Performance Evaluation of a Resource Discovery Service

Dartmouth Technical Report TR2004-513

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Master of Science

in

Computer Science

by

Jue Wang

DARTMOUTH COLLEGE

Hanover, New Hampshire

October 5, 2004

Examining Committee:

(chair) David Kotz

Robert Gray

Susan McGrath

Charles K. Barlowe
Dean of Graduate Studies

© Copyright by

Jue Wang

2004

Abstract of the Thesis

Performance Evaluation of a Resource Discovery Service

by

Jue Wang

Master of Science in Computer Science

Dartmouth College, Hanover, NH

October, 2004

Professor David Kotz, Chair

In a pervasive computing environment, the number and variety of resources (services, devices, and contextual information resources) make it necessary for applications to accurately discover the best ones quickly. Thus a resource-discovery service, which locates specific resources and establishes network connections as better resources become available, is necessary for those applications. The performance of the resource-discovery service is important when the applications are in a dynamic and mobile environment. In this thesis, however, we do not focus on the resource- discovery technology itself, but the evaluation of the scalability and mobility of the resource discovery module in Solar, a context fusion middleware. Solar has a naming service that provides resource discovery, since the resource names encode static and dynamic attributes. The results of our experiments show that Solar's resource discovery performed generally well in a typical dynamic environment, although Solar can not be scaled as well as it should. And we identify the implementation issues related to that problem. We also discuss experience, insights, and lessons learned from our quantitative analysis of the experiment results.

Acknowledgments

I want to express my sincere gratitude toward my advisor and my mentor, Professor David Kotz, for his practical support, valuable guidance, and enthusiastic encouragement along the way. His diligence and commitment to science make a good example for me and his gentle nature helps me enjoy the time throughout my time at Dartmouth.

Special thanks to Guanling Chen for his precious suggestions, patient discussion, critical feedback and constructive criticism, with which my research life comes to be much more fruitful. And also I appreciate his brotherly help and warm encouragement when I felt frustrated because of tentative research stalemate.

I also thank Bob Gray, Ron Peterson, Susan McGrath, and Qun Li. They were helpful when discussing research ideas and were ready to answer my trivial questions at any time. Many of the research findings in this thesis result from discussions and collaborations with many brilliant people in addition to those I just mentioned. They include Tim Tregubov, Wayne Cripps, Kefei Cao, Libo Song, Ming Li, and other faculty and students in the Computer Science Department at the Dartmouth College.

Most of all, thanks to my parents and my grandma, who have been here for me at every moment. Their self-giving love makes me capable of weathering all the ups and downs throughout the ordeal of my life.

Sponsors

I also gratefully acknowledge the support of Dartmouth's Center for Mobile Computing and the Institute for Security Technology Studies. This project is also supported under NSF CISE research award CCR-9404919, and under Award No. 2000-DT-CX-K001 from the Office for Domestic Preparedness, U.S. Department of Homeland Security. Points of view in this document are those of the author(s) and do not necessarily represent the official position of the U.S. Department of Homeland Security or any other sponsor.

Contents

Abstract ii

Acknowledgments iii

1 Introduction 1

1.1 Pervasive computing middleware 5

1.2 Motivation 6

1.3 Problem scope 8

1.4 Outline of thesis 9

2 Background 10

2.1 Name service and resource discovery 10

2.1.1 Name service 12

2.1.2 Resource discovery service 13

2.1.3 Basic classification 14

2.1.4 Solar's choice 15

2.2 Solar's architecture 15

2.3 Resource Discovery scheme in Solar 16

2.3.1 Name processing in Solar 17

2.3.2 Different queries in Solar 18

3 Methodology **21**

3.1 Experiment scope 21

3.1.1 Five challenges 21

3.1.2 Number of Planets 23

3.1.3 Different name action 24

3.1.4 Request distribution 25

3.1.5 Number of strands 25

3.2 Metrics 26

3.3 Generator design 27

3.3.1 Name generator (Version 1) 28

3.3.2 Name value reassignment 30

3.3.3 Name generator (Version 2) 30

3.4 Key parameters 31

3.5 Experiment setup 33

3.5.1 Hardware configuration 33

3.5.2 Experiment setup 33

4 Experimental Results **37**

4.1 Request rate 37

4.2 Number of Planets 38

4.3 Name action 51

4.4 Request distribution 53

4.5 Number of strands 53

5 Related Work **57**

5.1 INS's Evaluation 57

5.2 INS/Twine 59

5.3 Comparison 61

5.4 Others 62

6 Conclusions and Future Work **63**

6.1 Contributions 63

6.2 Future work 64

6.3 Conclusion 65

Bibliography **66**

List of Tables

3.1 The table for all parameter definitions and brief explanations. 32

3.2 Configuration of each node's CPU. 35

4.1 Request loss ratio measured at a particular receiver for each experiment. . . 44

List of Figures

1.1 Context is defined as implicit input to applications. 3

1.2 Multiple devices provide resources. 4

1.3 A disaster scenario. 7

2.1 The basic classification of resource discovery services. 14

2.2 The structure of Solar. 17

2.3 The process flow on how Solar saves a name. 19

2.4 Two kinds of queries. 20

3.1 The definition of the metrics. 27

3.2 The attribute hierarchy for a name. 29

3.3 A name structure based on the name content hierarchy. 30

3.4 An illustration for name reassignment. 31

3.5 Jefferson's portrait. 34

3.6 A network setup illustration. 36

4.1 Results for real request trend. 39

4.2 The logs collected in our experiments. 40

4.3 The results for different numbers of Planets. 41

4.4 Examining the latency of individual requests. 43

4.5 The Planet `top` results for three pairs of cases. 46

4.6 The generator `top` results for three pairs of cases. 47

4.7 Latency behavior of Solar's request handling thread pool. 49

4.8 Cooperative name processing procedure. 50

4.9 The results for different name actions. 52

4.10 The results for different request distributions. 54

4.11 The results for different numbers of name strands. 56

Chapter 1

Introduction

Today, computer scientists are trying to conceive a new way of thinking about computers in the world, one that takes into account the natural human environment and allows the computers themselves to vanish into the background. The idea of integrating computers seamlessly into the world is often called *pervasive computing*, or *ubiquitous computing* [31, 14]. Although some authors identify a difference between pervasive computing and ubiquitous computing (pervasive computing aims to make information available everywhere while ubiquitous computing requires information to be available everywhere [24, 36]), we treat the two terms synonymously for the purpose of this thesis. The essence of pervasive computing is for computers to fit the human environment, instead of forcing humans to enter machine's environment [40, 30]. More practically, pervasive computing is the harnessing of many small, cheap computational devices that are present in users' physical environments, including the home, office and elsewhere. The term "ubiquitous" is intended to suggest that small computing devices will eventually become common in most everyday objects. Although the hardware was impossible to achieve when the concepts were proposed in the early nineties, current technologies for device miniaturization and heterogeneous wireless networks have led increasingly to the integration of small and portable devices, and have

thus led to a much better position to pursue pervasive computing research.

Pervasive computing is promising but impractical, however, unless we can solve the following problem. A pervasive-computing environment is so rich with information, and the user is continually interacting with a changing set of nearby computers or other devices and thus could be easily overwhelmed. To gracefully integrate a computation and communication saturated environment with human users, pervasive-computing applications need to be *context-aware*; that is, the applications must be aware of and adapt to the situation in which they are running to avoid exposing unnecessary information and complexity to end users.

The definition of *context* tends to be vague, however, because everything in the world happens in a certain context [4]. The term has been used in many ways in different areas of computer science, such as "context-sensitive help", "contextual search", or "multitasking context switch". Here we focus on the context used by applications in mobile computing. Not satisfied by a general definition, some researchers divide context into several aspects [32]:

- *computing context*, such as network connectivity, communication costs, and communication bandwidth;
- *user context*, such as the user's profile, location, people nearby, even the current social situation;
- *physical context*, such as lighting, noise levels, traffic conditions, and temperature;
- *time context*, such as time of a day, week, or month and season of the year.

In this thesis, we use a loose definition of context from the point of view of applications. Lieberman and Selker [21] define context to be any input other than the explicit input and

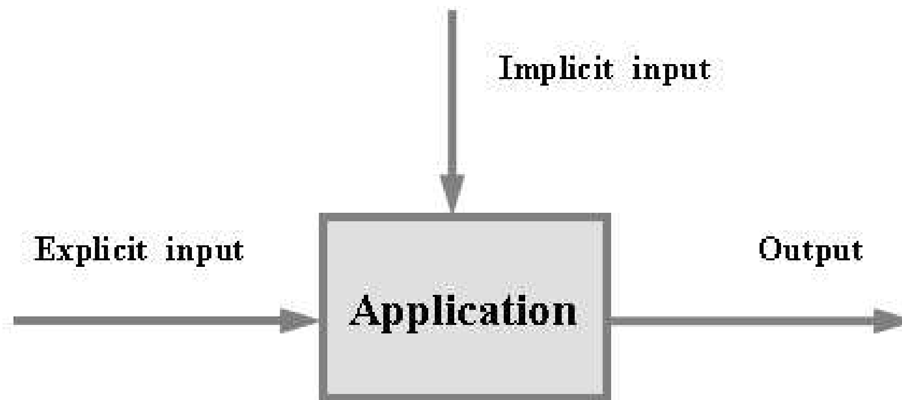


Figure 1.1: Context is defined as implicit input to applications.

output, as shown in Figure 1.1, where explicit input is the user's intentional action such as key strokes and mouse clicks.

Applications typically derive their context information (implicit input) from physical sensors and online information sources. These applications often run on portable but resource-constrained devices including:

- laptop computers;
- portable devices, including personal digital assistants (PDAs), mobile phones, pagers, video cameras, and digital cameras;
- networked sensors, such as location sensors that track the location of a devices and users, and environment sensors that provide some environmental situation around this sensor like temperature or humidity; and
- devices embedded in appliances such as washing machines, hi-fi systems, cars and refrigerators.

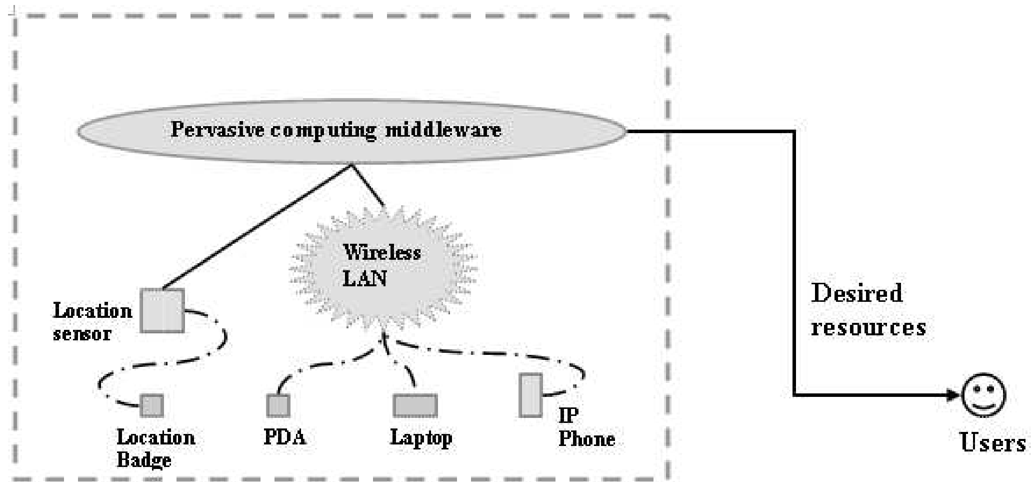


Figure 1.2: Many kinds of devices can work and provide resources in a pervasive computing environment. Here pervasive computing middleware is a platform for context before it is delivered to the user's application.

Figure 1.2 shows that many kinds of devices provide resources for the user in a pervasive computing environment.

Various devices produce information that might be useful as context, but the format, scope, or accuracy of the data may not be appropriate for the applications to use directly. Each device only has a limited view of the world and the hardware itself may be error-prone. Since only with a reasonably-accurate context can applications be confident to make adaptation decisions, the conversion of raw data into high-level context information, such as the user's current activity, is a requirement. The conversion from raw sensor data to higher-level understanding might involve simple filtering based on a value match, or sophisticated data correlation and machine-learning techniques. This process is called *context fusion*. In addition, it is necessary to enable applications to easily discover the necessary information sources; hence the need for a resource discovery service.

A *resource-discovery* service may monitor a resource's states and user's needs or re-evaluate accessible alternatives, and provide users the desired context information by establishing or rebinding network connections. Our research goal is not the resource-discovery

technology itself, but the evaluation of resource discovery performance because latency and scalability are important features of any practical pervasive computing system.

In this chapter, we first introduce the concept of pervasive computing middleware since resource discovery is a module in such middleware. Then we present an overview of the thesis.

