
      <ListOfTransitions>
        <TransitionDefault name="ToMigrateToMaps" nextstate="MigrateToMaps">
          <ListOfConditions>
            <ConditionMoreMappers/>
          </ListOfConditions>
        </TransitionDefault>
        <TransitionMigrationDefault name="ToDisplayResult" nextstate="DisplayResult" ip=" aaa.bbb.ccc.ddd">
          <ListOfConditions>
            <ConditionNoMoreMappers/>
          </ListOfConditions>
        </TransitionMigrationDefault>
      </ListOfTransitions>
      <OnEnter>
      <PrintState message="ReduceSnaps OnEnter"/>
      <ReduceSnaps/>
    </OnEnter>
    </StateReduce>

    <StateDefault name="DisplayResult">
      <ListOfTransitions>
        <TransitionDefault name="ToEnd" nextstate="End">
          <ListOfConditions>
            <ConditionCloseApp/>
          </ListOfConditions>
        </TransitionDefault>
      </ListOfTransitions>
      <OnEnter>
        <ShowReducerResult/>
      </OnEnter>
    </StateDefault>

    <StateAppShouldEnd name="End"/>

  </ListOfStates>
</StateMachineDefault>
```

When the *ConditionMapSignals* receives signals from the remote *Map* automata it records their respective addresses and stored them in the automaton memory. Data is kept in the automaton's memory since the automaton is the object whose entire state is migrated from machine to machine. The automaton presents a dictionary (key/value) to store transportable data across machines. When all the addresses are collected, the Reduce automaton enters its first state named *MigrateToMaps*. A special *Transition* called *TransitionMigrateToMapper* was devised to go through the heap of remote addresses constructed by the *ConditionMapSignals* and initiate a migration to the address popped from the top.

After the automaton has been migrated to the remote context, it enters the *StateReduce.* Similar to *StateMap,* this *State* provides an *OnEnter* function hook for custom code to be scripted in. Here <ReduceSnaps/> is related to the previous map function and goes through the intermediate set of *file name/number of faces* pairs on the local platform and compares the highest number of faces score with what it already has in the automaton's memory. If the score is higher, it stores the bitmap picture in memory and updates the new score. Access to the Map set of key/value pairs is possible since the set is a global variable of the *StateMap* module (Python module are singletons instances) and the *StateReduce* and *StateMap* are part of the same Python OS process.

After reduction, the automaton hops to the next "*Map* machine" if the address heap is not empty. The same process described above will take place, ultimately reducing all intermediate results to a single one. When the heap of addresses runs empty, the automaton is migrated back to its originating platform and the picture with the highest number of faces is displayed. The following figure is the state diagram of both automata.

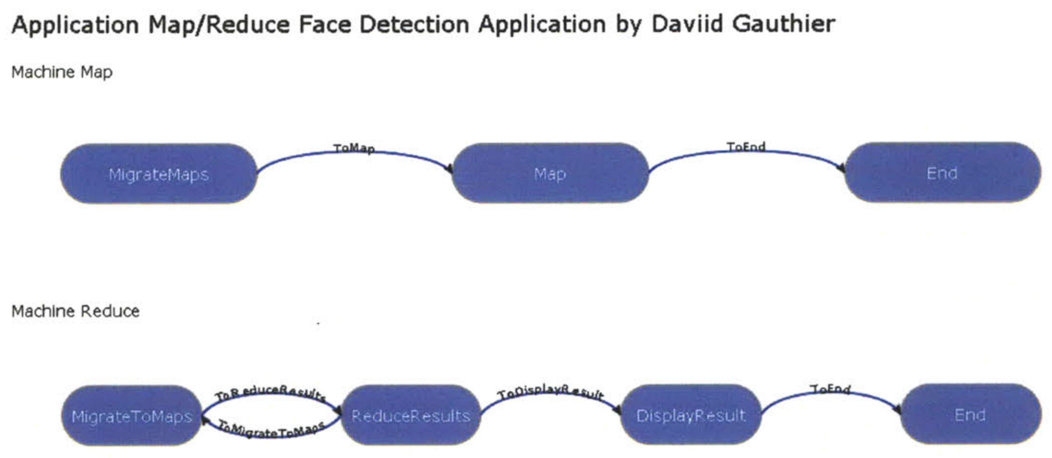


Figure 6.2 – Generated state diagram of the Map/Reduce Face detection application

Discussion

In this chapter we presented two applications built with *Substrate*'s authoring environment.

Notepad is a parasitic application capable of projecting itself from an Amulet into an environment featuring display devices. The script only contains 41 lines of XML code. Simple, the application is intended to introduce to application composers the simplicity of the authoring framework and library developers the mechanisms from which functionality is virally injecting into an environment. The application could be generalized to richer interactive applications, such a gaming, web browsing, or even drawing.

Map/Reduce Face Detection is an advanced application composed of two automata interacting with each other. This application shows the flexibility of *Substrate* to implement diverse programming models and agility to configure a set of loosely coupled machines. Here the Map/Reduce model has been implemented and adapted to the Third Cloud. We showed that parallel execution of code is possible by migrating the same instance of an automaton to different machines (Map automaton) – this model relates to the remote invocation (REV) paradigm introduced in chapter 3. We also showed that a mobile agent can be constructed crawling from machines to machines (Reduce automaton), where the route is established on the fly with the cooperation of other automata (Map automata). In Google's data centers, Map/Reduce is supported by an underlying central system responsible for scheduling jobs across a well know set of machines and mediating access to the file system [Dean, J. and Ghemawat, S., 2004]. In our case, the Map/Reduce task is a *viral process* capable of self-configuration and self-operation on a set of heterogeneous and loosely coupled machines. Task and configuration are part of the same executable.

In both application examples we demonstrated the process of writing an application script and extending functionality of the base system. Let's now revisit and evaluate *Substrate*'s design principles.

- As illustrated by *StateNotepad, StateMap,* and *StateReduce,* Substrate's authoring framework permits the extension the "high-level" language with semantics exported form a custom library description. In turn, the generated application code directly links library code. The framework does indeed unify low-level software engineering and high-level application design.

- As demonstrated in the context of the Map/Reduce Face Detection application, the authoring framework gives the ability to model a wide variety of complex contexts and configurations.

- The generated state diagrams are also important tools to understand what will be generated in executable code. These blueprints simplify the understanding of the internal logic of an automaton by representing it is an accessible visual format. It assists dialog between application composers and their emergent application. Moreover, idea transfers between non-technologists and technologist is greatly improved by this sort of state diagram.

