At Home In Service Discovery

Vasughi Sundramoorthy

AT HOME IN SERVICE DISCOVERY

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. W.H.M. Zijm,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op woensdag 20 september 2006 om 15.00 uur

door

Vasughi Sundramoorthy

geboren op 7 mei 1977
te Maleisie

Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. P. H. Hartel

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. Pieter Hartel	University of Twente, promotor
Dr. Kevin Mills	US National Institute of Standards and Technology (NIST), USA
Prof. dr. ir. Boudewijn Haverkort	University of Twente
Prof. dr. ir. Mehmet Aksit	University of Twente
Prof. dr. Wim Vree	Delft University of Technology
Prof. dr. sc. Thomas Plagemann	University of Oslo, Norway
Mr. Antonio Kung	Trialog, France, referent

Cover: Abstract illustration of the three classes of devices, and the communication protocol in FRODO. Uses the traditional "kerawang" motive, typical to Malay wood carvings.

The work in this thesis has been carried out under the auspices of the Institute for Programming Research and Algorithmics (IPA) research school and within the context of the Centre for Telematics and Information Technology (CTIT).

IPA Dissertation Series No. 2006-17
CTIT address: P.O.Box 217, 7500 AE Enschede, The Netherlands
Series title: CTIT PhD. Thesis Series
ISSN number: 1381-3617
CTIT number: 06-93

Printed by Wohrmann Print Service, Zwolle
Cover design by Kalai Kumar Poopalan @ Studio Werkz Concepts

ISBN 90-365-2392-3

Author email: s.vasughi@gmail.com

To my parents and my husband

All the powers in the universe are already ours. It is we who have put our hands before our eyes and cry that it is dark.
Swami Vivekananda

Abstract

Modern computer systems (since 1950s) evolved from being bulky, unreliable and expensive, to being tiny, reliable, cheaper and faster. Improvements in size, expense and performance was due to the evolution in computer hardware; from vacuum tubes, magnetic tapes, integrated circuits, to very large-scale integration (VLSI) of thousands of transistors and other circuit elements onto a single chip. The drop in cost and size led to the introduction of personal computers (PCs) for use in office, schools, and homes. Together with developments such as network technologies (LAN, WAN, WLAN, the Internet, etc), and application softwares, computer systems are no longer just calculators, but multipurpose devices. We use these devices for watching movies, making phone calls, sending emails, controlling remote devices, driving, etc.

The next evolution of computer systems is referred to as *pervasive computing.* In pervasive computing, computers (regardless of size) intelligently and autonomously work together, without human intervention. To achieve this goal, devices will have to be able to automatically discover each other's capabilities, or services. Therefore, *service discovery* has emerged as a potential solution for achieving the objectives of pervasive computing.

This thesis focuses on how to build an autonomous service discovery system within the boundaries of the home. There are two major challenges that this thesis addresses; the first challenge is how to build a reliable system, that is robust against communication and device failures. A reliable system ensures minimum human intervention, while increasing the chances of discovering the available service. The second challenge is to build a lightweight service discovery protocol that can be supported by devices with limited resources (low memory, processing power, etc). This is because the home consists of a variety of devices, with varying resource limitations. Devices from different manufacturers using different technologies should also be supported.

Our contributions to the challenges above are as follows. We introduce a

Framework for Robust and Resource-aware Discovery (FRODO) for the home. FRODO is a reliable and lightweight service discovery protocol that is also portable over a variety of devices. During the design, analysis (through model-checking and simulations) and development of FRODO, we formulate the *Service Discovery Principles*, which are the fundamental properties of a small-scale service discovery system. We also identify the failure recovery rules that facilitate the service discovery system to satisfy these principles. As a result of our design approach, FRODO is robust and suitable for the home environment which may include resource lean devices. FRODO has equivalent or better performance than its competitors.

Samenvatting

Sinds de 50-er jaren zijn computers gevolueerd van volumineus, onbetrouwbaar, duur en langzaam naar klein, betrouwbaar, goedkoop en snel. Deze verbeteringen zijn mogelijk gemaakt door de nog steeds toenemende integratie van componenten, van radiobuizen tot complete computers op een chip. De introductie van de personal computer en het grootschalige gebruik ervan, is het gevolg van deze ontwikkelingen. Computers worden, mede dankzij betere netwerktechnologien (LAN, WAN, WLAN en Internet) en betere software, niet langer uitsluitend gezien als rekenmachines. Ze worden voor meerdere doeleinden ingezet, zoals het verzenden van email, het bekijken van films, of de besturing van apparatuur thuis.

De volgende stap in de evolutie van computers is "pervasive computing", waarbij computers alom tegenwoordig zijn en op een niet-opdringerige manier ondersteuning bieden bij alledaagse taken. Netwerken van computers werken daarbij, zonder direct menselijk ingrijpen, op intelligente en autonome wijze samen. Om dit te bereiken is het noodzakelijk dat computers kunnen ontdekken welke andere computers, en diensten die aangeboden worden, aanwezig zijn in zo'n "pervasive" netwerk. "Service discovery", het automatisch vinden van diensten in een netwerk, is daarom een van de bouwstenen voor de verwezenlijking van pervasive computing.

Het onderwerp van dit proefschrift is een service discovery systeem voor netwerken thuis, waarbij twee grote uitdagingen worden aangepakt. De eerste uitdaging is het maken van een robuust en betrouwbaar systeem, dat diensten kan ontdekking terwijl er fouten kunnen optreden in communicatie en computers. De tweede uitdaging is dit systeem zo klein mogelijk te maken, zodat ook apparatuur met beperkt vermogen (geheugen, rekenkracht, energie) ondersteund wordt. Bovendien moet het mogelijk zijn apparatuur van verschillende fabrikanten met verschillen netwerktechnologien te koppelen.

Het proefschrift adresseert de uitdagingen als volgt. Allereerst wordt een service discovery raamwerk voor netwerken thuis (Framework for Robust and

Resource-aware Discovery: FRODO) gentroduceerd. FRODO is een robuust en lichtgewicht protocol dat geschikt is voor een scala aan verschillende apparaten. Daarna worden de service discovery principes geformuleerd die de fundamentele eigenschappen beschrijven waaraan een service discovery systeem moet voldoen. Opdat een service discovery systeem kan voldoen aan de principes zijn regels gedentificeerd voor het herstellen van de juiste toestand als een fout is opgetreden. Door middel van het toepassen van formele methoden in het ontwikkelproces wordt bewerkstelligd en aangetoond dat FRODO deze regels goed heeft gemplementeerd en voldoet aan de geformuleerde principes. Het resultaat is een robuust service discovery systeem dat even goed, of beter, presteert dan vergelijkbare systemen. FRODO is bovendien, in tegenstelling tot de concurrenten, geschikt voor apparaten met beperkt vermogen.

Acknowledgements

If I have seen further, it is by standing on the shoulders of giants.
Isaac Newton

I take this opportunity to thank the wonderful people who gave me their time, companionship, professional and personal help during my stay in the Netherlands and during the growth of this thesis.

I would first of all like to thank my promotor, Pieter Hartel. Thanks to him, I learnt not just how to do research, but to do "proper" research. He showed me what it takes to write a publication, and taught me how to methodologically produce scientific results. His support for my personal well-being was also immeasurable. Without him, this thesis wouldn't have made it this far.

I thank my daily supervisor, Hans Scholten for our interesting discussions, both on research ideas, and on more general topics. His support allowed me to acquire the tools I required for my research, and travels to various workshops and conferences. I also learnt much about the wonderful Dutch culture and people from him.

Kevin Mills and Christopher Dabrowski from the US National Institute of Standards and Technology (NIST) built the platform on which this thesis is standing. They showed me that service discovery is more than simply a set of libraries. For the last 3 years, they patiently answered my emails, contributed to various discussions and guided me through several experiments. I also thank them for the opportunity to work in NIST as a Foreign Guest Researcher. The result of the close contact I had with them produced the core of my thesis.

I extend a heartfelt appreciation to my promotion committee; Prof. Pieter Hartel, Dr. Kevin Mills, Prof. Boudewijn Haverkort, Prof. Mehmet Aksit, Prof. Wim Vree, Prof. Thomas Plagemann and Mr. Antonio Kung for reviewing and commenting on this thesis. I appreciate and respect your esteemed knowledge, dedication and commitment.

The time I spent in NIST allowed me to make the acquaintance of Ceryen Tan from M.I.T. During the formal verification of FRODO, we practically talked service discovery to death! His unique perspective to various issues was priceless. Gerard van de Glind, my Master student helped shape the FRODO protocol to its current state. Our online chat sessions during my stay in the US is still burning the wires... Jerry den Hartog with his brilliant mind and good humor helped me to abstract problems and solutions. Who needs a model checker, when we can have Jerry on the team? Theo van Klaveren developed FRODO in C, for which I am grateful. His efficiency and insight are invaluable.

My discussions with Sape Mullender, though brief, instilled important perspective into research directions in distributed systems. Pierre Jansen, Maria Lijding, Nirvana Meratnia, Ferdy Hanssen, Hylke van Dijke, Angelika Mader, and Georgi Koprinkov contributed to various discussions on several research questions. I also thank Marlous and Nicole for their administrative support.

I consider myself lucky to be part of a fantastic group of international researchers. Ricardo, Laura, Kavitha, Malohat, Jordan, Law, Supriyo, Stephan, Nikolay, Anka, Illeana, Raluca, Mihai, Ozlem, Michel, Mohammad, Yuanching and the list goes on and on. Please keep the party going!

I thank Kavitha and her husband Kiran for being such good friends. Kavitha, I appreciate your silent sufferings as my sounding board for my various problems in the last two years. To my other beloved confidant and parinimphen, Laura, you are simply "ammmaaazing!". Malohat and Mohabbat, your caring warmth in the final stages of my thesis was priceless. Sheela, your vivaciousness helped keep the loneliness away, for which I will always be grateful.

I thank Valerie Jones for her warmth and kindness when listening to my problems. I am also indebted to Anandhita for caring for me during the time I was sick at a critical point of my work (amazing doctor!). I also appreciate Sandro for insisting that I should stop working late nights! I extend a special"thank you" to Marijke Smit. She introduced me to the colorful world of "buikdansen". I will always love her for her warmth, and friendship, along with her insight into different cultures. You and Pieter will always have a home in Malaysia.

I would like to thank my Indian friends in Twente; Vijay Iyer, Pramod, Vishaka, Jay, Deepa, Vishy, Vishwas, Ambati, Rama, Ravi, Madhavi, Komal, Denni, and the many others for their support and friendship.

My brother-in-law, Kalai Kumar beautifully designed an abstract illustration of FRODO for the thesis cover, using a traditional Malay art.

Finally, I extend my deepest gratitude to my loving family. My parents have always believed in me, while teaching me to look and think beyond the boundaries. I thank my siblings; Vijaya, Preveen, Jegan, and Neesha, for their constant support. The unwavering love and sacrifice of my best friend and husband, *Siva* gave me the strength to start and finish my PhD.

Enschede, *Vasughi Sundramoorthy*
September, 2006

Contents

Chapter 1

Introduction

Mathematicians stand on each other's shoulders,
while computer scientists stand on each other's toes.
R. W. Hamming

This thesis fits within the context of three interconnected paradigms; autonomous computing, pervasive computing and pervasive home computing. We first explore the existing challenges in all three paradigms. We then describe the thesis focus and the Research Question for the home environment. We conclude by giving an overview of the contributions in this thesis.

1.1 The Context

Computer scientists can learn much from how the human body manages itself autonomously, and apply the same techniques to building distributed systems. This is how *autonomous computing*, an initiative of IBM [Mur04] sees the future of computer systems. An autonomous system has at least one of the following four properties [Gan03]:

(A1) Self-configuring. Systems that adapt automatically to dynamically changing environments. The systems can dynamically "on-the-fly" add new hardware and software to the system infrastructure with no disruption of services.

(A2) Self-healing. Systems discover, diagnose and react to disruptions. The objective of self-healing is to minimize outages to keep applications available at all times.

(A3) Self-optimizing. Systems monitor and tune resources automatically. Self-optimization enables hardware and software systems to maximize resource

utilization to meet end-user needs without human intervention. Resource allocation and workload management must allow dynamic redistribution of workloads to systems that have the necessary resources to meet workload requirements.

(A4) Self-protecting. Systems anticipate, detect, identify and protect themselves from threats. Self-protecting systems must have the ability to define and manage access to computing resources, to protect against unauthorized access, to detect intrusions and report and prevent these activities as they occur.

Autonomy of distributed systems is also one of the fundamental characteristics of the more ambitious *pervasive (or ubiquitous) computing*. The vision of pervasive computing is elegantly articulated in Mark Weiser's acclaimed seminal paper published in 1991 by Scientific American [Wei91]:

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it."

To realize the Weiser vision of pervasive computing, Satyanarayanan [Sat01] and Saha and Mukherjee [Sah03] pose the following research challenges:

(B1) Effective use of smart space. A smart space is an environment that has embedded computing capability in its surrounding physical infrastructure or material. One example of a smart space is the home environment, where a home owner's electronic profile allows the temperature and lighting in the home to adjust. Tangible user interfaces [Kle04; Sco02] effectively augment the physical world by integrating digital information with everyday physical objects, such as intelligent cups [Gel99], and clothing [Man96].

(B2) Invisibility. Human intervention should be so minimal, that technology disappears into the background of everyday life. Therefore, a pervasive system should continuously meet user expectations and rarely present unpleasant surprises.

(B3) Resource-constraints. The system should be able to support resource-lean devices. Embedded, wireless and mobile devices have weaker local resources (e.g. battery power, memory, processing power), compared to static and wired devices [Sat96].

(B4) Heterogeneity. The system should adjust, integrate and mask differences in environment, network, device and system platform.

(B5) Context-awareness and management. Applications should have perception on their environment, make timely and effective decisions, and act accordingly.

(B6) Localized scalability. Increased density of embedded devices in one area should not have severe implications on bandwidth, energy and distraction for mobile users.

(B7) Security and privacy. Devices embedded in smart spaces (and worn on our bodies) communicate seamlessly with any number of remote devices. In these interactions, information is shared and exchanged. Therefore, there is an increased risk to security and privacy.

This thesis focuses on the autonomy in a *pervasive home system*. The pervasive home system subsumes the constraints of pervasive computing, and also has challenges of its own. These challenges are illustrated by Edwards and Grinter from Xerox Palo Alto Research Center as the *Seven Challenges* for a ubiquitous home [Edw01].

(C1) No system administration at home. Unlike the more professional office environment, the home has no system administration. Therefore, self-configuration and self-healing are even more important in the home than in another environment like an office.

(C2) Reliability. Home owners are less tolerant to their appliances crashing than desktop users (a TV should work at least for 5 years). Therefore, the practice of designing for reliability must be integrated into the development cultures that build smart home technologies (by devoting substantial time and resource). The tradeoff to reliability is simplicity (design becomes more complex because of fault-tolerant measures), intelligibility, and ease of administration, which are also requirements for domestic technologies.

(C3) Impromptu interoperability. The home consists of devices with a number of components, acquired at different times, from different manufacturers, and created under different design constraint and conditions. And yet, devices should interconnect with each other with little or no advance planning or implementation. This challenge goes beyond issues of standardization for defining how devices should interoperate. New connectivity models are needed, beyond simple prior agreement or standard protocols and interfaces.

(C4) Understand and manage unpredictable technologies. Home owners need to be made aware of how technologies work in an easy-to-understand way. This is so that they can deal with unpredictable behaviors (such as when our wireless speakers connect with a neighbor's network). Therefore, "accidents" should be understood, repaired, or better yet, prevented, when new devices are added, old devices are removed, devices from different manufacturers coexist, and wireless connectivity extends beyond the walls of the home.

(C5) Social implications. Studies have to be done on how smart home technologies impacts the dynamics of the home, and society itself. Examples include studying the impact on privacy issues, and beliefs on good parenting.

(C6) Inference in the presence of ambiguity. The system should correctly infer human state and intent, based on received inputs, and take the necessary actions. The key question is on how much inference is required, and the problems caused by uncertain inferences and decisions. Context-aware applications usually concern themselves with the rules of inference (e.g. "when family members are gathered in the kitchen, heat coffee"). Work has to be done on determining how capable is the system in detecting or arriving at a conclusion. Provisions should be made for the user to override the system behavior.

(C7) Adapt technology for domestic use. Technologies should be grounded for the realities of the home. The stable and compelling routines of the home override even the abilities of the technology itself. Therefore, studies have to be carried out to understand how home owners appropriate and adopt new technologies (an example success story is the pervasive use of telephones).

Figure 1.1 shows the relationship among the research challenges for autonomous, pervasive and home computing. *Invisibility (minimum human intervention)* is a fundamental attribute of a pervasive system. *Invisibility* is achieved by implementing autonomous behaviors; self-configuration, self-healing, self-optimization and self-protection. The pervasive home environment inherits all the research challenges from pervasive computing, except for localized scalability (although it can be argued that future homes might require scalability because of a very large number of sensors).

1.2 Thesis Focus: Autonomous Service Discovery

All the challenges listed above for autonomous computing (A1 to A4), pervasive computing (B1 to B7), and pervasive home computing (C1 to C7) pose interesting research questions. However, as illustrated in Figure 1.1, self-configuration is a central theme in each of the three domains. One of the most widely used techniques contributing to self-configuration is service discovery. This observation, combined with our specific interest in the home environment gives rise to the following Research Question:

Research Question: *How to enable a variety of home appliances to discover each other's services effectively and efficiently?*

To address this Research Question, we focus on building a *service discovery system* with properties appropriate to the home environment. In the wider context, service discovery is a fundamental step for intelligent applications, before they can collaborate to perform a certain function. Service discovery in pervasive computing acts as a naming system (analogous to Domain Name Service for the Internet), but goes further by autonomously coordinating the discovery of

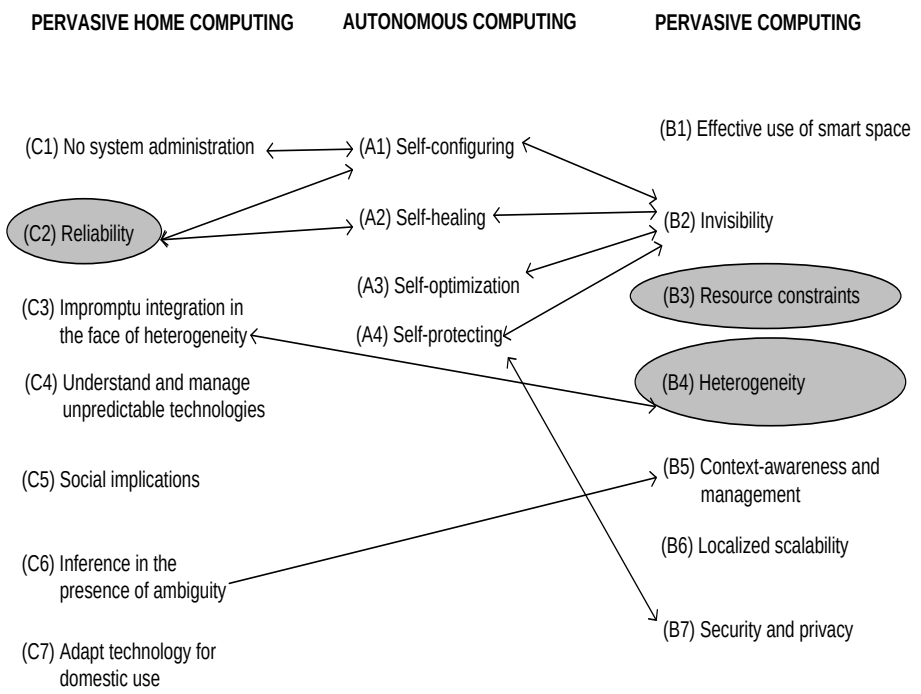


Figure 1.1: *Relating the research challenges in autonomous, pervasive and home computing. Arrows are used for challenges that are interrelated in the three paradigms. Self-configuration (A1) is an inherent property of service discovery, which removes (or requires only minimal) system administration (C1). We show that by implementing the properties of autonomous computing, invisibility (B2) is achieved. We specifically focus on the 3Rs; Reliability (C2), Resource-constraints (B3) and heteRogeneous devices and network (B4).*

new services, detection of defunct services, and updating applications on service availability and state changes. Hence, service discovery offers *self-configuration (A1)* to pervasive systems so that devices can "plug-and-play", with *no (or minimum) additional intervention (C1).* Therefore, A1 and C1 are inherent properties of service discovery.

The specific contributions of this thesis extend service discovery for the pervasive home by targeting what we call the 3Rs: *R*eliability, *R*esource-constraints and hete*R*ogeneity. We choose these challenges because they directly address the Research Question: a reliable system is *effective* if it can cope with communication and node failures. We take into consideration resource-constraints because we require an *efficient* utilization of resources. We also mask heterogeneity of devices and networks because the home consists of a *variety* of device architectures and networks (e.g. wired, wireless).

(C2) **Reliability**. We incorporate various fault-tolerant measures in our system design so that the system can recover from communication and node failures. Therefore, the system regulates its own recovery; the system *self-heals (A3)*.

(B3) **Resource-constraints**. Resource-awareness is required because new, sophisticated technologies should not increase the cost of devices. Since cost increases with resource usage, the system is made resource-aware. A resource-aware system not only reduces cost of implementation, but also makes available the services of resource lean devices.

(B4) **Heterogeneity**. The home consists of devices with a variety of different architectures and network connectivity. To enable these devices to discover each other, service discovery should minimize dependencies on the capabilities of underlying protocol stacks. Therefore, our service discovery protocol is portable over heterogeneous devices and networks.

The thesis focuses on the communication aspects of service discovery for the home environment. We do not address how to efficiently describe services (service ontology). This is because we give more importance to the 3Rs, which are not common factors in existing service discovery protocols. However, this work can be extended with existing formats of service descriptions. Furthermore, we believe that the results presented in this thesis are applicable not only for the home environment, but also for other ad-hoc smart spaces (such a conference venue, or an office environment). This is because the 3Rs are equally important in the home as elsewhere. On the other hand in other domains, scalability and security may be more important than in the home; these properties are beyond the scope of the thesis. From now on, we only focus on the selected challenges which we call as the 3Rs in this thesis. We do not discuss the rest of the As, Bs and Cs listed in Section 1.1.

Now that we have identified the research focus and objectives, we provide the thesis overview and the contributions in the next section.

1.3 Thesis Overview

We begin by describing the state of the art in service discovery in the next chapter. In Chapter 3, we zoom in on small-scale systems and state the fundamental properties of service discovery in seven *Service Discovery Principles*. We also give various self-healing methods to satisfy the principles. In Chapter 4, we describe our design approach. We then give an overview of our own service discovery system which satisfies the 3Rs. We analyze and evaluate our design choices in Chapter 5. Finally, we summarize the contributions in the thesis, and suggest future work in Chapter 6.

We now elaborate further the contributions of each chapter in some detail.

Chapter 2. We describe the general design space of service discovery, and give a taxonomy of the state of the art. We begin by clarifying the fundamental concepts of service discovery, in relation to the distributed system paradigm (architecture, functionality, and related distributed system models). We then specify the operational aspects that impact the design of a service discovery system, and give an overview of the design space. We proceed to give a taxonomy that first analyzes state of the art solutions with respect to the operational aspects, before comparing their functionalities. The result of our analysis enables us to identify opportunities for design improvements for the home, with respect to the 3Rs.

Chapter 3. We focus on how to provide *reliability* in service discovery, which is one of the challenges in the 3Rs. A reliable service discovery system is *effective* against communication and node failures, thus satisfying the effectiveness criteria in the Research Question. Towards this goal, we specify the functional principles of service discovery for small-scale systems. We present seven *Service Discovery Principles* that state the fundamental properties of service discovery. We then focus on how to satisfy the principles for consistency maintenance. Consistency maintenance allows applications to receive updates on service attributes. Our analysis reveals that to effectively regain consistency in the face of failures, correct behavior of the functionalities in service discovery is required. Therefore, we provide a novel classification of failure recovery rules that facilitate the system to satisfy the Service Discovery Principles.

Chapter 4. We focus on building a service discovery system for the home environment that satisfies the 3Rs. We first describe our design approach which inspired the formulation of the Service Discovery Principles and the recovery rules presented in Chapter 3. We then present a *Framework for Robust and Resource-aware Discovery* (FRODO), our service discovery system built for the home environment. We explore the design choices in FRODO, which determine a subset of the design space described in Chapter 2. FRODO is reliable in the face of communication and node failures, supports resource-constrained devices, and is portable over heterogeneous devices and networks.

Chapter 5. We analyze and evaluate our design choices in FRODO, using several different techniques such as model-checking, simulation, and prototyping. To our knowledge, no other service discovery design has been analyzed this thoroughly during its development. We model-check our design to identify and rectify design errors. The result of the model-checking is a list of hard-to-detect design errors that once rectified, enable our FRODO model to satisfy the Service Discovery Principles. We also use simulations to compare the performance of FRODO against Jini and UPnP, two well-known service discovery protocols appropriate for the home. As a result of our design efforts, the *responsiveness, effectiveness* and *efficiency* of our design is better than that of Jini and UPnP at failure rates

which are realistic in the home environment. Finally we describe the implementation of our prototype, and validate a selected set of simulation results against the implementation. The prototype is a resource-lean realization of the FRODO protocol.

Chapter 6. We summarize our contributions in this chapter. We also highlight several future research directions for the FRODO system, and for service discovery in general.

The contributions of this thesis are a step towards achieving the vision of a pervasive home system. We hope to convince the reader that the design concepts and principles apply not only to the home environment, but are also applicable to similar smart spaces.

1.4 Publications

To conclude, we list our refereed conference publications and technical reports that constitute the core of the thesis.

1.4.1 Refereed Publications

1. Sundramoorthy, V. and Scholten, J. and Jansen, P.G. and Hartel, P.H. Service discovery at home. In 4th International Conference on Information, Communications & Signal Processing and 4th IEEE Pacific-Rim Conference On Multimedia (ICICS/PCM), Singapore, Dec 2003, pp. 1929-1933. IEEE Computer Society Press.

2. Sundramoorthy, V. and Tan, C. and Hartel, P.H. and den Hartog, J.I. and Scholten, J. Functional Principles of Registry-based Service Discovery. In 30th Annual IEEE Conference on Local Computer Networks (LCN), Sydney, Australia, Nov 2005. pp. 209-217. IEEE Computer Society Press.

3. Sundramoorthy, V. and van de Glind, G.J. and Hartel, P.H. and Scholten, J. The Performance of a Second Generation Service Discovery Protocol In Response to Message Loss. In 1st International Conference on Communication System Software and Middleware (COMSWARE), New Delhi, India, Jan 2006. pages to appear. IEEE Computer Society Press.

4. Sundramoorthy, V. and Hartel, P.H. and Scholten, J. On Consistency Maintenance In Service Discovery. In 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Rhodos Island, Greece, Apr 2006. pp. 10 in CDROM. IEEE Computer Society Press.

1.4.2 Technical Reports

1. Sundramoorthy, V and van de Glind, G.J. Frodo High-Level and Detailed Design Specifications version 1.0. Technical Report TR-CTIT-06-25, Enschede, June 2006.

2. Sundramoorthy, V and Tan, C. Frodo Dt-Spin Models version 1.0. Technical Report TR-CTIT-06-27, Enschede, June 2006.

Chapter 2

A Taxonomy of Service Discovery

By viewing the old we learn the new.
Chinese Proverb

2.1 Introduction

Although our work is within the scope of the home environment, we first analyze the general concepts and issues in service discovery. This analysis places service discovery in the context of distributed systems by describing service discovery as a third generation naming system. Next, we describe the different architectures in service discovery. We then specify the objectives and functions of service discovery. We show how these functions are implemented, in relation to the different models in the distributed systems. We then proceed to show how service discovery fits into a system, by characterizing operational aspects. Subsequently, we describe how existing state of the art performs service discovery, in relation to the operational aspects and functions of service discovery.

Contributions: This work diverges from existing surveys [Bet00; Van05; Ric00] which categorize only the functional features of service discovery protocols, based on architectural and programming platform differences. We aim to:

1. Clarify the fundamental concepts of service discovery in relation to the distributed system paradigm.
2. Specify the operational aspects that impact the design of a service discovery system, and the resulting design solutions.
3. Provide a taxonomy that first analyzes state of the art solutions with respect to the operational aspects, before comparing functional behaviors.

4. Identify opportunities for design improvements to produce a small-scale, unattended system for the home.

This chapter is organized as follows. In Section 2.2, we provide an understanding of service discovery as a third generation name discovery system. In Section 2.3, we describe the service discovery system in terms of the participating entities, architecture and topology. We define the service discovery objectives and functions in Section 2.4. We identify the fundamental distributed models of service discovery in Section 2.5. In Section 2.6, we analyze the operational aspects for a service discovery system. We proceed to summarize a selection of widely used service discovery systems in Section 2.7. In Section 2.8, we give a taxonomy of state of the art solutions to the operational aspects, compare the functional implementations, and identify areas for improvements. We conclude our findings in Section 2.10.

2.2 Service Discovery: Third Generation Name Discovery

Service discovery allows applications in a distributed system to discover and share network entities. This goal is shared by a class of distributed systems that we refer to as *name discovery systems*. A *name* is a string of bits that is used to identify a variety of entities, such as computers, peripherals, applications, remote objects, files, etc. In the context of name discovery systems, a name is not the address of an entity (addresses are uninterpreted bit patterns such as Ethernet addresses [Nee93]), but a name is the persistent identifier for the entity, or human understandable textual description [Tan02b]. Consistently named entities enable computers to communicate with one another via a distributed system, and share (access to) the entities [Cou05]. A name has a list of *attributes* associated with it. An attribute is a name-value pair that describes a property of the entity

We classify *name services*, such as Grapevine [Bir82], GNS [Lam86] and DNS [Moc88], developed in the 1980s as first generation name discovery systems. A *name server* stores a set of bindings between the name and the attribute list of an entity, and resolves queries for the entity. A query based on the name simply returns the list of attributes that describes the entity.

Directory services such as Profile [Pet88], Univers [Bow90], X.500 [Cha94] and LDAP [How99], developed in the 1990s are classified as second generation name discovery systems. Directory services provide a more powerful mechanism for querying entities. Directory services perform *attribute-based queries* that return the names associated with the attribute. An example is a query that on the basis of a given telephone number, returns the name of the associated employee.

The first two generations of name discovery systems share the following context and limitations:

- Context: Computer-based environment with predominantly wired connectivity, where nodes are mostly static, names and attributes rarely change,

and the system is reliable

- Limitation: Static infrastructure of servers and directories, that requires configuration and maintenance by privileged system administrators.

Service discovery inherits the fundamental concepts of traditional name discovery systems, where entities are named according to a naming standard, and attribute-based queries return the names of the matching entities. A *service* is defined as the following:

A service is a distinct part of a computer system that manages a collection of related entities and presents their functionality to users and applications [Cou05].

We classify service discovery as a third generation name discovery system because service discovery satisfies the requirements of ubiquitous computing [Wei91], by *enlarging the context and relaxing the limitations of traditional name discovery systems.*

- Context: An environment consisting of a variety of embedded devices (not just PCs), with wired and wireless connectivity, static and mobile nodes, which undergoes frequent changes of resource availability and attribute values, and is vulnerable to failures. Therefore, service discovery is a fundamental building block for pervasive computing.
- Solutions: Service discovery systems deliver the promises of pervasive computing with the following solutions:

  1. **Service discovery provides self-configuring and self-healing capabilities for a spontaneous network of devices.** A service discovery system allows entities to enter and leave the system automatically, with *minimum supervision and maintenance.*
  2. **Service discovery supports access to services.** Traditional name discovery systems typically discover information, such as an IP address for a given name in DNS, employee information or phone numbers in LDAP. A service discovery system supports access to services, where discovered service descriptions may contain executable programs, or URLs to the service.

Therefore, service discovery is a solution for naming and discovering resources in a versatile, but dynamic network of devices.

2.3 Service Discovery Architecture

This section describes the entities and the different architectures of service discovery.