1.1 Pervasive computing middleware

Generally, a *pervasive computing middleware* is an infrastructure to process raw data and dispatch them to specific applications in a pervasive computing environment. The middleware processes raw data by fusing multiple incoming resource streams and thus improves the quality of computed context [9, 17, 13, 19, 33]. While it may be possible to integrate all sensors on a single platform for a particular application, such as an augmented mobile phone [34], we are concerned about larger scenarios, with distributed sensors and multiple applications on different devices that may benefit from the aggregation of multiple resources. These context-aware applications require customized context tailored to their needs and also demand a resource-discovery scheme to locate the most relevant information sources quickly in a changing environment.

In this thesis, we use an integrated pervasive computing middleware, Solar. It is an infrastructure with two kinds of clients: *sensors* as data sources and *applications* as data sinks. We provide more information about Solar in Chapter 2. In this section, we introduce the general features of Solar by examples.

Solar has two prototypes, both implemented in Java [3]. Both prototypes adopted an operator composition programming model [5] (an operator is an independent data processing module that takes one or more data sources as input and acts as another data source.) The first implemented prototype had a centralized architecture for simplicity [39], and suc-

cessfully achieved flexible data-fusion customization. To scale to a large number of applications and sensors, the second prototype was designed after experience with the first version, including an analysis of a sensor environment [7], performance and interoperability [41, 42], and the security and access control design [23, 25]. The second prototype had a fully distributed and self-organized architecture, including the resource discovery scheme we evaluated in this thesis.

1.2 Motivation

A *resource-discovery service* finds the available alternatives, and binds or rebinds clients to services as better alternatives become available. A detailed definition of Solar's resource discovery service is discussed in the next chapter.

Basically, a resource-discovery service should achieve three main goals: (1) handle sophisticated resource descriptions and query patterns; (2) handle mobility and dynamism in the pervasive computing environment, including changes of resource location, resource description and network attachment point; and (3) scale to a large number of distributed resources.

Little research exists that focuses on these performance characteristics of resource-discovery services. Many of these systems have addressed limited combinations of these properties and many resource-discovery schemes have been designed primarily for small networks, or for networks where dynamic updates are relatively uncommon or infrequent (e.g., DNS [27]).

Applications that are used in a mobile scenario (where the number of resources can change quickly and updates are common) require a resource-discovery component that can support a highly dynamic *name space*. For example, applications used in disaster-response scenarios where medical , fire and law enforcement personnel people might use such mobile

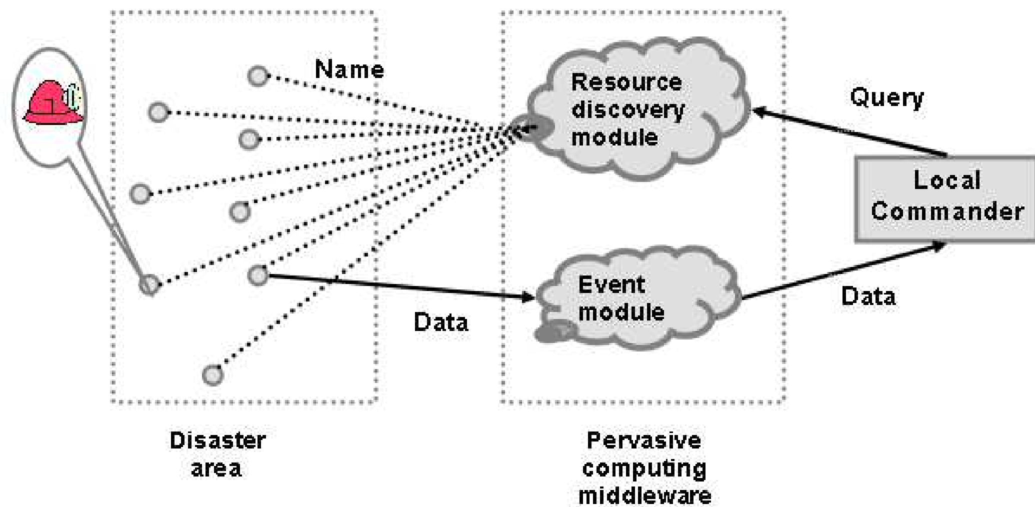


Figure 1.3: A disaster scenario. Some mobile sensors distributed at the disaster scene report the current situation to the local central commander, through a resource-discovery system. The commander manages this emergency and dispatches the rescue units.

devices for situational awareness. A typical scenario is illustrated by Figure 1.3.

Image that a fire is caused by some electronic equipment in a high-rise apartment building located in a high-density downtown. At the scene of a disaster, rescue workers might wear helmets with small attached cameras and a wireless network interface. All these cameras communicate wirelessly with a pervasive computing middleware, which advertises the cameras as resources. If these cameras are named according to their location, a local commander can manage the emergency with the assistance of the cameras. He may request photographs of a particular area by selecting cameras whose name (location) matches the area of interest, monitor the current situation on the scene and dispatch the rescuers on various assignments. In this scenario, the resource names may be added or deleted after the new rescue units are dispatched to the scene or withdrawn from the scene. A camera's name may change whenever its location changes, e.g., because it is attached to a rescue vehicle or personnel. Here the resource discovery system plays the key role because it must

quickly allow the commander to locate and subscribe to information resources relevant to the situation, and reflect any changes in the state of the resource.

This example helps us to understand that for some applications, it is important for a resource discovery service to support a highly dynamic name space, although some applications may use more slowly changing resources. Thus we set out to evaluate Solar's capability to handle a large and dynamic name space.

1.3 Problem scope

The main theme of this thesis is to evaluate the performance of a resource-discovery service on a large-scale context-fusion middleware with a highly dynamic name space. Any resource-discovery service must perform well in five ways. First, the service must work stably in most normal situations. Second, it should be flexible to application-specific customization and personalization. Third, it should be scalable to handle a large number of sensors, devices, applications, and users. Fourth, it should support dynamic resource attributes since some devices change their locations or states often. Finally, it should be robust enough to achieve some degree of self-management by automatically monitoring and recovering node failures. There are many ways to measure a system; our central problem is to measure the performance of this service's ability to scale given a highly dynamic pervasive-computing environment. That is, our research focuses on the evaluation of scalability in response to the number of components and their mobility.

A breakdown of our major research questions follows:

- Since we are interested in performance and scalability, we must build an environment containing a large quantity of resources in which many change their locations or states often. How do we simulate the dynamic environment with enough authenticity, and how do we scale the workload realistically?

- After building the environment, we must determine how to measure scalability. That is, what metrics are suitable? Perhaps latency, throughput, memory consumption, network bandwidth consumption, disk space consumption, or others. Even after choosing the metrics, how do we measure them precisely?
- How do we design experiments to measure the performance metrics under the desired workload so that the measurements do not perturb the system?

Throughout this thesis, we strive to answer these questions, and to measure and interpret the performance of Solar's resource-discovery service.

1.4 Outline of thesis

This thesis begins with a general introduction to resource discovery, then introduces Solar in Chapter 2. Then we present detailed experiment designs under different metrics in different scenarios, and our experimental environment, in Chapter 3. In Chapter 4 we report our experimental results and discuss their significance. In Chapter 5, we survey and discuss some other related work. Then we conclude in Chapter 6.

Chapter 2

Background

In this chapter, we first define and explain the concepts of name service and of resource discovery. Then we briefly discuss Solar's operator composition and its resource discovery module.

2.1 Name service and resource discovery

Pervasive computing leads to numerous networked devices and thus to a challenge for applications: how to locate a particular network resource out of hundreds or thousands of accessible resources. A *resource* is an entity wishing to be discovered by applications. It can be a service, a device, or a piece of information. There may be only one resource, but at times there are many, and users want to find the one that meets their needs. So designating a particular resource by a unique name can be useful for identifying the desired instance. For example, a name in the form of a URL is needed to access a specific web page. Processes can not share particular resources managed by a computer system unless they can name the resources uniquely and consistently. Similarly, users can not communicate with one another through a distributed system unless they can name one another, for example, with

an email address.

We say that a name is *resolved* when it is translated into data about the named resource or object, often in order to invoke an action upon it. The association between a name and an object is called a *binding*.

A name might contain information about the object that it names; for example, it might contain information about the location of the object. There are several ways to represent names, for example, a string, a path, or a set of attributes. A string name often has no strict structure. For example, a temperature sensor in Sudikoff room 210 can be represented as [temperature210]. A path name, however, is described from the root to a leaf in the tree that defines a name space. The temperature sensor might thus be [/Sudikoff/2F/210/temp-sensor]. The third alternative has less structure than a tree: a name is a set of attributes. An *attribute* represents the value of a property associated with an object, and consists of a value and a tag. The above temperature sensor might be named [sensor=temperature, room=210, floor=2, building=Sudikoff].

After explaining these basic terms, we may define a "resource-discovery service". First consider the difference between "name service" and "discovery service" [11, 12]. We can use a specific name to seek a particular resource; we also can use some descriptive attributes to achieve the same goal. Sometimes clients do not know the name of a particular entity that they seek, but they do have some information that describes it. Or the client requires a service and knows some of the characteristics that the required service must have. Based on different means of identification, name service and discovery service have a subtle difference. Name services use a complete name as identification. Discovery services may use any attributes as identification. Discovery services are sometimes called *yellow-page services*, and conventional name services are correspondingly called *white-page services*, in an obvious analogy with the different types of telephone directories. Discovery services are also sometimes known as *attribute-based name services*. Consider an example based

on this analogy. If we want to buy a car, we may search a telephone book to contact a car agent. Using a white-page service, we have to remember the name of some specific car agent, say Miller Automobile Co., and then we use the name to find the desired data, like telephone number, store location and open hours. Using a yellow-page service, we need only a descriptive attribute, such as automobile sales, and then we may find a section in the yellow pages where all the local car agents are listed with names, telephone numbers, and store locations. By using this descriptive attribute, not the name itself, we get all the desired information.

2.1.1 Name service

A *name service* stores a collection of one or more *contexts* – sets of bindings between textual names and attributes for objects. The major operation that a name service supports is to resolve a name tributes) from a given name. Operations are also required for creating new bindings, deleting bindings, listing bound names, and adding and deleting contexts.

Two main concepts for name services are name spaces and name resolution. Here we provide only general descriptions of these concepts.

A *name space* is the collection of all valid names recognized by a particular service. For a name to be valid means that the service will attempt to look it up, even though that name may prove not to correspond to any object – to be unbound. Names may have an internal structure that represents their position in a hierarchical name space, or in an organizational hierarchy. The most advantage of hierarchical name spaces is that each part of a name is resolved relative to a separate context, and the same name may be used with different meanings in different contexts. *Name resolution* is an iterative process whereby a name is repeatedly presented to naming contexts. A naming context either maps a given name onto a set of primitive attributes directly, or it maps it onto a further naming context and a

derived name to be presented to that context. To resolve a name, it is first presented to some initial naming context; resolution iterates as long as further contexts and derived names are output. Hierarchical file path names are an example; each directory is a separate context explored along the path.

2.1.2 Resource discovery service

In a name service, which stores a collection of <*name*, *attributes*> pairs, the attributes from a particular object are looked up for a given name. In a discovery service, attributes are used to look up resources. That is, we use some descriptive attributes, other than a complete name, to look up the desired objects. In these services, the "names" may be just another attribute.

A *discovery service* stores the collections of bindings between names and attributes and can look up entries that match attribute-based specifications. A discovery service returns attributes of any object found to match some specified attributes. So for example, the query "Telephone = 603-646-8614" might return ["Name = Jue Wang", "Telephone = 603-646-8614", "Email = jue.wang@dartmouth.edu"]. The client may specify that only a subset of the attributes is of interest – for example, just the email address of matching objects.

Attributes are clearly more powerful than names as designators of objects: programs can be written to select objects according to precise attribute specifications where names might not be known. Another advantage of attributes is that they do not expose the structure of organizations to the outside world, as do organizationally partitioned names, although the relative simplicity of use of textual names makes them unlikely to be thoroughly replaced by attribute-based naming in many applications. The Intentional Naming System [1], a resource discovery and service location system for dynamic and mobile networks of devices and computers, emphasizes the use of descriptive attributes for resources. Then applica-

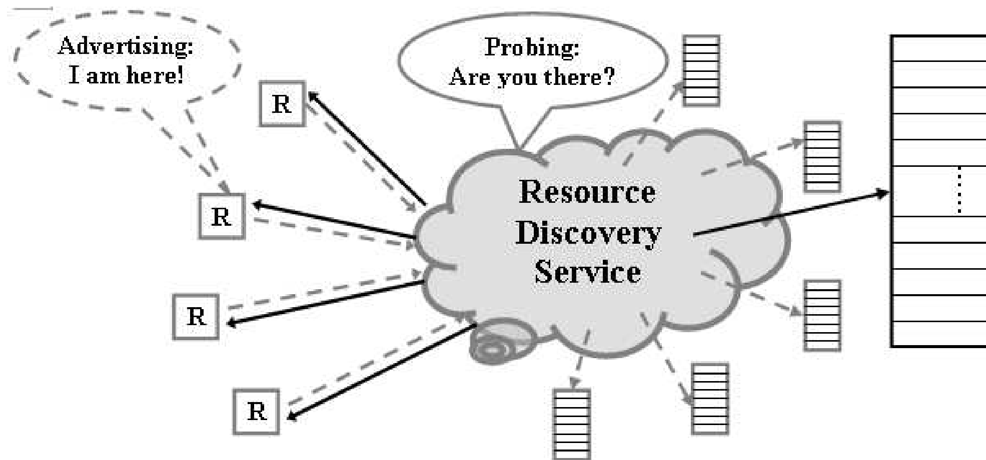


Figure 2.1: The basic classification of resource discovery services. Grey solid lines and arrows are probes, and blue dashed line and arrows are advertisement. On the other side, blue solid arrows represent a query to a centralized directory and grey dashed arrows are queries to a distributed directory.

tions may contact desired resources by describing the destination (i.e., their intent), not naming the destination (i.e., not host name).