- *Substrate*'s code generation mechanism supports an iterative design process since application can be assembled, modified, and deployed rapidly without the need of intermediary expert developers.

7 Conclusion and Future Work

In this thesis we introduced architectures of the Third Cloud. Based on concepts of *connectivity utility* and *resource utility*, the Third Cloud looks at bringing cooperative use of interactive resources in the in the public realm; creating an *interaction utility*. We can draw important similarities between the Third Cloud concept and the one of ubiquitous computing. The main difference relates to scope. Rather than embedding interactions in highly controlled spaces, contexts of interaction in the Third Cloud need to be spontaneously defined on the spot, based on less organized, perhaps decentralized set of public interaction resources presenting uneven computing conditions. This thesis looked at the topic of interoperability, modular composition, and application authoring in the design of pervasive and public interactive systems.

A typical approach to solve the problem of interoperability between devices is to define a fixed set of domain-dependent interfaces which would be installed and configured in the environment *a priori* a given interaction. A mobile device would then discover the availability of these services and establish a domain dependent protocol with a specific node hosting a given service. For example, in the case of the Notepad application presented in chapter 6, a Notepad service would need to be configured on a set of machines in the environment and made available on the network. A client application residing on the user's mobile device would need to discover the Notepad service and establish a connection using Notepad dependent protocol with the node hosting it. Another domain-dependent interface would also be needed to build the Map/Reduce Face Detection application presented in the same chapter. However, the problem with such domain-dependent interfaces in the context of the Third Cloud, of course, is that there are a plurality of them; the range of such interfaces means that there are always likely to be specific interfaces that a particular entity is not configured to use or understand.

Our alternative approach to this problem, presented in this thesis, is to consider a small set of domain-independent interfaces shared between devices and machines of the Third Cloud. Rather than making explicit a single specific interaction between communicating entities, as in the domain-dependent case, our domain-independent interface funnels interactions into one simple protocol. Funneling of interactions is achieved with the use of virtual machines, placed on each Third Cloud devices, and virtual code exchange between these devices. In other words, our approach is to consider the dynamic configuration of an *ad hoc* network of virtual machines composed of devices in proximity in order to move executing processes into nodes presenting local access to a given interaction resource. Using the systems we created, we have shown that there is no need for separate protocols for the Notepad application and the Map/Reduce Face Detection applications to execute in an environment but rather a need for a single protocol used to migrate desired functionality from a mobile device to its cooperative context.

Motivated by the concept of *viral process*, our approach in achieving interoperability tried to highlight the importance of decoupling computation from the underlying machines where it takes place; not to make computation location invariant per se, but rather to make it agile, mobile, and opportunistic. Augmented of the ability to capture social and infrastructural context in which it operates, computation can spontaneously and opportunistically span the boundaries of local and personal device and expand itself into the environment by using the proximate interaction infrastructure as a substrate; making the vision of *interaction utility* a reality.

Given the anticipated continued churn in devices, media formats, and interaction resources, it will be practically impossible for application developers to provide to end users custom-built applications that support the range of things that they want to do given the variety of resources available in a particular point of time. A central goal of this thesis was to find the middle ground in a spectrum that ranges between extreme ease of use – represented by pre-built applications proposing a unique and fixed interaction scenarios – and extreme flexibility in constructing a variety of tasks – using specific programming languages such as Java, Python, C++ to express in details a given application.

Our approach was to construct a set of modular and composable service types that can be assembled by end users in pre-defined ways, minimizing the responsibility of the software developer to solely create the application with which the user interact. We approached this goal by proposing a modular scripting language and an automaton computing model both entrenched in a single authoring framework we called *Substrate*. We decided to base user generated applications on an automaton model since it provides the necessary modular structure and expressive abstraction, such as a state diagram, to author self-operable interactive tasks capable of safely migrating execution to cooperative machines depending on their surrounding contexts. Our framework itself presents a descriptive layer where automaton-based applications are scripted in a high-level scripting language, and a constructive layer transforming the scripted computation into executable code. By separating description from construction, we isolate the specifics of implementation and unify both library software engineering and application design with the same semantics. As a result, emergent resources can automatically be added to our framework by having a software engineer describing a library to operate the resource, implementing the details in a target programming language, and exporting control semantics to our high-level language in a modular fashion so application designers can assemble an application based on the resource.