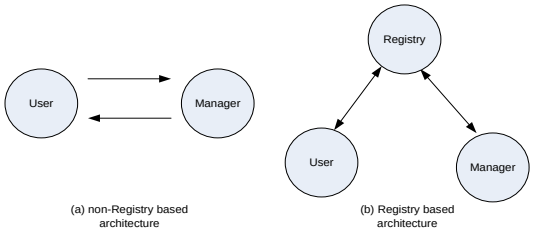


Figure 2.1: *(a) The non-Registry architecture consists of Users and Managers that multicast queries and service advertisements. (b) The Registry architecture uses unicast for registering services and sending queries.*

A service is specified by a *Service Description (SD)*, which typically describes the service in terms of: (1) device type (e.g. printer), (2) service type (e.g. color printing) and (3) attributes (e.g. location, paper size). There are three types of entities in a *service discovery system*; a *Manager* owns the SDs, a *User* has a set of requirements for the services it needs, and a *Registry* caches available services so that Users can discover the services through *queries* to the Registry. A node can behave as a User, Manager, Registry or a combination of these roles.

There are two basic types of architecture in service discovery, as shown in Figure 2.1; *Registry* and *non-Registry* based. In the Registry-based architecture, Registries can be deployed by a system administrator, or automatically elected by the nodes in the system. Once the Registry is available, the rest of the nodes in the system will have to discover the Registry before services can be registered and queried. Unicast communication in the Registry-based architecture reduces network traffic, thus increasing scalability. In the non-Registry-based architecture, Users and Managers can perform multicast queries and service *advertisements*. Therefore, unlike the Registry-based architecture, the system is not vulnerable to single point of failure issues. However, since extensive multicast is used, network traffic increases, causing scalability issues.

In the non-Registry architecture, there are two types of logical topologies: (1) *Meshed topology*, as shown in Figure 2.2(a), where all entities receive each other's multicast queries and service advertisements. (2) *Clustered topology*, where entities form clusters based on some criteria (e.g. service type or location). Members of a cluster communicate only with each other, thus service advertisements and queries are limited within the cluster. A node may also belong to a combination of clusters, and therefore has a wider scope for service discovery. Figure 2.2(b) gives an example of the cluster-based topology.

The Registry architecture has four types of logical topologies: (1) An *unconnected Registry topology*, as shown in Figure 2.3(a). In this topology, Registries do not communicate with each other, but Managers and Users may associate themselves with multiple Registries. (2) A *meshed Registry topology*, as shown in Figure 2.3(c), where Registries communicate with each other as peers. A Registry

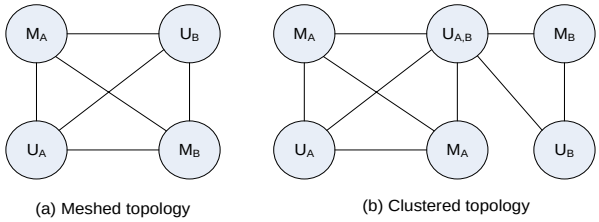


Figure 2.2: *Logical non-Registry topologies. (a) In the meshed topology, Users, U and Managers, M can listen to each other's queries and service advertisements. (b) In the cluster-based topology, Users, U and Managers, M may form a logical cluster according to some criteria. A and B denotes two types of clusters, where $U_{A,B}$ belongs to both clusters, and is able to discover services of both clusters.*

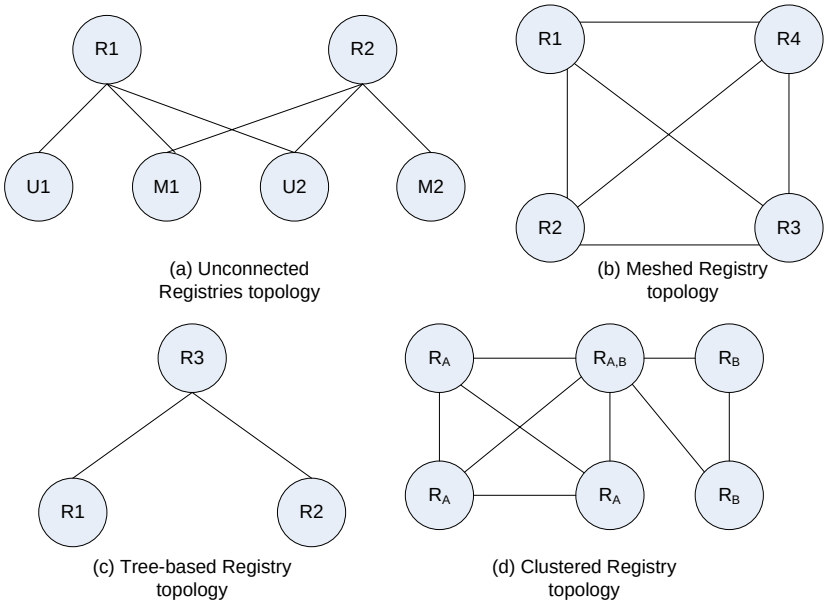


Figure 2.3: *Logical Registry topologies. (a) In the unconnected Registry topology, Registries, $R1$ and $R2$ do not communicate with each other, but User, $U1$ and Manager, $M2$ may register and discover services from both $R1$ and $R2$. (b) In the meshed Registry topology, Registries are peers to each other, and forward messages to all their peers. (c) In the tree-based Registry topology, Registries $R1$ and $R2$ are child Registries of $R3$. Child Registries may forward messages to parent Registries. (d) In the clustered Registry topology, Registries optimize the tree or mesh topology by limiting query processing to a select few Registries. A and B are two clusters, where Registries, R_A and R_B can only communicate with the members of their own cluster. $R_{A,B}$ is able to communicate within both clusters.*

forwards queries and replicas of its cache to all its peers. (3) A *tree-based Registry topology*, where Registries form a parent-child relationship, based on some criteria (such as location or resource-constraints). A child Registry, such as shown in Figure 2.3(b) forwards queries to its parent when it does not find matching services within its own cache. The query may traverse all or parts of the hierarchy, depending on some query processing optimization criteria. (4) A *clustered Registry topology*, as shown in Figure 2.3(d), where Registries form clusters based on service type or location. This topology optimizes the tree-based and meshed topology, where query processing is done only by a select few Registries.

2.4 Service Discovery Functions

We state the main objectives of service discovery as:

O1: *Discover services that match requirements*

O2: *Detect changes in service availability and attributes*

Toward accomplishing these objectives, we classify service discovery tasks into four main *functions*; Configuration Discovery, Service Registration, SD Discovery and Configuration Update. The term "configuration" refers to the entities in the system: Manager, User and Registry. The Configuration Discovery and Service Registration functions exist to facilitate service discovery. O1 is accomplished by the SD Discovery function, while O2 is accomplished by the Configuration Update function. Each of the four functions can be accomplished using several different *methods*. We use italics to indicate the methods:

1. **Configuration Discovery** - This function allows Registries to be setup, and identities of entities (e.g. Registries or cluster members) in the system to be discovered. There are two sub-functions of Configuration Discovery:

   (a) Registry auto-configuration - Allows the system to configure one or more Registries automatically through (a) *Registry election* algorithms, or (b) *Registry reproduction*, where a parent Registry spawns a child Registry. The Registry election or reproduction is done based on some criteria such as resource superiority, load threshold, service type or location. Registry auto-configuration is done on the fly, without supervision.

(b) Entity discovery - Allows entities in the system to discover a Registry or cluster through (a) *active discovery*, where nodes initiate the discovery by sending announcements, or (b) *passive discovery*, where nodes discover the required entities by listening for announcements. In some systems, discovery via active and passive methods is integrated with the underlying routing protocol to optimize bandwidth utilization.

2. **Service Registration** - This function allows Managers to register their services at a Registry. Registration methods include (a) *unsolicited registration*, where nodes request the Registry to register their services and (b) *solicited registration*, where Registries request new nodes to register. The Registry keeps a cache of available SDs, and updates them according to requests from the Managers.

3. **SD Discovery** - This function allows Users to obtain SDs that satisfy their set of requirements. Users may cache the discovered SDs to reduce access time to the service, and reduce bandwidth utilization by avoiding multiple queries. There are two sub-functions in SD Discovery:

   (a) Query - This is a pull-based model where Users initiate (a) *unicast query* to a Registry, or (b) *multicast query*. The query specifies the requirements of the User. The Registry or Manager that holds the matching SD replies to the query.

   (b) Service notification - This is a push-based model, where Users receive (a) *unicast notification of new services* by the Registry, or (d) *multicast service advertisements* by Managers.

4. **Configuration Update** - This function monitors the node and service availability, and changes to the service attributes. There are two sub-functions in Configuration Update:

   (a) Configuration Purge - Allows detection of disconnected entities through (a) *leasing* and (b) *advertisement time-to-live (TTL)*. In leasing, the Manager requests and maintains a lease with the Registry, and refreshes the lease periodically. The Registry assumes that the Manager who fails to refresh its lease has left the system, and purges its information. With TTL, the User monitors the TTL on the advertisement of a discovered Manager. The User assumes the Manager has left the system if the Manager fails to advertise before its TTL expires, and purges its information.

   (b) Consistency Maintenance - Allows Users and Registries to detect updates on cached SDs. Updates can be propagated using (a) push-based *update notification*, where Users and Registries receive notifications from the Manager, or (b) pull-based *polling for updates* by the User

Table 2.1: *Service discovery functions, methods and related distributed system models*

Function	Subfunction	Method	Distributed model
Configuration Discovery	Registry auto-configuration	(a) Registry election, (b) Registry reproduction	Periodic announce / listen
	Registry discovery	(a) active discovery, (b) passive discovery	
Service Registration		(a) solicited registration, (b) unsolicited registration	Caching, replication, periodic announce / listen
SD Discovery	Query	(a) unicast query, (b) multicast query	Query processing, caching, periodic announce/ listen
	Service notification	(a) Registry notification, (b) multicast service advertisement	
Configuration Update	Configuration Purge	(a) leasing, (b) advertisement TTL	Periodic announce / listen, eventual consistency, leasing
	Consistency Maintenance	(a) pull-based polling for update by Users, (b) push-based update notification by Registry to Users, (c) push-based update notification among Registries	Eventual consistency, Publish/subscribe

to the Registry or Manager for a fresher SD. (c) In a multiple Registry topology, push-based *update notifications among Registries* can be done to achieve consistency.

A service discovery function implements one or more distributed models to accomplish its tasks. In the next section, we describe the distributed models of service discovery.

2.5 Distributed Models Of Service Discovery

We now describe the fundamental models on which all implementations of service discovery are based.

A service discovery system inherits the general characteristics of *coordination-based* distributed systems. A coordination-based distributed system separates computation and coordination. The computation part allows a number of processes to manipulate data, while the coordination part is responsible for communication and cooperation between the processes [Pap98]. The coordination model

fully decouples the communicating entities (User, Manager and Registry) in terms of time and space, thus providing asynchronous communication and operations.

- *Space decoupling:* the communicating entities do not need to know each other. A User subscribes to receive messages related to the *subject* it is interested in (e.g. printing service). The Manager publishes an event to an event service (can be located in the Registry). The Manager does not need to hold references to the subscribers (e.g. destination addresses), nor does the Manager need to know how many Users have subscribed to its service. Similarly, the subscribed User does not need to hold references to the Managers, neither does it need to know how many Managers are participating in the interaction. This model is typically used when point-to-point communication needs to be kept minimum. Furthermore, resource-lean entities can depend on more powerful entities to manage event dispatching.

- *Time decoupling:* the communicating entities do not need to be actively participating in the interaction at the same time. The Manager can publish some events while the subscribed User is disconnected, and conversely, the User can be notified about the the event when the Manager that published the event is disconnected. This model is useful in environments where entities frequently connect and disconnect, such as in mobile networks.

Based on the concepts of coordination-based distributed systems, service discovery systems typically comprise a combination of the following distributed models:

1. ***Periodic announce/listen*** - Most service discovery protocols combine periodicity, and the announce/listen model to detect new services or nodes [Fen97; Han98]. The model helps Users to detect the presence of the Registry and the Manager. The lack of an expected announcement indicates to the User that the Manager has left the system. This model is applied in all four service discovery functions, and is especially useful to recover from communication and node failures.

2. ***Caching, replication and consistency*** - Caching is a technique for improving scalability and performance in distributed systems. In Service Registration, the Registry caches available SDs. In SD Discovery, Users typically cache the discovered SDs to reduce the access time to the service, and bandwidth usage. A service discovery system may contain a set of replicated Registries to increase reliability and improve performance. Caching and replication requires consistency maintenance to preserve the integrity of the cached SD. Service discovery complies with *eventual consistency*, because it is client-centric, which tolerates transient inconsistencies, just like Bayou [Pet96], the distributed database system for mobile users. The eventual consistency model is applied in Configuration Update.

3. ***Leasing*** - Leasing [Gra89], is a time-based mechanism that provides efficient consistent access to cached data in distributed systems. In service discovery. leases between communicating entities are used for automatic garbage collection. The lessor periodically refreshes its lease with its counterpart to indicate its continuous existence, or interest in a service. The lessee purges an entity that fails to refresh its lease. Flexible leasing, as proposed by Duvvuri et al. [Duv03] can also be used so that the expiration time is adapted to suit the demand of the lessor, while the lessee can hold the upper limit on the maximum lease period for a service. This model is typically used in Configuration Purge.

4. ***Query processing*** - Query processing [Jar84] has been extensively explored in conventional database systems [Gad85; Leh86]. In the context of service discovery, SD Discovery may apply a query processing model to optimize query propagation, such as by using DHTs [Sto03; Row01] to reduce communication cost. The function also uses query processing when the Registry matches the requirements of the the User against registered SDs.

5. ***Publish/subscribe*** - the publish/subscribe mechanism [Oki93] provides asynchronous communication via loose decoupling of space and time, and benefits large scale settings such as the Internet [Not04], and mobile environments [Hua01]. Subscribers express their interest in an event, or a pattern of events, and are subsequently notified of any event, generated by a publisher, which matches their registered interest. The event is asynchronously propagated to all interested subscribers, usually through multicast. This model is typically used in Consistency Maintenance.

Table 2.1 summarizes service discovery functions, methods and the related distributed models for each function. Service discovery systems use a combination of models described above as the building blocks of its functions. Every service discovery system implements the functions according to its own design rationale.

2.6 Operational Aspects of Service Discovery

This section analyzes the design aspects related to the *operational environment* of service discovery systems, and lists solutions found in the wider distributed system paradigm, but tailors them to the service discovery context.

The operational environment influences the design rationale of service discovery systems. For example, a stable, wired office environment, with good system administration may not require too much emphasis on fault-tolerance towards communication and node failures. However, in the context of the less controlled home-environment, it becomes a necessity, because home owners are not restricted in how they manage their appliances (unplugging, moving). In a wireless, mobile environment, the system becomes even more vulnerable to certain communication

and node failures. We identify the following as design aspects for service discovery in ubiquitous computing:

1. ***System size*** - We define "system size" in terms of two dimensions: distance and the number of nodes. Small sized systems such as *Personal Area Networks (PAN)* and *Local Area Networks (LAN)* contain a limited number of nodes, and do not require a high degree of scalability. Large systems such as *Metropolitan Area Networks (MAN)* and *Wide Area Networks (WAN)* including the Internet require a scalable service discovery system. Scalability measures include setting up multiple Registries, whether in a tree or mesh topology, and applying query optimization and load balancing techniques to conserve bandwidth.

2. ***Lossy environment*** - Service discovery systems in wireless and mobile networks must assume that they will operate in a lossy environment with communication and node failures. *Communication failures* include message corruption, message loss and link failures. Message corruption is due to interference, noise or multipath fading. Message loss is due to loss of signal caused by physical obstacles, collisions, bandwidth limitations, etc. Link failures, especially in ad-hoc networks, are caused by mobile nodes losing radio contact with the destination node. *Node failures* include crash failures and interface failures. Crash failures are caused when nodes abruptly disappear from the system due to energy depletion, pulled out without warning, and overloaded processors (nodes simply stop communicating). Interface failures mean receiver and transmitter failure. Therefore, service discovery systems should be fault-tolerant. Some examples of fault-tolerant mechanisms in service discovery systems include redundant and replicated Registries, caching of alternate services, primary-based recovery protocols such as Registry monitoring and Registry backup [Tan02b], retransmissions and acknowledgments for reliable transmission, and containment of unreliable behaviors by blacklisting suspicious nodes.

3. ***Resource constraints*** - Nodes with hardware constraints are known as *resource-lean nodes* (low memory, processing power and energy). Systems with resource-lean nodes require resource-aware service discovery functions. One solution is to delegate more tasks to more powerful nodes. In systems with *low bandwidth availability*, cross-layer dependencies such as service discovery with routing knowledge, and efficient query processing among Registries (e.g. through DHTs) can help conserve bandwidth. Furthermore, load balancing techniques help scale Registry-based architectures so that Registries do not overload.

4. ***Security*** - A secure service discovery system must support *confidentiality, message integrity and availability* [Avi04]. Methods to address these concerns include authentication of communicating entities, access control

so only a select few are able to communicate, protection of sensitive service attributes (e.g. location) by hiding the value, data integrity, so that communicating entities can detect when data is tampered during transit, and detection and blacklisting of malicious nodes (including authorized entities). The challenge for a secure service discovery system is to maintain self-configuration of the system, because the owner of the devices will most probably be required to provide authentication and access control. Security also consumes resource due to encryption algorithms. Most service discovery systems assume participating nodes are secure by delegating security to the application layer. However, full fledged deployment of a service discovery system will eventually require some secure measures integrated into the service discovery functions.

5. ***System heterogeneity*** - Nodes in heterogeneous systems contain different types of network connectivity (e.g. Ethernet, 802.11 a/b/g, IRDA), and a variety of network stacks (e.g. transport, routing, addressing). A service discovery protocol that abstracts away as much as possible the lower-layer protocol stacks, and can perform its functions with minimum dependencies allows easier deployment in a heterogeneous environment.

Figure 2.4 summarizes the five operational aspects and the respective solutions. By taking the operational aspects into consideration, it is possible to design a system that addresses more than one type of operational issue. For example, an architecture with replicated and redundant Registries supports a large and lossy system. State of the art systems usually base their design rationale on their own set of priorities for the design aspects, hence causing tradeoffs.

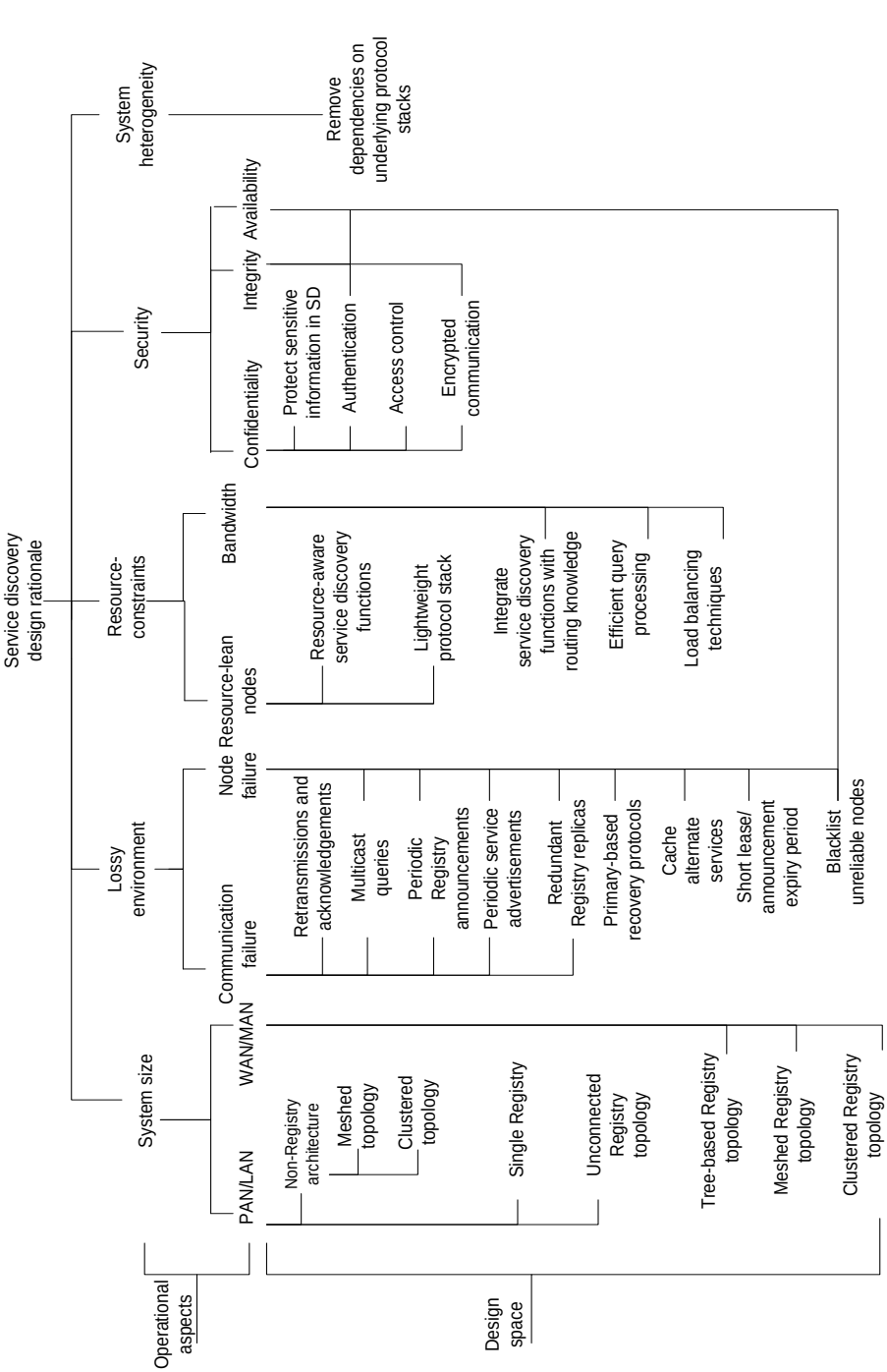


Figure 2.4: *Summary of operational design aspects and solutions, tailored for service discovery. The design rationale for a service discovery system depends on its own relevant set of operational aspects.*

2.7 State of the Art

Having described the nature of service discovery in the context of distributed systems from the point of view of (a) the architecture, (b) the functionality, (c) the models underlying the implementations, and (d) the operational aspects, we now investigate how existing service discovery systems fit into this mould.

We provide summaries of selected state of the art service discovery systems. We choose to describe these systems because of their popularity in the type of network and the size of system that they support. For ease of understanding, we will maintain the terms Manager, User and Registry to represent the protocol-dependent entities, even though the original papers use slightly different terms.

We divide the systems into two categories, based on the targeted system size: (1) *small systems*, which includes LAN and PAN, with limited number of nodes, and (2) *large systems*, which includes WAN and MAN, with a large number of nodes. The system size has the most influence on the design decisions in existing state of the art, where they implement similar service discovery functions and methods, and Registry topology (e.g. in large systems, the Registry-based architecture is chosen, where Registries are replicated, and arranged in either tree or mesh topology).

Unless mentioned specifically to support ad-hoc networks, the systems work in infrastructure-based wireless networks, and by default, also work on wired networks.

2.7.1 Small systems

Small systems are not usually concerned with scalability issues. The architecture type can be Registry, non-Registry or a combination of Registry with multicast query capability for Users (for resilience against single point of failure problems). Small systems also provide more support for consistency maintenance than large systems, because bandwidth utilization is a less critical issue.

1. Jini [Mic03b] - Jini was developed by Sun Microsystems, and is implemented using the Java programming language. Jini is a Registry architecture, where the Registry is called the *Lookup Service*. The Manager registers its service at the Registry, by uploading the service proxy code. The data stored is typically a part of a structured distributed shared memory, using tuple spaces, implemented through JavaSpaces [Mic03a]. The User queries the Registry for services matching its requirements, and receives the proxy code of the service. The User also requests the Registry to notify it if similar services register in the future. The Manager maintains a lease with the Registry, where it periodically refreshes the lease to indicate its continuous existence. The Registry automatically purges the information on the Manager that failed to refresh its lease. If the Manager updates its service description, it publishes an event to the Registry. The Registry

propagates the event to interested Users. The use of Java allows code mobility and operating system flexibility for Jini devices. However, Jini uses the Java Virtual Machine (JVM) and Java Remote Method Invocation (RMI), and depends on TCP/IP for reliable communication. These technologies cause dependencies on the underlying protocol stacks.

2. UPnP [Mic00] - Universal Plug and Play (UPnP) was developed by Microsoft, and is based on the Simple Service Discovery Protocol [Gol00]. UPnP is a non-Registry based architecture. The User is called the *Control Device*, and the Manager is simply called the *Device*. Service Description is described in XML. The Manager sends multiple multicast messages periodically to announce its presence and its services. The User also sends multicast queries to request services matching its requirements. The Manager sends XML documents to the User. The XML document provides the device and service descriptions along with URLs to view the user interface of the Manager. The User controls the remote service through the Simple Object Protocol (SOAP) [Gud03] and XML parsing of action requests. The Manager updates its service through General Event Notification Architecture Base (GENA) [Coh94]. GENA publishes notifications to subscribers. The Manager periodically sends multicast announcements, which is used by the interested User to monitor the continued existence of the Manager. The non-Registry based architecture eliminates single point of failure issues, and supports mobility. However, it increases network traffic due to extensive use of multicast messaging. Like Jini, dependencies on IP technologies causes network dependencies.

3. SLP [Gut03] - The Service Location Protocol (SLP) was developed by the IETF SvrLoc working group. It is a combination of Registry and non-Registry architecture. The User is called the *User Agent*, the Manager is the *Service Agent* and the Registry is the *Directory Agent*. When a Registry exists, the Manager registers its Service Description at the Registry and the User queries for services matching its requirement. SLP provides filters that allows attribute and predicate string search. When the Manager updates its Service Description, it re-registers at the Registry. The User has to query the Registry periodically if it wants to discover the update. When the Registry is unavailable, the User can send multicast queries to discover the Manager. The Manager also periodically refreshes its registration by re-registering its data. If the Manager fails to refresh its registration on time, the Registry removes the data, and assumes that the Manager is no longer available in the system. A typical implementation of SLP depends on reliable TCP/IP. SLP provides basic service discovery functions, with limited consistency maintenance support. Unlike Jini, the combination of of Registry and non-Registry based architecture reduces single point of failure issues.

4. Bluetooth SDP [Bra01] - Bluetooth was developed by the Bluetooth Special Interest Group, an industry consortium consisting of companies like Ericsson, Nokia and IBM. It is meant for low power, short range (within 10m), wireless (ad-hoc) radio system devices (PAN network) operating in the 2.4GHz ISM band. The Bluetooth Service Discovery Protocol (SDP) depends on the underlying connectivity, thus we first describe how devices establish their connectivity. Bluetooth devices periodically sniff for nearby Bluetooth devices and form a personal area network called piconets which has a maximum of 8 members. The member that initiates communication acts as the master of the piconet. Additional devices are supported by reusing addresses of a silenced existing member on a new member. Groups of piconets communicating with each other are called scatternets. The Bluetooth SDP [Blu01] is a non-Registry based architecture. The Manager runs an *SDP server*, while the User runs an *SDP client*. Services are divided into classes. Each service is represented by a *service record*. The User sends a query for a particular service type, and receives a response from the Manager if it offers a matching service. The User detects that the Manager is no longer available when it does not receive a response to a request. The Bluetooth SDP has simple basic functions, where there is no leasing, or subscription to receive updates. The tight dependency on the underlying protocol layers makes the Bluetooth SDP unsuitable for stand-alone deployment.

2.7.2 Large systems

Service discovery for large systems normally have a multiple Registry architecture because it reduces network traffic, thus increasing scalability. Since large systems are deployed over long distances, multiple, replicated Registries are available. To support a large number of nodes, Registries can do load balancing, and reproduce child Registries to help reduce load. To allow Registries to query and update each other efficiently, Registries are arranged in a logical mesh or tree topology.

1. GSD [Cha02] -GSD is developed in University of Maryland. It is targeted at ad-hoc networks, where service discovery is integrated with routing. GSD classifies services according to several groups and hierarchies, expressed in DAML. The protocol requires all nodes to behave as Registries. The Manager multicasts group information along with the information of groups and services within its own vicinity, within a limited number of hops. The neighbor nodes cache the group and service information. A service query from the User is matched by the neighbors against cached information, and replied if there is a cache hit. In case of a cache miss, the query is forwarded based on the probability of finding the requested group in their vicinity. The neighbor node caches the User's address, the query identifier and the previous hop address in a reverse-route table. The reply to the query is forwarded back to the User by the neighbor nodes based on their reverse-route tables. This protocol reduces network traffic by utilizing selective forwarding and limiting the number of hops for messages to be forwarded.

However, by limiting the number of hops that service advertisements can travel, fewer nodes are aware of the service existence. GSD provides limited consistency maintenance by restricting the lifetime of advertisement messages.

2. Ninja SDS [Cze99] - The Ninja project by the University of California, Berkeley developed the Service Discovery Service (SDS). SDS is a Registry architecture, where services are registered by Managers and discovered by Users through queries. The Registry in SDS is called the *SDS server*. For the purpose of scalability, Registries are organized into multiple shared tree-like hierarchies, so that tasks can be shared among several Registries. When a Registry is overloaded, it spawns a nearby node as a new Registry, which then becomes a child of the overloaded Registry. The new Registry is allocated a portion of the network extent, and thus, a portion of the load. Security is also supported by SDS, where service discovery functions are wrapped around steps to allow authentication, authorization and data integrity. For service queries and Service Descriptions, SDS uses an XML-based query and description language. SDS is implemented in Java and requires the use of Secure Remote Method Invocation (Secure RMI) to perform secure communication, hence it requires substantial resources.

3. INS/Twine [I.S01] - INS/Twine was developed at Massachusetts Institute of Technology. Like SDS, The architecture is based on the Intentional Naming System [AW99], where a number of Registries, called *resolvers*, map queries to destination addresses, and also distribute service information. Unlike tree-like Registries structures in SDS, INS/Twine Registries have a mesh-like topology, where Registries are peers with each other. A Manager is simply referred to as a *resource*. The Manager advertises its Service Description to the nearest Registry. INS/Twine is built on top of a distributed hash table (DHT), such as Chord [Sto03]. The Registry extracts prefix subsequences of attributes and values in the Service Description, into *strands*. The Registry then computes hash values for each of these strands, which constitutes numeric keys used to map resources to resolvers. To avoid being overwhelmed with registrations, Registries in INS/Twine use keying mechanisms to limit registrations. The service information is stored redundantly in all Registries that correspond to the numeric keys. When a User queries the nearest Registry, the Registry splits the query similar to how the Service Description is split. The Registry then queries other Registries that are identified by one of the longest strands. The query is further processed by the Registry, which returns the matching service information.

4. Jxta [Gon01] - Jxta was developed by Sun Microsystems. It is a combination of Registry and non-Registry based architecture. Registries in Jxta are known as Rendezvous peers. Managers (simply known as *peers*) send multicast advertisements to make their presence and services known to the network. Registries that receive the advertisements cache the service information. A User can send

multicast queries, and Managers and Registries with matching service information respond to the queries. Each Manager periodically refreshes its service advertisements. Users and Registries purge service information when the Manager fails to refresh the advertisements at the expected time. When a service changes, the Manager sends another advertisement (either immediately, or at the next periodic refresh time) so that Users and Registries can detect the change. An additional entity called the *relay peer* acts as name resolver to map a service to its destination address. The relay peer stores routing information and relays messages across firewalls. In Jxta, Users, Managers and Registries can form groups, based on a certain criteria (such as location, service type, etc). An entity can only communicate with members of the groups that it has joined. Unlike SDS and INS/Twine, Jxta does not provide load-balancing techniques to unburden overloaded Registries. It also provides limited consistency maintenance support.

2.8 Taxonomy of State of the Art

In this section, we first analyze how the state of the art service discovery systems address the five operational aspects; system size (which influences the choice of architecture), lossy environment, resource-constraints, heterogeneity and security. We then proceed to analyze the differences in functionality.