2.1.3 Basic classification

Using resource discovery, hundreds or thousands of resources can register descriptions, and clients can compose queries to locate these resources. There are many resource discovery schemes. They can be roughly classified into four basic classes according to the protocols they use [29, 15]: probing or advertising, and centralized or distributed directory. Here Figure 2.1 illustrates the classification.

Mobile devices wishing to discover resources in the environment can either passively listen to advertisements by other resources in the system or can actively probe the network with periodic discovery messages. Some people favor probing over advertisements because in an advertisement-based system the scope and frequency of the messages generated by a resource to advertise itself to the system can not be adjusted to suit the requirements

of any single client application. Furthermore, with resource advertisements arriving asynchronously at different frequencies from various resources, there may be excessive load on the network or on clients. From a design point of view, however, an advertisement-based system has the advantage that every change in resources will be immediately advertised to the system so that the clients can adapt the changes quickly without much delay.

Resources can be registered in a centralized or distributed directory. A centralized directory avoids query broadcasts and is simple, but it may not scale well. A distributed discovery direction may scale to a large number of resources and applications in a pervasive computing environment, but there is some overhead to keep the directory state consistent with the (often-changing) properties of resources.

2.1.4 Solar's choice

Solar represents names as a set of attributes for two reasons. While a hierarchical tree structure produces concise names and is easy to traverse and explore, the conventions used to structure the tree are likely stricter and harder to extend than those in a set of attributes, which may make the tree less attractive in a dynamic ubicomp environment. On the other hand, attribute-based names allow partial matches, e.g., [sensor=temperature, building=Sudikoff]. Although tree-based names may allow the same effect, e.g., [/Sudikoff/*/temp-sensor/], the syntax could be awkward. The Solar designers chose attribute-based names to allow much more flexibility in the structure of the name space.

2.2 Solar's architecture

Solar [8], a context-fusion infrastructure for pervasive computing, uses an advertising-based distributed directory; nodes in the context-fusion network collaborate as peers to distribute resource information and to resolve queries. It allows context-aware applications

to select distributed data sources and compose them with customized data-fusion operators into a directed acyclic information-flow graph. Using the filter-and-pipe [35] style, multiple graphs by different applications interconnect with each other to form a global *operator graph*. This approach allows two kinds of re-use: code-based reuse that allows applications to import existing modules from documented libraries, and instance-based reuse that allows applications to discover and use an already deployed data-fusion component. An illustration is in Figure 2.2.

First we define some terms from Solar [6]. A filter is an "*operator*" and a pipe is a "*channel*." A contextual information resource, whether sensing physical properties such as location or computational properties such as network bandwidth, *publishes* raw data in *events* to an *application* through Solar. An application sends a *query* to Solar to request specific resources. Solar consists of a set of functionally equivalent hosts, named *Planets*, which host operators and peer together to form a service overlay using a peer-to-peer protocol (specifically Pastry [28]). A resource may connect to any Planet to advertise its availability and an application may send its queries to any Planet to select resources and supply data-fusion operators. A Planet is an execution platform for operators, and it is responsible for tracking subscriptions, which is a continuous query, and delivering events in the operator graph. Thus each Planet is a peer node in a context-fusion network and all Planets are functionally equivalent.

2.3 Resource Discovery scheme in Solar

In this section, we introduce the name processing procedure and the different query types in Solar.

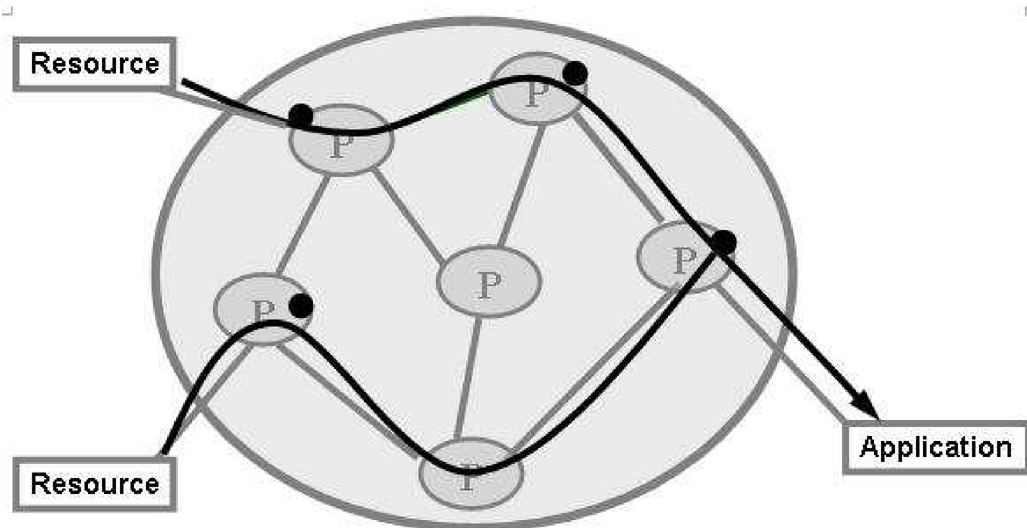


Figure 2.2: The structure of Solar. Here the grey circles are Planets, which are software virtual machines running on the workstations. Here the little black dots are operators, which are data processing modules. Events flow from resources though operators on the Planets, to provide context information to the applications.

2.3.1 Name processing in Solar

Solar's resource discovery mechanism is similar to INS/Twine [2], though using the Pastry routing protocol. In the Patry protocol each Planet is a peer and is given a random numeric key; a message with a given key is routed by Pastry to the Planet whose own key is close to the message's key.

Solar allows resources to advertise their existence by registering a name, and applications to discover resources by querying a name. We first define the name specification format and then describe Solar's resource (name) discovery scheme.

A *name specification* is a set of attributes, each of which has a tag string and a value string. Resources use name specifications to define their name; applications use a name specification to query for matching names. For example, the advertisement

```
[sensor= printer, class= color, building= Sudikoff, room= 020]
```

has four attributes; for the first attribute, the tag is "sensor" and the value is "printer." The

query

```
[sensor = printer, building= Sudikoff]
```

matches the above name because all attributes of the query exist in that advertisement name, with the same values.

For flexibility, Solar allows both flat attributes and hierarchical ones. For example, an attribute-based name can be flat as [sensor=printer, class=color, building=Sudikoff, room=002] or hierarchical as [sensor=printer, class=color, building= Sudikoff [room=002]]. Solar splits the name into several name strands. Every path from the root to a leaf is a name strand. Each name strand is hashed to a key; the fullname is sent to the Planet with the closest key. If a name is split into several strands, it will be duplicated and saved on several matching Planets depending on the hash keys of its strands. For example, consider the above advertisement name; Figure 2.3 shows the process flow.

As in the example above, a name matches a query if the query's strands are a subset of the name's strands. By hashing each of the query's strands and looking on the Planets identified by those keys, the query can be checked against every name containing a strand that also hashed to that key. Thus, Solar's directory is distributed among all Planets. The name directories on Planets are different but overlapping.

2.3.2 Different queries in Solar

There are two kinds of queries: a continuous query or a one-time query. To distinguish between them, we call the first one *subscribe* and the second one *query*. A subscription is a persistent query with a long time expiration so the directory will actively notify the querying entity about any changes in names that match the name specification. Thus we reduce the overhead of looking up names frequently, especially when monitoring resources that may change attributes often (e.g., due to the resource moving to a new location). A

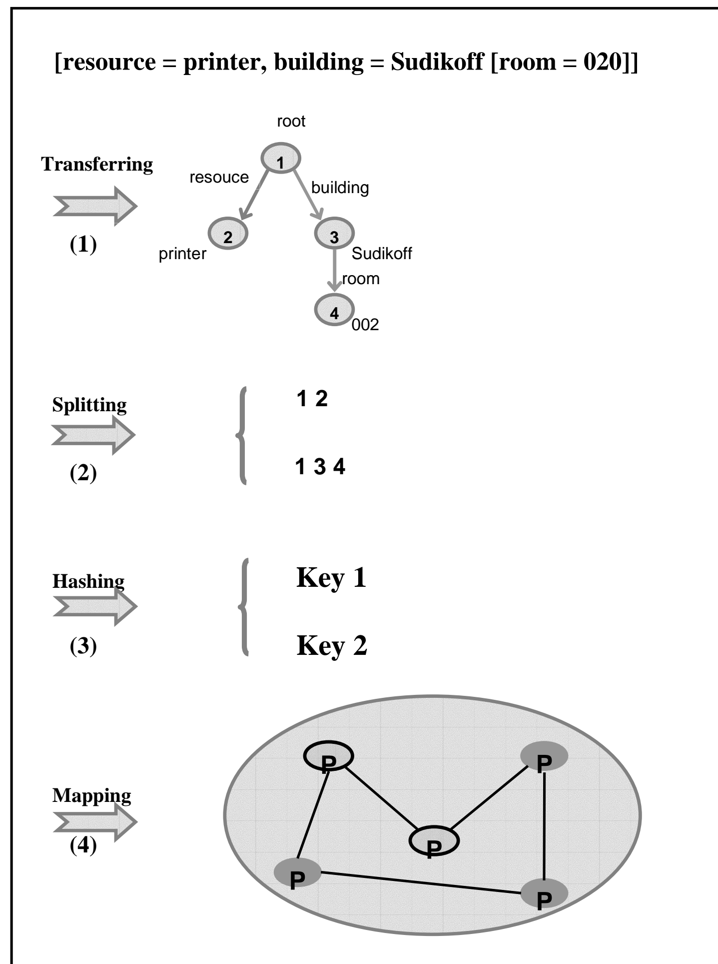


Figure 2.3: The process flow on how Solar saves a name. An advertisement name first is translated into a name tree, then split into two name strands. The two name strands are hashed to two keys, and mapped to two Planets with similar keys. Thus the name is saved on those two Planets with two copies; and both copies are the same.

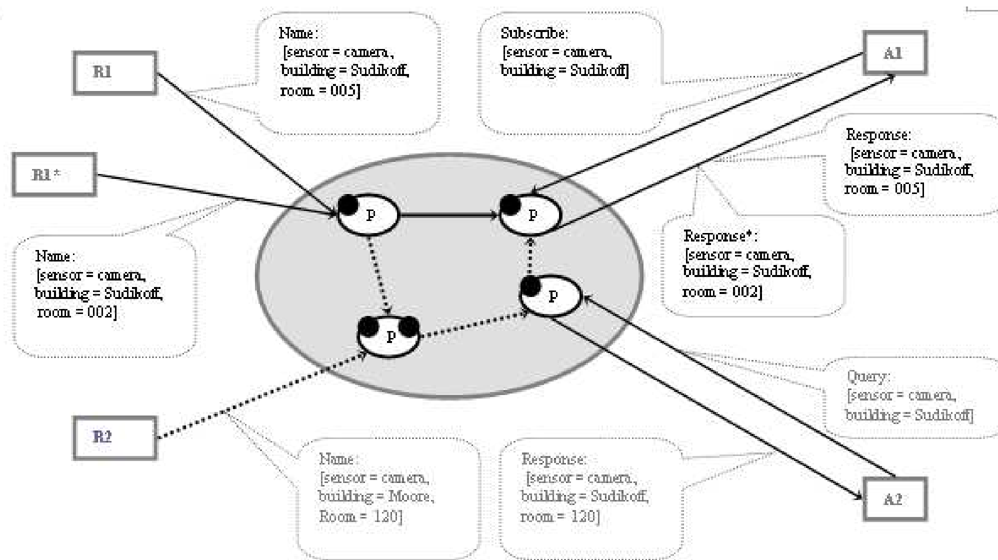


Figure 2.4: Resources R and applications A may connect to any Planet P. At time t, two resources R1 and R2 advertised and registered their name with Solar. Then Application A1 sent a subscription to Solar and Application A2 sent a query to Solar. A1 and A2 received the responses from Solar. Later, at time t*, Resource R1 moved and updated its name with Solar; A1 received the Response* automatically but A2 did not.

query returns a result only once. Figure 2.4 illustrates the difference between "subscribe" and "query."

Chapter 3

Methodology

In this chapter, we define the scope of our experiments and the metrics we measure, design the name generator, and identify the key parameters. We also introduce our experimental platform including its hardware configuration and network environment.

3.1 Experiment scope

We begin with a discussion of five challenges we face in designing an experiment. We then identify our primary goals for these experiments, and describe four aspects of our experimental design.