We demonstrated the flexibility and power of our approach in chapter 6 where two different applications, Notepad and Map/Reduce Face Detection, were authored and constructed using *Substrate*'s framework. In these applications, specialized modular automaton components such as *State, Condition,* and *Transition* have been extended with specific functionality in a set of libraries; functionality ranging from automaton migration to discovering resources in proximity. These components naturally integrated our computing model and were made available to application designers through *Substrate*'s high level language. In turn, using the same automaton paradigm to describe computation and the same authoring environment, Notepad and Map/Reduce Face Detection were constructed. Through these applications we showed that *Substrate* can adapt to diverse programming models and demonstrated the agility of an automaton-based *viral process* to configure a set of loosely coupled machines to carry a given task.

Our systems could certainly be improved. Future work on *Substrate* would consider creating an editing tool based on automata state diagrams. This editing front-end could easily generate *Substrate* scripts. The same compiler toolchain described in this thesis would then be used to generate executable code. The *Awareness and Discovery* engine could propose adaptable advertising rates based on work presented in [Ypodimatopoulos, P., 2008]. Our *Identity and Authentication* protocols could form the basis of an identity management system presenting a richer semantic to describe, present and manage social relations entrenched within the Amulet. The Amulet could be embedded with a set of local sensors and its form at could be made smaller.

In fact, all architectures presented in this thesis were devised around the open source philosophy in the hope of bringing user innovators as key actors in the development of the Third Cloud [von Hippel, E., 2007]. *Substrate* was constructed around the assumption that by constructing a set of modular and composable service types that can be assembled by end-users in pre-defined ways, we could minimize the responsibility of the software developer to solely create the application with which the user interact. The open source Amulet was also designed in reaction to the pathological limitations of our devices to interoperate with each other; limits imposed by network operators and machine makers who are building technology in silos. This is perhaps the reason why Weiser's vision of ubiquitous computers may have become a reality, but his vision of invisible computing and "calm" interaction has not. In an effort to address this issue, all our systems are open and have been integrated horizontally, from the bootloader loading the Linux kernel into the Amulet microprocessor's memory to *Substrate*'s "high-level" language used to compose viral applications. We truly believe that empowering user innovators is the only avenue which will lead us to the Third Cloud. It is time to start treating different materials – people, machines, "ideas" and all the rest – as interactional effects from which social, interactive, and innovative networks emerge, rather than primitive causes.

Appendix A

Substrate Implementation Notes

A.1 Authoring Environment

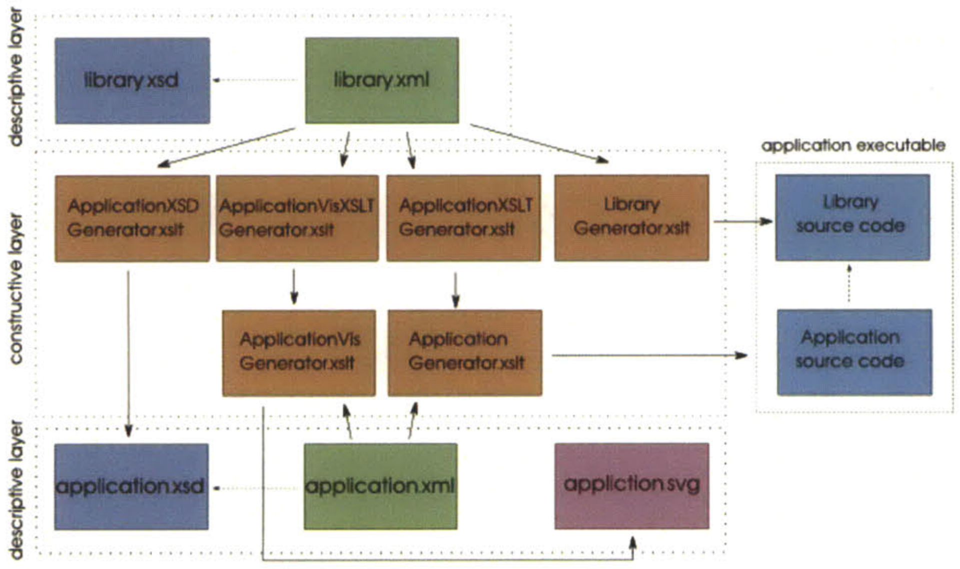


Figure 4.2 – Diagram illustrating components of Substrate's authoring environment

Substrate's scripting language is based on extensible markup language (XML) and related technologies. It was elected language of choice since has the following feature:

- provides a formal semantic to describe program/application
- is easier to read and write than bare source code
- tools exist to validate content
- tools exist to transform into another format

Used in tandem with both XML Schema (XSD) and Extensible Stylesheet Language Transformations (XSLT) technologies, we can easily engineer a basic compiler; where the XSD-validating parser figures as the compiler's front-end and the XSLT code generator as the back-end. Substrate's authoring framework features four of these compilers:

(1) **Library definition compiler**

Core elements of the finite state automaton can be extended at the library level. A library description is written in XML and validates against *Substrate*'s library XML schema (library.xsd). The schema defines possible base types that may be extended (subclassed) by a library. Such types include: State, Transition, Condition and Procedure. Extension of these basic types is performed by providing a class definition, consisting of input parameters and a set of methods. Hence, a library definition exposes new class names and method signatures, defining a specific namespace which can be referenced by applications. Code is generated when the library definition validates against the library XML schema.

Specific XML to Python transformations of are defined in an XSLT format (LibraryGenerator.xslt). This file defines the process of generating Python classes derived from *Substrate*'s base classes. Depending on the content of the library definition, a set of files are generated. Each of them represents a derived class comprised of method placeholders. These methods are generated according to their signatures, specified in the library definition, or derived from *Substrate*'s base classes. In programming language term, each generated classes are facades [Gamma et al., 1995] comprising method stubs for code to be written in.

Hence a current Python library can easily be added to *Substrate*'s code base. It only needs to define an *export interface* in XML (library definition) and the authoring framework will in turn create placeholders to put desired functionality in. To make a custom piece of code available to application composers, a library developer only has to write a few lines of code: write an XML interface and hook its code into generated method stubs.