2.8.1 Taxonomy of State of the Art Solutions to Operational Aspects

Figure 2.5 shows a summary of our analysis on the solutions provided by state of the art systems for the operational design aspects. The shaded columns for each system expose the aspects that the system considers, and the solutions. We also show which system provides the most support for each design issue by the number of shades for the issue across the systems.

For small sized, mobile PAN (less than ten nodes) the non-Registry architecture (UPnP and Bluetooth SDP) is the most suitable. This is because the number of nodes is small, and service discovery tasks will be accomplished faster, than if a Registry is required to be setup, and maintained. For LAN (tens to several hundred nodes), the Registry-based architecture would be more suitable, to help conserve bandwidth. The Registry can either be statically deployed (Jini, SLP, GSD), or dynamically elected (Jxta), depending on the degree of fault-tolerance required. For large systems, scalability is the primary concern. Registries are scoped according to location or services (SDS, Jxta), and arranged in a tree (SDS), or mesh (GSD, Jxta, INS/Twine) topology to optimize query processing and conserve bandwidth.

State of the art systems provide fault-tolerance for a lossy environment by implementing on-the-fly Registry setup (SDS), or multiple replicas of the Registry (as can be done in Jini, SLP, GSD, INS/Twine) to provide redundancy in the face

of single point of failure problems caused by mobile Registries. However, redundancy increases design complexity and consumes additional resources. Registries can also be monitored by other nodes for node crash failures (SDS, Jxta). The non-Registry based architecture of UPnP is the most robust against crash failure and message loss, because Users and Managers can hear each others' multicast queries and announcements directly. However, the tradeoffs are scalability and conservation of resource consumption. A Registry-based architecture, with the ability for nodes to multicast queries when the Registry disappears (SLP, GSD, Jxta), increases robustness against message loss and crash failure, while also increasing scalability and resource consumption. To provide reliable transmission, TCP is used in Jini, SLP, and UPnP. None of the state of art protocols address message corruption (assume lower protocol layers will address this problem), nor do they detect and recover from Byzantine failures.

None of the presented systems makes a distinction among resource-lean and resource-rich nodes at the functional level by partitioning service discovery tasks. At the architectural level, it is implicitly understood that Registries have powerful hardware resources. Service discovery for large systems provide load balancing techniques so that Registries are not overloaded; SDS and INS/Twine allow overloaded Registries to spawn another to take over a portion of their tasks. For conserving bandwidth, some systems use efficient replication and query processing such as by using DHTs (INS/Twine).

To integrate nodes into different types of connectivity (e.g. wired Ethernet to wireless 802.11b), nodes are assumed to have the necessary interfaces to the different networks, or have connectivity via access points. SDS and Jxta abstract away underlying protocol stacks (transport layer and beyond), therefore providing more flexibility for deployment over different types of connectivity and protocol stacks. Service discovery functions in these systems are self-sufficient, with minimum network layer assumptions (multicast and unicast capabilities required). Systems built for the ad-hoc networks (Bluetooth SDP and GSD) integrate routing knowledge with service discovery, and become more dependent on the network stack. Some systems explicitly require a certain type of technology, such as TCP and IP (Jini, UPnP, SLP and INS/Twine). INS/Twine also uses the integrated INS as the underlying name mapping framework.

Most systems depend on higher layers in the protocol stack to provide authentication, authorization, privacy and data integrity. Additional steps are required in between service discovery functions. For example, once the Manager discovers a Registry, it decrypts the message and verifies the signature in the message to authenticate the Registry (as is possible in SLP, Jxta and SDS). These are intermediate steps, before service registration. Users who discover the service can only access the service if they are authorized by a capability manager (as in SDS), by applying for group membership (possible in Jxta), or if allowed by the Manager (can be performed by security applications in all systems). The service discovery systems discussed here do not support detection and blacklisting of an authenticated and authorized entity that has turned malicious. One other

protocol that does reputation-based service discovery to support this problem is SuperstringRep [Wis05]. Security measures consume substantial resources, increase complexity of the system extensively, and reduce self-configuration of the system. Due to these reasons, a full-fledged secure service discovery system is yet to be deployed successfully.

Operational Issues			Design Solutions		State of the art solutions to operational issues							
Issue	Specifics				Jini	SLP	UPnP	Bluetooth SDP	GSD	Jxta	SDS	INS/ Twine
System size	1	Small systems (PAN / LAN)	Non-Registry architecture	Meshed topology								
				Clustered topology								
			Registry architecture	Single Registry								
				Unconnected topology								
	2	Large systems (MAN / WAN)	Multiple Registry architecture	Meshed Registry topology								
				Tree-based Registry topology								
				Clustered Registry topology								
Lossy environment	3	Communication and node failure	Retransmissions and acknowledgements		TCP	TCP	TCP					
			Multicast queries									
			Periodic Registry announcements									
			Periodic service advertisement									
			Redundant Registry replicas									
			Primary-based recovery protocols									
			Cache alternate services		Appl	Appl	Appl	Appl	Appl	Appl	Appl	Appl
			Short lease/announcement expiry period		Appl	Appl	Appl	Appl	Appl	Appl	Appl	Appl
			Blacklist unreliable nodes									
Resource constraints	4	Resource-lean nodes	Resource-aware functions									
			Lightweight protocol stack									
	5	Bandwidth constraint	Integrate routing and service discovery									
			Efficient query processing									
			Load balancing techniques									
Security	6	Confidentiality, integrity and availability	Remove sensitive information from SD		Appl	Appl	Appl	Appl	Appl	Appl	Appl	Appl
			Authentication		Appl	Appl	Appl	LM	Appl	Appl		Appl
			Access control					LM				
			Encrypted communication		Appl	Appl	Appl	LM	Appl			Appl
			Blacklist malicious nodes									
System heterogeneity	7	Heterogeneous protocol stack	Abstract away underlying protocol stacks		TCP/IP	TCP/IP	TCP/IP	Bluetooth	Routing			INS/IP
Operational issues that are given priority by the system					1,3	1,3	1,3	1,6	2,3,4	1,2, 3,6,7	2,3, 6,7	2,3,5

Appl: The solution can be provided by the application layer
LM: The underlying Link Manager in the Bluetooth protocol stack provides authentication and encryption
RMI/JVM: Jini is supported by the security features in JVM and RMI/TCP: Reliable communication is provided using TCP
TCP/IP, Bluetooth, Routing, INS/IP: The system depends on the listed underlying protocol stack

Figure 2.5: *Taxonomy of state of the art solutions to operational aspects. Shaded service discovery systems support the proposed solutions. Appl means the solution to the operational issue is supported by the application layer. Some systems depend on solutions provided by the underlying protocol stacks, such as TCP, IP, Bluetooth and ad-hoc routing protocols.*

Service Discovery Functions		Methods	State of the art functional implementation							
			Jini	SLP	UPnP	Bluetooth SDP	GSD	Jxta	SDS	INS/ Twine
Configuration Discovery	Registry auto-configuration	Registry election			N/A	N/A				
		Registry reproduction			N/A	N/A				
	Registry or cluster discovery	Passive discovery				N/A				
		Active discovery	*		N/A	N/A				
Registration		Solicited registration			N/A	N/A				
		Unsolicited registration			N/A	N/A				
SD Discovery	Query	Unicast query				**				
		Multicast query								
	Service notification	Service notification by Registry			N/A	N/A				
		Multicast service advertisement								
Configuration Update	Configuration purge	Leasing expiry								
		Advertisement TTL expiry								
	Consistency maintenance	Poll for updates (by the User;)	Appl	Appl	Appl	Appl	Appl	Appl	Appl	Appl
		Notification of updates (by the Manager / Registry)								
		Update among Registries			N/A	N/A				

* Jini only does active discovery when the node initializes (powers on)
** Bluetooth SDP depends on the underlying Bluetooth network to detect neighboring nodes, so that it can query through unicast
N/A: the method is not relevant for the non-Registry architecture
Appl*:* the method can be supported by the application layer, by using the handles provided by the service discovery protocol

Figure 2.6: *Taxonomy of state of the art functional implementation. The more shades a function has, the higher the effectiveness of the function. However, the choice of method impacts the efficiency, responsiveness and resource consumption.*

2.8.2 Taxonomy of service discovery functions and methods

Once architectural decisions are made on how to address the operational design aspects, the service discovery functions are implemented. Figure 2.6 summarizes the functional capabilities of state of the art systems. In Chapter 5, we show that the functional differences impact system performance such as responsiveness, effectiveness, efficiency and resource consumption.

For the Configuration Discovery function, the Registry reproduction method in INS/Twine and SDS requires the first set of Registries in both systems to be manually deployed by a system administrator. None of the described systems use a more dynamic election method to establish the initial set of Registries. For discovering Registries and cluster members, systems that do periodic passive and active discovery (SLP, GSD, INS/Twine and SDS) have higher responsiveness than systems that implement only one of the methods. Passive and active discovery are especially useful to allow the system to recover from failures that cause network partitioning.

For the Service Registration function, no systems are built with solicited registration, which would have allowed the Registry to recover speedily the purged information of a Manager, after communication failures. The state of the art systems allow only unsolicited registration, after a Registry is discovered.

For the SD Discovery function, UPnP and Jxta allow both multicast queries and multicast service advertisements. The combination of these two methods gives the highest probability for successfully discovering a service, even after message loss and temporary node failures (e.g. mobile nodes getting temporarily disconnected). Among Registry systems, SLP allows multicast queries and advertisements when the Registry is not discovered. This method conserves bandwidth more efficiently. In Jini, the Registry can notify the Users of newly registered services matching the requirements of the Users. This method increases the chances of discovering new services in Registry-based systems.

For the Configuration Update function, Jini uses leasing for Configuration Purge, where the Registry can request Managers to lengthen or shorten their lease period, according to the Registry's processing capability. The rest of the systems require Users and Registries to monitor the advertisement TTL of Managers to detect defunct services. Leasing is more efficient in terms of bandwidth and resource utilization, compared to periodic multicast advertisements. For Consistency Maintenance, the state of the art systems provide handles to allow the application on the User to query the Registry or Manager periodically for updates. Only Jini and UPnP allow the Registry or the Manager to update the User directly on changes in the SD. In large systems (Jxta, INS/Twine and SDS), Registries achieve consistency by updating each other on changes in the cached SDs. Unlike small systems, updates are not propagated each time an SD changes, but in bulk (when a threshold is reached), thus providing weaker consistency maintenance (but necessary to conserve bandwidth).

2.9 Discussion and Conclusion

We have analyzed the field of service discovery by first characterizing service discovery as a third generation name discovery system that solves the limitations of legacy naming systems for pervasive computing. We have described the different architectures and the main functions of service discovery that allows services to be discovered, and changes in service availability and attributes to be detected. We have showed how a small number of models of distributed systems can be used as a basis for the implementation of the service discovery functions. We have classified the main operational design aspects for service discovery, and have compared the state of the art solutions to these aspects.

We now focus on providing a design which satisfies the 3Rs in Chapter 1; a *reliable*, unattended small scale system, which consists of *heterogeneous* nodes with different *resource-constraints*. Towards satisfying these requirements, we choose to build a Registry-based system. Registry-based systems are vulnerable to single point of failure issues. However, when resource-constraints are taken into consideration, it is necessary to implement a Registry-based architecture so that resource-lean nodes can participate in service discovery using the help of the Registry.

Our analysis reveals several areas for improvements for a new Registry-based service discovery system for the home:

1. Reliability. State of the art systems either depend on underlying reliable transmission protocols such as TCP for retransmissions and acknowledgments to support temporary communication and node failures, or have no provisions for such failures. Therefore, a new service discovery system should incorporate *retransmissions and acknowledgments* for critical messages, should a reliable transport layer not be available in all devices.

   Existing Registry systems are vulnerable to *single point of failure* issues due to Registry crashes. A new service discovery system should implement Registry election to allow *automatic Registry setup*, and incorporate *primary-based recovery protocols* [Tan02b] such as monitoring and backing up the Registry to increase fault-tolerance against Registry crashes. This solution also eliminates the need for Registry replicas in small systems, and is therefore appropriate for our home context.

2. Resource constraints. Cost of devices is a major concern for the home environment. Since cost increases with resource consumption, we require a *resource-aware service discovery system*. State of the art systems typically assume nodes in the system have homogeneous resources. The systems are also not suitable for resource-lean nodes, because of heavyweight protocol stacks and implementation (e.g. TCP, IP, JVM, XML). Therefore, we should *partition service discovery tasks* across nodes based on their resource

constraints. Resource-lean Users and Managers need only implement lightweight methods for each service discovery function, while more powerful Users and Managers can support the weaker nodes by implementing more sophisticated methods. Registries are only implemented in the powerful nodes.

3. Heterogeneity. Most service discovery systems depend on a specific set of underlying protocol stacks such as TCP, IP or certain routing protocols. We should ensure a more flexible implementation across heterogenous nodes by *abstracting away the underlying protocol stacks*. Instead, our system should only need unicast and multicast capabilities from the network, with a set of specifications for the two communication models. For example, unicast is defined as "point-to-point communication for nodes at the opposite edges of the system", while multicast is defined as "one-to-many communication for nodes within the system".

We conclude by stating that the behavior of a service discovery system is heavily influenced by the operational aspects that the system gives priority to. In the home environment, the size of the system is small. The system should also be robust against failures, and support resource-constrained and heterogeneous devices. Therefore, our system described in Chapter 4 implements a resource-aware Registry-based architecture, uses Registry election and primary-based recovery protocols to eliminate single point of failure issues, and is portable over heterogeneous devices and networks.

In the next chapter, we state that it is not sufficient for a service discovery system for the home to simply provide design solutions to the operational aspects. The service discovery system should also provide some *guarantees of correct behavior*, based on a set of principles.

Chapter 3

Functional Principles of Service Discovery for Small Systems

An expert is a person who has made all the mistakes
that can be made in a very narrow field.
General Omar Bradley

3.1 Introduction

As explained in Chapter 2, service discovery systems that aim to be self-configuring should achieve the following objectives:

O1: *Discover services that match requirements*

O2: *Detect changes in service availability and attributes*

The dynamic nature of the environment of a service discovery system requires the objectives to hold true, even when nodes appear and disappear at random. In some cases we can rely on human intervention to maintain the services, but in the home environment, this is impossible. Therefore, service discovery systems for the home environment are expected to recover from unexpected failures, and continue to function correctly.

This chapter states the functional principles of service discovery for small scale, unattended systems. We also identify the essential recovery behaviors in service discovery that help satisfy the principles when the system faces communication

and node failures. The recovery rules are essential for improving *reliability* in service discovery (one of the 3Rs from Chapter 1). A reliable service discovery system is *effective*, because it increases the chances of satisfying O1 and O2. Effectiveness is a required criteria in the Research Question in Chapter 1.

Contributions: The contributions of this chapter to the field of service discovery are:

1. Pioneering work for specifying the functional principles of service discovery for small systems. We present seven *Service Discovery Principles* that state the fundamental behavior of service discovery for small, unattended systems.

2. Providing a detailed analysis of consistency maintenance in service discovery, and a novel classification of consistency maintenance recovery rules according to common failure scenarios.

The chapter is organized into two parts; In Section 3.2, we model the service discovery environment for a small-scale system, and introduce seven *Service Discovery Principles*[1] .

In Section 3.3, we introduce *recovery rules*[2] that specify how service discovery systems can recover from failures, and even after failure, eventually satisfy the Service Discovery Principles. Here, we focus on how to satisfy the principles for consistency maintenance. Consistency maintenance allows Users to receive updates on service attributes. Our analysis reveals that *for Users to regain consistency in the face of failures, the correct behaviors of all service discovery functions are required.* In the home environment, it is the task of the service discovery system to maintain the integrity of cached data, so that applications that rely on the information in the service description (SD) do not behave incorrectly due to inconsistency. Furthermore, guarantees on consistency maintenance in service discovery safeguards the system from services that do not provide their own consistency maintenance at the application layer when the service attributes change.

We use the term "failures" for all transient failures that temporarily disable nodes from communicating with each other, such as message loss, link failure and temporary node crashes due to power or hardware failure. We do not address persistent failures such as unreliable nodes that require physical removal, or severe network failures that require network reset, because human intervention is necessary in such cases.

[1]This section has been published in Proceedings of the 30th Annual IEEE Conference on Local Computer Networks (LCN 2005), Sydney, Australia, pp. 209-217, IEEE Computer Society Press.

[2]This section has been published in Proceedings of the 20th IEEE Int. Parallel & Distributed Processing Symposium (IPDPS 2006), Rhodos Island, Greece, page to appear, IEEE Computer Society Press.

3.2 Functional Principles of Service Discovery

A system is flawed if a User is never able to discover an available Manager whose services match the User's requirements. To make the notion of correct behavior precise, we define *Service Discovery Principles*. These fundamental principles also define the nature and constraints of service discovery. Systems that adhere to the principles provide guarantees on their behaviors.

3.2.1 Related Work

The only other work done to develop principles for service discovery is the work on *Service Guarantees* from Dabrowski, Mills and Quirolgico [Dab05] from the US National Institute of Standards and Technology (NIST), which proposes general guarantees that service discovery protocols should satisfy. The work on the Service Discovery Principles was done in parallel with the development of the Service Guarantees at NIST, and is the result of joint research. The Service Guarantees do not specify the system size nor the environment that they apply to. The Service Discovery Principles refine and simplify the Service Guarantees for small systems, focused on the home environment. To our knowledge, none of the existing state of the art systems offer guarantees on functional behaviors.

3.2.2 The Service Discovery Environment

If the environment of a service discovery system causes it to fail, we should like the system to recover from failures.

Below we provide a formal description of a system.

1. **System**: A system consists of a set of entities with attributes $(\mathsf{e} \in)\mathsf{E}$, a set of services $(\mathsf{s} \in)\mathsf{S}$. The attributes of the entities, described below, evolve over time $(\mathsf{t} \in)\mathsf{Time}$. Based on the attributes, the entities are divided into three (not necessarily pairwise disjoint) sets $(\mathsf{u} \in)\mathsf{U}$ for Users, $(\mathsf{m} \in)\mathsf{M}$ for Managers, and $(\mathsf{c} \in)\mathsf{C}$ for Registries.

2. **Entity attributes**: Each entity, e has the following attributes, which are subject to change over time:

   - $\mathsf{C}(\mathsf{e}) \subseteq \mathsf{E}$ is the set of Registries discovered by e. An entity is called a Registry, i.e. $\mathsf{e} \in \mathsf{C}$, if it has discovered itself in the role of Registry, $\mathsf{e} \in \mathsf{C}(\mathsf{e})$.
   - $\mathsf{OfferedSD}(\mathsf{e}) \subseteq \mathsf{S}$ is the set of services, offered by e. An entity is called a Manager, i.e. $\mathsf{e} \in \mathsf{M}$, if it offers at least one service, $\mathsf{OfferedSD}(\mathsf{e}) \neq \emptyset$.
   - $\mathsf{Requirement}(\mathsf{e}) \subseteq \mathsf{S}$ is the set of services required by entity e. An entity is called a User, i.e. $\mathsf{e} \in \mathsf{U}$, if it requires at least one service, $\mathsf{Requirement}(\mathsf{e}) \neq \emptyset$.

- $\mathsf{DiscoveredSD}(\mathsf{e}, \mathsf{e}') \subseteq \mathsf{S}$ is the set of services discovered by e at entity e'. This attribute is typically only used when e is a User and e' is a Manager. We put $\mathsf{DiscoveredSD}(\mathsf{u}) := \bigcup_{\mathsf{m} \in \mathsf{M}} \mathsf{DiscoveredSD}(\mathsf{u}, \mathsf{m})$ for the set of all services discovered by a User, u at any Manager, m.
- $\mathsf{RegisteredSD}(\mathsf{e}) \subseteq \mathsf{S}$ is the set of registered services in e. This attribute is only used for a Registry.

There are several protocol-dependent parameters used in the Service Discovery Principles:

1. **Connectivity condition**: $\mathsf{Conn}(\mathsf{e}, \mathsf{e}')$: The service discovery protocol is responsible for providing the definition of Connectivity. An example is "if a message is transmitted from either e to e', or vice versa, the message is received." The Connectivity condition is not restricted to valid communication paths. It can also be defined by an application, for example, a security application that detects a malicious entity, and indicates to the service discovery protocol that the node is not available for any further operations.

2. **Disconnect condition**: $\mathsf{DisConn}(\mathsf{e}, \mathsf{e}')$: $\neg\mathsf{Conn}(\mathsf{e}, \mathsf{e}')$ for "sufficiently long", where "sufficiently long" is a protocol-dependent parameter. The definition of "sufficiently long" includes the Connectivity definition and the period communication did not occur. Examples are the limit of retransmissions or the time period for waiting for acknowledgements.

3. **Global Connectivity**, GC: The condition is satisfied if all entities are connected.
   $\mathsf{GC} : \forall \mathsf{e}_1, \mathsf{e}_2 \in \mathsf{E} = \mathsf{Conn}(\mathsf{e}_1, \mathsf{e}_2)$

4. **Number of Registries**, N: the desired number of Registries in the system.

5. **Required Registries or cluster members**, $\mathsf{G}(\mathsf{e})(\subseteq \mathsf{E})$: the Registries or cluster members required by e. For example, an entity may not be interested in knowing all Registries, but only those of a certain type, i.e. those which satisfy its "Registry requirements".

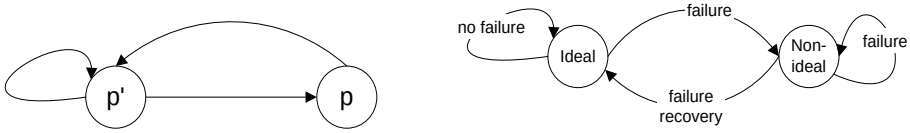
6. **Registry election**, Rank: a function for electing the Registries in the system, where the set C has highestRank if
   $\neg(\exists \mathsf{e}' \notin \mathsf{C} : \mathsf{Rank}(\mathsf{e}') > \min\{\mathsf{Rank}(\mathsf{e}) \mid \mathsf{e} \in \mathsf{C}\})$

3.2.3 Service Discovery Principles

We use *Linear Temporal Logic (LTL)* [Hut00] to define the Service Discovery Principles. This is essentially positional logic with temporal operators such as $\Box$ (*always*) or $\Diamond$ (*eventually*). As shown in Figure 3.1(a), the recovery of a service discovery system is defined as the *response* pattern, $\Box(p' \rightarrow \Diamond p)$, where p' is the

state of Connectivity, Global Connectivity or Disconnect, and p is the response of the service discovery system toward satisfying a Service Discovery Principle.

The Service Discovery Principles define the precise goals of the service discovery functions described in Chapter 2. Each function must perform correctly, and recover from failures, as shown in Figure 3.1(b). There are two states in a service discovery function. When there are no failures, the function performs correctly and is in the *ideal* state. When failures occur, the function may behave incorrectly, but has to recover from failures and leave the *non-ideal* state when failure ends.



(a) Response pattern for service discovery, $\Box(p' \rightarrow \Diamond p)$

(b) States of a service discovery function,

Figure 3.1: *Service discovery system states. (a)* p' *is the state of Connectivity, Global Connectivity or Disconnect, and* p *is the response of the service discovery system to satisfy a Service Discovery Principle. (b) The ideal state for a function has no failure, and the function performs correctly. In the non-ideal state, the function may perform incorrectly. But when the failure ends, the service discovery function should eventually recover from failures and return to the ideal state.*

We need some auxiliary definitions before giving the Service Discovery Principles.

1. The operations "$\Diamond\subseteq$", "$\Diamond\supseteq$" and "$\Diamond\supset\subset$"
   $\mathsf{a} \Diamond\subseteq \mathsf{b}$ holds at time t_0 when a at time t_0 is a subset of b at some future time t_1 where $\mathsf{t}_1 \geq \mathsf{t}_0$ (in LTL, the future includes the present). Similarly, for the symbols "$\Diamond\supseteq$" and "$\Diamond\supset\subset$" where $\mathsf{a} \supset\subset \mathsf{b}$ denotes that the two sets are disjoint ($\mathsf{a} \cap \mathsf{b} = \emptyset$).

2. $\mathsf{ServiceSearch}(\mathsf{u}, \mathsf{c})$ states that a User, u is looking for a specific service from a Registry, c

3. $\mathsf{MatchingSD}(\mathsf{c}, \mathsf{u}) :=$
   $\mathsf{RegisteredSD}(\mathsf{c}) \cap \mathsf{Requirement}(\mathsf{u})$
   is the set of services registered at the Registry, c which match the requirements of User, u.

4. $\mathsf{UpdateSD}(\mathsf{m}) :=$
   states that the $\mathsf{OfferedSD}(\mathsf{m})$ of Manager, m has changed.

5. $\mathsf{ServiceFound}(\mathsf{c}, \mathsf{u}) :=$
   $\mathsf{MatchingSD}(\mathsf{c}, \mathsf{u}) \Diamond\subseteq \mathsf{DiscoveredSD}(\mathsf{u})$
   states that the matching services at a Registry c are discovered by the User u.

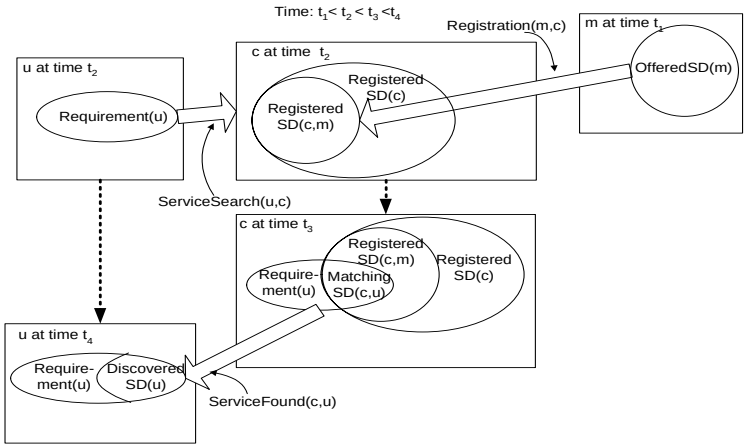


Figure 3.2: *System flow and relations between sets during Registration and Service Discovery.* Registration, ServiceSearch *and* ServiceFound *are shown as messages sent between entities. A service is registered by the Manager at time t_1, then discovered by the Registry at t_2, and User searches for the service at or before t_2. The Registry processes the request at t_3, where it finds matching services for the User. The User discovers the service at t_4.*

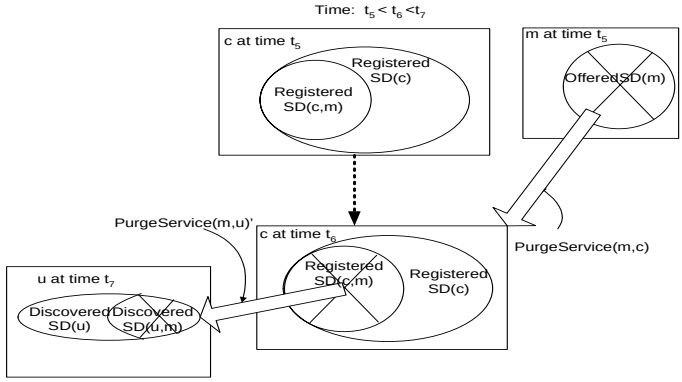


Figure 3.3: *System flow and relations between sets during Configuration Purge. A service is purged by the Manager at t_5, then the Registry is notified at t_6, which then purges the registration. The Registry notifies the User at t_7, and the User purges the service from its* DiscoveredSD *cache.*

6. $\mathsf{Registration}(\mathsf{m},\mathsf{c}) :=$
   $\mathsf{OfferedSD}(\mathsf{m}) \Diamond \subseteq \mathsf{RegisteredSD}(\mathsf{c})$
   states that all services of Manager, m are registered at Registry, c.

7. $\mathsf{PurgeService}(\mathsf{m},\mathsf{u}) :=$
   $\mathsf{OfferedSD}(\mathsf{m}) \Diamond \supset\subset \mathsf{DiscoveredSD}(\mathsf{u})$
   states that the services of Manager, m are eventually purged from the discovered services of User, u.
   $\mathsf{PurgeService}(\mathsf{m},\mathsf{c}) :=$
   $\mathsf{OfferedSD}(\mathsf{m}) \Diamond \supset\subset \mathsf{RegisteredSD}(\mathsf{c})$
   similarly states that the services of Manager m are eventually purged from the registered services of Registry, c. If an entity is both a Registry and a User then both properties must hold.

8. $\mathsf{Uptodate}(\mathsf{u},\mathsf{m}) :=$
   $\mathsf{OfferedSD}(\mathsf{m}) \cap \mathsf{Requirement}(\mathsf{u}) \Diamond \subseteq \mathsf{DiscoveredSD}(\mathsf{u},\mathsf{m}) \ \wedge$
   $\mathsf{OfferedSD}(\mathsf{m}) \Diamond \supseteq \mathsf{DiscoveredSD}(\mathsf{u},\mathsf{m})$
   states that User u will find new services at Manager m and remove services that are no longer available at m.

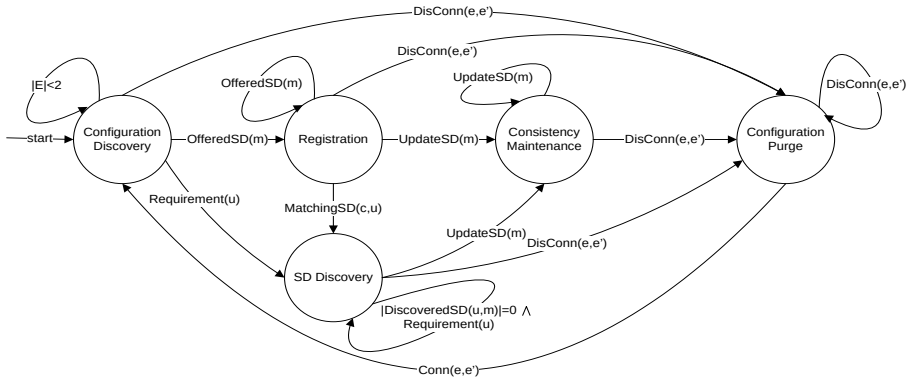


Figure 3.4: *Service discovery system life cycle. When a system consists of more than one entity, Configuration Discovery is performed, followed by Registration or SD Discovery. Registration is triggered when there exists a Manager with* $\mathsf{OfferedSD}(\mathsf{m})$. *SD Discovery is triggered when there exits a User with* $\mathsf{Requirement}(\mathsf{u})$, *or if the Registry has* $\mathsf{MatchingSD}(\mathsf{c},\mathsf{u})$. *SD Discovery is done every time there is a new requirement or as long as the User has not discovered the required service, where* $|\mathsf{DiscoveredSD}(\mathsf{u},\mathsf{m})| = 0$. *When* $\mathsf{UpdateSD}(\mathsf{m})$ *occurs in the Manager (SD changes), Consistency Maintenance is performed. Configuration Purge occurs every time entities face* $\mathsf{DisConn}(\mathsf{e},\mathsf{e}')$ *due to failures. When failures end, and* $\mathsf{Conn}(\mathsf{e},\mathsf{e}')$ *is restored, the cycle is restarted.*

Figures 3.2 and 3.3 provide scenarios for a Registry-based system, where the relationship between entities, and the sets are explained, as time progresses. In

Figure 3.2, ServiceSearch is the message providing the requirements of the User to the Registry, while in Figure 3.3, PurgeService and PurgeService′ are the messages that notify the Registry and User respectively of a defunct service.

We summarize the general life cycle of a service discovery system in Figure 3.4. We show that transitions between functions are triggered by the appearance of services, $\mathsf{OfferedSD}(\mathsf{m})$, requirements, $\mathsf{Requirement}(\mathsf{u})$, service changes, $\mathsf{UpdateSD}(\mathsf{m})$ and conditions $\mathsf{DisConn}(\mathsf{e},\mathsf{e}')$ and $\mathsf{Conn}(\mathsf{e},\mathsf{e}')$.