3.1.1 Five challenges

There are five basic challenges to be addressed to provide the resource-discovery service in a context fusion network (CFN), given a heterogeneous and volatile ubiquitous environment.

- Stability: a resource-discovery system in a CFN should work stably in normal situations.

- Flexibility: a resource-discovery system in a CFN must be flexible. It must allow deployment of well-known context services with both application-specific customization and user-specific personalization.

- Scalability: a resource-discovery service in a CFN must be scalable to handle a large number of sensors, devices, applications, and users. It should be easy to increase the service capacity to handle the increased load as necessary.

- Mobility: a resource-discovery service must explicitly support mobility, both at the physical and logical level. A moving device connecting to a CFN may traverse both geographic and network boundaries. Also a CFN may automatically migrate some mobile devices to balance the system load or to use the bandwidth more efficiently. Another form of logical mobility includes attribute changes, such as a printer that re-advertises itself to the resource discovery system after changing its working state from idle to busy. The term *dynamics* may be better to refer to name changes that result from physical movement or other logical changes in the device; both have the same effect on a resource-discovery system. For consistency, we still use *mobility* in this thesis.

- Robustness: the overall complexity of a pervasive environment requires a resource-discovery system to be robust yet require minimum user intervention. A dependable resource discovery system must be to some degree self-managed. That is, it may pro-actively monitor node failures, automatically recover lost components and garbage collect application-specific fusion components that are no longer in use.

Those five goals challenge every context-fusion system in a pervasive environment. In this thesis, however, we focus on the scalability of the resource-discovery service, especially in a dynamic environment. And we focus more on the speed than on the accuracy,

flexibility, or reliability of the name service.

We consider four aspects of our experiments. First, Solar is a distributed context-fusion network in a pervasive computing environment, and all the nodes (Planets) work collaboratively. Accordingly, we should vary the number of Planets. Second, to simulate various dynamic scenarios, we must consider different name actions, i.e., to insert various names, to submit various queries, or to change names frequently. Thus, the experiments can be divided into three types: 1) name additions, 2) name changes and 3) name queries. Third, the request distribution may have an impact on performance. In particular, a uniform request distribution may behave differently than a burst distribution. Fourth, we are interested in how the system performance varies when handling both long and short names; recall that a name with multiple strands will be saved on multiple Planets. Since every Planet needs time to communicate with other collaborative Planets, we expect that names with more strands can affect the processing time and system overall performance.

We describe each of these four aspects of our experiments in the following subsections, although in the experiments multiple aspects were involved at the same time.

3.1.2 Number of Planets

We experimented with a low, medium, and high number of Planets separately. Although Solar can run multiple Planet instances on one host, we ran one Planet per host so that the computation and network resources scale in proportion with the number of Planets. Below we describe how we use "generators" to create a synthetic workload. We ensure that each Planet is assigned the same number of generators to ensure equitable load for every Planet. For example, we assigned 30 generators as follows: six for each of 5 Planets, three for each of 10 Planets or two for each of 15 Planets.

3.1.3 Different name action

To simulate scalable and dynamic scenarios, we may create plenty of different names and queries to simulate the situation that resources send their names to Solar for advertisement and applications send queries or subscriptions to Solar for desired resources. Then, we change the name content to simulate the situation that the resource moves geographically or changes status physically. Thus, these three kinds of experiments are all scalability experiments, and the second one is motivated by dynamic environments.

- Generating name additions: to simulate an environment with many resources, we need to generate plenty of name additions.
- Generating name updates: as we discussed above, names change due to changes in the location or state of a resource that wishes to reflect location or state in its name.
- Generating name queries: we simulated the application behavior by generating plenty of name queries and measuring the performance changes when we vary the query rate.

In particular, the second case includes following situations:

1. A name is updated by changing the value for some attributes because of geographical location changes. For example, *[sensor = camera, building = Sudikoff, room = 020, accessibility = public]* $\Longrightarrow$ *[sensor = camera, building = Sudikoff, room = 001, accessibility = public].*

2. A name is updated by changing the value for some attributes because the resource's status changes. For example, *[sensor = camera, building = Sudikoff, room = 020,*

accessibility = public] $\Longrightarrow$ *[sensor = camera, building = Sudikoff, room = 020, accessibility = private].*

3. A resource disappears from the CFN and did not return before the expiration time in Solar's directory. Thus, Solar deletes the name from its directory.

4. A resource disappears from the CFN for a while and returns before the expiration time in Solar's directory. Thus, Solar extends the expiration deadline for this name.

The first two cases can be simulated by changing names, with different frequencies. In our current system, we did not simulate the last two cases.

3.1.4 Request distribution

When we scale the request rate to test the scalability, the request volume per unit time is not the only concern; request distribution is another problem. A uniform distribution is a gentle way to distribute the load; a bursty distribution is much tougher than uniform because all requests come in short bursts. To make it more realistic, we also consider an exponential distribution.

3.1.5 Number of strands

Because of the structure of Solar's distributed directory, we varied the number of name strands. We experimented with a variety of name sizes, choosing a number of name strands between 1 and 100.

One challenge arising from the different name strand numbers was to generate realistic names; it is hard to create a practical name with 100 name strands. So, for large names, we let names have simple content, although losing some realism. More details are in Section 4.3.

3.2 Metrics

Since we care about the speed of resource-discovery, we used three basic metrics: *name addition latency*, *name update latency* and *name lookup latency*, both shown in Figure 3.1.

Name addition latency is the time to communicate a name from its resource to its final destination (the applications with a subscription matching the new name). It consists of sending a name to Planet, processing the name in Solar and delivering a message to the destination application. (We assume that there are always subscribing applications, for the purpose of this metric.)

Name update latency is the time to update a name change from its resource to its destination (the applications with a subscription matching the new name). It consists of unadvertising the old name, Solar deleting the old entry in the name directory, sending a new name to Planet, processing the name update in Solar and delivering a message to the destination application. (We assume that there are always subscribing applications, for the purpose of this metric.)

Name lookup latency is the time to send a query to Solar, look up the name in the resource directory, and send back a response to the application that sent the query.

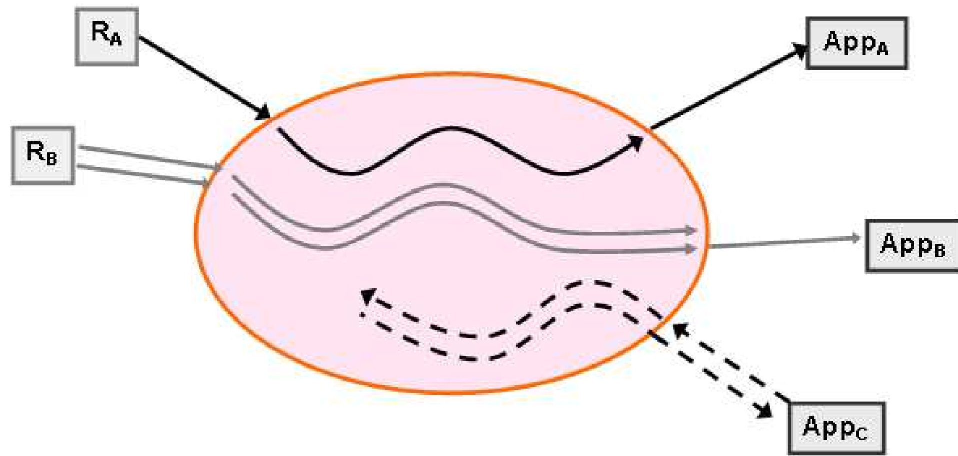


Figure 3.1: The definition of the metrics. The left squares are resources, and the right rectangles are applications with subscriptions. In this example, name addition latency is represented by the black solid arrows from R_A and App_A, name update latency is represented by grey solid arrows from R_B and App_B, and name query latency is represented by black dash arrows which is round-trip. A curved line means it may route through several Planets in Solar.

3.3 Generator design

To generate name updates, name additions, and name queries realistically, we wrote two programs capable of acting like a resource or application client to Solar. Version 1 simulated practical names with realistic name content and Version 2 generates names or queries with a large number of strands but without practical name content.

In this section, we present the Version 1 generator design, and discuss the difference between name templates and names in Section 3.3.1. Then we discuss some details about value assignment for names from the Version 1 generator in Section 3.3.2. Then we introduce the Version 2 generators briefly in Section 3.3.3.

3.3.1 Name generator (Version 1)

We use *name generators* to simulate resources and *query generators* to simulate the applications. A name generator has two layers. The first layer creates a name template, which is a set of attributes without values. The second layer fills in the template by assigning every attribute a reasonable value, and then publishes the name to Solar. Every name template is built with several existing attributes and sub-attributes, e.g.,

```
[sensor= printer [class= ], building= Sudikoff [room= ],
 others= accessibility [class= ]].
```

Here the name template has three attributes and every attribute has its own sub-attributes without any specified value. After filling an assigned value, a complete name advertises the resource in its current state, as follows:

```
[sensor= printer [class= color], building= Sudikoff [room= 020],
 others= accessibility [class = public]].
```

Thus, by separating the name template creation and the name value assignment, we can create names automatically and at the same time generate name updates easily. We generate the names with attributes as follows.

- Resource entity: the moving resource may be a digital video camera, a digital still camera, a portable personal computer, a PDA, an IP phone, a location sensor, or an environmental sensor. Every kind of resource has some different sub-features. Without losing universality, we use four kinds of resources. They are printer, camera, location sensor, environmental sensor. The sub-features for the printer are type (color or gray), for the camera are type (still or video), for the location sensor are badge ID (values from 0000 to 1000) and collector ID (values from 000 to 1000), and for the environmental sensor are type (temperature or humidity).

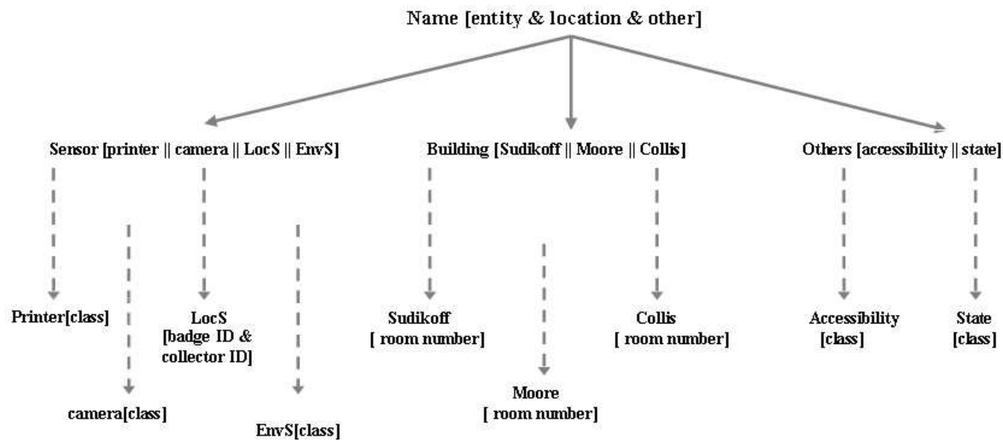


Figure 3.2: The attribute hierarchy for a name. A solid arrow represents an "and" relation between the attributes. A dashed arrows represents an "or" relation between attributes.

- Resource location: the location includes building (Sudikoff, Moore and Collis) and room number (1 to 20).

- Other features: some other features like the accessibility and resource state are representative. The resource might be open to the public, or just for private access; the resource state can be "busy" or "idle."

Every name template fits in the hierarchy in Figure 3.2.

Based on this attribute hierarchy, every name template has three kinds of features. The first one is sensor entity, the second one is location and the last one is other features. Every kind of feature has an attribute and a sub-attribute. The assignor fills attributes with a value that is randomly chosen with a uniform distribution across its range, leading to a large number of potential names.

```
(2 + 2+ 1000* 100 + 2) * (20 + 20 + 20)* (2 + 2) = 24,001,440
```

Since queries have the same format as names, a query generator is almost the same as a name generator except that queries may not include the full attribute hierarchy and thus a partial matching may happen.

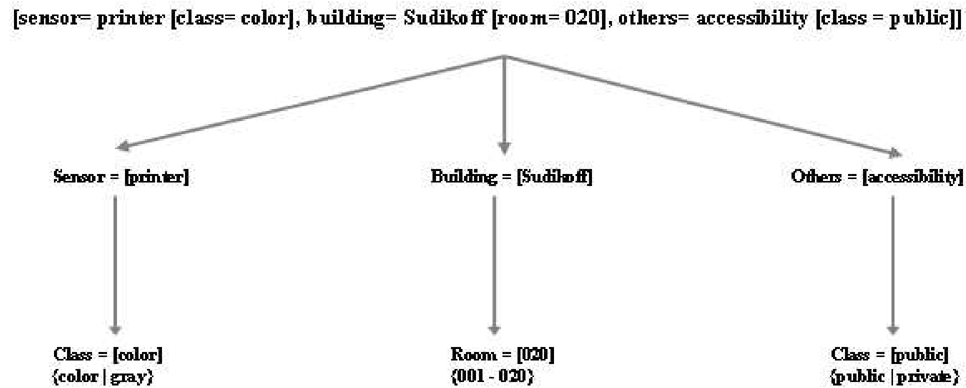


Figure 3.3: A name structure based on the name content hierarchy.