(2) **Library definition to Application XSD and XSLT compiler**

From a library description in XML the framework constructs a namespace used to reference generated classes. This namespace is constructed by producing an application XML schema (application.xsd) from the library description. This generated schema contains all possible class names and method signatures described in the library which can be referenced in an application and against which the application XML description is validated.

Bindings between generated namespace and the actual Python library code are defined in an generated XSLT document (ApplicationGenerator.xslt). This document contains code transformation descriptions which have as input elements of the generated namespace and produce as output related Python class and method invocations. Thus applications' description in XML are transformed into valid Python code capable of binding previously generated libraries.

This intermediate code generating phase is central to the authoring framework's extensibility. By providing both an XML schema generator (ApplicationXSDGenerator.xslt) and a code-generator generator (ApplicationXSLTGenerator.xslt) it is possible to seamlessly extend the framework's code base with optimized library code that is automatically available to applications scripts.

(3) **Library definition to Application state diagram XSLT compiler**
From a library definition, a state diagram generator is constructed. Currently *Substrate* outputs a state diagram based on the Scalable Vector Graphics (SVG) format. The generated SVG code-generator contains shape descriptions and draw functions to construct a representation of an application's automaton logic. It gives a visual feedback to application composers on the flow control of their script and provides a blueprint of what will be generated in executable code.

(4) **Application compiler**
An application is scripted in an XML format which validates against the generated XML schema (application.xsd) described in the previous section. This

schema also contains format description of a finite state automaton. Hence, an application is required to be scripted in a finite state automaton format describing a set of States, Transitions and Conditions. These building blocks can be derived from generated libraries or from *Substrate*'s base implementation.

Code generation of an application is two fold. First the application XML will be transformed into an SVG document featuring a state diagram representation of the automaton (ApplicationVisGenerator.xslt). Second, the application XML is processed according to the generated Application XSLT document (ApplicationGenerator.xslt) and produces a Python executable file binding both *Substrate*'s core finite state automaton classes and extended library code.

The build process is automated by the various Ant [Ant.apache.com, 2009] scripts produced alongside the code generation phases. These Ant scripts are similar to Makefiles, as they contains all necessary commands to validate and generate applications and libraries when invoked.

A.2 Awareness and Discovery

Byte offset	**0-1**	**2-3**	**4-5**	**6-7**	**8-9**	**10-11**
0	RNH		RENH		TRAN	
12	CB			OM	VP	PL
24	Resource Description Payload					

RNH: Resource Name Hash (4 bytes)
RENH: Resource Estimator Name Hash (4 bytes)
TRAN: Transaction (information)(4 bytes)
CB: Callback address (6 bytes)
OM: Overload Metric (2 bytes)
VP: Valid Period (2 bytes)
PL: Payload Length (2 bytes)
Resource Description Payload (1 or more bytes)

Table 4-1 *Awareness and Discovery* frame header

Our protocol uses raw sockets to directly embed beacon information in layer-2 frames.

We utilize the provided link-layer MAC address to keep track of beacons' provenance. Upon reception of a beacon, the *Awareness and Discovery Engine* looks its remote resources table if the given *MAC address/Resource Name Hash* pair is present.

If the pair is <u>not</u> present:

(1) Constructs a data structure containing all the information provided in the beacon.
(2) Timestamps the structure to keep track of its validity.
(3) Inserts the structure in the remote resource table.
(4) *Signal*s observers about the presence of this new resource.

If the pair <u>is</u> present:

(1) Compares the content of the beacon with the given data structure.
(2) Updates data fields if necessary (update the overload metric for example).
(3) If the resource as been updated, signals observers about this resource's update.
(4) Gives the structure a new timestamp.

In order to keep its view of the network up-to-date, the *Awareness and Discovery Engine* refreshes its remote resources table periodically by comparing the *Validity Period* of each resource entry with the difference of times between the current time and the entry's timestamp. If the comparison exceeds a given threshold *T*, then removes the entry from the table and signals observers about this resource's removal.

Local resources are also advertised through the *Awareness and Discovery Engine.* Node wishing to make a resource available on the network, calls the *publish* method of its local engine and provides all resource's information as arguments. In turn, the engine constructs a data structure representing the resource and inserts it in its local resource table. A node can dynamically call the *update* method when it needs to change a given resource's state (e.g. Transactional Information, Overload metric). Local resources are removed from the engine's local resource table by invoking its *remove* function with a valid *Resource Name Hash*. Advertisement of local resources on the wireless network is done periodically by constructing beacon frames of all the local resource present in the local resource table at that time, and transmitting them in through the network interface. The engine inserts its own advertisement period in each beacons' *Validity Period*.