With the system and basic properties in place we are ready to give the 7 *Service Discovery Principles*.

(P1) **Registry Setup Principle**
When there is global connectivity, N Registries of the highest rank are selected in the system.

$\Box(\mathsf{GC} \rightarrow \Diamond((|\mathsf{C}| = \mathsf{N}) \wedge \mathsf{highestRank}))$
Assume all entities have the same rank if no ranking function is used.

(P2) **Configuration Discovery Principle**
An entity discovers all available Registries or cluster members in the system that it is interested in.

$\forall \mathsf{e} : \Box(\mathsf{C}(\mathsf{e}) \subseteq \mathsf{G}(\mathsf{e}) \wedge$
$\quad \forall \mathsf{c} \in \mathsf{C} \cap \mathsf{G}(\mathsf{e}) : \mathsf{Conn}(\mathsf{c},\mathsf{e}) \rightarrow \Diamond \mathsf{c} \in \mathsf{C}(\mathsf{e}))$

(P3) **Registration Principle**
A Manager registers its service description at each Registry it discovers.

$\forall \mathsf{m}, \forall \mathsf{c} \in \mathsf{C}(\mathsf{m}) :$
$\quad \Box(\mathsf{Conn}(\mathsf{m},\mathsf{c}) \rightarrow \mathsf{Registration}(\mathsf{m},\mathsf{c}))$

(P4) **SD Discovery Principle**
A User discovers the service descriptions that match its requirements directly from the Manager, or from a Registry.

$\forall \mathsf{u}, \mathsf{e} : \Box(\mathsf{Conn}(\mathsf{e},\mathsf{u}) \rightarrow \mathsf{ServiceFound}(\mathsf{e},\mathsf{u}))$

(P5) **Configuration Purge Principle**
A User or Registry purges the services of a Manager that has become disconnected.

$\forall \mathsf{m}, \forall \mathsf{e} \in \mathsf{U} \cup \mathsf{C} :$
$\quad \Box(\mathsf{DisConn}(\mathsf{m},\mathsf{e}) \rightarrow \mathsf{PurgeService}(\mathsf{m},\mathsf{e}))$

(P6) **2-Party Consistency Maintenance Principle**
A User remains consistent with a Manager when its services change. This principle applies to consistency maintenance between the User and the Manager, hence the term "2-party".

$\forall \mathsf{m}, \mathsf{u} : \Box(\mathsf{Conn}(\mathsf{u},\mathsf{m}) \rightarrow \mathsf{Uptodate}(\mathsf{u},\mathsf{m}))$

(P7) **3-Party Consistency Maintenance Principle**
A User remains consistent with a Manager when its services change, through the Registry. This principle applies to consistency maintenance between the User, the Registry, and the Manager, hence the term "3-party".

$\forall \mathsf{c} \in \mathsf{C}(\mathsf{m}) : \Box((\mathsf{Conn}(\mathsf{c}, \mathsf{m}) \wedge \Diamond \mathsf{Conn}(\mathsf{u}, \mathsf{c})) \rightarrow \mathsf{Uptodate}(\mathsf{u}, \mathsf{m}))$

Although the Service Discovery Principles are tailored for small systems, it is possible to extend the principles to large systems:

- Configuration Discovery Principle: A Registry may need to discover other relevant Registries, based on the type of topology. In a meshed Registry topology, Registries need to discover each other for forwarding queries, while in clustered Registry topology, Registries need to discover the relevant cluster members. The Configuration Discovery Principle can be directly applied in such cases where e is the Registry that has a set of requirement for discovering peer Registries.
- 2-party Consistency Maintenance Principle: A Registry replica requires fresh service information from its parent or peer when there is indication that its cache is inconsistent (e.g. reaches maximum cache misses). In such cases, the Registry requiring the update follows the behavior of the User, u and the Registry providing the update behaves as the Manager, m in the 2-party Consistency Maintenance Principle.

How well a system delivers the Service Discovery Principles can be measured from the effectiveness of the Consistency Maintenance function, when the system faces communication and node failures. Users cache a discovered SD, and ultimately require consistency with the Manager when the SD changes, thus satisfying the Consistency Maintenance Principles (P6 and P7). When faced with failures, Users become inconsistent with the Manager, and need to implement some fallback mechanisms to regain consistency. We call the set of fallback mechanisms *recovery rules*. If failures occur for a long and uninterrupted period of time, nodes are assumed to have crashed, and entities will purge the cached data of the non-communicating entity (satisfying P5). The recovery rules facilitate rediscovery of the purged entity (satisfying P1, P2, P3, and P4), so that Users can regain consistency. In the next part of this chapter, we identify and classify several recovery rules, based on the failure scenario.

3.3 Failure Recovery Rules During Consistency Maintenance

Users typically cache the discovered SD to reduce the access time to the service, and reduce bandwidth usage (by avoiding repeated queries to rediscover the

service). Caching requires consistency maintenance so that the User and the Registry keep a consistent view of the service. *Polling* for updates (pull model), and *notification* by the Manager when the service changes (push model) are two consistency maintenance methods in service discovery. However, communication and node failures may cause the consistency maintenance methods to fail to update the Users. We find that communication and node failures create a number of failure scenarios. How well a service discovery system performs depends on the type of recovery rule that the system adopts when dealing with each failure scenario. We classify consistency maintenance recovery rules according to the failure scenarios, and propose rules that improve performance, and satisfy the Service Discovery Principles.

3.3.1 Related Work

While consistency maintenance has been studied extensively in conventional, large-scale distributed databases and filesystem, there has been little work aimed specifically at evaluating consistency maintenance in service discovery protocols. Consistency maintenance in service discovery protocols conforms to the client-centric consistency model, which originates from the work on Bayou [Pet96; Ter94], a database for mobile systems with unreliable connectivity. The worldwide naming system, Domain Name Service (DNS) and the World Wide Web satisfy the *eventual* consistency guarantee in this model, which states that *in the absence of updates, all replicas converge toward identical copies of each other* [Tan02b].

Consistency maintenance in service discovery protocols is similar to maintaining cache coherence in distributed systems [Min90]. Cache coherence ensures the integrity of the data stored in the local cache of the User. The data stored may be part of a structured distributed shared memory, using tuple spaces such as implemented in Jini through JavaSpaces [Mic03a], or simply copies of service descriptions, as is stored in SLP and UPnP. Franklin et al. describe push-based and pull-based cache coherence strategies [Fra97]. Service discovery protocols use asynchronous update notifications to achieve cache coherence.

Existing work on consistency maintenance in service discovery is done by Dabrowski and Mills from the US National Institute of Standards and Technology (NIST). They evaluate the consistency maintenance mechanisms in UPnP and Jini. An architectural-based approach using an Architectural Description Language (ADL) is used to analyze service discovery systems [Dab01]. They benchmark the performance of service discovery systems according to their *Update Metrics*, in a dynamically changing environment, with increasing message loss [Dab02b] and interface failure [Dab02a] as the communication failure models. Dabrowski and Mills also propose a generic model encompassing the design of first-generation service discovery systems [Dab05]. They suggest service discovery protocols should provide guarantees of *correct* behavior against a set of properties which they call Service Guarantees. They report that first-generation

service discovery systems do not provide guarantees of correct behavior.

Frank and Karl [Fra04] study the impact of caching discovered services in mobile ad-hoc networks. They show that Users in mobile ad-hoc networks remain inconsistent with a service Manager that has become unavailable, when service lease periods are long. They propose that the service Manager explicitly announce when its service becomes unavailable. A cross-layer approach is taken, where Users learn about the continuous existence of Managers from the underlying routing protocol. Frank and Karl claim that this method allows Users to regain consistency faster than if they depend on the periodic announcements of the Manager, especially in a highly loaded network. They assume that the demand for a service Manager increases in a highly loaded network, hence increasing the possibility for Users to receive routing packets from the Manager. We find the dependance on the routing protocol is not a proper solution when the service Manager is not in high demand, therefore reducing the chances for interested Users to refresh their cache on the Manager. Furthermore, this solution is more useful for service removals than service updates. We suggest that the Manager and the User maintain a subscription lease, which is refreshed periodically through unicast, so that the Manager will know which Users it should update when the service changes. We also focus on small systems, where periodic announcements through multicast are acceptable.

3.3.2 Consistency Maintenance and Failure Recovery In Service Discovery

"Consistency" is the state where the User obtains the correct service information after the service changes. Users become consistent with the Manager when they successfully receive update information from the Manager. To explain update information, we use an example of a Manager which offers a printing service. The service description is a list of attribute-value pairs.

```
SD = {DeviceType=Printer,
        ServiceType=ColorPrinter,
        AttributeList{PaperSize=A4,
        Location=Study, PaperTray=High}}
```

If the attribute or value in the SD changes, for example, "PaperTray" changes from "High" to "Low", an update can be sent to the service subscribers to indicate that the printer is running out of paper. The printer may update the Users via the Registry, or directly.

As mentioned in Chapter 2, service discovery complies with *eventual consistency*, because it is client-centric, which tolerates transient inconsistencies. For discovered services to be useful, it is important that the consistency guarantees are specified clearly. The Consistency Maintenance Principles (P6 and P7) require the User and/or Registry to *always eventually* regain consistency with the Manager after the service changes. The User detects the change in the Manager, and regains consistency by obtaining the correct view of the service, either from the Manager directly (P6), or via the Registry (P7). The principles hold true only

when there is *connectivity* among the communicating entities (e.g., valid communication paths). The term *always eventually* states that a successful update invariably takes place at some time in the future, without giving a concrete time constraint.

3.3.3 Consistency maintenance mechanisms

Before we delve deeper into the issues facing consistency maintenance in an environment with communication and node failures, we first introduce the basic mechanisms that existing service discovery systems implement. The User has to *subscribe* either directly to the Manager (*2-party subscription*) or to a Registry (*3-party subscription*) to receive updates. A subscription between the User and the Manager or between the User and the Registry remains valid as long as the subscription *lease* does not expire. To maintain a valid subscription lease, Users are required to send messages periodically to the lessee to indicate their continued interest with the service.

There are two basic consistency maintenance methods:

(CM1) *Notification (push-based update)* - The User receives an update when the service description changes. In 3-party subscription, the Manager notifies the Registry which then propagates the update to subscribed Users. In 2-party subscription, the Manager notifies subscribed Users directly. Update notification is a built-in mechanism in a service discovery protocol. Examples of state of the art protocols that have this capability are UPnP and Jini.

(CM2) *Polling (pull-based update)* - The User regains consistency by polling the Manager or the Registry to retrieve the updated service description. In 3-party subscription, the Manager updates the Registry by re-registering its services. In both subscription schemes, periodic queries from the User eventually retrieve the updated service description. Typically, polling is implemented in the application layer, with hooks from the service discovery protocol.

Dabrowski and Mills [Dab02a] show that periodic polling is the more effective method if the application allows persistent polling (even when the lower protocol layers signal a connection failure), therefore increasing the chances for the User to retrieve the updated service description eventually. However, Dabrowski and Mills show that polling is a slower mechanism than update notification because of the dependency on the period of polling. We find that polling is also a less efficient mechanism than update notification in scenarios where services rarely change, causing multiple redundant polls. Thus, in this chapter, we focus only on recovery for consistency maintenance through notification.

During update notification, the Manager updates the Users by: (1) propagating an *invalidation* message that indicates that the service has been updated. The Manager notifies the interested User that a change has occurred, whenever

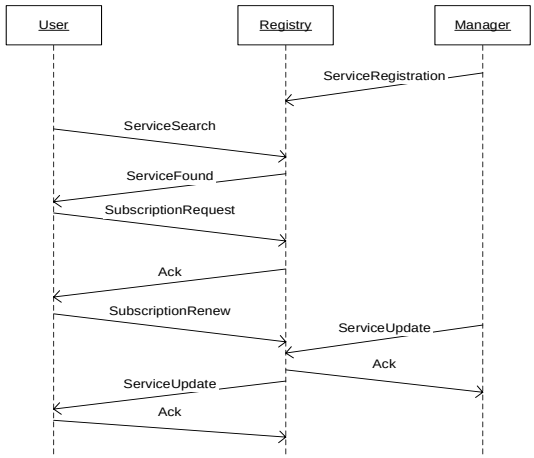


Figure 3.5: *Consistency maintenance through notification with 3-party subscription. The User discovers the Manager and subscribes to receive updates via the Registry. The User periodically renews the subscription lease by sending SubscriptionRenew messages. The Manager sends a ServiceUpdate message when the service changes.*

the service changes. Consecutive polling by the User retrieves the updated data. This method is efficient for a service that has frequent updates, but causes unwanted redundancy and delay for services that rarely change. (2) Propagating the *updated data*. This method is fast and efficient for a service that changes infrequently. An adaptive method that dynamically switches between sending an invalidation message or sending the update can be implemented, as done in the Alex protocol [Cat92], a filesystem that adapts the type of update propagation based on the age of the file (it assumes older files are less likely to be modified than younger files). However, to our knowledge, no existing service discovery protocols adopt the adaptive mechanism, due to the complexity in implementation.

Consistency maintenance in Registry architectures relies on the successful communication between the Manager and the Registry, *and* between the Registry and the User (*3-party subscription*). Peer-to-peer architectures rely only on the successful communication between the User and the Manager (*2-party subscription*). An example of consistency maintenance using 3-party subscription is shown in Figure 3.5.

3.3.4 Recovery rules for consistency maintenance

Due to temporary communication failures or node failures, notification of updates may fail. Nevertheless, when connectivity among entities is restored, the service discovery system is expected to recover and regain consistency, as stated by the Configuration Update Principles. Our in-depth analysis of the behavior of service

discovery during communication and node failures results in a novel method of identifying, classifying and proposing new recovery rules based on the type of update and failure scenario. Table 3.1 is a summary of our classification of recovery rules.

Table 3.1: *Classification of recovery rules for consistency maintenance. Subscription-recovery rules for each type of update take effect when subscription still remains valid. Purge-rediscovery rules occur when subscription expires, and may coincide based on the failure scenario.*

Subscription-recovery		Purge-rediscovery	
Update scenario	Recovery rule	Purge and rediscover scenario	Recovery rule
Critical update	SRC1	Manager rediscovers the Registry (and vice-versa)	PR1
	SRC2	User rediscovers the Registry	PR2
Non-critical update	SRN1	Registry rediscovers the User	PR3
	SRN2	Manager rediscovers the User	PR4
		User rediscovers the Manager	PR5

During communication and node failures, the subscription between the entities may remain valid, even though update notification fails. This is because the participating entities may face short-term failures, and restore connectivity before the subscription lease expires. Hence, it is up to the continuing subscription process to ensure Users regain consistency. We call this type of recovery *subscription-recovery*. When the subscription lease expires, consistency maintenance depends on the inherent capability of the service discovery protocol to detect, and rediscover purged nodes and services. Hence, this type of recovery is called *purge-rediscovery*.

1. Subscription-recovery. The success of consistency maintenance using this type of recovery depends on how persistently the subscription process tries to update the User. The degree of persistence in notifying updates depends on the type of update scenario: *critical update* and *non-critical update*. This is because not all applications require the same level of persistence in sending and receiving updates. By specifying the update scenario, we isolate necessary rules for successful consistency recovery.

Critical update. This update scenario applies to services that are critical, and need correct information urgently. An example is a fire alarm Manager that changes the value of an attribute, "status" from "ON" to "OFF", and is required to update the PDA of the homeowner. For critical updates, we propose two types of recovery rules.

(SRC1) Acknowledgments and retransmissions of notification - Update notifications

sent by the Manager or the Registry must be acknowledged to indicate success or failure. We propose no retransmission limit for the notification messages. Retransmission is only stopped when (a) the subscription expires, (b) acknowledgment for the notification is received, or (c) the application layer indicates loss of connectivity. Update retransmissions can be spaced in a periodic manner, until acknowledged by the Registry or the User.

(SRC2) Active User and Registry monitoring of updates - This rule takes effect if the User requires a history of missed updates from the Manager. The User and the Registry monitor either the sequence number on the update notifications, or the time period for the next notification (the latter applies only to Managers that provide fixed, periodic updates). When an expected update is missed, the User or the Registry requests the update. The Manager caches the history of service changes and only purges the cached updates after all interested Users successfully obtained the complete view of the service.

Non-critical update. Unlike the critical update scenario, the non-critical update scenario applies to services that are not sensitive to, or not overly affected by missed updates. An example is a printer Manager that updates a User when its paper tray empties. We propose the following recovery rules to improve consistency maintenance performance.

(SRN1) Acknowledgments and retransmissions of notification - Update notifications sent by the Manager or the Registry are acknowledged to indicate success or failure. Retransmissions of unsuccessful notification is done until either (a) retransmission limit is reached, (b) acknowledgment is received, (c) the subscription expires, (d) the application layer indicates lack of connectivity, or (e) the service changes again, requiring the Manager to reset the notification process.

(SRN2) Future retry of unsuccessful notification - This rule occurs after SRN1 fails to update the User. The Manager caches information on inconsistent Users and retries notification once a message from the inconsistent User is received (such as the subscription lease renewal message). The status of the inconsistent User is cached until (a) the subscription expires, (b) the service changes again, requiring the Manager to reset the notification process, or (c) the update is acknowledged.

The Consistency Maintenance Principles only require the User to regain consistency eventually, but not necessarily recover particular values of previously missed updates. Therefore recovering updates caused by multiple changes are not treated in the non-critical update scenario.

2. Purge-rediscovery. The success of consistency maintenance using this type of recovery depends on the proficiency of the service discovery protocol to detect, register and rediscover nodes and services after the subscription expires. We propose the following recovery rules for the User to regain consistency, based on the "purge" scenario. A combination of purge-rediscovery rules take effect if several failure scenarios occur simultaneously.

(PR1) The Manager purges the Registry, or vice-versa: the Manager and the Registry rediscover each other through (a) the Registry's periodic multicast announcement, or (b) the Manager's periodic multicast announcement (here, the Registry contacts the Manager). When the Manager re-registers, the Registry notifies interested Users of the new registration. The User regains consistency from the Registry notification. Users receive notifications of new service registrations by explicitly requesting for service notification, when they first establish contact with the Registry.

(PR2) The User purges the Registry: the User rediscovers the Registry through (a) the periodic Registry announcement, or (b) the User's periodic multicast announcement (here, the Registry contacts the User). The User then queries the Registry for the required service to regain consistency with the Manager (provided that the Manager registers the updated service description).

(PR3) The Registry purges the User: subsequent lease renewal from the User to the Registry results in a re-subscription process, where the User then receives the updated service description from the Registry.

(PR4) The Manager purges the User: subsequent messages received from the purged User allows a re-subscription process, where the User then receives the updated service description.

(PR5) The User purges the Manager: the User can purge the Manager when the service lease expires, or when the Registry notifies the User when it purges the Manager. The User purges the Manager only if the application layer is not communicating with the Manager. The User rediscovers the Manager through (a) multicast query with its requirements, where the matching Manager replies with the updated service description, or (b) periodic multicast service advertisements of the Manager, where the User then queries the Manager for the service description, or (c) unicast query to the Registry for the service

Further analysis in Chapter 5 shows that retransmissions and acknowledgments through SRC1 and SRN1 are useful for short term communication failures, as long as subscription remains valid. SRC2 and SRN2 are essential for satisfying the eventual consistency guarantee in the Consistency Maintenance Principles. During short-term node failures (where nodes recover from failures before the subscription expires), SRN2 is the most effective rule. When the subscription

expires, PR5 in 2-party subscription is found to be most effective, where Users can listen to multicast service advertisements by the Manager, and retrieve the updated service. PR1 increases the effectiveness of 3-party subscription, where the Registry notifies the Users when the Manager registers.

3.4 Discussion and Conclusion

Our analysis in Chapter 2 leads to a resource-aware Registry-based architecture, and uses Registry election and primary-based recovery protocols to eliminate single point of failure issues in a lossy environment. We also provide a more flexible implementation across heterogenous nodes by abstracting away the underlying protocol stacks. These design solutions satisfy the 3Rs from Chapter 1; reliability, resource-constraints and heterogeneity of devices and networks.

In this chapter, we focus on *reliability*, so that the system can be *effective* against communication and node failures. A reliable service discovery system satisfies the effectiveness criteria in the Research Question presented in Chapter 1. Towards this goal, we specify the fundamental behavior for service discovery in small, unattended systems through the Service Discovery Principles. We then propose that service discovery systems apply the recovery rules so that they can self-heal from failures, and thus satisfy the principles.

We identify the following as opportunities to produce a reliable service discovery system which gives some guarantees on functional correctness:

1. A system that provides guarantees on correct functional behavior - Certain existing state of the art systems like SLP do not satisfy the Consistency Maintenance Principles. Other service discovery systems like Jini and UPnP implicitly claim that they support the fundamental behaviors of service discovery. None, however, provide guarantees of correct behavior [Dab05]. We design a system for the home that adheres to the Service Discovery Principles so that applications can rely on the service discovery protocol to function correctly.

2. Formal verification of models of the system - To claim that our system gives guarantees of correct behavior, we formally verify our design against the Service Discovery Principles, and improve the design until the principles are satisfied. The design choices from Chapter 2 enable models of the system to be formally verified against the Service Discovery Principles: (a) there are no dependencies on the underlying protocol stacks for reliability and detection of available entities in the network. Therefore we can verify (a model of) a stand-alone service discovery system. (b) We successfully verify models of the system with resource-lean nodes because we partition the service discovery tasks based on the resource-constraints; resource lean nodes depend on more powerful nodes to perform correct service discovery. The results of the verification depend on the accuracy of the models [Bri02].

3. Incorporate recovery rules for the common failure scenarios - The recovery rules described in this chapter enable a service discovery system to recover from short and long term failures. We implement some recovery rules that are not found in existing systems (e.g. SRN2). We also implement the rules that support the context of our operational issues (e.g. for PR5, instead of implementing multicast service advertisements and multicast queries like UPnP, we only implement multicast queries when the Registry fails to reduce bandwidth consumption).

These opportunities lay the foundation for the design of FRODO which we present in Chapter 4. In Chapter 5, we show that FRODO satisfies the Service Discovery Principles through formal verification. FRODO is the first service discovery system that provides some notion on functional correctness. We also analyze the effectiveness of the recovery rules in FRODO, UPnP and Jini though simulations.

Chapter 4

FRODO System Overview

Programs for sale: Fast, Reliable, Cheap: choose two.
Anonymous

4.1 Introduction

In this chapter, we describe a *Framework for Robust and Resource-aware Discovery* (FRODO)[1] [Sun03], our service discovery system built for the home environment.

First, we describe our design approach, that leads to an in-depth understanding on the behavioral properties of service discovery systems, and eventually the formulation of the Service Discovery Principles and the recovery rules already presented in Chapter 3. As summarized in Figure 4.1, there are three phases to our design approach:

- In Phase 1, we analyze the requirements for service discovery in the home. We also systematically identify areas for improvement in existing state of the art; our taxonomy of service discovery systems in Chapter 2 is a result of this phase.
- Phase 2 is divided into two parts: (2a) High-level system design of FRODO, and (2b) Design evaluation. For the initial high-level design of the protocol, we use diagrammatic notations. For a more detailed and precise specification, we use the executable Architectural Description Language (ADL), Rapide [Luc98]. To evaluate the design, we test the Rapide-based specification of FRODO against different simulated failure scenarios. We then

[1] This work is published in the 4th Int. Conf. on Information, Communications & Signal Processing and 4th IEEE Pacific-Rim Conf. On Multimedia (ICICS/PCM), Singapore, vol. III, IEEE Computer Society Press, Dec. 2003, pp 1929-1933

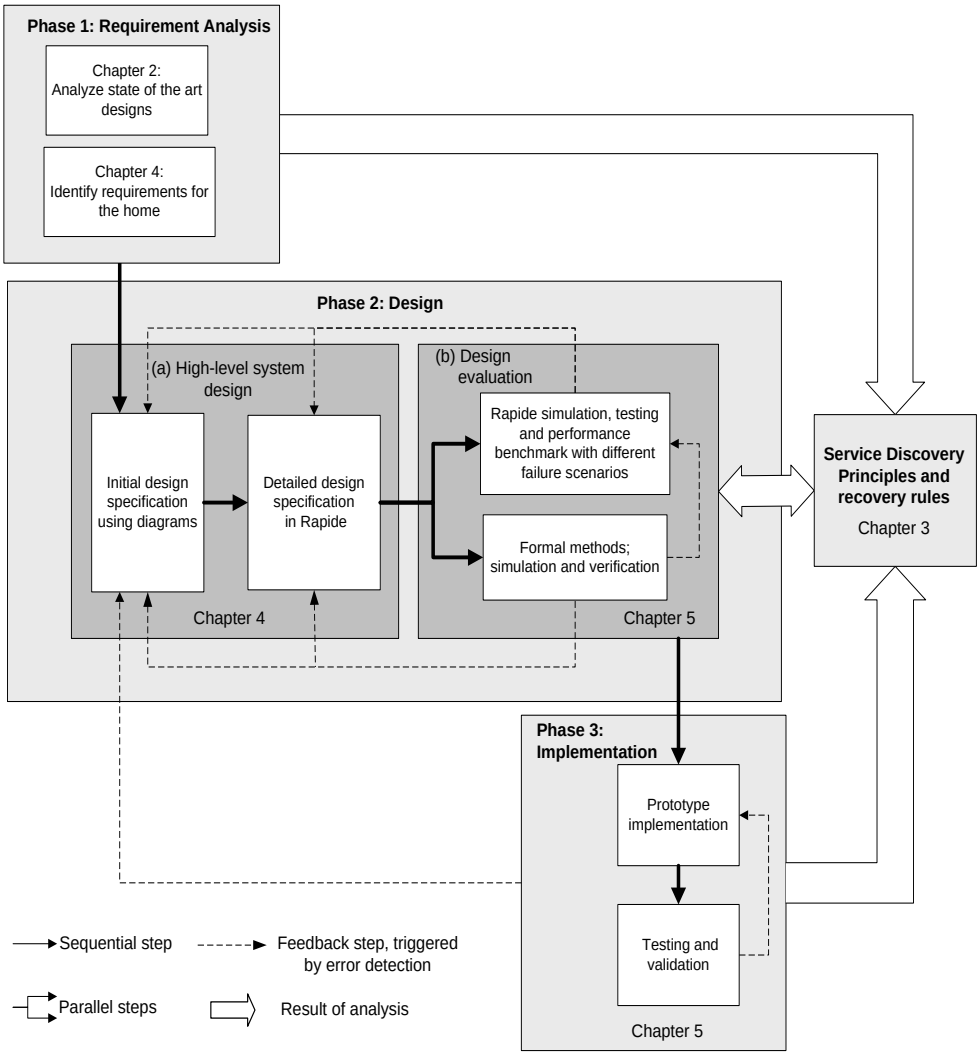


Figure 4.1: *The FRODO design approach. In Phase 1, we analyze state of the art designs and identify areas for improvement. We also identify requirements for service discovery for the home. In Phase 2, we specify the high-level design of the protocol in flowcharts and Rapide. We then improve and evaluate the design through model-checking and simulation. In Phase 3, we implement our prototype, which we use to compare the implementation performance against the simulated model*

use formal methods to detect hard-to-find design errors, and to give some notion on functional correctness. We enhance and evaluate the performance of FRODO by comparing it against the Rapide-based models of Jini and UPnP.

- Phase 3 is the realization of our design through prototyping, and validation of a selected subset of the simulations via measurements.

We introduce several innovations to service discovery systems:

1. Classification of devices based on resource-constraints, so that service discovery tasks can be partitioned effectively.
2. Primary-based recovery protocols [Tan02b] such as a Backup for the Registry, and Registry monitoring.
3. Registry election by a class of powerful nodes, so that a single Registry is elected. Registry election allows the system to continue functioning, even when the Registry and the Backup face persistent failures simultaneously.

This chapter is organized as follows. Section 4.2 describes the overall design approach of FRODO, but focusing more on the requirements (Phase 1). Section 4.3 gives an overview of the high-level design of FRODO (Phase 2a), which is the product of our design approach. Since we give details on the design evaluation (Phase 2b) and prototype implementation (Phase 3) in Chapter 5, we do not explain these phases in this chapter. We discuss the advantages and disadvantages of our approach to designing FRODO in Section 4.4.

4.2 FRODO Design Approach

We base our design approach shown in Figure 4.1, on the well-known Waterfall model [Roy70]. The emphasis in this thesis is more on the design, and less on the development. This approach saves implementation time because most of the protocol errors are detected and rectified early in the design phase (less than four months for developing the prototype). Furthermore, it is easier to modify and strengthen the protocol in the design phase, compared to the implementation phase. We strengthen the design by incorporating formal methods (model-checking) and simulations (with a performance benchmark) into the design phase.

Table 4.1: *Requirements, design solutions and assumptions in FRODO*

Requirement	Operational Issue	Design solution	Assumption
R1	**System size**	Small sized system, with a single Registry architecture	The home consists of a limited number of devices
R1	**Resource-constraints**	Service discovery functions are partitioned across device classes, based on increasing resource-constraints; 3C, 3D and 300D. The Registry architecture allows 3C and 3D nodes to depend on 300D nodes to elect and maintain the Registry. 3C and 3D Users and Managers behave differently from 300D Users and Managers.	At least one 300D node is available for the role of Registry (e.g. TV, set-top box, PC).
R2	**Lossy environment**	(1) Automatic Registry election to eliminate a single point of failure. (2) A Backup that monitors the liveliness of the Registry, and automatically replaces the Registry. (3) Retransmission and acknowledgements of critical messages. (4) Leasing and polling for automatic garbage collection. (5) Periodic announcements to detect purged entities. (6) Negotiation among multiple Registries to maintain a single Registry. (7) Cache discovered entities, and cross-check senders of selected messages, so that a purged entity can be rediscovered	(1) Temporary communication failure (message loss, link failure, etc.) and node failure (crashes, interface failure, etc.). (2) No system administration.
R3	**System heterogeneity**	Detection of entities, and retransmissions and acknowledgements are done in the service discovery protocol	Any pair of nodes in the system can receive and send messages to each other. *Unicast* is "one-to-one communication", and *multicast* is "one-to-many communication".
R4	**Security**	Security measures are not included in the service discovery functions	The application layer provides security.

Phase 1: Requirement Analysis. We define the following as the requirements for service discovery in the home:

(R1) Low cost of devices - New, sophisticated technologies should not consume too much resources, nor should they increase cost.

(R2) Robust - System administration is not available, therefore the system should continue to meet user expectations, and automatically recover from failures as fast as possible.

(R3) Portable design - the technology should be able to run on a variety of network stacks, because devices are manufactured by different vendors.

(R4) Secure - the system should not be vulnerable to security threats such as unauthorized access, denial of service, repudiation, etc.

The following describes the design solutions that we provide to satisfy the requirements above. When necessary, we refer to Table 4.1, which summarizes the design solutions and the assumptions in FRODO, according to our study done in Chapter 2(Sections 2.6 and 2.9).

(R1) As shown in Table 4.1(first row), FRODO is a single Registry-based architecture. This is because FRODO is a small-scale system, and a single Registry allows resource-lean nodes to depend on the more powerful Registry for service discovery. Multiple Registries unnecessarily waste resources. We also classify devices based on resource-constraints. The device classification enables the service discovery tasks to be partitioned based on the limitations of the devices:

- 300D (Dollar) device class - powerful devices, controlled by a complex embedded computer. Their memory requirements are more than 1MB (e.g. set-top boxes). A node in this class can be a Manager, a User, and a Registry.
- 3D device class- medium complex devices (e.g. temperature controller). A node in this class can be a Manager and a User, with fewer functions compared to the 300D entities.
- 3C (Cent) device class - simple devices with restricted resources (e.g. simple sensors). Nodes in this class are only Managers.

Figure 4.2 shows the three device classes in FRODO, and the entities contained in each class. Table 4.1(second row) summarizes the resource-aware measures that FRODO implements.

(R2) We incorporate fault-tolerant measures in the protocol, so that the system can recover from failures. Table 4.1(third row) lists the various recovery behaviors in FRODO.