3.3.2 Name value reassignment

We needed to generate name updates to simulate the situation that resources are moved (or change state) frequently. Consider the example in Figure 3.3; we decompose it on a tree.

The name (or query) has three attributes, thus three sub-trees. Every attribute has a sub-attribute, thus two layers. A name template is the tree without all leaf nodes, which is the first layer. A name is the whole tree, including the values at leaf nodes. We assign every template a unique ID for identification. Then the second layer assigns values in this template and publishes it to Solar.

Note that we do not allow two values consecutively assigned to be the same. For example (Figure 3.4), a device exists at location A at t_1, then moves from A to B at t_2, and comes back from B to A at t_3. The name produced from the device at t_1 may be same as the one at t_3. Actually, this situation seems likely to often in a pervasive computing environment.

3.3.3 Name generator (Version 2)

When we do experiments with names that have a large number of strands we need a different name generator. The Version 2 name generator can build names with a large number

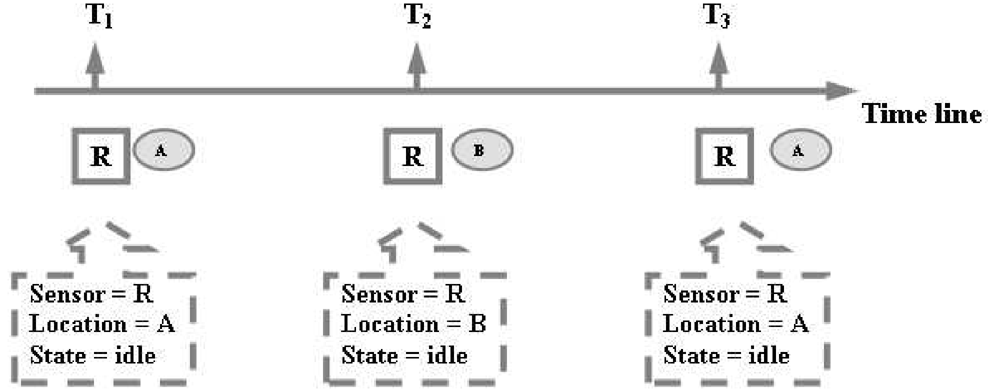


Figure 3.4: An illustration for name reassignment.

of strands but with no concern for realistic content. Thus the attributes are not hierarchical and a name looks like:

$$[attribute_1 = value_1, attribute_2 = value_2, \ldots, attribute_n = value_n]$$

N attributes will be divided into n name strands; every name strand is a copy to be processed. If a Planet receives multiple copies for one name, it will do multiple operations. For example, if we have a name with 100 name strands, then 100 copies, for 10 Planets. On average each Planet receives 10 copies, and each Planet will process 10 for each name addition, query, or update.

3.4 Key parameters

We used several parameters in our experiments. For clarity, we arrange them in Table 3.1 and explain them briefly here.

The parameters λ and I control the arrival times of new requests. Thus in burst experiment, all $\lambda * I$ requests come at the start of the interval; in non-burst experiment, the requests arrive every $1/\lambda$ time. For example, in a name addition experiment $\lambda = 10$, it means the

Parameter	Notation	Brief Explanation
Request rate	λ	Request rate (name addition rate, name update rate, or query rate). The unit is request number / second.
Interval size	I	Time of an interval with the unit as second. This parameter is used only in burst experiments.
Name generator number	N_N	Number of name generators.
Query generator number	N_Q	Number of query generators.
Planet number	N_P	Number of Planets running.
Host number	N_H	Number of hosts that run Planets.
Running time	T	Running time for every experiment.
Name strand number	A	Number of name strands for a name, used only for experiments varying the number of name strands.

Table 3.1: The table for all parameter definitions and brief explanations.

name generation produces 10 new names per second uniformly, that is, it produces one name every 0.10 second. On the contrary, in a burst experiment for name addition, if I is 10 seconds, the name generator produces 100 new names (if $\lambda * I = 10 * 10 = 100$) as fast as possible at the start of the 10-second interval. After finishing 100 names, it generates no names until the next 10-second interval. Although the overall request rates in these two cases are the same (100 per 10 seconds), the performance of Solar may be quite different.

In our experiments, we needed several name generator machines and query generator machines so that we could generate requests fast enough and so that Solar would be faced with concurrent requests. Note that λ is a total request rate; since every generator generated names or queries at the same rate, each of n generators worked at rate λ/n. We chose to require $N_p = N_H$ as discussed above, although it is possible to run multiple Planets on the same host.

3.5 Experiment setup

All the experiments ran on the department's Linux cluster (Jefferson; Figure 3.5). Because the experiments are time sensitive, this cluster can provide an exclusive environment. We had no automatic mechanism to ensure that no other people were using the cluster, so we built a cluster mailing list to arrange times for exclusive use of the cluster. Then we can avoid the sharing of machine hardware resources to affect the experiment results. In this section, we briefly introduce our cluster configuration.

3.5.1 Hardware configuration

Jefferson[1] is a 32 node cluster machine with a head node for management. It has 64 processors, that is, two processors for each node. Each node runs RedHat Linux 9 and Java virtual machine 1.4.1. Each node has 4GB RAM, an Ultra320 36GB hard drive, gigabit Ethernet, and pretty blinking lights. See Figure 3.5 and Table 3.2.

3.5.2 Experiment setup

Considering the four aspects, different Planet numbers, different request distributions, different name actions and different name strand numbers, we briefly introduce the network for our experiments. In our experiments we always ran 30 generators on 3 hosts, so there were 10 generator instances on each host.

As an example, consider the 30 generators and 5 Planets shown in Figure 3.6. The Jefferson master node controls the experiment flow, and five Planets run on five hosts separately. Suppose they are Jefferson node1 to node 5, although actually we did not use the hosts with the exact same numbers in the experiments. Jefferson node 11 to node 12 ran multiple generator instances; each generator process connects to a specific Planet.

[1] http://jefferson.cs.dartmouth.edu

Figure 3.5: Jefferson’s portrait.

processor	0	1
vendor_id	GenuineIntel	GenuineIntel
CPU family	15	15
model	2	2
model name	Intel (R) Xeon CPU 2.80GHz	Intel (R) Xeon CPU 2.80GHz
stepping	7	7
CPU MHz	2795.330	2795.330
cache size	512 KB	512 KB
fdiv_bug	no	no
hlt_bug	no	no
foof_bug	no	no
coma_bug	no	no
fpu	yes	yes
fpu_exception	yes	yes
CPUid level	2	2
wp	yes	yes
bogomips	5583.66	5583.66

Table 3.2: Configuration of each node's CPU.

To monitor the CPU utilization for every host (both generators and Planets), we ran a little program on each host. When the experiment began, the master first ran `top` for each host by running a shell script, then started the Planets automatically, and then starts up the generators. The `top` program recorded the CPU utilization for its Planet host every 5 seconds.

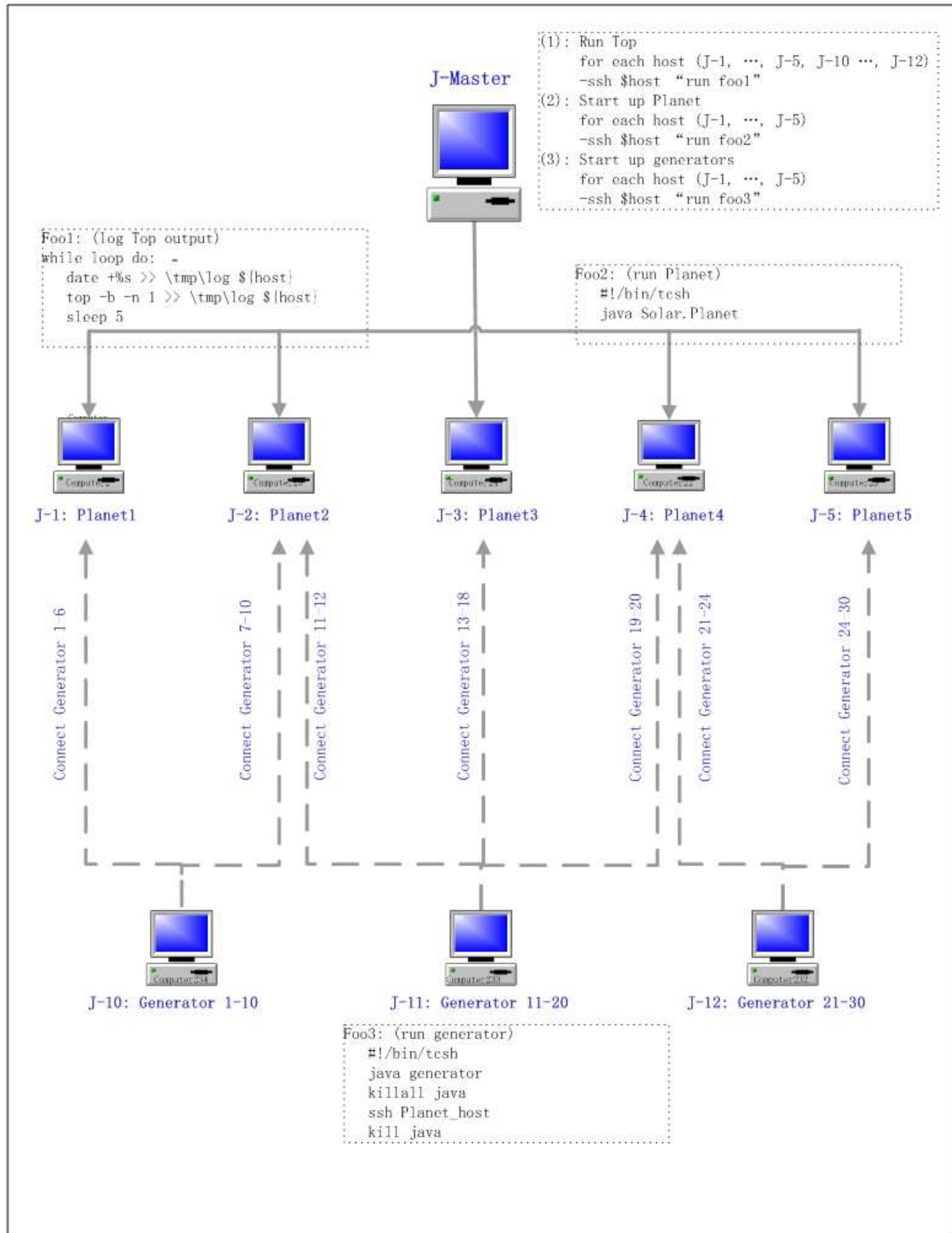


Figure 3.6: A network setup illustration for the case 30 generators and 5 Planets. Dashed text boxes are some scripts, using pseudocode.

Chapter 4

Experimental Results

In this chapter, we present and explain our experimental results. For each of our four experiment aspects, we use one section to discuss the purpose of these experiments, the values and ranges of all the parameters used in the actual experiments, and the plotted results. First we discuss how we measure the real request rate, because it often differed from our desired request rate.

4.1 Request rate

One issue we should explain before presenting the result plots is the request rate. In the experiments, we used 30 generator processes to generate new requests to Solar at a particular rate, say, 3 requests per second. The generator issued a request, waited for the reply, slept until $1/3$ second had passed since the time the prior request was issued, and issued the next request. In each new experiment, we increased each generator's request rate by 1, so that the total request rate increased by 30. When we examined our data, however, we found that our desired rate was often not achieved. For example, suppose the experiment involved 15 Planets and 30 generators, 3 strands per name, and a uniform request rate of

name addition. As the desired request rate increased from 3 to 23, the interval between requests decreased and became smaller than Solar's response time. We plot the relationship between the request time and the sequential request number in Figure 4.1.