Applications and the *Awareness and Discovery Engine* do not communicate directly – not part of the same OS process. In its current incarnation, the engine is encapsulated in a system wide *dbus* [freedesktop.org] deamon, an inter-process communication (IPC) bus system portable to a plethora of Linux based systems and machine architecture (x86, ARM, MIPS, PPC, etc.). In isolating *Substrate*'s application and system domains with a well defined set of signals and methods, our beaconing scheme is more consistent as the central engine broadcasts periodically *all* available local resources to the environment, rather than singularly and sporadically, giving a better spontaneous picture of its state to its neighborhood. Also, since the access to the network interface is mediated through the engine, we could easily change wireless medium or network protocol without impacting applications' composition.

A.3 Execution Migration Process

Our implementation of the *Execution Migration Process* is based on Python's Interpreter functionality (VM). We constructed a special Python *class loader* sitting in between *Substrate* applications and the Python Interpreter. It is intended to capture all *Substrate* modules (libraries) imported by an application. More precisely it performs the following:

(1) Locates a given *Substrate* module on the local machine or request it from a peer wishing to migrate to the local platform (detailed in the *Protocol* section)
(2) Stamps the module of being *Substrate* related and keep a reference to it.
(3) Loads the module into the Python Interpreter

The class loader is configured to know a priori the location of the *Substrate* library directory on the local machine. Consequently, it can quickly locate *Substrate* libraries locally or infer that the local environment needs to request them from a peer. It also keeps track of all libraries that have been loaded into the Interpreter and thus directly accessible in memory.

The point of this mechanism is for the class loader to have an accurate picture of the local *Substrate environment* capabilities. It is thus able to provide access to local or already loaded modules to requesting peers on the network.

Protocol

As explained earlier, upon a *Transition* an automaton reaches a state where it can be serialized and migrated from the local environment to another environment (see phase 4 of *figure4-X*). Part of our automaton implementation, we provided the necessary means to encapsulate its state and self-initiate a migration. Here we explain the protocol involved in the migration process.

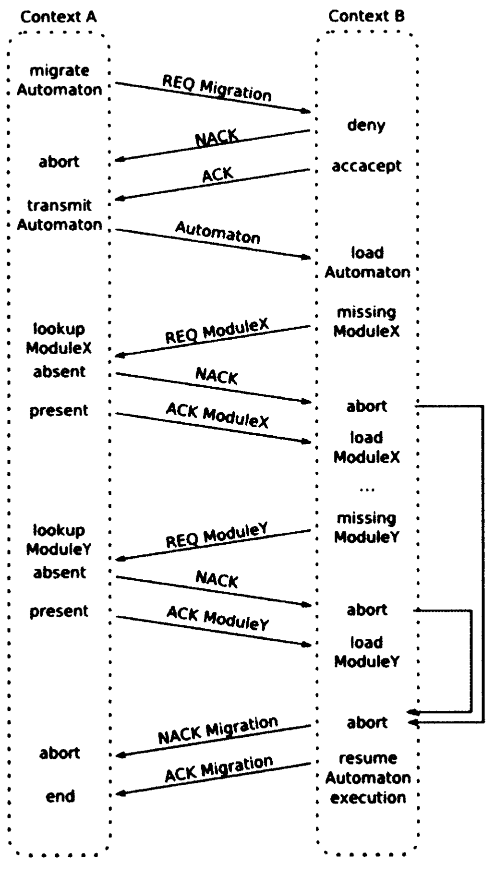


Figure 4.5 – Diagram illustrating migration protocol

A *Context* is the overarching structure creating threads of execution and keeping a global reference to all local *Automata* running on the platform. Each platform has a single *Context* who is responsible of instantiating each automaton and performing migration to other remote *Contexts* upon request. Consequently, an automaton initiates a migration by informing its *Context* about the destination it needs to migrate to. This destination is usually provided by the *Transition* who initiated the migration process. Destination addresses and ports can currently be hardcoded in the *Transition* or retrieved from a discovery beacon (*Callback* field).

We focus on a situation where an automaton in *Context A* requests a migration to *Context B*, who is listening to incoming connections at a given address. *Context A* serializes the automaton and initiates a connection to *Context B*. When the connection is established the following protocol is executed (see *figure 4-X*):

(1) *Context A* sends a *REQ Migration* to *Context B.*

(2) *Context B replies* with an *ACK* or a *NACK.*

(3) If *Context A* receives an NACK it aborts migration, if receives an *ACK* continues to next phase.

(4) *Context A* sends the serialized automaton to *Context B.*

(5) *Context B's* class loader evaluates the automaton and signal *Context B* if a module is missing from its environment denying it from instantiating the automaton. If the automaton can be instantiated go to phase 9, if not continue to next phase.

(6) *Context B* sends a REQ for the missing module X.

(7) *Context A* asks its class loader if module X is available locally. If available replies with an *ACK* and byte-code for the module, if not sends an *NACK.*

(8) If *Context B* receives an *ACK,* loads the byte-code through the class loader and proceeds to phase 5. If receives a *NACK* go to phase 11.

(9) *Context B* sends an *ACK Migration* to *Context A* and go to phase 13.

(10) Context A receives an *ACK Migration* and go to phase 14.

(11) Context B sends an *NACK Migration* to *Context A.*

(12)Context A receives an *NACK Migration* and aborts migration.

(13)Context B resumes automaton execution.

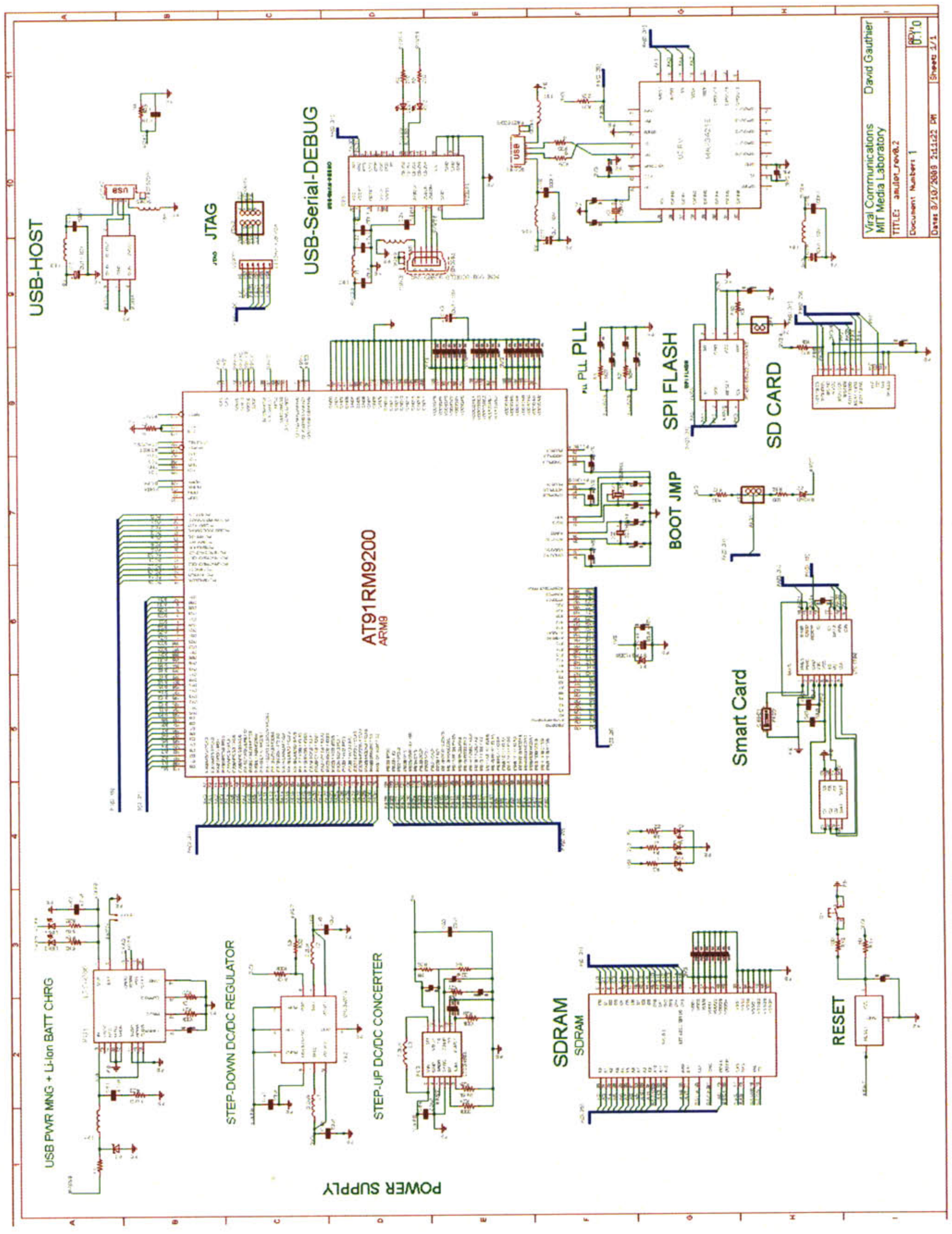
(14)Context A ends automaton execution.

When *Context B* resumes the automaton it has all the necessary *State*s byte-code it needs in order to operate the automaton. It retrieved this information in a peer-to-peer fashion from *Context A.* This process is described in phases 5 to 8. An automaton resumes by instantiating the *current state* identified by its program counter (PC) and enters the state. In other words, it hands execution control to the *current state* which can now access resources locally and present new functionality on its new host. This process illustrates how a mobile device using *Substrate* can dynamically inject functionality in its proximate environment. The next section explains in details applications that were constructed around this model.

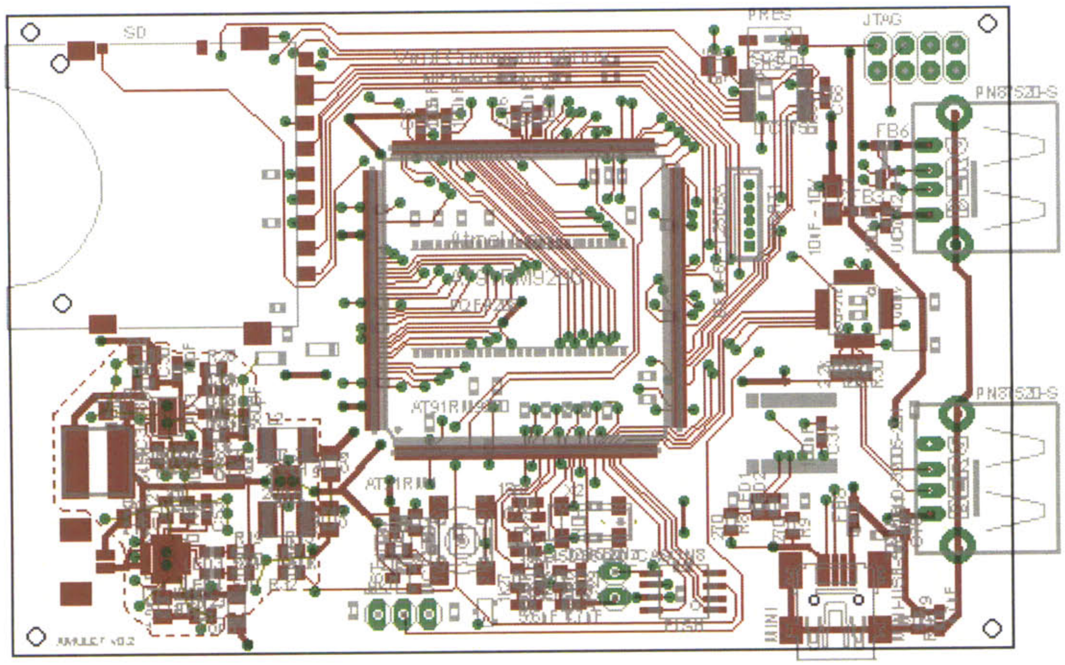
Appendix B

The Amulet Implementation Notes

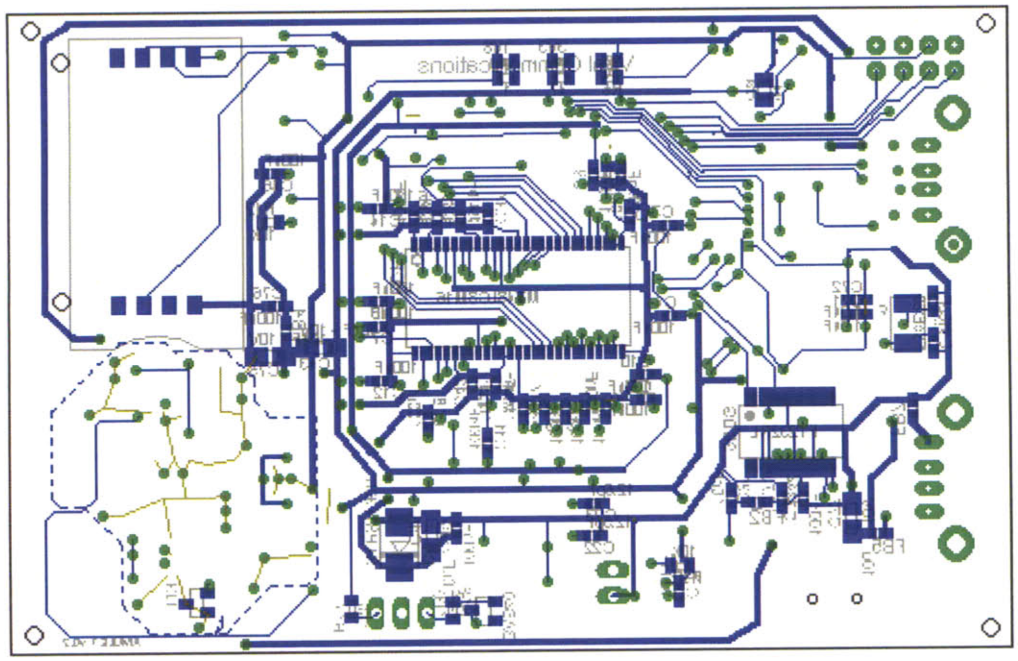
B.1 Amulet Schematic



B.2 Amulet Top Board Layout



B.3 Amulet Bottom Board Layout



Appendix C

Applications Implementation Notes

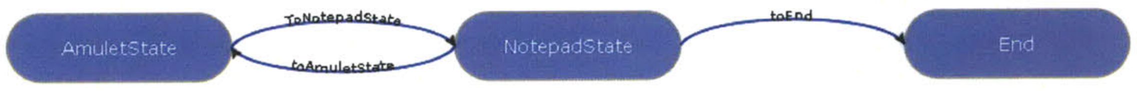
C.1 Notepad XML script