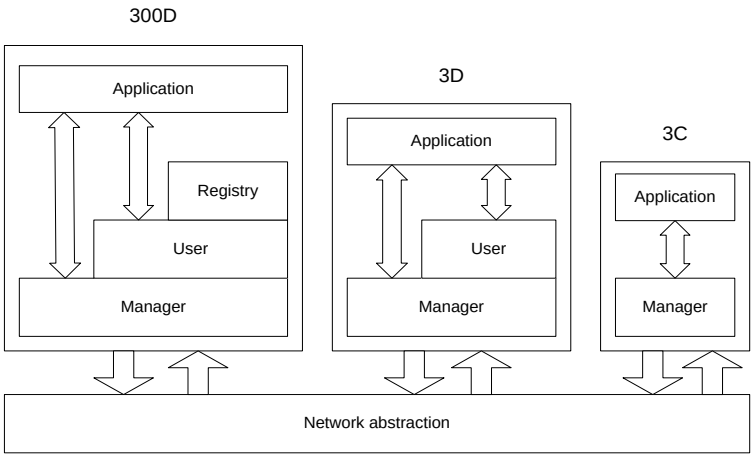


Figure 4.2: *FRODO device classes. 300D devices are the most powerful, thus the Registry is elected among these devices. The bigger the box that depicts the Manager or the User, the heavier the tasks for that device class. The application triggers the Manager and the User entities by providing the values for the attributes, and prompting when the User should discover the service*

(R3) FRODO is not influenced by the underlying protocol stacks, such as the type of transport layer and whether the network is based on wired or wireless communication. We abstract from the underlying protocol stacks, and simply require unicast and multicast capabilities from the network. Table 4.1(fourth row) summarizes our abstraction levels, and the definitions for unicast and multicast.

(R4) We do not integrate security features in our prototype, as stated in Table 4.1(fifth row). However, we require a full-fledged implementation and deployment of FRODO to include security measures (at least in the application layer), such as authentication, access control, encryption, etc.

Next in our requirements analysis is to specify the service discovery functions in FRODO. Chapter 2(Table 2.1) provides a study of the inherent functions of service discovery. We find Configuration Discovery, Registration, SD Discovery and Configuration Purge are implicitly implemented in the state of the art designs, hence we incorporate these functions in FRODO. However, only Jini and UPnP implement the complete Consistency Maintenance function (the User is updated). Partial implementation of Consistency Maintenance is available in most Registry systems (Registries are updated by Managers, or by each other, but the Users do not receive update notifications). We recognize Consistency Maintenance as a vital function in service discovery, because unattended systems with collaborating applications depend on the service discovery protocol to perform correctly. We

analyze in more detail the Consistency Maintenance function in Chapter 3 and Chapter 5.

Phase 2: Design. Once the requirements are clear, we enter the design phase. The design phase consists of the high-level system design of FRODO, and the design evaluation, which improves the design and performance. Further details on design evaluation are given in Chapter 5.

- Initial design - We specify the initial design using diagrammatical notations. Diagrammatical notations are beneficial because we focus only on the the protocol behavior, and are not distracted by implementation details (such as programming language syntax, packet size, timer values, etc.).

- Detailed, executable design using Rapide - The design specification using diagrams is not complete, because it does not provide details. Therefore, we use an ADL tool, called Rapide to further specify the design. Rapide is designed to support component-based development of systems by utilizing architecture definitions as the development framework. It offers event-based simulation for distributed, time-sensitive systems [Luc98]. The FRODO behavioral specification defines how an interface in FRODO (an ADL interface is analogous to a "class" in object-oriented programming) reacts to various inputs, produces output events, and change values of variables.

- Model-checking - To check the functional correctness of FRODO, we build several abstracted models of FRODO using the model-checker DT-Spin [Bos97]. Model-checking is useful because it gives an in depth understanding of how processes interact concurrently in a protocol, how to identify the properties that a service discovery protocol should satisfy, and how to detect design errors that are not captured during simulations. The properties we model-check inspired the formulation of the seven Service Discovery Principles.

- Rapide simulation and performance benchmark - We now focus on the non-functional aspects of the design. The resulting Rapide-based FRODO specification is executable, and can react to event-based simulation of communication scenarios. Simulations can be used at early stages of the design phase to investigate correctness and performance, before the system is built [Luc02]. We improve our design by testing with different failure scenarios. We also compare multiple versions of our design, and against Rapide models of Jini and UPnP (built at NIST), to improve performance.

Phase 3: Implementation. We implement a prototype of FRODO in C to validate the simulation and to measure the resource consumption. We show that FRODO is a lightweight protocol, even with various fault-tolerant measures. We successfully test and validate the prototype against the simulated model, to show that the

experimental observations corroborate the simulations. Details on this phase are given in the next chapter. On-going work in this area is to improve the prototype so that FRODO can be deployed as a full-fledged implementation.

Our design approach gives confidence in the design, which is demonstrated by the fact that only *around three design errors* were found during prototype testing and validation, (e.g. UDP packet size limit requires serialized FRODO messages to be packetized and numbered sequentially). Other possible errors were captured and rectified during the earlier design phase.

The design approach also gives a deep understanding of the complexity that underlies a service discovery system, and the behavioral properties. This understanding provides the inspiration for formulating the Service Discovery Principles and the recovery rules in Chapter 3.

4.3 FRODO Overview

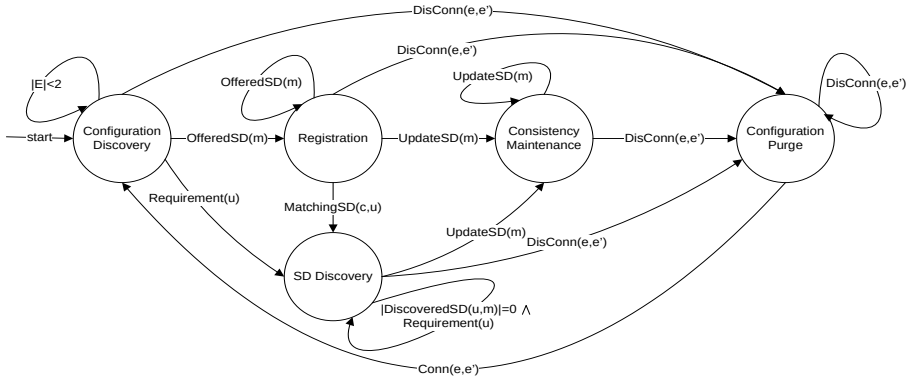


Figure 3.4: *Service discovery system life cycle. When a system consists of more than one entity, Configuration Discovery is performed, followed by Registration or SD Discovery. Registration is triggered when there exists a Manager with* $\mathsf{OfferedSD(m)}$*. SD Discovery is triggered when there exits a User with* $\mathsf{Requirement(u)}$*, or if the Registry has* $\mathsf{MatchingSD(c,u)}$*. SD Discovery is done every time there is a new requirement or as long as the User has not discovered the required service, where* $|\mathsf{DiscoveredSD(u,m)}| = 0$*. When* $\mathsf{UpdateSD(m)}$ *occurs in the Manager (SD changes), Consistency Maintenance is performed. Configuration Purge occurs every time entities face* $\mathsf{DisConn(e,e')}$ *due to failures. When failures end, and* $\mathsf{Conn(e,e')}$ *is restored, the cycle is restarted.*

We now provide an overview of the service discovery functions in FRODO. We show Figure 3.4 from Chapter 3 again in this section to describe the general lifecycle of the FRODO service discovery system. There are four main functions in FRODO; Configuration Discovery, Registration, SD Discovery, and Configuration Update (which consists of Configuration Purge and Consistency Maintenance).

Table 4.2: *Summary of the functions in FRODO, and the methods to achieve the functional objectives*

Function	**Subfunction**	**Method**
Configuration Discovery	Registry auto-configuration	(a) Registry election among 300D nodes, (b) Backup assignment by the Registry, (c) Registry and Backup handover upon detecting more powerful nodes
	Active discovery	Multicast announcements by the Registry, 300D nodes, and 3D Managers
	Passive discovery	All nodes listen to the multicast announcements of the Registry
Registration	Solicited registration	Registry requests an unregistered node that announces itself to register
	Unsolicited registration	Managers register when they detect a Registry that they have not seen before
SD Discovery	Unicast query	Users send unicast queries to the Registry to discover services
	Multicast query	Users send multicast queries to discover services, when the Registry is not available
	Service notification	Users request notification from the Registry, when Managers that match their requirements register
Configuration Update	Configuration Purge	(a) Lease renewals by 300D nodes, (b) Polling of 3D and 3C Managers by the Registry
	Consistency Maintenance	(a) 2-party subscription (300D Managers update the Users directly), (b) 3-party subscription (the Registry relays the updates from 3D and 3C Managers to the Users)

The transitions between functions are triggered by the appearance of services, $\mathsf{OfferedSD}(\mathsf{m})$, requirements, $\mathsf{Requirement}(\mathsf{u})$, service changes, $\mathsf{UpdateSD}(\mathsf{m})$ and conditions $\mathsf{DisConn}(\mathsf{e}, \mathsf{e}')$ (disconnection) and $\mathsf{Conn}(\mathsf{e}, \mathsf{e}')$ (valid communication paths exits between the nodes).

Table 4.2 gives more details on how each service discovery function from Figure 3.4 is implemented in FRODO. This table can also be used as a roadmap for reading the remainder of this chapter.

In this section, we first describe the ideal state of the function, when there is no failure of communication or nodes. Where necessary, we give insight into failure scenarios and explain the recovery behavior. For further details on the design, which includes failure scenarios, we refer to the design specifications [1] .

[1]Further design details can be found in the Frodo High-Level and Detailed Design Specifications, version 1.0. Technical Report TR-CTIT-06-25, published at http://eprints.eemcs.utwente.nl/2710, Enschede, June 2006.

4.3.1 Configuration Discovery

FRODO provides zero-configuration through Registry election and automatic entity discovery through multicast announcements. FRODO is the first Registry-based service discovery system to: (1) automatically elect a Registry based on resource-constraints, (2) maintain the liveliness of the Registry through primary-based recovery protocols such as automatically appointing a Backup, and monitoring the Registry, and (3) conduct negotiation among multiple Registries so that the system converges successfully, after network partitioning (caused by communication failures).

Managers need to discover the Registry to register and update their services, while Users need to discover the Registry to query for services. 300D and 3D nodes are able to discover the Registry through *active* discovery, where the nodes initiate the discovery by sending multicast announcements. All nodes can also discover the Registry through *passive* discovery, by listening to Registry announcements (3C Managers discover the Registry using only this method, since they are assumed not to have the capability to do periodic announcements).

We give the details of Registry auto-configuration, active and passive discovery in the following.

4.3.1.1 Registry auto-configuration

Unlike competitor Registry-based systems like Jini and SLP, the Registry in FRODO is elected without the intervention of a system administrator. There are several methods to ensure that a single Registry is automatically elected, and maintained. First, *Registry election* is carried out by 300D nodes to elect the most powerful node as the Registry. Next, *Backup assignment* is done by the Registry. *Registry and Backup handover* is done when a more powerful node than either the Registry or the Backup is detected (the new node must have processing power above a certain threshold, defined at system implementation).

1. Registry election. A 300D node sends a multicast *MyResource* message at initialization (in this chapter, we do not give details on message format, as this information is available in the high-level and detailed specifications [Sun06a]). If the 300D node receives another *MyResource* message, the election is started. Else, the 300D node simply times out and becomes the Registry. The most powerful node which has the highest rank is elected as the Registry. The Registry sends a multicast *IamRegistry* message to announce its existence. Henceforth, this announcement is sent periodically.

   The following are the parameters for the election:

   (a) device orientation, where a purely wired device is given the highest priority, and a purely wireless device is given the lowest priority. A device with both orientations (such as a laptop) is given medium priority.

(b) processing power
(c) available memory size
(d) unique device identifier

A 300D device checks the parameters of a remote device contained in the received *MyResource* message to determine whether it is superior. The values of the device orientation and processing power are given more priority thus will be compared first. Memory size and the value of the unique device identifier will be considered to break a tie between two devices with the same device orientation and processing power. This is because we assume that any device with a minimum available memory size of 1MB is sufficient to store configuration information (e.g. registrations, subscriptions). The universally unique device identifier is used to break a tie between two devices of the same type by comparing which device has the larger identifier.

Failure recovery:

- It is possible that a Registry is successfully elected, but crashes before it can announce itself. To ensure that such a scenario does not cause the system to halt, we make 300D nodes that lost the election wait for the Registry announcement. Upon detecting that the Registry has become silent after the election, a 300D node can restart the election by requesting (through multicast) another election round. 300D nodes that receive this request wait for some time before actually restarting the election, to give the existing Registry a chance to announce itself again (just in case the initial Registry announcement is lost).
- Failures can cause multiple Registries to be elected. Registries *negotiate* to produce a single Registry, by sending messages containing their resource information, exactly as done in the *MyResource* messages.

2. Backup assignment. During the election, a node that eventually becomes the Registry creates a *RankList* consisting of resource information of 300D nodes that participated during the election. This allows the Registry to assign or replace a Backup. The Registry assigns the next powerful node in the *RankList* as the Backup. Once the Backup acknowledges the assignment, the Registry sends the tables that contain configuration information. There are several tables that the Registry maintains and backs up:

   (a) *ServiceLookup* - registration data on available SDs
   (b) *RankList* - list of available 300D nodes and their resources
   (c) *ServiceNotification* - list of Users and the services they seek, so that Users can be notified if a required service becomes available
   (d) *SubscriptionTable* - list of Users and the Managers for update propagation

Failure recovery:

- To indicate success or failure of receiving the tables, the Backup sends a response (an *Info* message), with the flag for each table turned on (if successful). The Registry resends the missing table, where upon successive failures, the Registry cancels the Backup and assigns another as the Backup.
- The Registry and the Backup poll each other periodically, by sending *Hello* messages. The Registry polls first. If the Backup does not receive the poll until a waiting period expires, the Backup initiates the poll. The Backup waits for the poll from the Registry, before it assumes the Registry has crashed, and sends the *IamRegistry* announcement.
- Other 300D nodes may detect that the Registry is no longer responding, in which case, they send a request for reelection. Upon receiving this request, the Backup quickly polls the Registry to confirm the validity of the request. The Backup then sends the *IamRegistry* announcement to stop the full election from taking place, if there is no response from the Registry.
- The Backup may receive multiple assignments from different Registries, due to network partitioning. The Backup notifies these Registries of the presence of each other. This speeds up Registry negotiation.

3. Registry and Backup handover. A 300D node may enter the system after the Registry election is completed. The resources of the new node are compared to those of the Registry and the Backup. If there is a substantial difference, the role of the Registry or the Backup is handed over to the new node.

Failure recovery: The Registry is able to cancel Backups in case of multiple Backup assignments due to network partitioning.

Future work includes investigating how the Backup and 300D nodes can monitor the Registry's reliability. Nodes should be able to isolate the Registry based on a "reliability factor" which is reduced every time the Registry is disconnected. Nodes can agree on whether the Registry should be isolated based on a majority consensus [Pea80; Wan05].

4.3.1.2 Active discovery

A 300D node discovers the Registry when it receives a response to the *MyResource* message, which it sends upon initialization. A 3D node sends a multicast *SmallDevAnnouce* message periodically to indicate its presence to the Registry. 3D nodes uses a *bounded exponential back-off algorithm* [Tan02a] to conserve resources (the waiting period for the next announcement is twice the wait period for the previous announcement, until a limit is reached, where upon the node then resets the waiting period to the initial period).

4.3.1.3 Passive discovery

All nodes listen to the periodic Registry announcements. A 300D node keeps a list of recently discovered Registries, so that it may register with a new Registry. This is because the Backup may take over the role of the Registry, with no knowledge of the 300D node. The 300D node has to initiate registration upon discovering a new Registry.

300D nodes also monitor the periodic announcement of the Registry, so that they can restart Registry election when the expected announcement does not arrive. 3D and 3C nodes cache the Registry identifier of the most recent announcement.

Failure recovery: Both active and passive discovery are the failure recovery mechanisms for rediscovering nodes that are purged (because of communication failure), but are still available in the system.

4.3.2 Registration

Registration takes place when a Manager has at least one SD (Service Description) to register. The SDs are stored by the Registry, in the *ServiceLookup* table. There are 2 types of registration methods; *solicited* and *unsolicited* registrations. Solicited registration allows the Registry to initiate the registration when it discovers an unregistered Manager. Unsolicited registration allows a Manager to register with a Registry that it has not seen before. We describe each of these methods in the following.

4.3.2.1 Solicited registration

Solicited registration is initiated by the Registry when the Registry receives an announcement (a *SmallDevAnnounce* or a *MyResource* message) from an unknown source. The Registry requests the unknown source to register (by sending a *SrvRegReq* message). If the node is a Manager, it replies with a *SrvReg* message for each SD. Part B of Figure 4.3 shows this interaction.

4.3.2.2 Unsolicited registration

For 300D nodes, unsolicited registration is done when a new Registry is detected through passive discovery. 3D and 3C nodes only initiate unsolicited registration when they purge the Registry (when an expected response does not arrive), and upon receiving a fresh Registry announcement.

Failure recovery:

- Every service registration message is acknowledged to indicate successful registration. If the acknowledgement is not received, the Manager retries

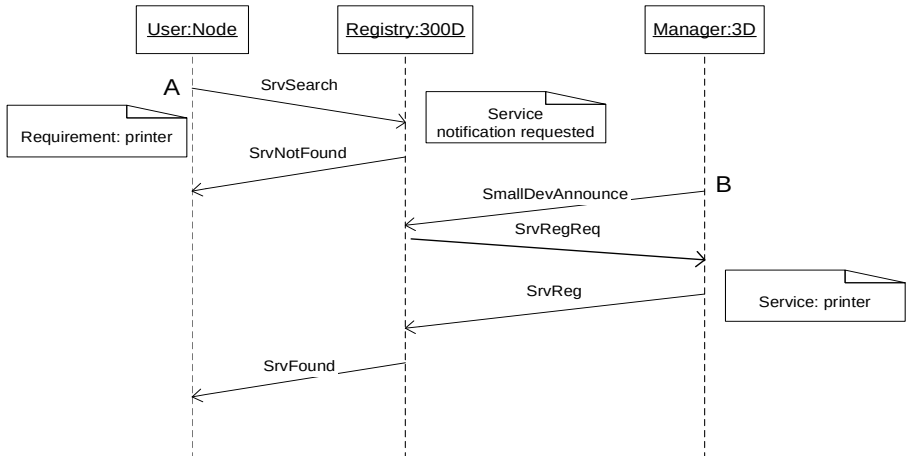


Figure 4.3: *Unicast query, with notification for unavailable services. The Registry notifies the User with a SrvFound message when a matching service becomes available*

.

once, before assuming the Registry has crashed. When this happens, a 300D Manager would request Registry re-election, while a 3D Manager would restart its bounded exponential back-off announcements.

- The *IamRegistry* message is used to solicit registration if the Backup takes over as the Registry with empty tables (the Registry crashes before it can send the tables to the Backup). The Managers check the appropriate flag in the *IamRegistry* announcement to determine if they need to re-register.

4.3.3 SD Discovery

There are three methods for Users to discover services: *unicast query* to the Registry, *multicast query*, and *service notification* from the Registry. We discuss each of these methods in the following.

4.3.3.1 Unicast query.

The User queries the Registry using a unicast *SrvSearch* message. If the service is registered in the *ServiceLookup* table, the Registry returns the SD through a *SrvFound* message; otherwise a *SrvNotFound* message is sent.

4.3.3.2 Multicast query.

When the User is unable to discover or communicate with the Registry, the User sends a multicast *SrvSearch* query. The Managers with services that match the

requirements of the User reply with *SrvFound* messages.

4.3.3.3 Service notification.

The User can also request the Registry to notify it if services that match its requirements become available in the future (the appropriate flag in the unicast *SrvSearch* message is turned on). Part A of Figure 4.3 shows a scenario where the Registry notifies the User when the required Manager registers.

Failure recovery: The Registry may not respond to the unicast *SrvSearch* message because of temporary communication failures. In this case, the User sends a multicast query to discover the required service. When the Registry receives the multicast query from the User, the Registry replies appropriately to the query (*SrvFound* or *SrvNotFound* message). The Registry also solicits registration from the User to notify the User of its presence. Consequently, the User reverts to unicast queries in the future.

4.3.4 Configuration Update

In this function, (1) the Registry automatically detects nodes that have left the system, and (2) both the User and the Registry discover changes to cached SDs.

4.3.4.1 Configuration Purge.

This function detects nodes that have left the system through two automatic garbage collection mechanisms:

1. Lease renewals - 300D nodes are expected to refresh their registrations periodically. Thus, a 300D node maintains a "lease" with the Registry for every service registered. If the lease is not renewed, the Registry assumes the node has left the system, and purges all configuration data on this node. The lease period is requested by the Manager, according to its own criteria (e.g. a mobile node may request a short lease period). If the lease period exceeds the limit set in the Registry, the Registry requests the Manager to shorten the lease period. This method is similar to leasing in Jini.

2. Polling - The Registry periodically polls 3D and 3C Managers. These nodes need not keep track of leases as done by the 300D nodes.

Failure recovery:

- Configuration Purge is by itself the failure recovery mechanism for detecting defunct nodes and services.

- Lease renewal and poll messages may be lost, and Managers can be wrongly assumed to have left the system. Therefore we specify that only if two consecutive lease renewal or poll messages are lost, the registration of the Manager is purged.
- After the first lease renewal or poll expires, the SD is set as "inactive". This prevents positive replies to queries on this service.

4.3.4.2 Consistency maintenance.

This function updates cached SDs. Users *subscribe* to either the Registry or the Manager to receive updates on cached SDs. There are two types of subscriptions:

1. 2-party subscription. 2-party subscription requires 300D Managers to administer the subscription, and update the Users directly.
2. 3-party subscription. 3-party subscription supports resource lean 3D and 3C Managers by delegating subscription administration to the Registry.

Once a subscription request is acknowledged, the User periodically renews its subscription lease to indicate continuous interest to receive updates.

Failure recovery:

- As in lease renewals, the subscriber is purged by the Manager and the Registry after two consecutive times the subscription lease period expires.
- Upon receiving *SubscriptionRenewal* message from a purged User, the Manager or the Registry sends a *Resubscribe* message so that the User can refresh its subscription.
- The *ServiceUpdate* message is retransmitted (we use 4 retransmissions, as done in TCP connection setup). Once the retransmission limit is reached without an acknowledgement, the Manager or the Registry gives up, until the User renews its subscription, and the update is resent.

We evaluate consistency maintenance and the recovery behaviors of FRODO, Jini and UPnP in more detail in the next chapter.

4.4 Discussion and Conclusion

In Chapter 2, we characterize the design space for service discovery, as summarized in Figure 2.4. In this chapter, we explore a subset of the design space, as described in Table 4.1. This subset is defined by the requirements for our home environment specified in Section 4.2; low cost of devices, robust, and portable design. These requirements follow from the 3Rs in Chapter 1; *reliability* is satisfied because the

system is robust against communication and node failures. *Resource-constraints* is taken into account because the system is resource-aware (thus cost of devices are not significantly increased). *Heterogeneity* of device architectures and networks is masked by removing dependencies on the capabilities of the underlying protocol stacks.

To satisfy the requirements, our service discovery system has the following properties: (1) *self-configuration*, where the most powerful device is elected as the Registry, and multicast communication is leveraged for detecting new entities, (2) *self-healing*, with various failure recovery behavior against communication and node failures, (3) *resource-aware*, where devices are classified so that service discovery tasks can be partitioned according to resource-constraints, and (4) *portable design*, where there are no dependencies on the capabilities of the underlying protocol stacks, other than the requirement for unicast and multicast communication. As a result, we build a small-scale service discovery system that does not require administration, supports resource-lean nodes and can be used over heterogeneous devices and networks.

The areas in the design space that we do not explore are scalability and security. We offer the following justifications for these omissions.

Our design does not solve scalability issues such as how the Registry can efficiently process queries and cache registration data, or whether a single Registry can cause a bottleneck. This is because we expect that there will be a limited number of devices in the home environment, where a single Registry (and a Backup) is sufficient. A scalable design includes multiple Registries, and efficient query mechanisms (e.g. DHTs and Bloom filters, as mentioned in Chapter 2). Scalability is also not addressed by state of the art designs for the home, such as Jini, SLP and UPnP. Even though Jini and SLP allow multiple Registries, Registries do not forward queries and update each other, nor is the message transmission optimized to support an internet scale network, unlike truly scalable systems like INS/Twine and SDS (described in Chapter 2).

We also do not address security issues because our focus is on designing a system that regulates itself through self-configuration and self-healing. As described in Section 1.2, we do not address self-protection of the system. Security reduces self-configuration, because there must be a pre-configured and trusted authority in the system (to authenticate and authorize entities). Security also consumes resource due to encryption algorithms and protocols. As a consequence, the FRODO design is vulnerable to certain types of threats (such as denial of service). Therefore, as mentioned in our requirement analysis in Section 4.2, it is essential to incorporate security at the application level during a full-fledged implementation and deployment of FRODO (as also required for implementing Jini and UPnP). Important future work in this area is to investigate how security can be integrated into service discovery, while preserving self-configuration.

We do not investigate the best ways to implement the Service Description (SD). We simply require that the service provides information on the device type, service type and includes a list of attributes. For a successful implementation

of the FRODO system, a consortium such as The European Application Home Alliance (TEAHA) [TEA04], consisting of device and application manufacturers should standardize the SD format and values.

Having justified the omissions, we now discuss the design space that we do explore.

To enable self-configuration, we implement Registry election. The 300D node with the most powerful resource is elected as the Registry. We also use periodic multicast announcements so that entities can automatically discover (or rediscover) each other. We allow the role of the Registry and the Backup to be handed over to more powerful devices that appear later in the system. This allows the system to self-optimize so that only the most powerful nodes are given the heaviest tasks. The downside to Registry election is uncertain behavior of the system during severe communication failure (such as message loss or link failures) which causes network partitioning. When the network is partitioned, multiple Registries can be elected at the same time. We do support this scenario by requiring the Registries to negotiate and reelect a single Registry, when the network converges. However, during network partitioning, Registries may not have a consistent view of the available services, and therefore return false replies to queries from Users. Nevertheless, by implementing both solicited and unsolicited registrations, we can eventually regain the lost and inconsistent registration data.

Our design does not depend on the capabilities of specific protocol stacks, such as the type of transmission (e.g. TCP, UDP), routing protocol (e.g. IP, AODV), MAC layer (e.g. CSMA/CA in WLAN, CSMA/CD in Ethernet) and physical layer (wired or wireless). The advantage of abstracting away the underlying protocol stacks is a portable design that can be implemented over heterogenous protocol stacks. This choice is justified because the resource-lean 3D and 3C device classes are unable to support heavy protocol stacks, unlike the 300D device class. Therefore, the FRODO design is appropriate for an environment such as the home that consists of a variety of devices.

We provide failure recovery for various failure scenarios, including some which may seem extreme (such as the Registry and the Backup failing at the same time). However, to exclude such scenarios leads to serious consequences (e.g. no other Registry is elected after the first crashes!). In our design, at every "if-else" fork, we try to capture the failure scenario and provide the recovery behavior (around 20% of the lines of code in the prototype for 300D, and 10% for 3D). This approach is necessary, because we require both a self-configured and self-healing system. As a result, our service discovery protocol is slightly more complex than the discovery behavior in Jini and UPnP. Jini is a fixed Registry architecture, which requires the Registry to be deployed by a system administrator. UPnP is an even more straightforward design because it is a non-Registry based architecture. On the other hand, Jini and UPnP depend on heavy protocol stacks. Thus, even with various failure recovery mechanisms, FRODO still maintains good (and in some cases, better) performance when compared to Jini and UPnP, and has low resource consumption as shown in Chapter 5.

The main disadvantage of designing a self-healing system is that we increase the protocol complexity. Protocol complexity increases chances for more design and implementation errors. Furthermore, resource consumption is also increased to support various failure recovery mechanisms. Nevertheless, because we classify devices based on resource constraints, most failure recovery tasks are assigned to the powerful 300D device class. Therefore partitioning tasks across device classes allows us to achieve robustness through 300D devices, while still allowing resource-lean 3D and 3C devices to take part in the system. State of the art systems like Jini and UPnP require all entities (Users, Managers and Registries) to have the same behavior, thus excluding resource-lean nodes from taking part in service discovery.

In the next chapter, we give our evaluation of FRODO and compare the performance of FRODO against UPnP and Jini, which are two well-known state of the art systems for the home environment.

Chapter 5

Evaluation

No amount of experimentation will ever prove me right,
but ONE experiment can prove me wrong.
Albert Einstein

5.1 Introduction

This chapter summarizes our analysis of the FRODO system. To detect and rectify design errors, we formally verify FRODO against the Service Discovery Principles [1] . To enhance the performance of FRODO, we benchmark through simulation, the performance of FRODO against Jini and UPnP, during communication [2] and node failures [3] . We incorporate functionalities that provide equal, and in some cases, better responsiveness, effectiveness and efficiency than the competitor systems. Finally we validate that the simulation results of FRODO and Jini are consistent with the real-life performance of these systems.

Contributions: This chapter contributes to a better understanding of the design and analysis of service discovery systems by:

[1] Results have been published in Proceedings of the 30th Annual IEEE Conference on Local Computer Networks (LCN 2005), Sydney, Australia, pp. 209-217, IEEE Computer Society Press.

[2] Results have been published in Proceedings of the 1st Int. Conf. on Communication System Software and Middleware (COMSWARE), New Delhi, India, pages to appear, IEEE Computer Society Press.

[3] Results have been published in Proceedings of the 20th IEEE Int. Parallel & Distributed Processing Symposium (IPDPS 2006), Rhodos Island, Greece, page 10pp, IEEE Computer Society Press.

1. Providing an approach for formally verifying a model of a service discovery system against the Service Discovery Principles. FRODO is the first service discovery system that gives a notion of correct behavior though formal verification.

2. Identifying the failure scenarios that cause small scale, unattended service discovery systems, with unreliable transmission to violate the Service Discovery Principles, and the resulting design solutions.

3. Analyzing the effectiveness of the recovery rules implemented in UPnP, Jini and FRODO, and comparing the overall consistency maintenance performance of the three systems in a lossy environment.

The rest of the Chapter is organized as follows. In Section 5.2, we describe the formal verification of FRODO, including the modeling approach, properties to verify, and the results of the verification. In Section 5.3, we benchmark through simulations, the models of FRODO against the models of UPnP and Jini. We describe the modeling approach, the performance metrics, and compare the consistency maintenance mechanisms in the three systems. We benchmark the models of the three systems in two experiments with different failure types; message loss and interface failure. We compare the effectiveness of the recovery rules presented in Table 5.6 (in Chapter 3), that are implemented in UPnP, Jini and FRODO. In Section 5.4, we give a brief description of our prototype, and the resources consumed by the different device classes in FRODO. We compare the consistency maintenance performance of our implementations of FRODO and Jini, and validate the simulation results for a Registry failure scenario.

5.2 Modeling and Verification of FRODO

We use model checking [G.J03] to verify that FRODO adheres to the Service Discovery Principles. If there are cases where the protocol fails the verification, we identify whether the error lies in the modeling process, or in the design phase. The former requires the model to be corrected (and often our understanding!). However, it is the latter which is most beneficial, since detecting a design flaw leads towards discovering design solutions that improve and strengthen the protocol, until the Service Discovery Principles are satisfied.

We verify the desired behavioral properties of the protocol through exhaustive enumeration (explicit or implicit) of all the states reachable by the system and the behaviors that navigate through them, given a limited number of system scenarios (but carefully selected based on the service discovery functions in Figure 3.4). We use the model-checker DT-Spin [Bos97], for this purpose. DT-Spin is an extension of the well-known SPIN tool [G.J03]. DT-Spin is used instead of the original SPIN model checker because we need multiple timeouts that occur at any time, even if

other processes are still enabled in the model. Timeouts are essential requirements for the recovery processes of the protocol against failures. The *timeout* variable in standard SPIN cannot be used as it is only true when the system is idle. However, DT-Spin also increases the state space with respect to standard Spin, because time is modeled as ticks, and each tick occupies a state. A reasonable number of ticks is set for each timer, to make verification feasible.