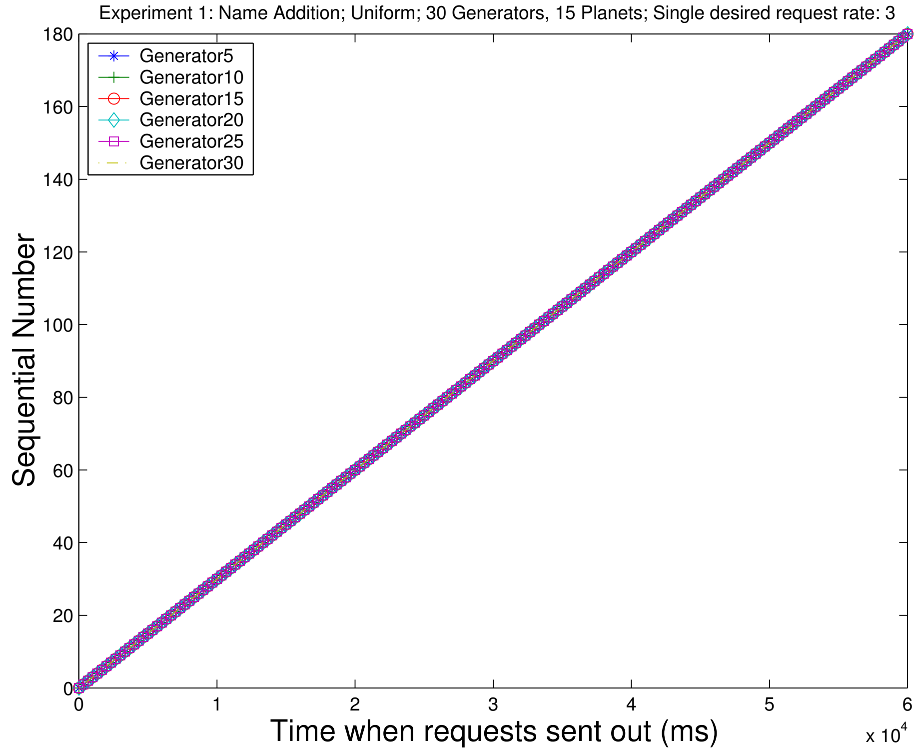
Figure 4.1 (a) and (b) show the desired rates 3 and 23 respectively. For clarity, we randomly chose a subset of generators; otherwise the plots would be messy with all 30 generators. When the desired rate was low, like 3, the real rate was almost equal to the desired rate, so all Planets have a diagonal line reflecting the desired rate. On the other hand, plot (b) shows that the real request rate deviated from the desired rate the first few requests, because Solar became busier with higher load and its response time increased, thus the name generators were limited to send out more requests.

To calculate the real request rate, we saved several logs in the processing procedure, shown in Figure 4.2.

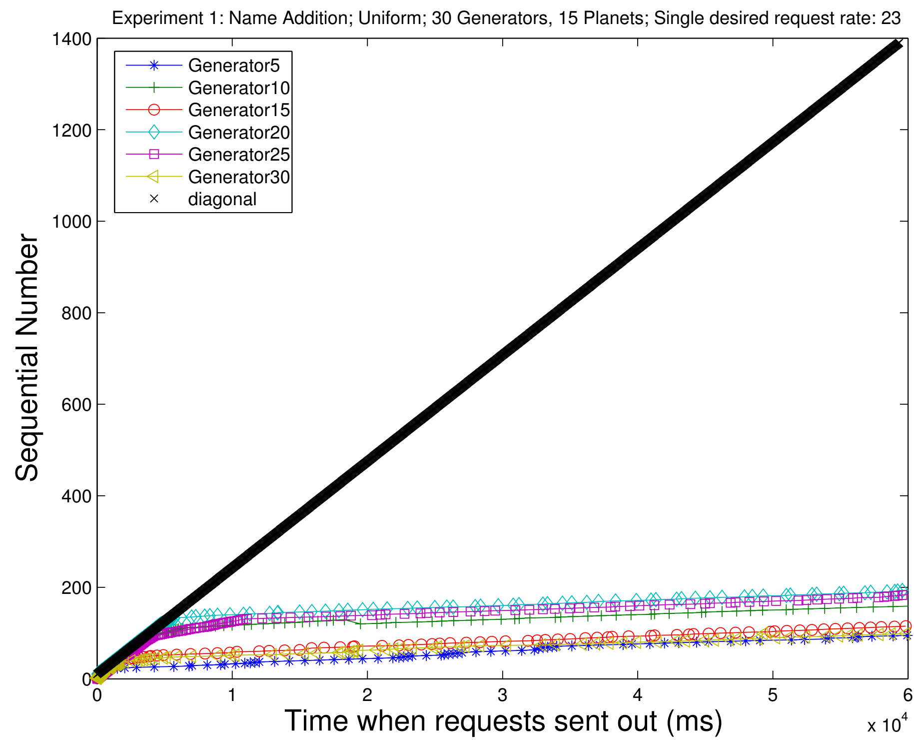
We log the subscribe operations, the names generated and the time submitted to Solar, and the time the response came from Solar. Since we had carefully arranged for the sender and receiver to be located on the same generator host, clock skew was not an issue. Using the sender log and the receiver log, we computed the latency for every request. We counted the number of the successful request records in the receiver log and divided by the running time for one trial, 60 seconds in our case. So we calculate the real request rate for every trial. In the following result plots, we use the real request rate rather than the desired request rate.

4.2 Number of Planets

In this section, we plot the results of our experiments where we varied the number of Planets. To estimate how Solar scaled with the number of Planets, we ran experiments with 5, 10, and 15 Planets. We first present the results in which we discovered an unusual



(a)



(b)

Figure 4.1: Results for real request trend.

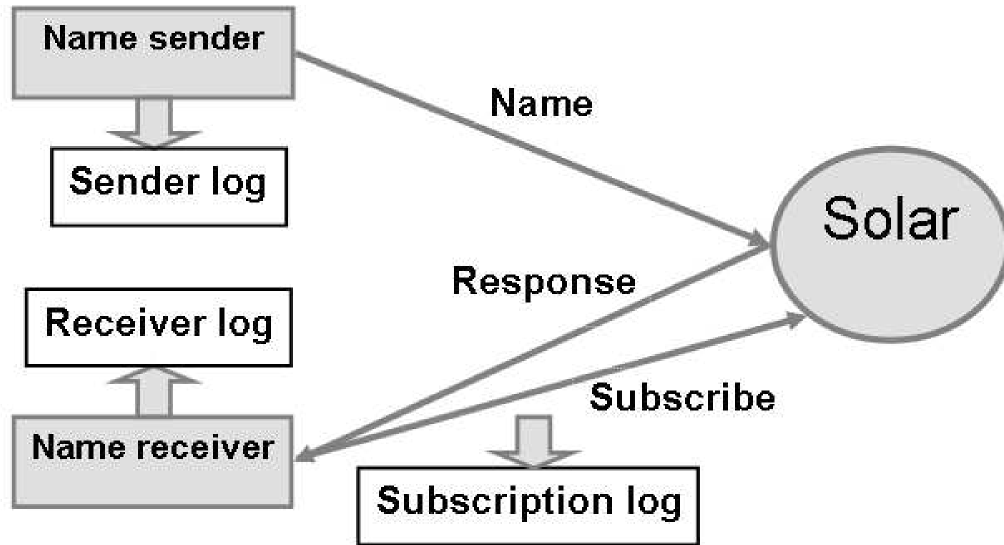
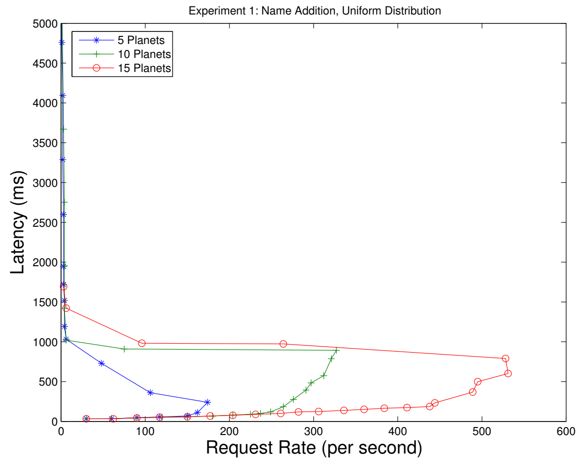


Figure 4.2: The logs collected in our experiments.

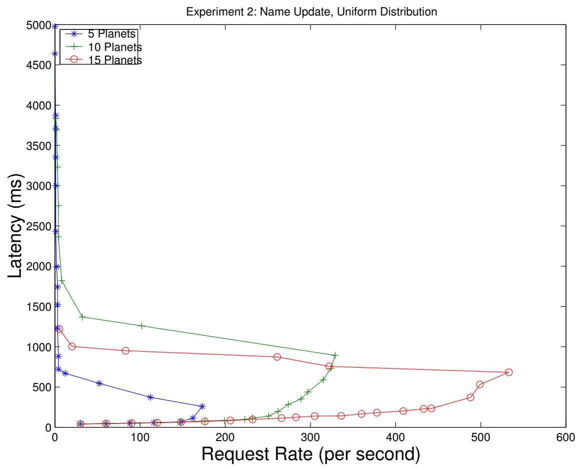
phenomenon. We then ran additional experiments to dig into the details.

In all the experiments, we increased Solar's capacity by adding working nodes (Planets), and also increased the workload to Solar by adding to the request rate. With these results, we may be able to choose a suitable number of Planets for a specific project, to make good use of available resources by neither starting up too many Planets nor too few Planets.

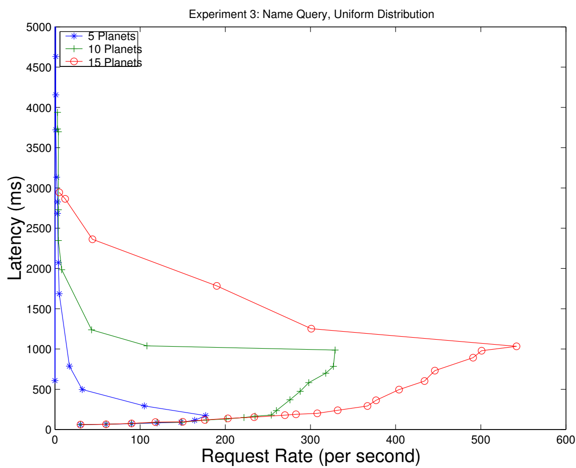
In all the tests, the request rate followed a uniform distribution, and the simulated names are from the first-version name generators, and all names had 3 name strands. We tested all the different name actions: name addition, name update, and name query. For each of the three name actions, we have a single plot in Figure 4.3 to show the results, where y-axis is average latency across all receivers and x-axis is total real request rate. We increased the desired request rate from every name generator by 1 request/second; thus the total increment of the desired request rate was 30 request/second. Note that x-axis is real request rate and may not reflect such uniform increase. The lines connecting points are drawn in the order of increasing desired request rate.



(a)



(b)



(c)

Figure 4.3: The results from the experiment with different number of Planets, for name addition (a), update (b), and query (c).

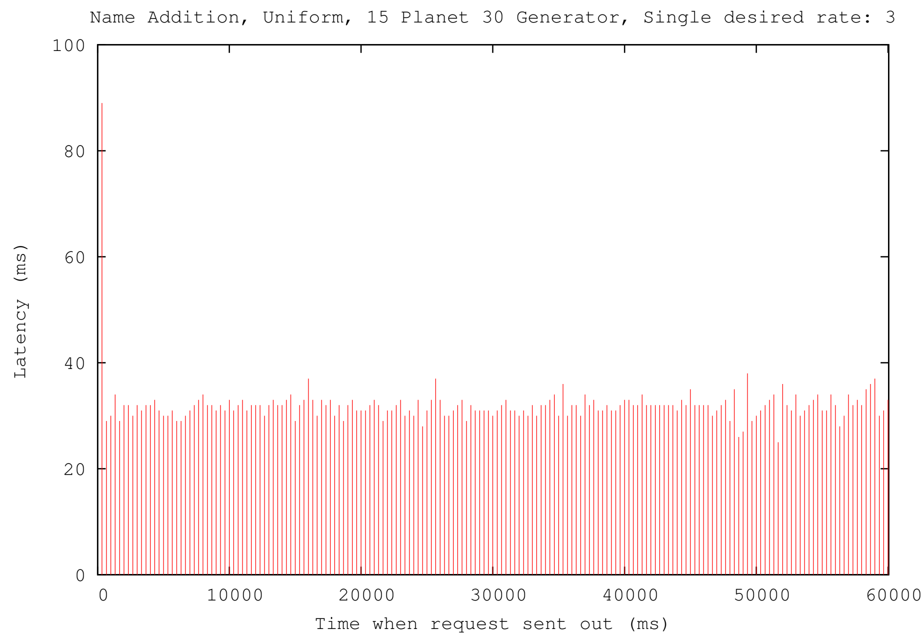
On each plot we compare the results of different numbers of Planets. Our test bed limited us to 15 Planets, but 15 was enough for us to see how Solar reacted to increased request load and varying capacity. Every curve has a similar trend; for low request rates the latency was low and increased slowly with increased request rate; at some point the curve turned backward with a larger latency increase and decreased real request rate.

As we expected, Solar could handle higher request load with more Planets. We found that 5 Planets could handle between 150 and 200 requests per second; 10 Planets could handle between 300 and 350 requests per second; and 15 Planets could handle between 500 to 550 requests per second.