```xml
<Application name="Viral Notepad Application" author="Daviid Gauthier" xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xsi:noNamespaceSchemaLocation='../generators/viral_framework.xsd'>
  <Interfaces>
    <ServiceDefault/>
    <ServiceNotepad/>
  </Interfaces>
  <Resources>
    <ListOfScreens>
      <Screen name="aScreen" resX="2560" resY ="1600"/>
    </ListOfScreens>
  </Resources>
  <Machines>
    <StateMachineDefault name="Application Automaton" resetOnResume="false" suspendOnLeave="false" maxPlays="10">
      <ListOfConditions>
        <ConditionAlways/>
      </ListOfConditions>
      <ListOfStates>
        <StateDefault name="AmuletState">
          <ListOfTransitions>
            <TransitionMigrateOnResource name="ToNotepadState" nextstate="NotepadState" resourceRef="aScreen"/>
          </ListOfTransitions>
        </StateDefault>
        <StateNotepad name="NotepadState" title="Zapman's Notepad">
          <ListOfTransitions>
            <TransitionMigrationDefault nextstate="AmuletState" name="toAmuletState" ip="00:XX:E7:XX:D2:XX">
              <ListOfConditions>
                <ConditionMigrateNotepad/>
              </ListOfConditions>
            </TransitionMigrationDefault>
            <TransitionDefault nextstate="End" name="toEnd">
              <ListOfConditions>
                <ConditionExitNotepad/>
              </ListOfConditions>
            </TransitionDefault>
          </ListOfTransitions>
        </StateNotepad>
        <StateAppShouldEnd name="End"/>
      </ListOfStates>
    </StateMachineDefault>
  </Machines>
</Application>
```

C.2 Notepad SVG State Diagram

Application Viral Notepad Application by Daviid Gauthier

Machine Main Application Automaton

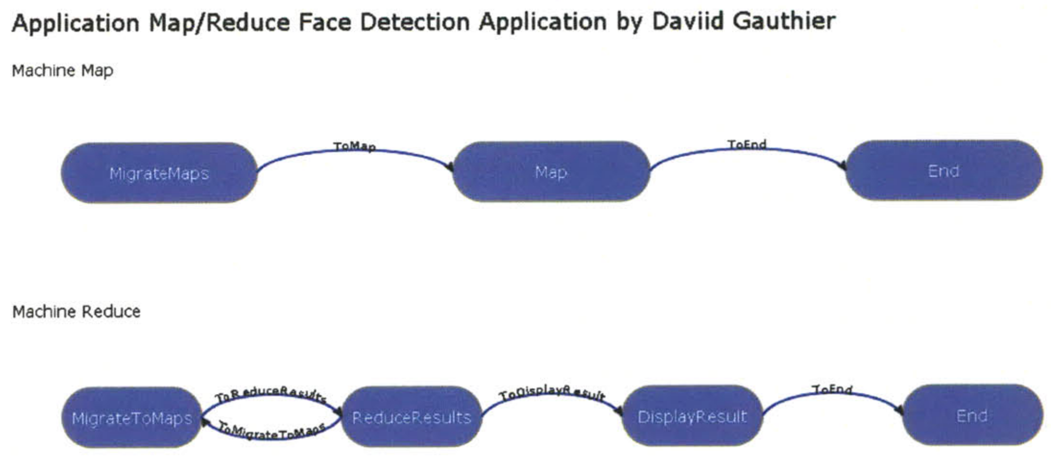


C.3 Map/Reduce Face detection XML script