5.2.1 Modeling Approach

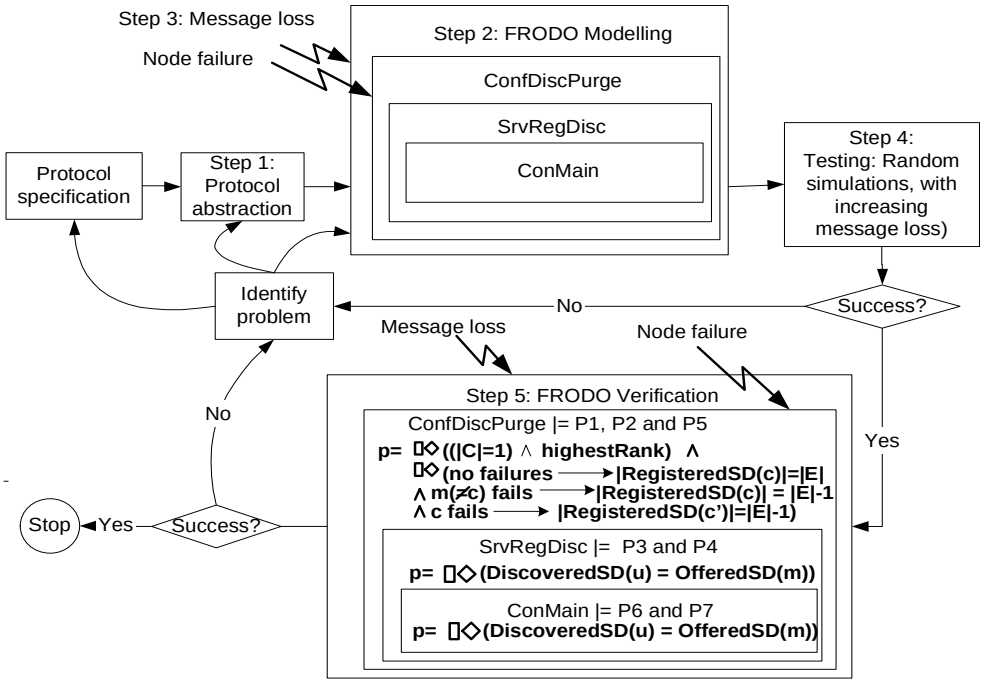


Figure 5.1: *Modeling FRODO. The FRODO Modeling box shows the abstraction link between the 3 modules, where the outer boxes abstract some functions from the inner boxes. Connectivity and Global Connectivity are modeled with/without message loss respectively, and Disconnect is modeled as node failure.*

It is impractical to work with monolithic models of complex protocols such as FRODO [Ruy00]. In the first place error traces would be too cluttered with irrelevant detail to be able to spot the real problems, and secondly the state space would grow beyond the bounds of what the current tools can cope with. Therefore we split the FRODO protocol in a number of modules, each of which corresponds roughly to one of the four functional areas or sub-functions thereof (Chapter 4 describes the functions in FRODO). Each model is then provided in

several versions, including a concrete model with the most detail, some versions that represent worst case behavior, and a number of abstract versions with as little detail as possible. By combining a concrete version of one module with appropriate abstract versions of all others, the system as a whole can be verified, focusing on the behavior of the concrete module. The main challenge of this method is to keep the different versions of each module consistent. Figure 5.1 shows the approach that we use to model, simulate, and verify FRODO against the Service Discovery Principles.

Step 1: Protocol abstraction. Each assembly of modules builds on a common layer of protocol abstraction. We deliberately abstract irrelevant detail such as message format.

Step 2: Modular decomposition. FRODO consists of three modules: a) *ConfDiscPurge* models Configuration Discovery and Configuration Management. This module is actually broken into four sub-functions as follows: *ConfDiscPurge-1* models the leader election protocol which elects one Registry. *ConfDiscPurge-2* models two worst-case scenarios of the protocol with (i) all nodes claiming to be Registry after a network partitioning, and (ii) all registration information in the Registry being lost because of prolonged communication failure. *ConfDiscPurge-3* models a Backup taking over as the Registry when the existing Registry leaves the system. This model uses node failure, instead of message loss to model network disturbance. *ConfDiscPurge-4* models the Registry handing over to a superior node that enters late into the system. This model is an abstraction of ConfDiscPurge-1, where message loss is already checked, and therefore not included in ConfDiscPurge-4. b) *SrvRegDisc* models both Registration and Service Discovery functions. Configuration Discovery is abstracted away in this model, i.e. a Registry is successfully elected. The model consists of one 300D node which is the Registry. Service discovery is done through the Registry using the directed search mechanism, thus the service cannot be discovered unless registration occurs. The discovery of each type of Manager (3C, 3D and 300D) is modeled separately. c) *ConMain* models the Consistency Maintenance function, which propagates updates to Users through 2-party and 3-party subscription.

Step 3: Message loss. Failure is modeled as message loss. All models except ConfDiscPurge-3 and ConfDiscPurge-4 are simulated and verified with and without message loss. A receiver will continue executing its tasks, unaware of any message loss, which may lead towards a violation of the Service Discovery Principles. We use a counter for every message *type*, which increments whenever a particular type of message is lost. The message may be lost, until a constant *MAX_LOSS* for its type is reached. The following is an example of how a *SrvRegReq* message is sent or lost. The variables *rcvrID, OfferedSD*, and *srcID* are the

receiver node's identifier, the service identifier that represents a service and the sender node's identifier respectively.

```
if
:: loss_counter[type]<= MAX_LOSS ->
          lossCounter[type]++
/* transmit message */
:: send!SrvRegReq,OfferedSD,srcID,rcvrID;
fi
```

Message loss increases the number of states because of the additional counter and timing variables and non-deterministic choice. The impact of message loss on the verification is shown in Table 5.1. In the example, when MAX_LOSS > 1, state space explosion causes the model-checker to halt due to machine memory limitation (we use machines with available memory of 20GB).

Table 5.1: *An example of the impact of message loss on state space. In this example, when MAX_LOSS > 1, the verification halts because of machine memory limitation.*

Model	**State vector (bytes)**	**Depth**	**States stored**
ConfDiscPurge-1, exhaustive mode, no message loss	304	37328	38345
ConfDiscPurge-1, supertrace mode, MAX_LOSS=1	308	61993	497,662 billion

Step 4: Testing. We use the simulator tool in DT-Spin to test and debug the models. We test different, random simulation scenarios and increase the MAX_LOSS for every simulation (up to 10 message losses to capture extreme scenarios)

Step 5: Verification. DT-Spin checks correctness claims that are generated from logic formulas expressed in LTL. When a claim is invalid for a model, the tool produces a counter example that explicitly shows how the property was violated. The counter example is a feedback for the simulator tool of DT-Spin to show the execution trail that causes the violation.

5.2.2 Property Modeling

We now model the desired behavioral properties of FRODO. We use the parameters and notations defined in Section 3.2.3 of Chapter 3 to model the properties of FRODO. The interpretations of the protocol-dependent parameters, described in Section 3.2.2 of Chapter 3 are:

1. Connectivity condition: $\mathsf{Conn}(\mathsf{e}, \mathsf{e}')$: *if a message is transmitted from either* e *to* e'*, or vice versa, and the expected acknowledgement arrives, the entities are reachable.*

2. Disconnect condition: $\neg\mathsf{Conn}(\mathsf{e}, \mathsf{e}')$: *for twice the timeout period* (as explained in Section 4.3.4, Chapter 4).

3. $\mathsf{Rank}(\cdot)$, the function used in Registry election, that returns the highest ranked 300D node as the Registry (as explained in Section 4.3.1, Chapter 4).

4. $\mathsf{G}(\mathsf{e}) = \mathsf{E}$ the nodes are interested in any Registry as there is only a single Registry in FRODO.

5. $\mathsf{N} = 1$, a single Registry is elected.

We model the response pattern $\Box(\mathsf{p}' \rightarrow \Diamond\mathsf{p})$ (described in Chapter 3), by building the property p' directly into the models, so that p is verified as a *recurrence property* [G.J03]. *The recurrence property* $\Box\Diamond\mathsf{p}$ *states that if the state formula,* p *happens to be false at any given point in a run, it is always guaranteed to become true again if the run is continued.* For p' = Global Connectivity, we verify p in a model without any message loss. For p' = Connectivity, we verify p in a model with a limit on message loss. For p' = Disconnect, we verify p in a model with node failure.

The descriptions of each property, p are given below. For the purpose of readability, we left out some technical details. For more details, we recommend the DT-Spin models of FRODO [1] .

$$\mathsf{ConfDiscPurge} \models \mathsf{P1} \wedge \mathsf{P2} \wedge \mathsf{P5}$$

In this property, the number of entities in the system varies according to node failures (e.g. node crashes). We use $|\mathsf{E}|$ to represent the original number of nodes in the system, before a node failure occurs.

For P1, all entities have to agree on the highest ranking node, say c, becoming the single Registry. For P2, all nodes must discover this Registry (and no others). In the verification we check $|\mathsf{RegisteredSD}(\mathsf{c})| = |\mathsf{E}|$, which implies that each node m has registered at c. In this model, the Manager offers exactly one service. The Manager discovers the Registry, and registers. For P5, the Registry purges the service registration of disconnected nodes. Thus, $|\mathsf{RegisteredSD}(\mathsf{c})| = |\mathsf{E}| - 1$. If the failing node is the Registry itself then the Backup c', which is the second highest ranking node, must detect this and take over as the new Registry, resulting in $|\mathsf{RegisteredSD}(\mathsf{c}')| = |\mathsf{E}| - 1$. Thus, for the ConfDiscPurge module, we check the following property (each line is checked separately, in different models)

$\mathsf{p} := \Box\Diamond(|\mathsf{C}| = 1 \wedge \mathsf{highestRank}) \wedge$ $\quad /*\,\mathsf{P1}\,*/$

[1] Frodo Dt-Spin Models, version 1.0. Technical Report TR-CTIT-06-27, published at http://eprints.eemcs.utwente.nl/2715, Enschede, June 2006.

$$\begin{array}{ll} \Box\Diamond(\text{ no failures } \rightarrow |\mathsf{RegisteredSD(c)}| = |\mathsf{E}|) \wedge & /*\,\mathsf{P2}\,*/ \\ \Box\Diamond(\ \mathsf{m}(\neq \mathsf{c})\text{ fails } \rightarrow |\mathsf{RegisteredSD(c)}| = |\mathsf{E}| - 1) \wedge & /*\,\mathsf{P5}\,*/ \\ \Box\Diamond(\ \mathsf{c}\text{ fails } \rightarrow |\mathsf{RegisteredSD}(\mathsf{c}')| = |\mathsf{E}| - 1) & /*\,\mathsf{P5}\,*/ \end{array}$$

Together with some basic properties of the behavior of each of the nodes (e.g. discovering only one Registry), it is sufficient to obtain that P1, P2 and P5 all hold.

$$\mathsf{SrvRegDisc} \models \mathrm{P3} \wedge \mathrm{P4}$$

We consider the situation where a User u is interested in the service that is offered by some Manager m in the system, $\mathsf{Requirement(u)} = \mathsf{OfferedSD(m)}$. We verify the property

$$\mathsf{p} := \Box\Diamond(\mathsf{DiscoveredSD(u)} = \mathsf{OfferedSD(m)})$$

Which gives that services can be found (P4) and also that services are registered (P3) as the User can only discover registered services.

$$\mathsf{ConMain} \models \mathrm{P6} \wedge \mathrm{P7}$$

We consider the situation where the service at m discovered by a User u changes but still satisfies the requirements of the User. To satisfy P6 and P7, the User has to update its discovered services to remain consistent with the Manager, $\mathsf{Uptodate(u, m)}$. This is implied by $\mathsf{DiscoveredSD(u)} = \mathsf{OfferedSD(m)}$. We again check the property

$$\mathsf{p} := \Box\Diamond(\mathsf{DiscoveredSD(u)} = \mathsf{OfferedSD(m)})$$

where the value of $\mathsf{OfferedSD(m)}$ is changed after a waiting period, during the verification.

5.2.3 Verification Results

As shown in Figure 5.2, a service discovery function oscillates between the ideal state, and the non-ideal state, where communication or node failure is the trigger for the transition. For example, a node can successfully discover the Registry, but fail to register when the registration message is lost. Thus the Registry Setup and Configuration Discovery Principles are satisfied, but the Registration Principle is violated. The system must ensure that the Registration function can return to the ideal state, and satisfy the principle.

We find formally verifying a model of a lossy service discovery system a critical step in the design phase. This is because formal verification reveals design errors

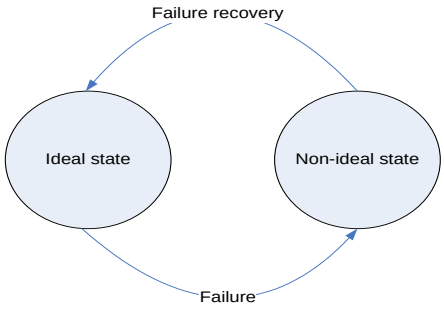


Figure 5.2: *States of a service discovery function. The ideal state for a function has no failure, and the function performs correctly. In the non-ideal state, the function may perform incorrectly. But when the failure ends, the service discovery function should eventually recover from failures and return to the ideal state.*

that cause the Service Discovery Principles to be violated. These errors may not be detected by simulating FRODO, or by testing the prototype.

We first give the analysis on the failure scenarios, before giving the results of our verification.

Table 5.2 lists the discovered failure scenarios that cause the Service Discovery Principles to be violated. For each failure scenario, we give the design solution that solves the failure in our FRODO models. The failure scenarios are relevant for any small-scale, unattended system with auto-configured Registries, with unreliable transmission.

We summarize the verification settings, and the results for each module in Table 5.3. In total, we developed and verified 21 models, out of which, 17 models achieved Exhaustive coverage (100% coverage of all reachable states for a model), while 4 models had to be run under the Supertrace mode (uses bit state hashing [G.J98], with coverage around 98%) because of state space explosion.

Table 5.2: *Summary of failure scenarios that violate the Service Discovery Principles, and the resulting design solutions. The "Ref" column is used in Table 5.3 to refer to the design solutions incoporated in the models.*

Ref	Failure scenario	Scenario description	Principles violated	Design solution
A	**More than the required N Registries are elected.**	Network partitioning causes multiple Registries to be elected. The system must eventually converge to N Registries.	Registry Setup, Configuration Discovery.	The Registry uses *periodic passive discovery* to detect other Registries, and negotiate to satisfy the value of N.
B	**Permanent Registry failure causes the Manager to never be discovered.**	The Registry fails *after* acknowledging the registration, and *before* updating a Backup. When the registration is acknowledged, the Manager keeps renewing its lease, but does not detect that the Registry is no longer available.	Configuration Discovery, Registration, SD Discovery.	The Manager *caches Registry information*, and initiates *unsolicited registration* to unknown Registries.
C	**Temporary Registry failure and cached Registry information causes the Manager to be never discovered.**	The Registry *recovers* from failure, but has *purged* the information on the Manager. The Manager does not attempt to re-register upon receiving the announcement of the Registry, because it has previously cached the Registry information, and assumes its registration is still valid.	Configuration Discovery, Registration, SD Discovery.	The Registry initiates *solicited registration* to detect purged Managers.
D	**Update is unsuccessful, when the Manager or the Registry gives up retransmitting the notification.**	Continuous communication or node failure cause the Manager or the Registry to stop retransmitting the update after the maximum number of retransmission is reached	2-party and 3-party Consistency Maintenance.	The Manager retransmits the update notification until successful (critical update), or retries the update in the future (non-critical update)

Table 5.3: *Verification results. The "success" results are obtained after correcting the failures using the design solutions from Table 5.2.*

Module	**Principle**	**Results**
ConfDiscPurge	Registry Setup, Registry Discovery, Configuration Purge	Success, after implementing design solutions A, B and C
SrvRegDisc	Registration, SD Discovery	Success
ConMain	2-party and 3-party Consistency Maintenance	Success, after implementing design solution D
Verify models during unconnected state	Disconnect check on all principles.	Success. Principles still hold

5.2.4 Discussion

The results of the verification depend on the accuracy of the models, and developing good models is an art [Bri02]. To ensure trustworthiness of the results, our models share the relevant properties with the original system, and maintain the structural similarity between the model and the system. We have made every effort to make our models as accurate as possible. However, we have had to make a number of simplifying assumptions: (1) the nodes provide correct information, (2) message losses are low, (3) the system does not require a fixed time constraint on satisfying the Service Discovery Principles, and (4) the system faces only the selected scenarios that we model. Based on these assumptions, the results show that FRODO guarantees that the functions of service discovery meet their objectives under the condition that nodes are able to communicate eventually. The results of the verification increases confidence in FRODO's capabilities.

In practical implementations, a protocol that violates some of the Service Discovery Principles can address its incompleteness through network and application layers. For example in SLP, the Configuration Update function is not implemented completely (Users are not automatically updated through notification). Service update has to be done through the application layer, where interested Users must periodically poll the Manager to receive service changes. In Jini, detection of disconnected nodes is mostly handled by the TCP layer (retransmission and acknowledgement). This creates a dependency on a reliable communication channel and restricts Jini to certain types of network. Therefore, delegating tasks away from the service discovery layer as done in SLP and Jini creates dependencies on other protocol layers. This leaves the system vulnerable to ambiguous interpretation by application developers on failure response.

The successful verification of FRODO models against the Service Discovery Principles does not necessarily mean that its performance is superior compared to other systems. We need to compare our design choices with other well-known systems like UPnP and Jini, so that we can improve the performance of FRODO

in terms of delay, effectiveness and efficiency, when the system faces failures. The next section evaluates the performance of FRODO.

5.3 Performance Benchmark through Simulations

In this section, we show that our design choices allow FRODO to perform better than well-known systems like UPnP and Jini, in the home context. We focus on the performance of consistency maintenance in the three systems during communication and node failures. The ability to recover from failures during consistency maintenance depends on the overall strength of the service discovery system. Our analysis also gives insight into the effectiveness of the recovery rules presented in Chapter 3.

5.3.1 Consistency Maintenance In Jini, UPnP and FRODO

Consistency maintenance involving three types of communicating entities (Manager, User and Registry) is known as 3-party subscription. Consistency maintenance involving only two types of entities (without the Registry) is known as 2-party subscription.

Jini uses 3-party subscription. The Manager sends an update to the Registry, and receives an acknowledgement. The Registry propagates the update to the subscribed Users. In UPnP, the Manager sends update notifications to the subscribed Users through 2-party subscription. The notification (an invalidation message) indicates only that the service has changed. A User receives the actual update after it requests the change. In both Jini and UPnP, a message is sent only if the reliable transmission using TCP successfully sets up a connection between the sender and the receiver. Messages for setting up the connection and notifying the update are acknowledged and retransmitted, as part of the TCP behavior.

FRODO with 3-party subscription supports resource lean 3D and 3C Managers, while 2-party subscription is used for 300D Managers. The task of maintaining subscriptions for resource-lean Managers is delegated to the Registry, so that the Manager needs only to notify the Registry if its service changes. The Registry notifies the subscribers when the Manager sends an update. In both subscriptions, every update message sent by the Registry and Manager is acknowledged. This is still a smaller overhead compared to that incurred by reliable transmission used by Jini and UPnP, as shown in Table 5.4.

Table 5.4: *Comparison of state of the art consistency maintenance mechanisms and recovery rules. The description of the recovery rules is presented in Chapter 3, Section 3.3*

Consistency maintenance mechanisms and recovery rules	**UPnP**	**Jini**	**FRODO**
Subscription type	2-party subscription	3-party subscription	3-party subscription (3C/3D Manager), 2-party subscription (300D Manager)
Configuration maintenance mechanism	CM1, CM2	CM1, CM2	CM1, CM2
Subscription-recovery rules	SRC1(TCP-dependent), SRN1(TCP-dependent)	SRN1(TCP-dependent), SRC1(TCP-dependent), SRC2	SRN1, SRN2, SRC1, SRC2
Purge-recovery rules	PR4, PR5	PR1, PR2, PR3	3-party subscription: PR1, PR3, PR5 (application dependent). 2-party subscription: PR1, PR4, PR5 (application dependent)
Number of update messages, for N Users, 1 Registry, and 1 Manager when there are no failures	$3N(noRegistry)$	$N+2$. If Users and Managers are registered with y Registries: $y(N+2)$	$N+2$, because there is only one Registry

5.3.2 Recovery Rules in Jini, UPnP and FRODO

As explained in Chapter 3, during short-term failures, entities that fail to update each other have to recover through the still valid subscription process. This is known as *subscription-recovery*. Continuous, long-term failures cause the subscription leases to expire, and entities purge the knowledge of each other. Entities use the Configuration Discovery, Registration and SD Discovery functions to rediscover the purged entities, and regain consistency. This is known as *purge-recovery*.

The type of recovery rule used by a system during purge-recovery depends on whether the architecture of the system is non-Registry or Registry-based. The competence of the systems in performing consistency maintenance rely upon *how the systems actually implement* the recovery rules. For example, as seen from Table 5.4, both UPnP and FRODO with 3-party subscription implement PR5, but Users use different approaches in how they rediscover the Manager; UPnP uses multicast User queries and Manager announcements, while FRODO uses unicast queries to the Registry, before trying multicast queries. We analyze the differences in implementation in Section 5.3.5.

Table 5.4 also shows that FRODO is unique because it supports both 2-party and 3-party subscriptions. FRODO is also the only protocol to support SRN2, where the Manager retries an unsuccessful update when it receives a subscription renewal message from the User. In small scale systems, FRODO with a single Registry is the most efficient protocol, because it propagates the least number of messages to get N Users updated. This is because FRODO is a single Registry architecture, which uses inexpensive UDP, and propagates the updated data, unlike TCP-based UPnP (which uses invalidation), and Jini (which becomes inefficient with redundant Registries).

5.3.3 Performance Metrics

We can benchmark the consistency maintenance performance of state of the art service discovery systems by using the *Update Metrics*, developed by Dabrowski and Mills [Dab02b]. The Update Metrics measure the consistency maintenance performance of service discovery systems against a particular *failure rate*.

Below, we give the general definitions used in the Update Metrics.

Definition 2: We use two types of failure models: *message loss* and *interface failure*. The failure rate, λ ($0 \leq \lambda \leq 1$) is defined as:

1. Message loss rate: number of messages lost compared to the total number of messages propagated in the system during the lifetime of the system, D, or

2. Interface failure rate: proportion of time that a node is unable to communicate during the lifetime of the system, D.

Definition 3: Let X be the number of runs repeated in the experiment, N the number of Users in the system, $C(i)(< D)$ the time when the service changes, and $U(i, j, \lambda)$ the time a User receives the update and reaches consistency, where $j \in 1...N$, and $i \in 1...X$.

We now give the Update Metrics. The results (data points in the graphs) for Update Effectiveness, Update Efficiency and Efficiency Degradation are averaged over $j \in 1..N$ and $i \in 1..X$. Update Responsiveness uses median calculation to eliminate biasing from extreme scenarios where only messages from the Manager or the Registry are lost (outliers), unlike the mean calculation.

1. **Update Responsiveness, $R(\lambda)$.** Measures the ratio of the time left after the update is propagated to a User, before a deadline, D compared to the total time available for the Manager to propagate the update before D.

$$\text{Update Responsiveness, } R(\lambda) \text{ is the median of } 1 - L(i, j, \lambda),$$

taken over $j \in 1..N$ and $i \in 1..X$.

$$\text{where } \textit{Relative change-propagation latency, } L(i, j, \lambda) = \frac{U(i, j, \lambda) - C(i)}{D - C(i)}$$

2. **Update Effectiveness, $F(\lambda)$.** Measures the average probability of success for a User to reach consistency.

$$\text{Update Effectiveness, } F(\lambda) = \frac{\sum_{i=1}^{X} \sum_{j=1}^{N} chg(i, j, \lambda)}{X.N}$$

$$\text{where } chg(i, j, \lambda) = \begin{cases} 1 & : U(i, j, \lambda) < D \\ 0 & : \text{otherwise} \end{cases}$$

3. **Update Efficiency, $E(\lambda)$.** Measures the effort required to maintain consistency. Let m be the minimum number of messages across all systems to propagate a change to the Users. Let $y(i, \lambda)$ be the total number of messages for *all* Users to regain consistency in the system (therefore, this metric only depends on i and λ). The Update Efficiency, $E(\lambda)$ is the ratio of m to $y(i, \lambda)$, averaged over the number of runs.

$$\text{Update Efficiency, } E(\lambda) = \frac{\sum_{i=1}^{X} [m/y(i, \lambda)]}{X}$$

where m is the minimum number of messages across all systems to propagate a change to the Users, and y is the total number of messages propagated in the system during inconsistency.

The Update Efficiency metric fixes the minimum number of messages, m and requires all protocols to base their efficiency measurement against the protocol which is the most efficient at 0% failure rate. This gives the protocols that exhibit the value of m an advantage over other protocols. However, the metric does not reflect how the protocols perform when the failure rate increases. It is possible that a protocol that propagates more messages at 0% failure rate degrades slower than the baseline protocols at higher failure rates. This leads us to consider the degradation of efficiency as a new metric.

4. **Efficiency Degradation, $G(\lambda)$.** We make a simple modification to the Update Efficiency metric by replacing m with a protocol's own minimum number of messages to propagate the update, m'. This metric permits a more accurate evaluation of protocol efficiency because it reflects how heavily the protocol has to propagate messages as failure rate increases, to ensure that all Users achieve consistency.

$$\text{Efficiency Degradation, } G(\lambda) = \frac{\sum_{i=1}^{X} [m'/y(i,\lambda)]}{X}$$

where m' is the system's own minimum number of messages to propagate the update

5.3.4 Modeling Approach

We use *Rapide* [Luc98], an Architectural Description Language and tool suite to build an executable model of FRODO. Rapide is designed to support component-based development of systems by utilizing architecture definitions as the development framework. It offers an event-based execution for distributed, time-sensitive systems.

We benchmark a total of four models: (1) UPnP, (2) Jini with 1 Registry, (3) FRODO with 3-party subscription using 1 300D node as the Registry and (4) FRODO with 2-party subscription, using 8 300D nodes (but still a single Registry system). Our model of FRODO with 2-party subscription contains only 300D nodes, because the nodes have resources similar to the nodes in Jini and UPnP. We reproduce the results of UPnP and Jini models by Dabrowski and Mills.

We now simulate the models with two types of failures: (1) communication failure modeled as *message loss*, and (2) node failure modeled as *interface failure* (receiver and/or transmitter failure). The following steps describe our approach.

Step 1: Modeling FRODO. Since FRODO and Jini are both Registry-based architectures, we build our FRODO model based on the Jini model by Dabrowski and Mills. The class diagrams in Figure 5.3 and 5.4 show the difference in the structure of our FRODO model against the Jini model. The main challenge in modeling FRODO is in developing a framework of behaviors for User and Manager, according to the type of device class. In UPnP and Jini, nodes are homogenous, allowing more straightforward models. In our simulations, we do not include 3C Managers because they behave exactly the same as 3D Managers during consistency maintenance.

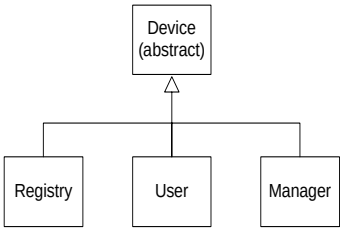


Figure 5.3: *In the Jini model from NIST, a device can instantiate as a User, a Manager, or a Registry. The Registry component is only relevant for Jini).*

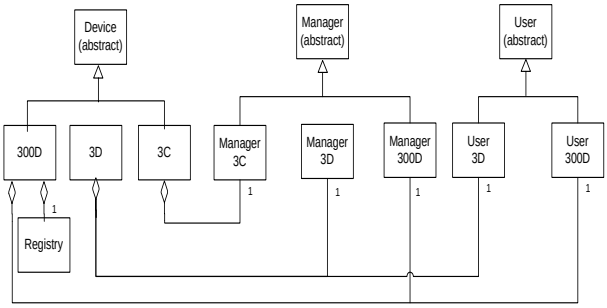


Figure 5.4: *In the FRODO model, a device can instantiate as a 3C, 3D or 300D class. A 300D node has a Registry component which performs Registry election, and which is triggered when it is elected as the Registry. 300D and 3D nodes can instantiate as a User and a Manager, while 3C nodes are only Managers. The User and Manager behaviors are tailored according to the device class limitations.*

Step 2: Constructing the failure response of transmission protocols. All three models use unreliable multicast transmission (UDP). For unicast transmission, FRODO also uses inexpensive UDP, while Jini and UPnP use reliable unicast transmission (TCP). In UDP, when a message is discarded, the source does not learn of the loss. In TCP, a *Remote Exception (REX)* is sent to the service discovery layer

of UPnP and Jini when an acknowledgement is not received after retrying and waiting, as explained further in Table 5.5.

Table 5.5: *Network characteristics. UPnP and Jini rely on notifications from the transport layer to detect transmission failures. FRODO does not rely on lower layers to detect failures. Redundant multicast transmissions also do not occur in FRODO because it does not fit the resource-aware context.*

Network behavior and failure response	**UPnP and Jini**	**FRODO**
Multicast	UDP	UDP
Unicast	TCP	UDP
Transmission delay	0.14s-0.42s	0.14s-0.42s
Unreliable protocol (UDP) response to message loss	No retransmission of lost messages, or acknowledgements. Redundant 6 times transmission for all UDP messages	Selected messages are retransmitted and acknowledged.
Reliable protocol (TCP) to message loss	Connection setup: 4 retransmission attempts with delays 6s, 24s, 24s, 24s, then REX if unsuccessful. Data transfer: retransmit until success, increasing, timeout by 25% on each retry (first time-out is round trip time)	-

Step 3: Constructing the service discovery behavior and recovery rules. In our UPnP scenario, the Manager sends 6 multicast announcement messages every 1800s. In Jini, the Registry sends 6 multicast announcements messages every 120s, while in FRODO, the Registry sends 2 multicast announcements every 1200s. We recommend 1200s for the Registry announcement period in FRODO (longer than in Jini), to avoid unwarrented Registry election that degrades efficiency, and consumes additional resources. This is because 300D nodes monitor the Registry's announcements, and random, short term Registry failure or message loss causes 300D nodes to restart Registry election. We also allow the Registry to announce itself only twice at a time, to conserve resources of both the Registry and receiving nodes. In short, we deliberately put our FRODO simulation at a disadvantage against the Jini and UPnP simulations, to conserve resources.

In Jini and FRODO, when the Registry is purged, the Manager rediscovers the Registry by listening for the Registry announcements. FRODO also requires 3D Managers to announce their presence periodically until the Registry is discovered. 300D Managers multicast announcements to start the Registry election process to setup the Registry. We deliberately model FRODO parameters to reflect resource-awareness by not requiring all messages to be retransmitted and acknowledged (only a selected few). We set the period of the Registry announ-

cements so that it is short enough for the discovery process, but long enough so that severe interface failures at high failure rates do not imbalance the system by continuously restarting the Registry election process.

The registration/advertisement lease period (the period where the service remains valid in the cache of the Registry or the User) is set to 1800s for all three protocols. In UPnP and FRODO with 2-party subscription, the User subscribes to a discovered Manager. The subscription lease (the period where the User is interested in receiving updates) is renewed every 1800s for both systems.

Table 5.6 compares the recovery rules in the models, and shows the differences in implementation. Compared to Jini, FRODO has stronger implementation of the recovery rules. UPnP uses PR5 because it is a non-Registry based architecture, thus it uses multicast service advertisements.

Table 5.6: *Recovery rules, as implemented in the UPnP, Jini and FRODO models. For the message loss experiment, we only use the single Registry topology of Jini. The gray areas indicate stronger implementation of the recovery rule.*

.