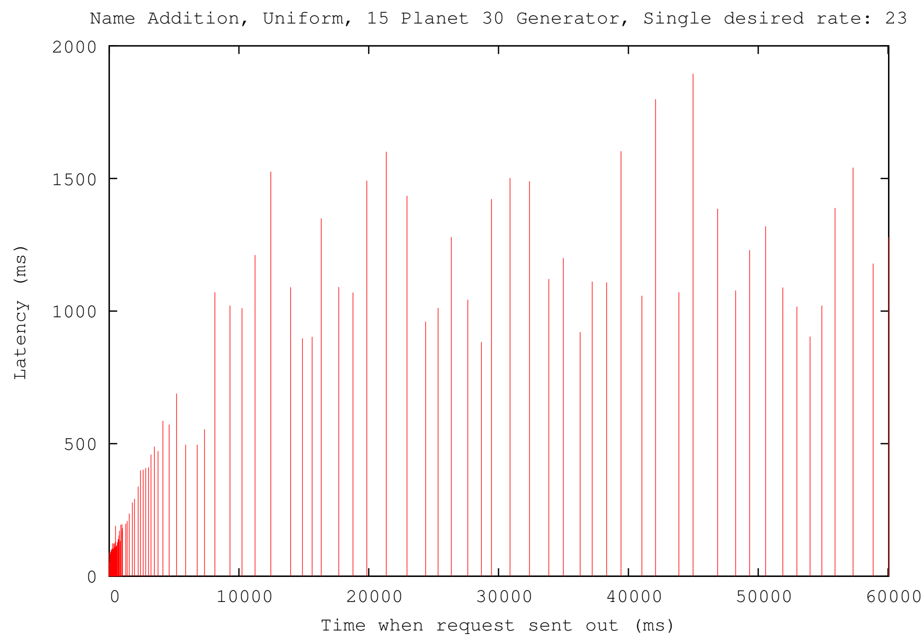
The backward turning curves, however, are rather surprising, since we expect the latency should stabilize as the request rate reached Solar's capacity. Examining further the "backward" curves in these plots, consider plot 4.3 (a). Both the desired rate 3 requests per second and 23 requests per second for 15 Planets had similar real request rates, about 90 requests per second. Oddly, the receivers perceived much different latency even though Solar was under similar load.

We plot these two cases in Figure 4.4, which shows the sequence of latencies experienced by one of the generators. Figure 4.4 (a) shows the relationship of the request start time and latency when the desired rate was 3 requests per second. Generally the latency for each request was similar with small variation. The first request clearly had a long latency, because Solar needed to do some initialization and connection setup for the first request.

Figure 4.4 (b) shows the results when the desired rate was 23 requests per second. We can easily see the requests' increasing latency in the first several seconds. Why does the latency increase? With higher request rates, Solar's incoming request queue started to accumulate as multiple generators raced to put in their requests. Later requests had to wait for previous ones to finish, and thus effectively had a longer latency. As Solar became busy and started to push back on the generators, it should be able to maintain a roughly constant



(a)



(b)

Figure 4.4: Examining the latency of individual requests, under low (3/second) and high (23/second) request rates.

Number of Planets	Single desired rate	Request loss ratio
5	7	0%
	8	0%
	9	1.9%
	10	18.4%
	11	23.5%
10	17	0.2%
	18	1.8%
	19	14.6%
	20	17.0%
	21	19.6%
15	20	0.2%
	21	0.1%
	22	0.7%
	23	3.1%
	24	18.6%

Table 4.1: Request loss ratio measured at a particular receiver for each experiment.

processing rate that led to a stable average latency. However, we saw different behavior in Figure 4.3. At the same time, we found another strange phenomenon; that was there were some dropped requests during the transition, shown in Table 4.1. Since every name had a unique identification number, we compared the sender's log and receiver's log by matching the name ID, and counted the dropped requests.

Our Planets used TCP connection and none of the Planets crashed during our experiments. The loss of requests thus might due to Solar's internal timeout facility. Namely, the party sending the request would time out and drop the request if it had not received an acknowledgment about that request for a certain threshold (5 seconds in our case). This is consistent with what we observed before, which means that the latency of completing a request increased beyond the threshold and resulting in dropped requests and reduced success request rate.

So why did the latency increase so much? We wanted to identify the bottleneck in Solar. So we ran `top`, a monitoring program, on each host and we configured `top` to

sample the local CPU utilization [1] every 5 seconds. Figure 4.5 plots the data from `top` for three pairs of cases. They are desired rate 2 requests per second and 9 requests per second for 5 Planets, desired rate 2 requests per second and 18 requests per second for 10 Planets, and desired rate 3 requests per second and 23 requests per second for 15 Planets. These pairs of cases all had a similar real request rate but different latency. Every `top` plot has an x-axis range of 60 seconds because every experiment ran for one minute. The y-axis shows the measured CPU utilization in percentage.

From Figure 4.5, we found that the Planet CPUs were not under high load, though indeed somewhat higher for high desired rate than for low desired rate. Although we may miss some higher peaks of CPU utilization due to our 5-second sampling rate, it is obvious that the CPUs were not consistently heavily loaded in any case. When the desired request rate increased, the CPUs' load increased then resulted in some latency increase, but not enough to account for the large latency. What other sources could have triggered the latency? It could not have been the generator CPUs since Figure 4.6 showed that the CPUs for generator hosts were not overladed in the experiments. It also could not have been the network, which was a 100 Mbps switched Ethernet within the cluster.

Consider how a Planet processes a name; it might involve multiple Planets since a name with multiple name strands will be sent to other Planets for processing. In our experiment, each generator was associated with a particular Planet in that it sent requests to a specific assigned Planet. Then the receiving Planet may forward requests to other Planets for cooperative processing. Every Planet had a single queue to store the requests before they were processed. There were multiple threads to handle the name requests for each queue. Generally, when many generators connect to one Planet and the desired rate was high, the queue fills quickly and the requests in the tail would have high latency.

[1] We measured CPU idle fraction with `top` and subtract from 1 to obtain utilization percentage. Although `top` itself possessed some CPU utilization, however, we did not need to know the exact CPU utilization and thus `top`'s occupation could be ignored in our case.

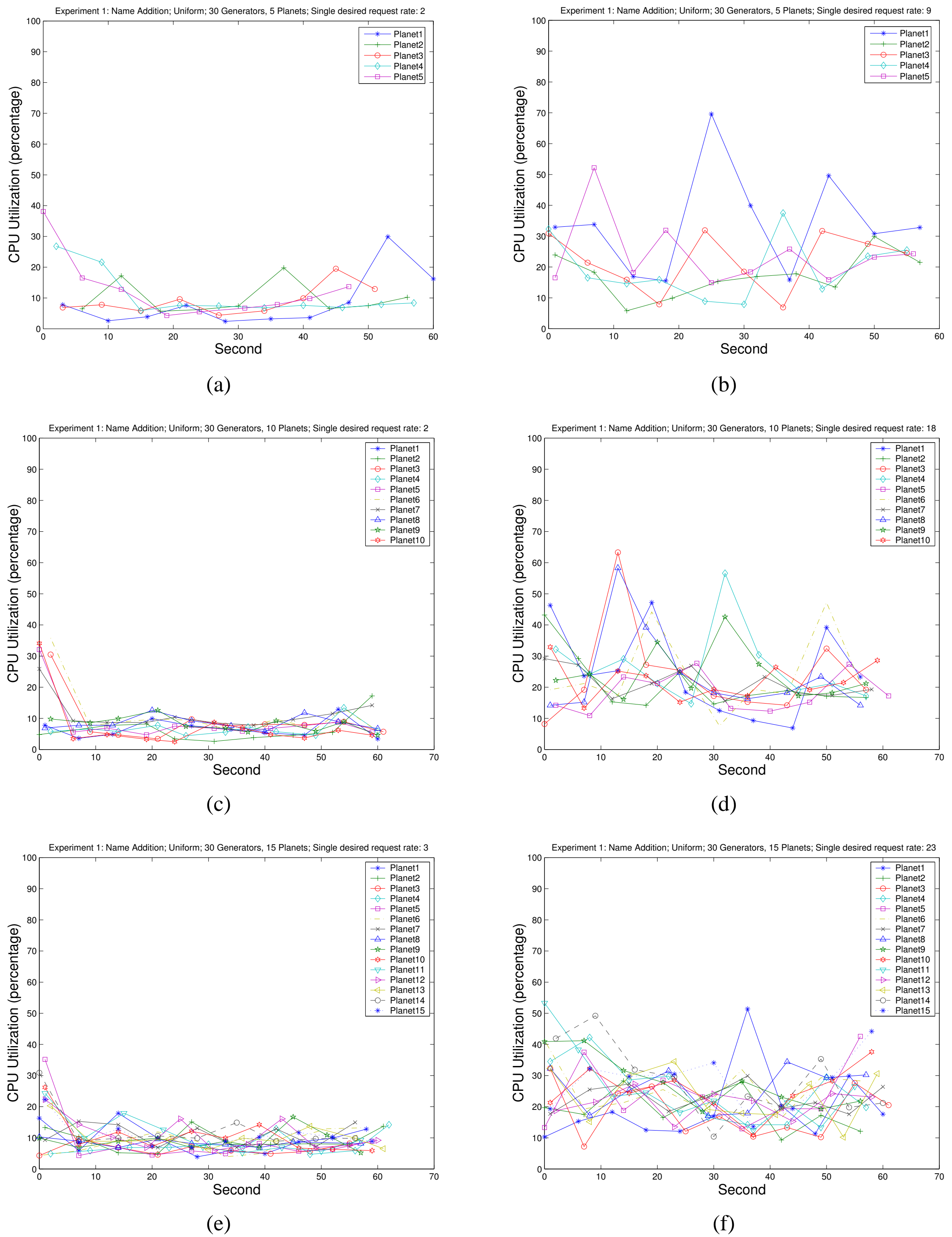
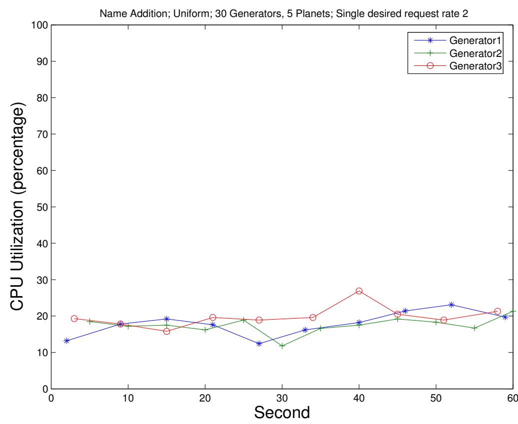
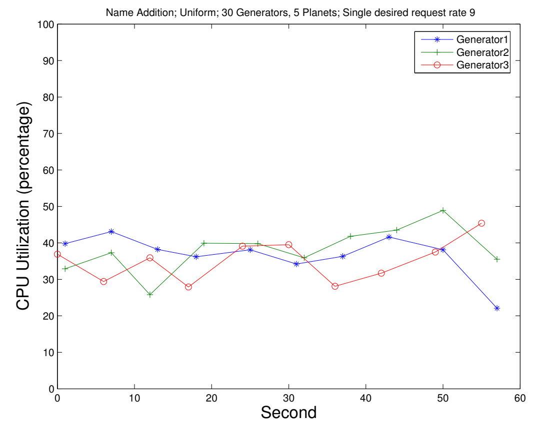


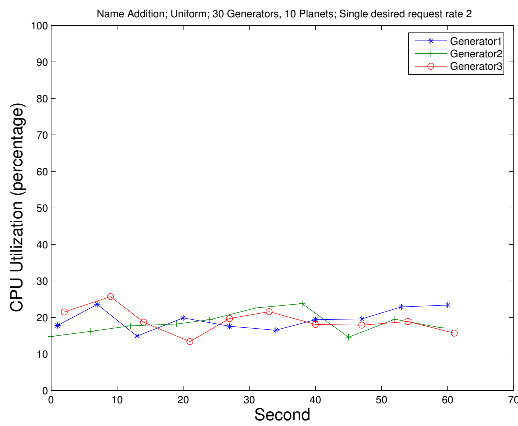
Figure 4.5: The Planet `top` results for three pairs of cases.



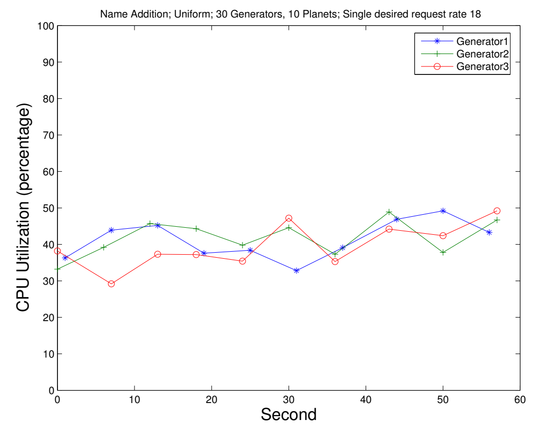
(a)



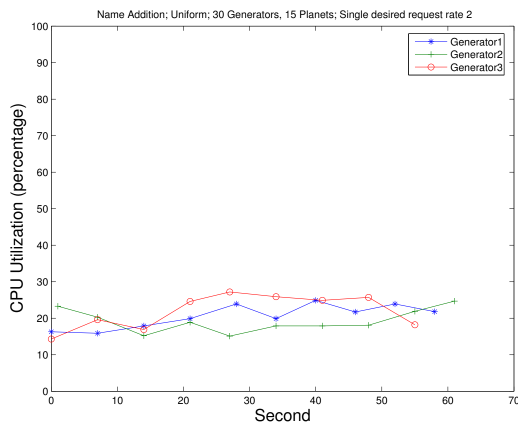
(b)



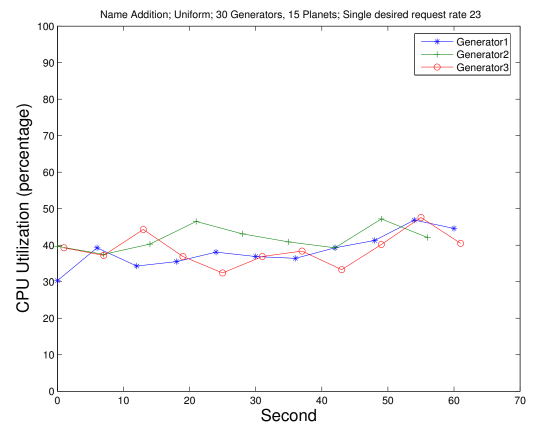
(c)



(d)



(e)



(f)

Figure 4.6: The generator `top` results for three pairs of cases.