```
<Application name="Map/Reduce Face Detection Application" author="Daviid Gauthier"
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xsi:noNamespaceSchemaLocation='../generators/viral_framework.xsd'>

  <Interfaces>
    <ServiceDefault/>
    <ServiceMapReduce/>
  </Interfaces>

  <Resources>
    <ListOfCameras>
      <Camera name="Camera"/>
    </ListOfCameras>
  </Resources>

  <Machines>

  <StateMachineDefault name="Map" resetOnResume="false" suspendOnLeave="false" maxPlays="10">

  <ListOfConditions>
    <ConditionAlways/>
  </ListOfConditions>

  <ListOfStates>

    <StateDefault name="MigrateMaps">
     <ListOfTransitions>
      <TransitionMigrateOnAllResources name="ToMap" nextstate="Map" resourceRef="aCamera"/>
     </ListOfTransitions>
    </StateDefault>

    <StateMap name="Map" reducerAddr="aaa.bbb.ccc.ddd" reducerPort="8188">
      <ListOfTransitions>
        <TransitionDefault name="ToEnd" nextstate="End">
          <ListOfConditions>
            <ConditionReducer/>
          </ListOfConditions>
        </TransitionDefault>
      </ListOfTransitions>
      <OnEnter>
        <PrintState message="TakeSnapAndAnalyze OnEnter"/>
        <SnapAndAnalyze numberSnaps="10"/>
      </OnEnter>
    </StateMap>

    <StateAppShouldEnd name="End"/>

  </ListOfStates>

</StateMachineDefault>
```

```
<StateMachineDefault name="Reduce" resetOnResume="false" suspendOnLeave="false" maxPlays="10">
<ListOfConditions>
  <ConditionMapSignals callbackPort="8188"/>
</ListOfConditions>
<ListOfStates>

  <StateDefault name="MigrateToMaps">
    <ListOfTransitions>
      <TransitionMigrateToMapper name="ToReduceResults" nextstate="ReduceResults"/>
    </ListOfTransitions>
  </StateDefault>

  <StateReduce name="ReduceResults">
    <ListOfTransitions>
      <TransitionDefault name="ToMigrateToMaps" nextstate="MigrateToMaps">
        <ListOfConditions>
          <ConditionMoreMappers/>
        </ListOfConditions>
      </TransitionDefault>
      <TransitionMigrationDefault name="ToDisplayResult" nextstate="DisplayResult" ip=" aaa.bbb.ccc.ddd">
        <ListOfConditions>
          <ConditionNoMoreMappers/>
        </ListOfConditions>
      </TransitionMigrationDefault>
    </ListOfTransitions>
   <OnEnter>
    <PrintState message="ReduceSnaps OnEnter"/>
    <ReduceSnaps/>
  </OnEnter>
  </StateReduce>