Consistency maintenance recovery rules	**UPnP**	**Jini**	**FRODO**
Topology	1 Manager, 5 Users	Two topologies. (a) 1 Registry, 1 Manager, 5 Users, (b) 2 Registries, 1 Manager, 5 Users	Two topologies. (a) 1 300D Registry, 1 3D Manager, 5 3D Users (b) 1 300D Registry, 1 300D Manager, 5 300D Users, 1 300D Backup
SRN1: Retransmissions and acknowledgements	TCP enables SRN1	TCP enables SRN1	Retransmissions and acknowledgements of selected messages
SRN2: Retry on unsuccessful notification	-	-	Manager in 2-party subscription retries update notification when it receives subscription renewals from inconsistent Users
PR1: The Manager re-registers, and the Registry notifies the User	-	Users are notified when the Manager registers in the future.	Users are notified if the Manager is available or registers in the future
PR2: The User queries the rediscovered Registry for service	-	Users query for the service when the Registry is rediscovered	-
PR3: The Registry rediscovers the User, and requests resubscription	-	The Registry responds to an unknown User with an error message that requires the User to rediscover the Registry	The Registry requests the User to resubscribe
PR4: The Manager rediscovers the User, and requests resubscription	The Manager requests purged Users to resubscribe	-	The 300D Manager in 2-party subscription requests purged Users to resubscribe
PR5: Users purges and rediscovers the Manager	Users rediscover the Manager through multicast queries, or by listening for multicast announcements from the Manager	-	3-party subscription: Users purge the subscription when the Registry purges the Manager. Managers are rediscovered by querying the Registry or by sending multicast queries when the Registry is not responding

Step 4: Failure modeling. We use two types of failure models.

- Message loss - There are several reasons for message loss; physical obstacles causing radio signal loss, packet collisions, packets dropped due to buffer overflow, etc. We model these failure scenarios as message loss. Messages are discarded randomly, at a loss rate λ, varying from 0.00 to 0.90, in increments of 0.05. Message loss is activated after 100s (grace period is 100s, so that Users can discover the Manager), it remains in effect until the simulation ends at 5400s (the reason for the simulation duration, $D = 5400s$ is given in Step 5).
- Interface failure - When a transmitter or receiver on a node fails, a node is only able to either send or receive messages. Simultaneous receiver *and* transmitter failure on a node models temporary radio disconnection, or power failure (in this case, the cached data is persistent). For each node, the transmitter and/or receiver are failed randomly, at a failure rate λ, varying from 0.00 to 0.90, in increments of 0.05. Interface failure occurs at a random time, from 100s to 5400s. Once the interface failure is activated, it remains in effect for a portion of the simulation duration (λ x 5400s).

Step 5: Experiment design. We use the application scenarios and parameters used by the UPnP and Jini models from Dabrowski and Mills for fair comparison. The simulation run time, D lasts for 5400s. The run time is based on the UPnP recommended service advertisement period of 1800s. All three systems use 1800s for maintaining a lease for registration and subscription. Thus, using three announcements provides a reasonable opportunity for a system to rediscover the Manager and regain consistency after a failure occurs. Five Users discover the Manager and obtain the SD. This process occurs within the first 100s without interface failure. At a random time between 100s to 2700s, the Manager's service changes, causing the Users to become inconsistent with the Manager. Users are notified of this change through 3-party or 2-party subscription. We repeat the experiment 30 times to obtain the average and median results.

We benchmark two experiments, which we will refer to as:

(A) *Message loss experiment.* This experiment uses message loss from Step 4 as the failure model. Here, we compare the performance of FRODO against UPnP and a single Registry topology of Jini.

(B) *Interface failure experiment.* This experiment uses interface failure from Step 4 as the failure model. Here, we compare the performance of FRODO against UPnP, and two types of Jini topologies (single and double Registry topology). This failure model has more severe consequences, because continuous failure to communicate causes nodes to purge each other faster than in the message loss experiment. Therefore, we compare the performance of FRODO against a more robust Jini model consisting of two Registries.

5.3.5 Results and Discussion

The results we present in this section are based on a detailed analysis of a random selection of 5 to 10 event logs (out of a total of 30 logs) for each simulation, at every failure rate.

A. Message Loss Experiment

We find that FRODO is the most responsive and effective protocol up to 75% loss rate, and has the overall highest efficiency and lowest latency for consistency maintenance. We also find that reliable unicast transmissions and redundant multicast transmissions in Jini and UPnP are not advantageous at loss rates lower than 75%.

To determine whether the performances of the four models have statistical significance, we perform paired, 2 tails t-test, at 95% confidence level (for all combinations of models across loss rates 0% to 90%). Our tests suggest that there are significant differences for (1) Update Effectiveness: between Jini and FRODO with 3-party subscription, and between both FRODO models, (2) Update Responsiveness: between UPnP and FRODO with 3-party subscription, and between both FRODO models, and (3) Efficiency Degradation: between each pair of models, except for UPnP and FRODO with 3-party subscription.

We present our results according to the Update Metrics. For ease of understanding, we refer to loss rates lower than 75% as *low loss rates*, and loss rates higher than 75% as *high loss rates*.

Update Effectiveness. Update Effectiveness (Figure 5.5) is the predominant metric that reflects the recovery ability of service discovery protocols to ensure Users regain consistency. We find that FRODO resolves the lack of aggressive retransmission policy by implementing stronger failure recovery rules. The ability of FRODO with 2-party subscription to resend the update at a later point of time (SRN2) proves advantageous when all previous update notifications are lost. This is shown in Figure 5.5, where FRODO with 2-party subscription maintains the highest effectiveness at low loss rates. UPnP is the least effective at low loss rates because message loss during invalidation (to indicate that an update has occurred) causes some Users in UPnP never to regain consistency. Unlike FRODO with 2-party subscription, UPnP does not cache inconsistent Users, nor does it retry the update. At low loss rates, FRODO with 3-party subscription has higher effectiveness than Jini because compared to FRODO, the User in Jini has to send an additional message before it queries; the User has to register for notification first.

At high loss rates, UPnP emerges as the most effective protocol (Figure 5.5(a)) because UPnP implements PR5, where Users receive the periodic multicast announcement from the Manager. Users eventually detect a previously purged Manager, and query its SD. Also, at high loss rates, we find that the aggressive re-

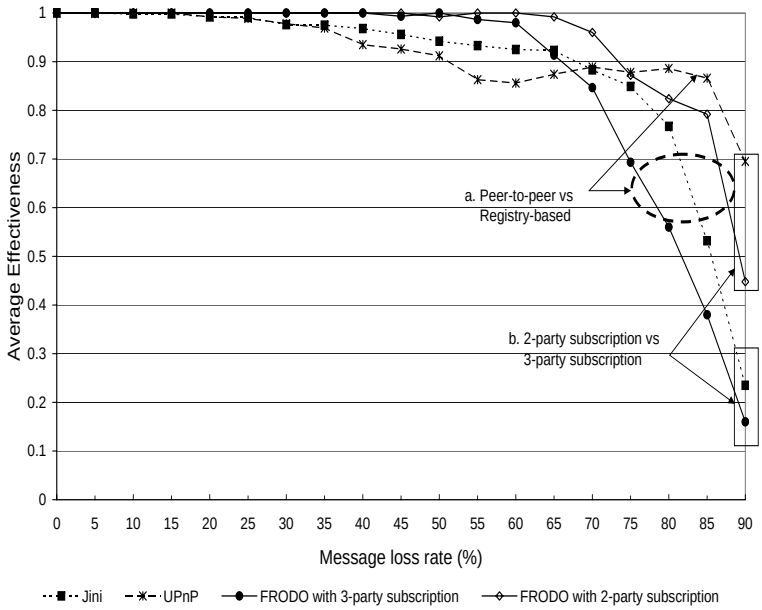


Figure 5.5: *FRODO with 2-party subscription has the highest effectiveness at low loss rates because the Manager retries the update notification at a later point of time. (a) UPnP's non-Registry architecture is the most effective at high loss rates. (b) 2-party subscription is more responsive then 3-party subscription at high loss rates.*

transmission policy in Jini yields better performance than FRODO with 3-party subscription. In general, protocols that implement 2-party subscription give better performance at high loss rates, because they are less vulnerable to single point of failure issues (Figure 5.5(b)).

Update Responsiveness. Update Responsiveness is shown in Figure 5.6. The results of Update Responsiveness are dependant on the type of subscription and transport protocol. 2-party subscription generally has faster responsiveness at high loss rates than 3-party subscription. 3-party subscription is more vulnerable to a single point of failure issue on the Registry, therefore causing higher delay, and lower probability of success for Users to regain consistency. Therefore, FRODO with 2-party subscription combined with faster UDP transmission allows FRODO to maintain the highest responsiveness at low loss rates. At high loss rates, UPnP has the best responsiveness, because its inherent non-Registry architecture uses PR5.

As in Update Effectiveness, UDP-based FRODO with 3-party subscription has faster responsiveness than Jini at high loss rates. However, the aggressive retransmission policy in Jini is only beneficial until 80% loss rate. With weaker

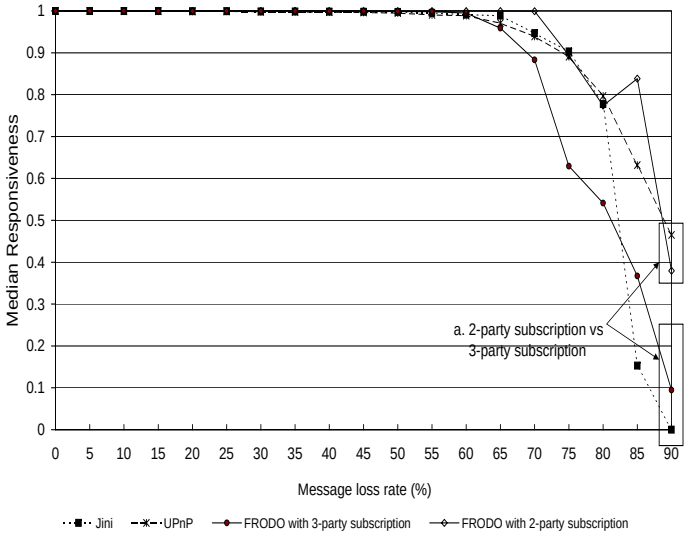


Figure 5.6: *The combination of fast UDP transmission, direct User and Manager communication and additional failure recovery in FRODO with 2-party subscription gives the highest responsive at low loss rates. (a) At high loss rates, architectures with 2-party subscription are more responsive than architectures with 3-party subscription.*

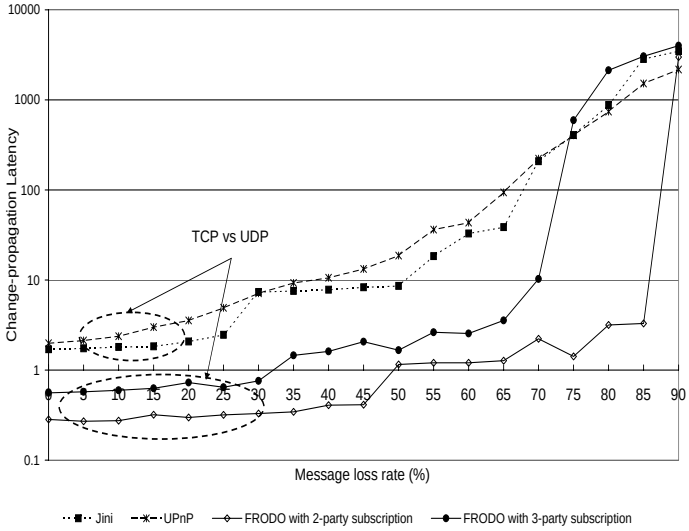


Figure 5.7: *Unreliable transmission allows fast update propagation at high loss rates for FRODO with 3-party subscription. Along with fast transmission, the combination of 2-party subscription and better failure recovery in FRODO gives the lowest delay at all loss rates. TCP in UPnP and Jini causes additional latency.*

failure recovery rules, the performance of Jini deteriorates sharply after 80% loss rate.

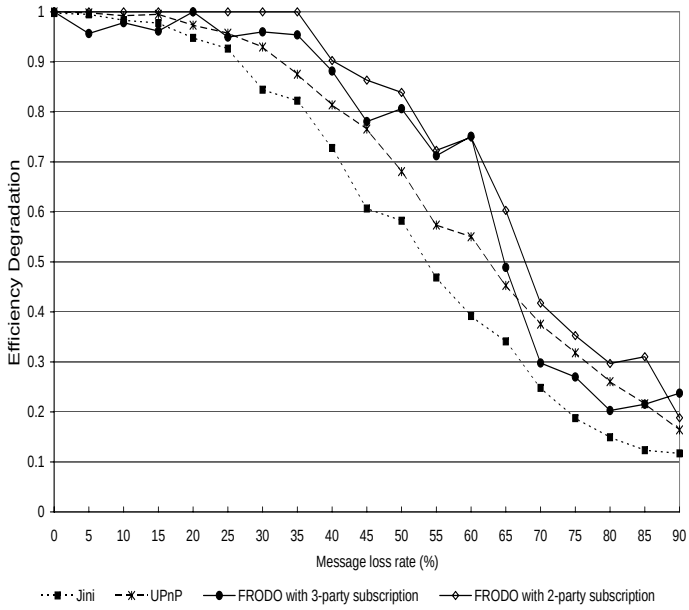


Figure 5.8: *FRODO and Jini propagate only 7 messages each, while UPnP propagates 15 messages at 0% loss rate. The efficiency of FRODO degrades the slowest, due to faster and more efficient failure recovery. The efficiency of Jini degrades the fastest because of weaker failure recovery. TCP messages are not included in the calculation for Jini and UPnP.*

We further investigate the impact of TCP and UDP by comparing the *median absolute latency* for Users in each protocol to regain consistency, for each failure rate. We remove the denominator from the relative change-propagation, L_λ, so that the absolute change-propagation latency for a message loss rate is:

$$\text{Absolute change-propagation latency, } L'_\lambda = t_{U(ij)} - t_{C(i)}$$

The result of this investigation is shown in Figure 5.7. TCP causes longer change-propagation latency in UPnP and Jini, compared to FRODO, as shown in Figure 5.7. TCP uses additional time for setting up the connection before actually sending the update message. FRODO does not require connection setup. Furthermore, only a selected few messages are retransmitted in FRODO. FRODO with 2-party subscription has the shortest delay because it generally does not depend on the Registry for consistency maintenance (unlike FRODO with 3-party subscription and Jini).

Table 5.7: *Comparison of consistency maintenance performance for UPnP, Jini and FRODO at 20% message loss rate. The shaded values show the protocol that has the best performance. Multiple shades on a single row indicate protocols that are not significantly different with FRODO at 95% confidence level for this loss rate. FRODO has the best performance here, while having the best overall performance for loss rates below 75%*

.

Update Metrics	UPnP	Jini	FRODO with 3-party subscription	FRODO with 2-party subscription
Update Responsiveness	0.999	0.999	1.000	1.000
Absolute change-propagation latency (s)	3.562	2.075	0.726	0.297
Update Effectiveness	0.992	0.992	1.000	1.000
Efficiency Degradation	0.973	0.948	1.000	1.000

Efficiency Degradation. FRODO has the best overall Efficiency Degradation. FRODO with 2-party subscription propagates the least number of messages to ensure Users regain consistency because communication is directly between the User and the Manager. When there is no message loss, UPnP has the worst efficiency (propagates a total of 15 messages, because of the invalidation step), while FRODO and Jini have the best efficiency (each propagates 7 messages). However, as the message loss rate increases, Jini shows the worst degradation in efficiency. Jini uses more effort to make Users to regain consistency, because compared to the other systems. Jini has weaker failure recovery ability, as explained in Update Effectiveness.

The UPnP and Jini models from Dabrowski and Mills do not take into account the messages used by the TCP layer. Therefore, the true results for Efficiency Degradation for Jini and UPnP is even lower than shown in Figure 5.8.

Discussion. FRODO has the best consistency maintenance performance until 75% loss rate. In Table 5.7, we highlight the performance of the three protocols at 20% loss rate because of the following: the worst-case scenario for message loss in IEEE 802.11b, ad-hoc mode wireless links with no error control is around 40% for the office environment, measured by Hoene et al [Hoe03] for 2 nodes at a critical distance of 18m. In base station mode, with error control, the measurements show that the typical loss rate does not exceed 5%. Since there is no agreement in the research community on the message loss rates in a typical home environment, we use 20% (about halfway between 5% to 40% range) as the reasonable loss rate to compare the performance of the three protocols in Table 5.7.

In a wired network, typical loss rates will be even lower.

The absolute change-propagation latency for FRODO is significantly lower compared to the other models. As explained earlier in Section 5.3.5, TCP in Jini and UPnP loses some time setting up the connection, and retransmitting the connection requests and the update messages. UPnP uses invalidation messages, while Jini requires more messages before the Registry can notify the Users when the purged Manager re-registers. Below 75% loss rate, FRODO has an advantage over TCP-based protocols because FRODO only incurs one-time transmission delay to send an update, while retransmitting only selected messages.

FRODO implements dynamic Registry election, adds robustness by removing the dependency on the recovery abilities of underlying protocol stacks, and tailors service discovery tasks according to device classes. This experiment shows that our design satisfies the performance required for wired and wireless (including ad-hoc) home networks, when compared to the less complex designs of Jini (manually deployed Registry) and UPnP (non-Registry system). The performance of FRODO starts to deteriorate only after 70% loss rate (well above failure rates for the home environment), which is expected because of the low number of message retransmissions. Although the performance can be increased by adding redundant transmissions for both unicast and multicast messages, we chose not to do so to satisfy the resource-aware context of FRODO.

B. Interface Failure Experiment

In this experiment, we: (1) compare the consistency maintenance performance of FRODO, UPnP and Jini, and (2) analyze the effectiveness of the recovery rules, as presented in Table 5.6. We can analyze the effectiveness of the recovery rules because interface failure makes a node unable to communicate properly for a continuous period of time, unlike intermittently at a random instance of time, as done in the message loss experiment. Therefore, we can decipher which recovery rule is predominant based on the length of the failure period. In this experiment, we find that at failure rates below 35% (we call these rates *low failure rates*), subscription remains valid because nodes recover from failures before the leases expire. At failure rates above 35% (we call these rates *high failure rates*), the nodes purge information of each other. Therefore, at low failure rates, subscription-recovery rules are used, and at high failure rates, purge-recovery rules are activated.

FRODO with 2-party subscription has the highest effectiveness at low failure rates because it implements SRN2 (the Manager retries the update propagation when the User renews an inactive subscription). UPnP has the highest effectiveness at high failure rates, because of PR5 (multicast service advertisments and queries). Both FRODO subscription models have the highest responsiveness and efficiency, across all failure rates.

As done in the message loss experiment, we determine whether the performances of the five models have statistical significance by performing paired, 2 tails t-test, at 95% confidence level (for all combinations of models across failure rates

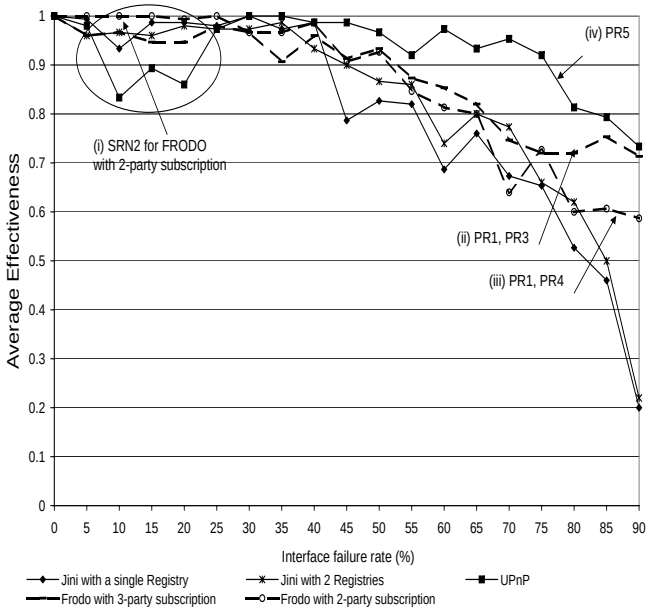


Figure 5.9: *(i) SRN2 is most effective because the Manager resends the update notification when the lease is renewed. (ii) Efficient PR1 in FRODO allows the Registry to update the Users when the Manager or the Registry recovers from failures. PR3 and PR4 in (ii) and (iii) allows Users to resubscribe to the Registry and the Manager respectively. (iv) PR5 is most effective at high failure rates where Users rediscover the Manager through the Manager's periodic announcements.*

0% to 90%). Our tests suggest that there are significant differences for (1) Update Effectiveness: between all combinations of models, except between both FRODO models, and between FRODO (both models) and Jini with 2 Registries, and (2) Update Responsiveness: between all combinations of models, except between both Jini models, and between FRODO (both models) and UPnP. For Efficiency Degradation, we find no significant difference in the performance of all five models. Therefore, we do not discuss in detail, or show the figure for this metric. Suffice to say, message retransmission is not an effective recovery method during interface failure, because the retransmission limit is reached before the failure ends.

Update Effectiveness. Update Effectiveness is the predominant metric that reflects the impact of recovery rules. We find that at low failure rates, the prominent recovery rule is SRN2, as implemented in FRODO with 2-party subscription (Figure 5.9(i)). At higher failure rates, PR5, as implemented in UPnP (Figure 5.9(iv)) is the most effective. In FRODO with 2-party subscription, Users rediscover the Manager via the Registry, as opposed to direct peer-to-peer commu-

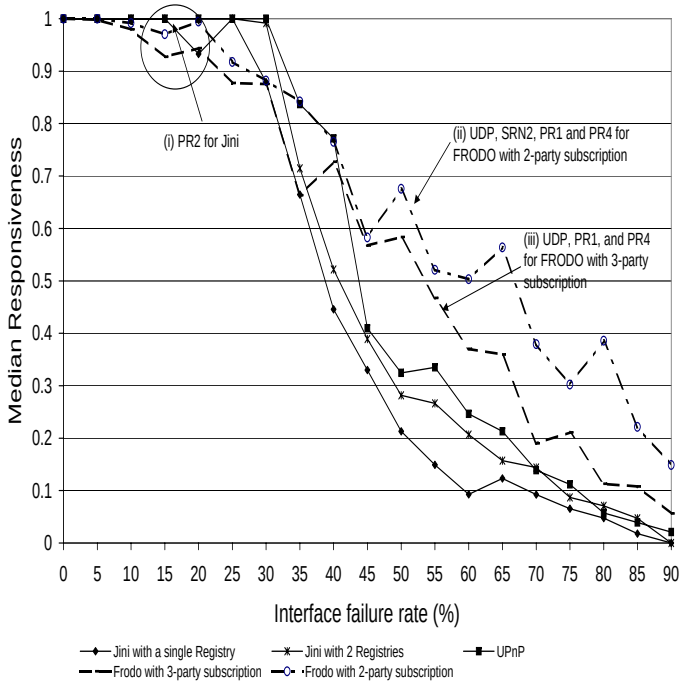


Figure 5.10: *(i) PR2 allows Users in Jini to regain consistency by querying the Registry. FRODO uses SRN2, which depends on the subscription lease period to regain consistency. (ii) UDP transmits messages faster than TCP. PR1 enables the Registry to update the Users when the Registry or the Manager recovers from failures. PR3 enables purged Users to resubscribe with the Registry. (iii) 2-party subscription, UDP, SRN2, PR1 and PR4 allow Users in FRODO to be the most responsive.*

nication in UPnP, causing the PR5 implementation in UPnP to be more effective than in FRODO. PR1 (the Registry notifies the User when the Manager registers) as implemented in FRODO (Figure 5.9(ii)) yields the next highest effectiveness.

Update Responsiveness. The Update Responsiveness metric as shown in Figure 5.10 reveals that FRODO with 2-party subscription incurs the overall shortest delay for Users to regain consistency, due to a combination of direct, peer-to-peer communication between the User and the Manager, fast UDP transmission, a low number of messages during consistency maintenance and the use of SRN2 and PR1 recovery rules.

Discussion. In this section, we elaborate on the results above by decomposing our analysis according to the type of recovery rule.

SRN1: The logs show that SRN1 has no apparent positive impact during interface failure because nodes typically fail longer than the total period for update retransmissions. SRN1 is more useful during high message loss rates, as shown in the message loss experiment.

SRN2: The impact of SRN2 is apparent especially at low failure rates because Users recover from failures quickly, before a subscription is purged. An example of a scenario without SRN2 is given below. The example shows the simulation result of UPnP at 15% failure rate. Tx and Rx mean transmitter and receiver, and the numbers represent time, in seconds. The service changes at $\texttt{t} = 2507s$, but the Manager fails to update the User which has both interfaces down from $\texttt{t} = 2023s$ until $\texttt{t} = 2833s$. The update notification fails, and the User never regains consistency! This is shown in the example below, when the User only "regains consistency" at the end of the simulation run, at $\texttt{t} = 5400s$. This is a failure to satisfy the 2-party Consistency Maintenance Principle.

```
Failure Rate: 15%

Manager Tx down at 381s, up at 1191s
User Tx and Rx down 2023s, up at 2833s

UPnP: User lost consistency at 2507s, regained consistency at 5400s
```

SRN2 has the highest effectiveness at low failure rates (Figure 5.9(i)), while PR5 has the highest effectiveness at high failure rates (Figure 5.9(iv)).

PR1: This rule is implemented differently in Jini and FRODO. In Jini, the Registry notifies interested Users of a new service registration *only* if the Manager registers *after* the User. If the Manager is already registered before the User discovers the Registry, the Registry does not notify the User. Therefore, service notification in Jini is only for *future* registrations. This anomaly is reported by Dabrowski and Mills [Dab01]. Jini overcomes this problem by forcing Users to always send queries after the User requests for service notification from the Registry, so that existing registered services can also be retrieved (PR2). Service notification in FRODO is more efficient since the Registry notifies interested Users of existing service registrations. A control experiment with and without PR1, shown in Figure 5.11 demonstrates the impact of PR1 on the Update Effectiveness of both FRODO systems.

In addition to a more efficient Registry notification, Users in FRODO discover the Registry not only by listening for Registry announcements (as used in Jini), but also by announcing their presence through multicast. This allows faster discovery of the Registry. This recovery rule helps FRODO to generally have a higher responsiveness, effectiveness and efficiency than Jini.

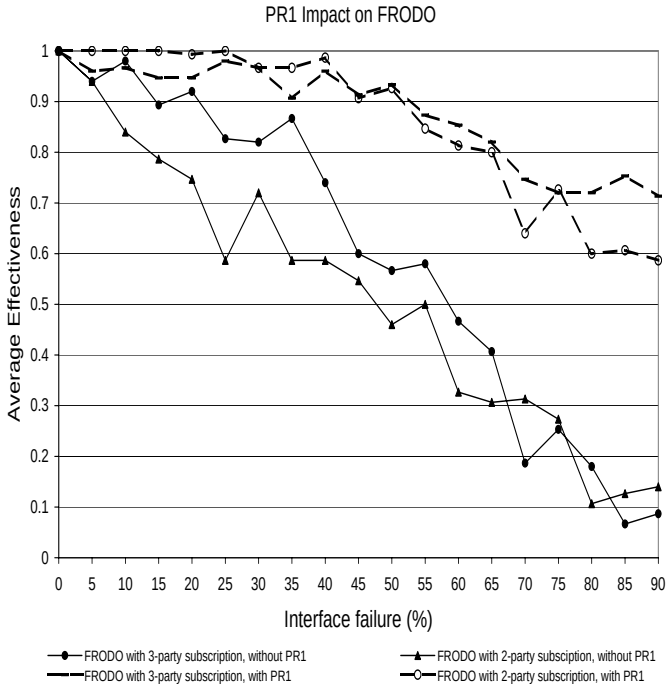


Figure 5.11: *Impact of PR1 recovery rule on the Update Effectiveness of FRODO with 2-party and 3-party subscriptions*

PR2: In Jini, PR2 is implemented together with PR1. As mentioned earlier, Jini requires Users to always query the rediscovered Registry to retrieve the service. Therefore, PR2 depends on whether the Manager was purged and re-registered before the User discovers the Registry. We see that such an implementation of PR2 benefits the responsiveness of Jini at low failure rates where Users regain consistency faster than FRODO, as shown in Figure 5.6(i).

PR3: This rule benefits the responsiveness and effectiveness of FRODO with 3-party subscription, as shown in Figures 5.9(ii) and 5.10(ii). The Registry in FRODO explicitly requests purged Users to resubscribe. The response to the resubscription is the updated SD. PR3 in Jini is implemented such that purged Users are simply returned with an error message from the Registry. The Users then redo Registry discovery, service notification request (PR1), and service query (PR2). Therefore, PR3 in FRODO with 3-party subscription allows faster and more effective consistency maintenance.

PR4: This rule benefits both 2-party subscriptions in UPnP and FRODO, where the Manager requests purged Users to resubscribe so that Users can obtain the

updated SD. The recovery is beneficial for the responsiveness and effectiveness of both systems, which remain high above Jini at failure rates above 40%.

PR5: This rule gives the highest effectiveness, as shown in Figure 5.9(iv) and Table 5.7. Users in UPnP listen to the periodic announcements of the Managers to rediscover the Manager. The Users have a high probability of regaining consistency because they can get updated when the Manager recovers from failures and announces its presence. FRODO with 3-party subscription employs a weaker recovery with PR5, where Users depend on the Registry to purge the Manager, and only then perform unicast or multicast queries to rediscover the service.

Table 5.8: *Average metrics results across interface failure rates from 0% to 90%. The higher the value for each metric, the better the performance. The shaded protocols offer the best performance for each metric.*

Update Metrics	**UPnP**	**Jini with 1 Registry**	**Jini with 2 Registries**	**FRODO with 3-party subscription**	**FRODO with 2-party subscription**
Update Responsiveness	0.553	0.474	0.476	0.580	0.666
Update Effectiveness	0.922	0.802	0.825	0.878	0.861
Efficiency Degradation	0.385	0.311	0.361	0.428	0.429

Table 5.8 shows that although FRODO is a single Registry architecture with unreliable transmissions, FRODO has the highest responsiveness, with the least degradation in efficiency compared to Jini (even Jini with two Registries) and UPnP, while maintaining a high degree of effectiveness.

5.3.6 Investigating the Impact of the Backup in FRODO

In FRODO, the Backup replaces the Registry when the Registry fails to respond to requests (primary-based recovery mechanism). The Backup is updated by the Registry when there is a change of the configuration information (service registration, notification request by the User for a particular service, subscription information, etc.).

In an experiment similar to the interface failure expirement, we change the SD of the Manager, and fail *both the transmitter and the receiver on the Registry, simultaneously* (according to increasing failure rate), at $\mathtt{t} = 100s$. The rest of the parameters remains the same as in the interface failure experiment. We only focus on Registry failure, to show that the Backup improves the consistency maintenance performance of FRODO.

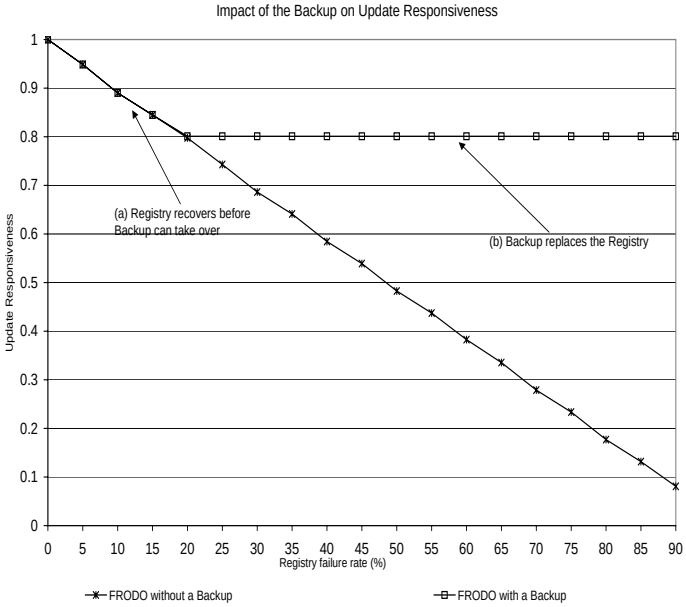


Figure 5.12: *Impact of Backup on Update Responsiveness*

Our results show that when the Manager fails to update the Registry, it purges the Registry information. When the Backup replaces the Registry, the Backup announces its presence, and the Manager re-registers its updated SD with the Backup. The Backup continues the 3-party subscription without requiring additional steps by the Manager and the User. In this experiment, the polling period between the Backup and the Registry is set to 120s.

Figures 5.12 and 5.13 show the Update Responsiveness and Update Efficiency of system with a Backup. The impact of the Backup on Update Responsiveness is only apparent after 15% failure rate, as shown in Figure 5.12(a) because the Registry recovers before the Backup detects Registry failure, and before the subscription expire. After 15%, we find that the Backup successfully replaces the Registry and continues the update notification seamlessly. The Update Responsiveness remains high at 0.8. The cost of Backup is lower Update Efficiency, which drops from 1.0 to 0.75 when there is no failure. However, when there is Registry failure, the cost of Backup on Update Efficiency is not significant, as shown in Figure 5.13. Since we allow the Registry to recover before the simulation ends, the Update Effectiveness, with and without the Backup is always 1.0, across all failure rates.

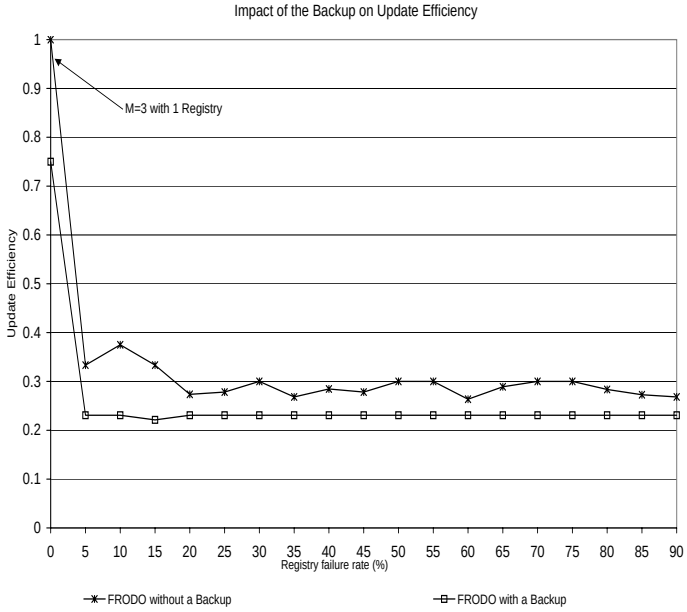


Figure 5.13: *Impact of Backup on Update Efficiency*

5.4 Implementation of FRODO

In this section, we describe our prototype of the FRODO system. We validate the results of the 300D simulation (without the Backup), described in Section 5.3.6, with the performance of the FRODO prototype. We also compare both results of FRODO with the performance of the simulated Jini model, and its Java implementation. The validation increases confidence in our simulation results presented in the previous sections.

The FRODO prototype is implemented in C (the compiler version is gcc 4.0.3). Our development platform is Linux (any Linux distribution that has Posix APIs). FRODO is also available for the Windows platform, and offers Java wrappers that act as interfaces to the FRODO executables to support Java-based applications. The current implementation of FRODO works for both Ethernet and WLAN, for a PC-based environment.

Table 5.9 summarizes the memory usage of the entities in our FRODO prototype. We implemented 300D and 3D device classes. The work on the 3C device class is still ongoing, but we can estimate the minimum memory size consumed by the 3C device class by deducting the irrelevant functions (e.g. periodic active discovery) from the 3D Manager. The memory requirement for FRODO entities are substantially smaller than for Jini entities. A Jini device based on Java 2

Micro Edition Connected Device Configuration (CDC) [Mic05], with RMI requires a minimum of 2.5 MB of ROM and 1 MB of RAM, plus TCP/IP network connectivity [Kam00]. These are resources only found in set-top boxes, network printers, network storage servers, etc.

Table 5.9: *Memory measurement for the prototype entities. We estimate the code size for the 3C Manager by removing irrelevant functions from the 3D Manager. The stack size for a 300D entity is ±3kb, while a 3D entity uses ±2.5kb*

Device class	**Static program size**	**Runtime memory**
300D	37.3kB	5kb
3D	25kb	3kb
3C	10kB (estimate)	-

Our validation experiment uses the same scenario presented in Section 5.3.6, for the single Registry topology of FRODO. The Registry is failed at the time when the Manager has changes to its SD (100s after the Manager initializes), and has to notify the Registry. The purpose of this scenario is to compare the Update Responsiveness of FRODO and Jini when the Registry recovers in both systems, and eventually updates the User. We compare the results of our Jini and FRODO implementations against the results of the respective simulations, to increase confidence in the results we present in the message loss and interface failure experiments. We fail the transmitter and the receiver on the Registry simultaneously, and allow the recovery to take place based on increasing failure rates (0% to 80%, with 20% increment). The topology for both systems contain one Registry, a Manager and a User. For each entity, we use Pentium II, 350MHz, Pentium II 400MHz and Pentium Pro 180MHz machines respectively, connected via a 100Mbps switch, so that the network is isolated from unwanted traffic.

Figure 5.14 shows the Update Responsiveness of the simulated FRODO and Jini models, and of the respective implementations. The results from the implementation corroborate the results from the simulations. We configure the processing speed and network latency in the simulations based on the implementation parameters. If these parameters are not accurate, the results from the simulations differ from the experimental results.

We also measure the average number of messages (taken over 30 runs) received and transmitted by each type of entity in Jini and FRODO. The results show that on average, Jini propagates *more than twice* the number of messages propagated in FRODO, mainly due to TCP retransmissions and acknowledgements.

The Update Effectiveness for all models and implementation is 1.0 because the Registry recovers eventually.

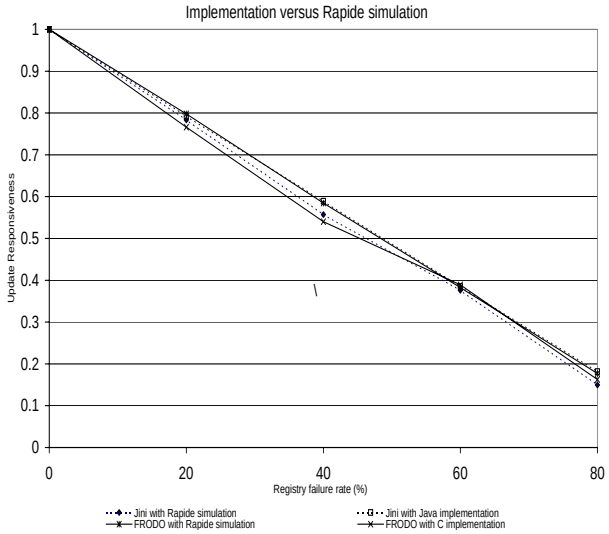


Figure 5.14: *Validation of simulation against real life implementation for a single Registry topology of FRODO and Jini.*

5.5 Discussion and Conclusion

In this chapter, we have shown how we satisfy the Research Question from Chapter 1, which requires an effective and efficient solution to service discovery at home. FRODO is effective because it is reliable against communication and node failures, while providing guarantees on functional correctness. FRODO is also efficient because it is lightweight, and consumes less bandwidth than its competitors.

Our approach of looking at the artifacts of our design in different ways improves our understanding of service discovery. We analyze and evaluate the FRODO system using different techniques; model-checking using DT-Spin, Rapide-based simulations and benchmarks (against models of Jini and UPnP), and testing and measuring the C-based prototype. We now discuss the advantages and the disadvantages of the approach.

During the modeling and analysis of FRODO, we obtain an in-depth understanding to the complexity and the behavioral properties of a robust service discovery system. This understanding inspired the formulation of the Service Discovery Principles and recovery rules in Chapter 3.

Most of the design errors are captured in the design phase, during simulation and model-checking. As shown in Table 5.2, we discovered loopholes in the protocol that would have been time-consuming and more difficult to rectify if the system had been deployed as a full-fledged implementation.

By comparing the performance of our Rapide-based model of FRODO, against the models of Jini and UPnP early in the design phase, we identify and incorporate best practices of the competitor systems (2-party subscription and multicast querying in FRODO are similar to UPnP, while 3-party subscription, leasing and service notification are similar to Jini). As a result, when the best practices of Jini and UPnP are combined with our own innovations (Registry election, primary-based recovery protocols, and device classification), FRODO gives overall better performance, especially at failure rates expected in the home environment.

By validating the results of our prototype performance against the results of the Rapide-based simulation, we gain the following benefits: (1) we check the integrity of our simulation results, and (2) we capture sequences of events in the implementation that deviate from the design specification (detected when the prototype performance does not corroborate the simulation results). Therefore, validation increases confidence on the presented results and the prototype itself. The prototype is a resource-lean realization of the FRODO protocol which allows us to estimate how much memory is needed in a full implementation.

The main challenge to analyzing FRODO using different approaches is to maintain the consistency of the design across different models (from different tools). We have made every effort to maintain the consistency but we have no way of formally establishing this. To achieve this, a common base specification would be needed for which specialized specifications could be generated. This is beyond the scope of the thesis.

Chapter 6

Conclusion

The important thing in science is not so much to obtain new facts,
as to discover new ways of thinking about them.
Sir William Bragg

We now summarize the contributions in this thesis, in relation to the Research Question described in Section 1.2. We also highlight areas for improvement in FRODO and future research directions in service discovery.

6.1 Contributions

In Chapter 1, we identify the following Research Question:

Research Question: *How to enable a variety of home appliances to discover each other's services effectively and efficiently?*

To address the Research Question, we built a small-scale, unattended service discovery system that enables devices to self-configure by discovering each other's services. Service discovery is necessary because there is no system administration in the home.

We focus on the 3Rs from Section 1.2 for building a service discovery system for the home; *R*eliability, *R*esource-constraints and hete*R*ogeneity. By addressing these challenges, our system satisfies the Research Question in the sense that we show how a system can be built. This is thus a kind of proof by existence. We do not imply that FRODO is the only system possible, or even that FRODO is the best system possible, However, we have been able to demonstrate qualitatively and quantitatively that our system is better than the state of the art. We incorporate reliability (and self-healing) so that our system is *effective* in the face of communication and node failures. We achieve this by implementing various

fault-tolerant measures. We also make the system *efficient* in terms of resource-consumption. The system is resource-aware, allowing resource-lean nodes to offer their services. A lightweight service discovery system also should not substantially increase cost of devices (so that the price of home appliances do not prevent home owners to adopt new technologies). Furthermore, we ensure that the system supports heterogeneous home appliances by abstracting away the underlying protocol stacks. Therefore, our protocol is portable over heterogeneous devices and networks, and it will work on a variety of appliances.

Before we design our service discovery system, we systematically analyze the fundamentals of service discovery in Chapter 2; the different architectures, the functionalities of service discovery, the underlying distributed system models for these functionalities, and the operational aspects of designing a service discovery system (system size, lossy environment, resource-constraints, system heterogeneity and security). We analyze the general design space, which leads to identifying appropriate design decisions on the type of architecture and service discovery functions that our system should incorporate.

We then define the requirements for service discovery in the home environment in Chapter 4; *low cost of devices, robustness, portable design* and *security* (we do not include security measures in our design, but delegate it to the application layer). We summarize in Table 6.1, the innovations in our service discovery system, FRODO from Chapter 4.

Table 6.1: *Summary of design solutions and system properties of FRODO (Chapter 4).*

3Rs for a Pervasive Home System	System Properties	Design Solutions
Reliability	Self-configuration	Registry election, so that a single Registry is elected and maintained.
	Self-healing	Primary-based recovery protocols such as a Backup for the Registry, and Registry monitoring. Robustness against various communication and node failures.
Resource-constraints	Resource-aware	Classification of devices based on resource-constraints. Service discovery tasks are partitioned across device classes. The roles of Registry and Backup are transferred to more powerful devices.
Heterogeneous devices and networks	Portable design	The design abstracts away the underlying protocol stacks.

We adopt the "designing for reliability" culture, as proposed by Edwards and Grinters in their Seven Challenges for the ubiquitous home [Edw01]. We do this by implementing recovery behaviors for various failure scenarios, as shown in Chapter 4. We also subject the design to model-checking and simulations early in the design phase, as described in Chapter 5. We evaluate and strengthen our

system against communication and node failures .

To our knowledge, no other service discovery design has been analyzed in such detail; high level design with diagrammatic notations, model-checking, simulations, and prototyping. As a result of our in-depth analysis, we produce the following results:

- Formulation of the Service Discovery Principles and the necessary recovery rules (Chapter 3). The principles and the recovery rules result from our in-depth understanding on the complex behavior of a robust service discovery system, in the face of node, and interface failures.
- FRODO is the first service discovery system that provides guarantees on its behavior (Chapter 5). FRODO has a stronger claim on functional correctness compared to other service discovery systems because we formally verify models of various parts of FRODO against the Service Discovery Principles. Our verifications represent a significant effort of approximately one year, thus providing confidence in the correctness of the design. However, more extensive verification would increase the confidence further.
- FRODO has good performance, and in circumstances appropriate to the home environment FRODO outperforms competitor systems like Jini and UPnP (Chapter 5). We obtain this performance because we combine best practices from Jini and UPnP with our own innovations.
- Our prototype in C is a resource-lean realization of the FRODO system. We show that even though FRODO implements various recovery measures, it has low resource consumption. We also successfully validate a selected set of Rapide-based simulation results with our prototype, which further increases confidence in our design.

The major contributions in this thesis; the FRODO service discovery system, the seven Service Discovery Principles and the failure recovery rules are significant towards building a small-scale, autonomous service discovery system.

6.2 Future Work and Reflections

FRODO does not require attendance of a system administrator. It is also resource-aware and lightweight, allowing resource-lean nodes to take part in service discovery. Furthermore, FRODO gives performance equal to or better than Jini and UPnP (at failure rates appropriate for the home environment), while giving guarantees on functional correctness. However, there are still areas for improvement, some of which are listed below.

- Security features need to be integrated into FRODO before it can be deployed successfully. Further investigation on the impact of malicious behavior, especially during Registry election needs to be undertaken.

- It is difficult to estimate the number of services that FRODO can support because we do not have data on the limitations and capabilities of 300D nodes that are available in an average home environment. Therefore, further analysis needs to be done to understand the limitations of FRODO in a typical, real-life home environment.

- As time progresses in the system, the available resources of 300D nodes may change (due to lack of memory, depletion or recharging energy, etc). Therefore, it should be possible to upgrade or downgrade the device class automatically, based on available resources.

- The impact of mobility on FRODO needs to be investigated. People do not walk around with TV sets and fridges, but smaller devices are moved around the home, such as the remote control. Therefore mobility may effect the available period of a service. One way to improve service discovery in a mobile system is to automatically adjust the lease, poll and announcement periods of the mobile service. Users can also adjust to a constantly mobile service, by polling the Manager directly, upon discovering the service. The User can then disregard the service and rediscover another when the service becomes unavailable too often.

Beyond the home context, there are still several interesting directions in which future research on service discovery can be taken.

- The semantics of device, service and attribute names still require much attention, to improve the context of a discovered service. For a truly unattended system deployment, different service discovery systems should adhere to a single, standardized method for describing services. This is important to ensure that applications that rely on discovered services can make the correct inference on the usage of the service.

- Service discovery systems should ensure that a service is discovered and accessed by authorized entities only. However, authorization should be dynamically allocated and revoked, as time progresses, and the requirement or capability of the entity changes.

- For a service discovery system in the pervasive environment to mature (as DNS has done in the Internet), applications that use service discovery need to be actively developed and promoted. One major hindrance to achieving this objective is the lack of agreement by manufacturers of devices and applications on a standard service discovery platform. A service discovery system that unifies well-known service discovery protocols is a step towards this objective.

- Existing service discovery architectures for wide-area networks focus more on scalability issues (such as bandwidth efficiency, and supporting a large

number of nodes). More work has to be done to produce a large-scale service discovery architecture, which is also robust against failures. A scalable and robust architecture is especially important in mobile ad-hoc networks, because nodes are easily moved, wireless connectivity is uncertain, bandwidth availability is limited, and energy is constrained. Scalable service discovery designs must be evaluated against various failure scenarios.

We conclude by stressing that a fast, reliable and yet lightweight service discovery system is the medium for propelling the power of computing beyond the realm of personal computers, such that information and services are accessible anywhere, and anytime.

List of Figures

List of Tables

Bibliography

[Avi04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. In *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11–33. IEEE Computer Society Press, Los Alamitos, California, Jan-Mar 2004.

[AW99] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 186–201. ACM Press, December 1999.

[Bet00] C. Bettstetter and C. Renner. A comparison of service discovery protocols and implementation of the service location protocol. In *Proceedings of 6th EUNICE Open European Summer School: Innovative Internet Applications*, pp. 101–108. University of Twente, September 2000.

[Bir82] A. D. Birrell, R. Levin, M. D. Schroeder, and R. M. Needham. Grapevine: an exercise in distributed computing. *Communications of the ACM*, vol. 25(4):pp. 260–274, 1982.

[Blu01] *Specification of the Bluetooth System, Core, Vol. 1*, Feb 2001.

[Bos97] D. Bosnacki. Implementing discrete time in promela and spin. In *Proceedings of the VIII Conference on Logic and Computer Science, LIRA '97,*, pp. 25–32. 1997.

[Bow90] M. Bowman, L. L. Peterson, and A. Yeatts. Univers: an attribute-based name server. *Software-Practices and Experiences*, vol. 20(4):pp. 403–424, 1990.

[Bra01] J. Bray, C. F. Sturman, and J. Mandolia. *Bluetooth 1.1 Connect Without Cables, 2nd Edition*. Prentice Hall, December 2001.

[Bri02] E. Brinksma and A. Mader. Model checking embedded system designs (invited). In *6th Int. Workshop on Discrete Event Systems (WODES)*,

pp. 151–158. IEEE Computer Society Press, Los Alamitos, California, Oct 2002.

[Cat92] V. Cate. Alex - a global file system. In *Proceedings of the USENIX File System Workshop*, pp. 1–11. USENIX, Ann Arbor, Michigan, 1992.

[Cha94] D. Chadwick. *Understanding X.500 The Directory.* Chapman & Hall, London, 1994.

[Cha02] D. Chakraborty, A. Joshi, Y. Yesha, and T. Finin. Gsd: a novel group-based service discovery protocol for manets. In *4th International Workshop on Mobile and Wireless Communications Network*, pp. 140–144. IEEE Computer Society, Stockholm, Sweden, 2002.

[Coh94] J. Cohen, S. Aggarwal, and Y. Goland. *General Event Notification Architecture Base: Client to Arbiter*, June 1994.

[Cou05] G. F. Coulouris and J. Dollimore. *Distributed systems: concepts and design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, fourth edn., 2005.

[Cze99] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. Katz. An architecture for a secure service discovery service. In *Proceedings of ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99)*, pp. 24–35. Kluwer Academic Publishers, 1999.

[Dab01] C. Dabrowski and K. Mills. Analyzing properties and behavior of service discovery protocols using an architecture-based approach. In *Proceedings of Working Conference on Complex and Dynamic Systems Architecture (CDSA).* Distributed Systems Technology Centre, December 2001.

[Dab02a] C. Dabrowski, K. Mills, and J. Elder. Understanding consistency maintenance in service discovery architectures during communication failure. In *Proceedings of the Third International Workshop on Software and Performance*, pp. 168–178. ACM Press, July 2002.

[Dab02b] C. Dabrowski, K. Mills, and J. Elder. Understanding consistency maintenance in service discovery architectures in response to message loss. In *Proceedings of the 4th International Workshop on Active Middleware Services*, pp. 51–60. IEEE Computer Society, July 2002.

[Dab05] C. Dabrowski, K. Mills, and S. Quirolgico. *A Model-based Analysis of First-Generation Service Discovery Systems.* Special Publication 500-260, National Institute of Standards and Technology, 2005.

[Duv03] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive leases: A strong consistency mechanism for the world wide web. *IEEE Transactions on Knowledge and Data Engineering*, vol. 15(5):pp. 1266–1276, 2003.

[Edw01] W. K. Edwards and R. E. Grinter. At home with ubiquitous computing: Seven challenges. In *UbiComp '01: Proceedings of the 3rd international conference on Ubiquitous Computing*, pp. 256–272. Springer-Verlag, London, UK, 2001.

[Fen97] W. Fenner. Internet group management protocol, version 2, rfc-2236, 1997.

[Fra97] M. J. Franklin, M. J. Carey, and M. Livny. Transactional client-server cache consistency: alternatives and performance. *ACM Transactions on Database Systems*, vol. 22(3):pp. 315–363, 1997.

[Fra04] C. Frank and H. Karl. Consistency challenges of service discovery in mobile ad hoc networks. In *Proceedings of the 7th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM)*, pp. 105–114. 2004.

[Gad85] S. K. Gadia and J. H. Vaishnav. A query language for a homogeneous temporal database. In *PODS '85: Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pp. 51–56. ACM Press, New York, NY, USA, 1985.

[Gan03] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, vol. 42(1):pp. 5–18, 2003.

[Gel99] H.-W. Gellersen, M. Beigl, and H. Krull. The mediacup: Awareness technology embedded in a everyday object. In *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pp. 308–310. Springer-Verlag, London, UK, 1999.

[G.J98] G.J.Holzmann. An analysis of bitstate hashing. In *Formal Methods In System Design*, vol. 13, pp. 287–305. Springer-Verlag, November 1998.

[G.J03] G.J.Holzmann. The model checker spin, primer and reference manual. Addison-Wesley, September 2003.

[Gol00] Y. Goland, T. Cai, P. Leach, and Y.Gu. *Simple service discovery protocol, version 1.0*, 2000.

[Gon01] L. Gong. Jxta: A network programming environment. *IEEE Internet Computing*, vol. 5(3):pp. 88–95, May-June 2001.

[Gra89] C. Gray and D. Cheriton. Leases: An efficient fault tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 202–210. ACM Press, Austin, Texas, December 1989.

[Gud03] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, and H. Nielsen. *Simple Object Access Protocol (SOAP) V.1.2, Part 1: Messaging Framework*, June 2003.

[Gut03] E. Guttman, C. Perkins, J. C. Veizades, and M. Day. *Service Location Protocol, V.2, RFC-2608*. Internet Engineering Task Force (IETF), December 2003.

[Han98] M. Handley and V. Jacobson. Sdp: Session description protocol, rfc-2327, 1998.

[Hoe03] C. Hoene, A. Gunther, and A. Wolisz. Measuring the impact of slow user motion on packet loss and delay over ieee 802.11b wireless links. In *28th Annual IEEE Conference on Local Computer Networks (LCN 2003), The Conference on Leading Edge and Practical Computer Networking*, pp. 652–662. IEEE Computer Society, 2003.

[How99] T. Howes, M. Smith, and G. S. Good. *Understanding and Deploying LDAP Directory Services*. Macmillan Technical Publishing, Indianapolis, Indiana, 1999.

[Hua01] Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile enviroment. In *MobiDe '01: Proceedings of the 2nd ACM international workshop on Data engineering for wireless and mobile access*, pp. 27–34. ACM Press, New York, NY, USA, 2001.

[Hut00] M. Huth and M. Ryan. Logic in computer science: Modelling and reasoning about systems. Cambridge University Press, First Edition, January 2000.

[I.S01] I.Stoica, R.Morris, D.Karger, M.F.Kaashoek, and H.Balakrishnan. A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*. 2001.

[Jar84] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys (CSUR)*, vol. 16(2):pp. 111–152, 1984.

[Kam00] A. Kaminsky. Jinime: Jinitm connection technology for mobile devices. white paper, August 2000.

[Kle04] S. R. Klemmer, J. Li, J. Lin, and J. A. Landay. Papier-mache: toolkit support for tangible input. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 399–406. ACM Press, New York, NY, USA, 2004.

[Lam86] B. W. Lampson. Designing a global name service. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing (PODC '86)*, pp. 1–10. ACM Press, New York, NY, USA, 1986.

[Leh86] T. J. Lehman and M. J. Carey. Query processing in main memory database management systems. In *SIGMOD '86: Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, pp. 239–250. ACM Press, New York, NY, USA, 1986.

[Luc98] D. Luckham. Rapide: A language and toolset for simulation of distributed systems by partial ordering of events. In Y. Masunaga, T. Katayama, and M. Tsukamoto, editors, *Proceedings of Worldwide Computing and Its Applications, International Conference, WWCA '98, Second International Conference*, vol. 1368 of *Lecture Notes in Computer Science*, pp. 88 – 96. Springer-Verlag, March 1998.

[Luc02] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Man96] S. Mann. Smart clothing: wearable multimedia computing and personal imaging to restore the technological balance between people and their environments. In *MULTIMEDIA '96: Proceedings of the fourth ACM international conference on Multimedia*, pp. 163–174. ACM Press, New York, NY, USA, 1996.

[Mic00] Microsoft. *Universal Plug and Play Architecture, V1.0*, Jun 2000.

[Mic03a] Sun Microsystems. *JavaSpaces Service Specification , version 2.0*, June 2003.

[Mic03b] Sun Microsystems. *The Jini Architecture Specification, version 2.0*, June 2003.

[Mic05] Sun Microsystems. Homepage at http://java.sun.com/javame/ reference/docs/index.html, 2005.

[Min90] S. L. Min and J.-L. Baer. A performance comparison of directory-based and timestamp-based cache coherence schemes. In *In Proceedings of the International Conference on Parallel Processing, Volume I*, pp. 305–311. CRC Press, 1990.

[Moc88] P. Mockapetris and K. J. Dunlap. Development of the domain name system. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pp. 123–133. ACM Press, New York, NY, USA, 1988.

[Mur04] R. Murch. *Autonomic Computing*. Prentice Hall, March 2004.

[Nee93] R. Needham. Names. In S. Mullender, editor, *An Advanced Course In Distributed Systems*, pp. 315–326. ACM Press/Addison-Wesley Publishing Co., Wokingham, England, 1993.

[Not04] W. W. G. Note. Web services architecture. http://www.w3.org/TR/ws-arch/, February 2004.

[Oki93] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus: an architecture for extensible distributed systems. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pp. 58–68. ACM Press, New York, NY, USA, 1993.

[Pap98] G. A. Papadopoulos and F. Arbab. Coordination models and languages. In *Advances in Computers*, vol. 46, pp. 329–400. Academic Press,, Orlando, Florida, USA, August 1998.

[Pea80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, vol. 27(2):pp. 228–234, 1980.

[Pet88] L. L. Peterson. The profile naming service. *ACM Trans Comput Syst*, vol. 6(4):pp. 341–364, 1988.

[Pet96] K. Petersen, M. Spreitzer, D. Terry, and M. Theimer. Bayou: replicated database services for world-wide applications. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pp. 275–280. ACM Press, New York, NY, USA, 1996.

[Ric00] G. G. Richard. Service advertisement and discovery: enabling universal device cooperation. In *IEEE Internet Computing*, vol. 4, pp. 18–26. IEEE Computer Society Press, Los Alamitos, California, September-October 2000.

[Row01] A. I. T. Rowstron and P. Druschel. Chord: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *In Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems (Middleware 2001)*, pp. 329–350, November 2001.

[Roy70] W. W. Royce. Managing the development of large software systems. In *Proceeding of IEEE WESCON, Reprinted in Proceedings of the 9th International Conference on Software Engineering (1987*, pp. 328–338. IEEE Computer Society Press, Los Alamitos, California, Nov 1970.

[Ruy00] T. C. Ruys. Low-fat recipes for spin. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pp. 287–321. Springer-Verlag, London, UK, 2000.

[Sah03] D. Saha and A. Mukherjee. Pervasive computing: A paradigm for the 21st century. vol. 36, pp. 25–31. IEEE Computer Society, Los Alamitos, CA, USA, 2003.

[Sat96] M. Satyanarayanan. Fundamental challenges in mobile computing. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pp. 1–7. ACM Press, New York, NY, USA, 1996.

[Sat01] M. Satyanarayanan. Pervasive computing: Vision and challenges. vol. 8, pp. 10–17. IEEE Computer Society, Los Alamitos, CA, USA, August 2001.

[Sco02] J. Scott, F. Hoffmann, M. Addlesee, G. Mapp, and A. Hopper. Networked surfaces: a new concept in mobile networking. *Mob Netw Appl*, vol. 7(5):pp. 353–364, 2002.

[Sto03] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, vol. 11(1):pp. 17–32, 2003.

[Sun03] V. Sundramoorthy, J. Scholten, P. G. Jansen, and P. H. Hartel. Service discovery at home. In *4th Int. Conf. on Information, Communications & Signal Processing and 4th IEEE Pacific-Rim Conf. On Multimedia (ICICS/PCM)*, p. 1929. IEEE Computer Society Press, December 2003.

[Sun05] V. Sundramoorthy, C. Tan, P. H. Hartel, J. I. den Hartog, and J. Scholten. Functional principles of registry-based service discovery. In *30th Annual IEEE Conf. on Local Computer Networks (LCN)*, pp. 209–217. IEEE Computer Society Press, Sydney, Australia, November 2005.

[Sun06a] V. Sundramoorthy and G. van de Glind. Frodo high-level and detailed design specifications –version 1.0. Technical Report TR-CTIT-06-25, Enschede, June 2006.

[Sun06b] V. Sundramoorthy, G. J. van de Glind, P. H. Hartel, and J. Scholten. The performance of a second generation service discovery protocol in response to message loss. In *1st Int. Conf. on Communication System Software and Middleware*, p. to appear. IEEE Computer Society Press, New Delhi, India, Jan 2006.

[Sun06c] V. Sundramoorthy, P. H. Hartel, and J. Scholten. On consistency maintenance in service discovery. In *20th IEEE Int. Parallel & Distributed Processing Symp. (IPDPS 2006)*, p. 10 in CDROM. IEEE Computer Society Press, Los Alamitos, California, April 2006.

[Tan02a] A. Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 2002.

[Tan02b] A. S. Tanenbaum and M. V. Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[TEA04] TEAHA. Homepage at http://www.teaha.org, 2004.

[Ter94] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS)*, pp. 140–149. IEEE Computer Society Press, Austin, Texas, September 1994.

[Van05] K. Vanthournout, G. Deconinck, and R. Belmans. A taxonomy for resource discovery. *Personal Ubiquitous Comput*, vol. 9(2):pp. 81–89, 2005.

[Wan05] S. C. Wang, M. L. Chiang, K. Q. Yan, and K. F. Jea. Streets of consensus under unknown unreliable network. *SIGOPS Operating Systems Review*, vol. 39(4):pp. 80–96, 2005.

[Wei91] M. Weiser. The computer for the 21st century. In *Scientific American*, vol. 265, pp. 94–104. IEEE Computer Society Press, Los Alamitos, California, September 1991.

[Wis05] R. Wishart, R. Robinson, J. Indulska, and A. Josang. Superstringrep: Reputation-enhanced service discovery. In V. Estivill-Castro, editor, *28th Australasian Computer Science Conference (ACSC2005)*, vol. 38 of *CRPIT*, pp. 49–58. ACS, Newcastle, Australia, 2005.

Curriculum Vitae

Vasughi Sundramoorthy was born in Kuala Lumpur, Malaysia on May 7, 1977. After finishing secondary school in 1997, she studied B. Eng in Computer and Communications Systems Engineering in University Putra Malaysia. She graduated in 2000, and went on to work as a Software Development Engineer in Motorola Malaysia Software Center. In April 2002, she became a PhD student (in Dutch: *Assistent in Opleiding* or AIO) with the Faculty of Electrical Engineering, Mathematics and Computer Science at the University of Twente. She was a member of the At Home Anywhere project, under the supervision of ir. Hans Scholten and Prof. dr. Pieter Hartel. She has deep interests in developing distributed systems for pervasive computing, with emphasis on fault-tolerance.

At the time of this thesis publication, Vasughi is a Research Associate in the Department of Computing at Lancaster University, UK.