  <StateDefault name="DisplayResult">
    <ListOfTransitions>
      <TransitionDefault name="ToEnd" nextstate="End">
        <ListOfConditions>
          <ConditionCloseApp/>
        </ListOfConditions>
      </TransitionDefault>
    </ListOfTransitions>
    <OnEnter>
      <ShowReducerResult/>
    </OnEnter>
  </StateDefault>
  <StateAppShouldEnd name="End"/>
 </ListOfStates>
</StateMachineDefault>

 </Machines>
</Application>
```

C.4 Map/Reduce Face detection SVG State Diagram



Bibliography

Amazon.com "Amazon Elastic Compute Cloud (Amazon EC2)". July 31, 2009. http://aws.amazon.com/ec2/. Retrieved on August 1, 2009.

Ant.apache.com (2009) "Apache Ant 1.7.1". http://ant.apache.org/manual/index.html. Retrieved July 2009

Arbib, M. A. (1969) *Theories of Abstract Automata (Prentice-Hall Series in Automatic Computation)*. Prentice-Hall, Inc.

Bluetooth SIG, 2009. Specification of the Bluetooth System , Version 3.0 http://www.bluetooth.com/Bluetooth/Technology/Building/Specifications/.

Brew.qualcom.com, "Brew", http://brew.qualcomm.com/brew/en/

Brooks, R. A.(1997). The Intelligent Room project. In *Proceedings of the 2nd international Conference on Cognitive Technology (CT '97)* (August 25 - 28, 1997). CT. IEEE Computer Society, Washington, DC, 271.

Cáceres, R., Carter, C., Narayanaswami, C., and Raghunath, M. (2005). Reincarnating PCs with portable SoulPads. In *Proceedings of the 3rd international Conference on Mobile Systems, Applications, and Services* (Seattle, Washington, June 06 - 08, 2005). MobiSys '05. ACM, New York, NY, 65-78

Corbató, F. J., Merwin-Daggett, M., and Daley, R. C. 2000. An experimental time-sharing system. In *Classic Operating Systems: From Batch Processing To Distributed Systems*, P. Brinch Hansen, Ed. Springer-Verlag New York, New York, NY, 117-137.

Carzaniga, A., Picco, G. P., and Vigna, G. 1997. Designing distributed applications with mobile code paradigms. In *Proceedings of the 19th international Conference on Software Engineering* (Boston, Massachusetts, United States, May 17 - 23, 1997).

ICSE '97. ACM, New York, NY, 22-32.

Clark D. D., Blumenthal M. S., 2009, The end-to-end argument and application design: the role of trust.

Clarke, D., Elien, J., Ellison, C., Fredette, M., Morcos, A., and Rivest, R. L. 2002. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security.* 9, 4 (Feb. 2002), 285-322.

Coen M, Phillips B., Warshawsky N., Weisman L., Peters S., and Finin R.. (1999) Meeting the computational needs of intelligent environments: The metaglue system. In *Proceedings of MANSE'99*

Dean, J. and Ghemawat, S. (2004). MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (San Francisco, CA, December 06 - 08, 2004). Operating Systems Design and Implementation. USENIX Association, Berkeley, CA, 10-10.

Elias, N. (1978). *The History of Manners. The Civilizing Process: Volume I.* New York: Pantheon Books

Fiat, A. and Shamir, A.(1987). How to prove yourself: practical solutions to identification and signature problems. In *Proceedings on Advances in cryptology---CRYPTO '86* (Santa Barbara, California, United States). A. M. Odlyzko, Ed. Springer-Verlag, London, 186-194.

Figueiredo, R. J., P. A. Dinda and J. A. B. Fortes. (2003) A case for grid computing on virtual machines. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 550. IEEE Computer Society

Flanagan, D., Ferguson, P. (2002). *JavaScript: The Definitive Guide* (4th ed.). O'Reilly & Associates

Foster, Ian, Carl Kesselman, Jeffrey M. Nick and Steven Tuecke. (2002) The Physiology of the Grid. An open Grid services architecture for distributed systems integration. Draft.

Ford, B., Strauss, J., Lesniewski-Laas, C., Rhea, S., Kaashoek, F., and Morris, R. 2006. Persistent personal names for globally connected mobile devices. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington, November 06 - 08, 2006). Operating Systems Design and Implementation. USENIX Association, Berkeley, CA, 233-248.

freedesktop.org. D-bus. http://www.freedesktop.org/wiki/Software/dbus. Retrieved July 2009.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley.

Ginsburg, S. (1982) *Introduction to Mathematical Machine Theory*. Addison Wesley Longman Publishing Co., Inc.

Gnu.org, "GNU Operating System", http://www.gnu.org/

Goland, Y. Y., Cai, T., Ye, G., Leach, P., Albright, S., (1999) IETF draft version 3, Simple Service Discovery Protocol/1.0, http://tools.ietf.org/html/draft-cai-ssdp-v1-03

Google code. (2008) V8 Javascript Engine, Retrieved July 2009 from http://code.google.com/p/v8/

Google code. (2009) Google App Engine, Retrieved July 2009 from http://code.google.com/appengine/

Greenstein, B., Gummadi, R., Pang, J., Chen, M. Y., Kohno, T., Seshan, S., and Wetherall,

D. 2007. Can Ferris Bueller still have his day off? protecting privacy in the wireless era. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems* (San Diego, CA, May 07 09, 2007). G. Hunt, Ed. USENIX Association, Berkeley, CA, 16.

Greenstein, B., McCoy, D., Pang, J., Kohno, T., Seshan, S., and Wetherall, D. 2008. Improving wireless privacy with an identifierfree link layer protocol. In *Proceeding of the 6th international Conference on Mobile Systems, Applications, and Services* (Breckenridge, CO, USA, June 17 20, 2008). MobiSys '08. ACM, New York, NY, 4053.

Grit, L., D. Irwin, A. Yumerefendi and J. Chase. (2006) Virtual Machine Hosting for Networked Clusters: Building the Foundations for Autonomic Orchestration. In *Proceedings of First International Workshop on Virtualization Technology in Distributed Computing* (VTDC).

Gumstix.com, "gumstix", http://www.gumstix.com/

Guttman E. (1999) IETF RFC 2608, Service Location Protocol, Version 2, http://www.openslp.org/doc/rfc/rfc2608.txt, June 1999

Hand, S., T. Harris, E. Kotsovinos and I. Pratt. (2003) Controlling the XenoServer Open Platform. Open *Architectures and Network Programming*, IEEE Conference on, pages 3–11.

Hansen, J. G. and Jul, E. (2004). Self-migration of operating systems. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop* (Leuven, Belgium, September 19 - 22, 2004).

Holtman, K. & Mutz, A. (1998) IETF RFC 2295, Transparent Content Negotiation in HTTP, http://www.ietf.org/rfc/rfc2295.txt, March 1998

IEEE 802.11. IEEE 802.11 LAN/MAN Wireless LANS.

http://standards.ieee.org/getieee802/802.11.html

IEEE. Zigbee: IEEE 802.15 WPAN Task Group 4 (TG4). http://www.ieee802.org/15/pub/TG4.html.

IrDA. IrDA page. http://www.irda.org/

Jeronimo M. and Weast J., (2003) *UPnP Design by Example: A Software Developer's Guide to Universal Plug and Play*, Intel Press.

Johanson, B., Fox, A., & Winograd, T. 2002. The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. *IEEE Pervasive Computing* 1, 2 (Apr. 2002), 67-74.

Johanson, B. and Fox, A. (2002). The Event Heap: A Coordination Infrastructure for Interactive Workspaces. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications* (June 20 - 21, 2002). WMCSA. IEEE Computer Society, Washington, DC, 83.

Juels, A. (2006) RFID security and privacy: A research survey. In IEEE Journal on Selected Areas in Communications

Kernel.org, The Linux Kernel Archive, http://www.kernel.org/

Kozuch, M., & Satyanarayanan, M., (2002) Internet Suspend/Resume, in *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications*, Callicoon, NY

Kozuch, M. , Satyanarayanan, M., Bressoud, T., Helfrich, C. & Sinnamohideen, S. (2004) Seamless mobile computing on fixed infrastructure. Computer, 37(7):65–72

Latour B.(1987). *Science in Action: How to Follow Scientists and Engineers Through*

Society, Milton Keynes: Open University Press

Latour, B. (1988). *Mixing humans and nonhumans together: The sociology of a door-closer. Social Problems* **35**(3): 298-310.

Latour B.(2005). *Reassembling the Social: An Introduction to Actor-Network-Theory,* Oxford University Press

Lindholm, Tim and Frank Yellin. (1996) The Java Virtual Machine Specification. Addison-Wesley.

Micali, S. and Shamir, A.(1990). An improvement of the Fiat-Shamir identification and signature scheme. In *Proceedings on Advances in Cryptology* (Santa Barbara, California, United States). S. Goldwasser, Ed. Springer-Verlag New York, New York, NY, 244-247.

Microsoft.com, "Azure Service Platform", http://www.microsoft.com/azure/default.mspx. Retrieved on August 1, 2009.

Nelson, Michael, Beng-Hong Lim and Greg Hutchins. (2005) Fast transparent migration for virtual machines. In *Proceedings of the 2005 Annual USENIX Technical Conference.*

Nintendo.com, "Wiimote", http://www.nintendo.com/wii, Retrieved July 2009.

NXP Semiconductor, I2C-bus specification (V3.0) (2007), from www.nxp.com/acrobat_download/usermanuals/UM10204_**3**.pdf

openembedded.org, Open Embedded, http://wwwopenembedded.org

Pang, J., Greenstein, B., McCoy, D., Seshan, S., and Wetherall, D. (2007) Tryst: The case for confidential service discovery. In HotNets .

Popek, G. J. and Goldberg, R. P. (1974) Formal requirements for virtualizable third generation architectures. *Commun. ACM* 17, 7 (Jul. 1974), 412-421.

RealVNC, (2009), The RFB Protocol version 3.8, http://www.realvnc.com/docs/rfbproto.pdf. Retrieved July 2009

Rudolph, L. (2001). Project Oxygen: Pervasive, Human-Centric Computing - An Initial Experience. In *Proceedings of the 13th international Conference on Advanced information Systems Engineering* (June 04 - 08, 2001). K. R. Dittrich, A. Geppert, and M. C. Norrie, Eds. Lecture Notes In Computer Science, vol. 2068. Springer-Verlag, London, 1-12.

Ruth, Paul, Junghwan Rhee, Dongyan Xu, Rick Kennell and Sebastien Goasguen. (2006) Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. In *Proceedings of the 3rd IEEE International Conference on Autonomic Computing.*

Saltzer, J., Reed, D., and Clark, D.D. 1984. *End-to-end arguments in system design.* ACM Transactions on Computer Systems, Vol. 2, No. 4, Nov., pp.277-288.

Satyanarayanan M., Kozuch M., Helfrich C., & O'Hallaron D., (2005) *Towards Seamless Mobility on Pervasive Hardware,* Pervasive & Mobile Computing, vol 1, num 2, pp 157-189

Saponas, T. S., Lester, J., Hartung, C., Agarwal, S., and Kohno, T. 2007. *Devices that tell on you: privacy trends in consumer ubiquitous computing.* In Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium (Boston, MA, August 06 10, 2007). N. Provos, Ed. USENIX Association, Berkeley, CA, 116.

Smaldone S., Gilbert B., Toups M., Iftode L., & Satyanarayanan M, (2008) *Smart Phones* *as Self-Cleaning Portable Caches for Infrastructure-Based Mobile Computing,* CMU-CS-08-140

Sousa, J. P. & Garlan, D. (2002) Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. In Proceedings of *the 3rd Working IEEE/IFIP Conference on Software Architecture*. pp. 29-43, August 25-31.

Sud, S.; Want, R.; Pering, T.; Rosario, B.; Lyons, K., 2008 *Enabling rapid wireless system composition through layer2 discovery,* Network, IEEE , vol.22, no.4, pp.1420, July/Aug.

Surie A., Lagar-Cavilla A., de Lara E., & Satyanarayanan M., (2008.) *Low-Bandwidth VM Migration via Opportunistic Replay,* HotMobile '08, Napa Valley, CA

Symbian.org, "Symbian", http://www.symbian.org/about/

Taylor, B. T., & Bove, V. M. (2008). The bar of soap: a grasp recognition system implemented in a multi-functional handheld device. In the extended abstracts on Human factors in computing systems (CHI '08) (pp. 3459-3464). Florence, Italy: ACM Press.

Thurlow R., (2009) IETF RFC 5531, RPC : Remote Procedure Call Protocol Specification Version 2, http://tools.ietf.org/html/rfc5531, May 2009

von Hippel, E., (2007) Horizontal innovation networks - by and for users, *Industrial and Corporate Change Advance Access*, published May 16, 2007, p. 22

Waldo, J. 2000 *The Jini Specifications*. 2nd. Addison-Wesley Longman Publishing Co., Inc.

Want, R., Pering, T., Sud, S., and Rosario, B. 2008. Dynamic composable computing. In *Proceedings of the 9th Workshop on Mobile Computing Systems and Applications* (Napa Valley, California, February 25 26, 2008). HotMobile '08. ACM, New York, NY, 1721.

Weiser, M. (1991). The Computer for the 21st Century. Scientific American, 265 (3), pp. 94-104.

Williams, A. (2002). Zero Configuration Networking, Internet-Draft: Requirements for Automatic Configuration of IP Hosts. from http://files.zeroconf.org/draft-ietf-zeroconf-reqts-12.txt

WiMAX Forum. WiMAX Forum Technical Documents. from http://www.wimaxforum.org/technology/documents/

Windows.com, "Windows CE", http://www.microsoft.com/windowsembedded/en-us/products/windowsce/default.mspx

Wood, T., P. Shenoy, A. Venkataramani and M. Yousif. (2007) *Black-box and Gray-box Strategies for Virtual Machine Migration.* In Proceedings of the Fourth Symposium on Networked System Design and Implementation (NSDI '07).

Ypodimatopoulos, P., *Cerebro: Forming Parallel Internets and Enabling Ultra-Local Economies,* S.M Thesis, MIT 2008