We performed a separate experiment to measure the performance of that threadpool. In the main thread of our test process, we used a tight loop to inject 500 requests into the 5-thread threadpool (loaded from an external library that used Java threads). We simulated request handling by sleeping 20 milliseconds. We found that the first couple of requests had large latency due to creation of new threads, as shown in Figure 4.7. Then the latency stabilized as requests filled threadpool's queue. After some period, however, some requests had an abnormally large processing time. The reason for this phenomena is unfair thread scheduling. Namely, some threads (requests) were unfairly held in the pool longer than other threads before they got a chance to finish, which means it is not a strictly round-robin thread scheduling policy. This behavior could be caused either by JVM or the threadpool library itself. In Java 2 Release, the Hotspot virtual machine uses system threads to implement Java threads. Because Linux threads are essentially implemented as a cloned process,[2] and process priority is dynamic in Linux, the scheduler keeps track of all processes (not only our test threads)'s activity and adjusts their priorities periodically, and could not guarantee round-robin policy for our threadpool.[3] The threadpool we used is an external module and we could not confirm whether it also contributed to the biased scheduling.

Also, we expected that the cooperative name-processing procedure may be a factor. Note that the requests coming from the generators and the requests from other Planets go through the same queue and thus may result in longer latency. For example, in the Figure 4.8, Planet A forwards Request 1 to Planet B, C and D for cooperative processing. Because of the higher desired rate, queues for those Planets are accumulated, so Planet B, C and D can not process this request immediately. It means that Request 1 will not be finished until all the requests before Request 1 in those queues are processed. At the

[2]http://java.sun.com/developer/technicalArticles/Programming/linux/
[3]http://www.oreilly.com/catalog/linuxkernel/ chapter/ch10.html

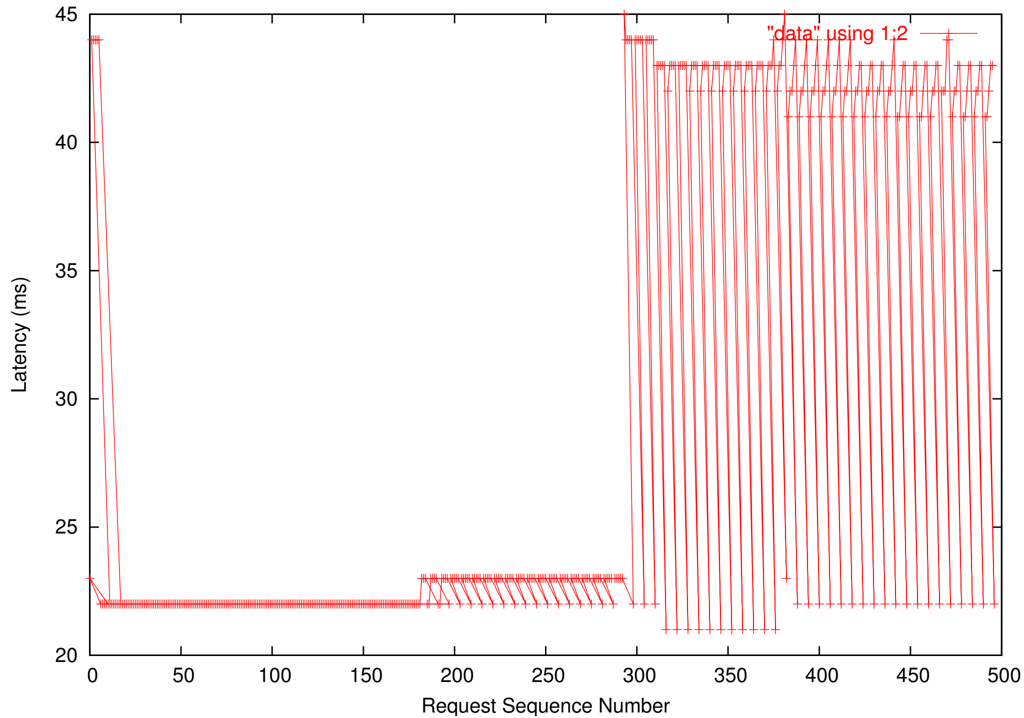


Figure 4.7: Latency behavior of Solar's request handling thread pool.

same time, other requests before Request 1 also need a cooperative processing by other Planets. Consider the case that many Planets forward requests to some Planet, e.g., Planet B and D forward their requests to Planet C, then Planet C has longer queue than any of others. Thus it may lead even longer latency and probably timeout happens. We believe that the cooperative name processing, which forced requests to go through multiple queues contented with other requests, had a big impact on the request processing latency.

As a possible tail problem from this cooperative name-processing procedure, a potential "deadlock" may happen because that multiple Planets forward requests to each other at the same time and mutually wait other parts to finish the processing. This situation, not surprisingly, leads to reduce the number of successful requests and increase the latency largely. We do not, however, expect this deadlock would happen frequently since each Planet had a thread pool (5 concurrent threads) handling incoming requests. Even this

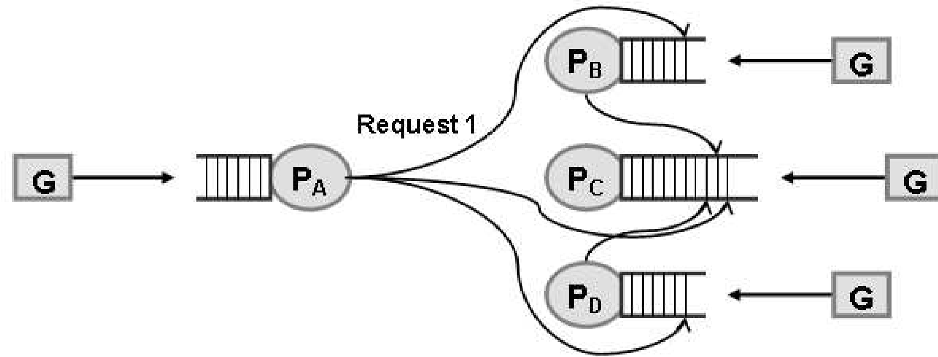


Figure 4.8: Cooperative name processing procedure.

situation happens in the name processing, it is not like the classic problem in operating system [10, 37, 18]. It is not either necessarily, complete or forever. It is not necessarily because deadlock will only occur if all the threads are blocked and form circular waiting, although higher request rate and more name strands make the probability higher. And it is not complete because other Planets may still work after two or some Planets happen this situation. Also it is not forever because Solar can get this situation away by passive expiration. As the active solution, two queues on each Planet, one for "external" requests directly from generators and the other one for "internal" requests from other Planets, and high priority on the internal queue may work.

In summary, we found that there were several sources that caused large request latency, which eventually led to request timeout and decreased successful request rate. All the latency sources, busier CPU load, biased thread scheduling in the request handling pool, and contentions of multiple queues that the request had to go through, are all legitimate reasons causing the backward curves in Figure 4.3. We could not, however, identify exactly which one or combined ones happened due to the lack of a fine granularity logging facility in Solar. We do recommend Solar use separate queues to handle "internal" and "external"

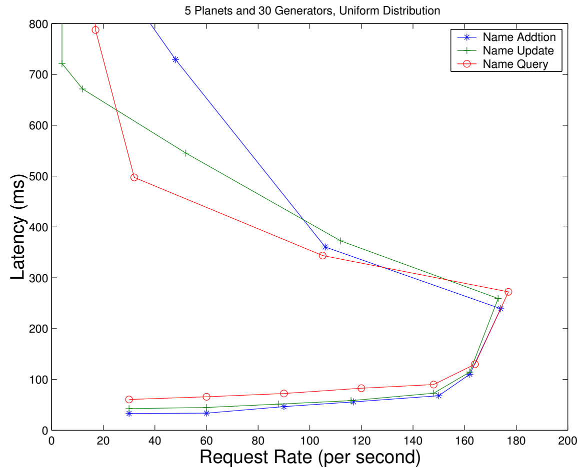
requests, and employ a thread pool using a round-robin scheduler.

4.3 Name action

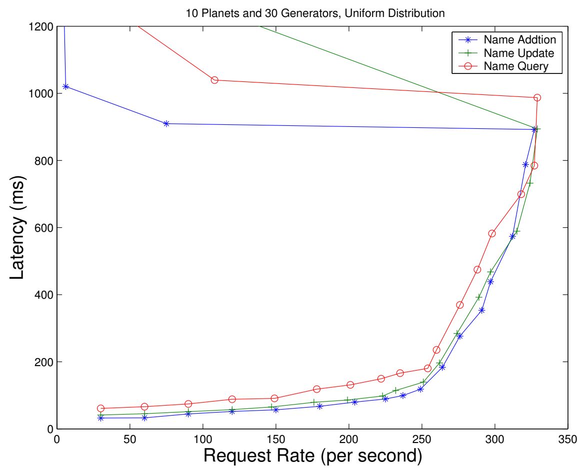
We have three name actions: name addition, name update; described in Section 3.1.3. These experiments help us to understand some of Solar's inner operations on different name actions.

We rearrange the result logs from previous experiments and compare the performance of different name actions, using a uniform distribution and 3 name strands. We show the results in Figure 4.9, focusing on the lower-latency portion of the result space.

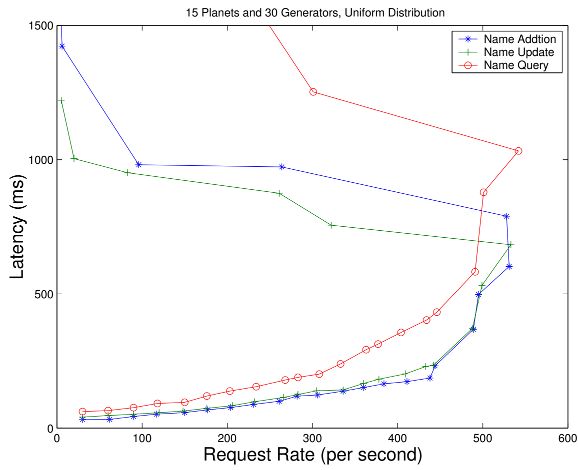
Solar has different operations for different requests: name addition, name update or name query, so naturally Solar needed different time to handle different kind of name actions. Although name update is similar to name addition, it has one additional operation: unadvertisement. Solar requires that any existing resources unadvertise its old name before it may advertise a new name. This additional operation resulted in the name update being slower than name addition. For name query operations, the name-matching operation dominated the whole transaction. Solar first filters some possible desired names for matching, and then matches the desired names with the query to select a name to return. Before the name query operation occurred, we warmed up the name directory by adding some names to Solar. In our case, the busier test had larger directory. Thus when the desired query rate was higher, the name directory in every Planet was larger, causing Solar to use more time to match queries.



(a)



(b)



(c)

Figure 4.9: The result plots for different name actions (focused on the lower latency portion of the results).

4.4 Request distribution

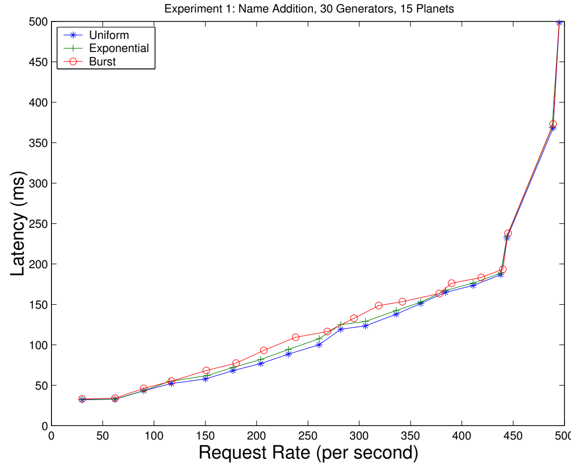
In addition to the request rate, we assume that the request distribution may also affect Solar's workload and that different distributions may lead to different performance. We designed three request distributions: uniform distribution, burst distribution, and exponential distribution. In the uniform distribution a generator sends requests evenly spaced in time (although, as we noted earlier, it may get behind schedule). In the burst case a generator sends requests one after another without pause, until the desired number per interval has occurred. The exponential distribution submits requests whose arrival times follow an exponential distribution of the given mean.

We ran experiments with 15 Planets and 30 generators, and 3 strands per name. For each of the name actions, (name addition, name update, and name query), we have a single plot with a curve for each of the three distributions within Figure 4.10.

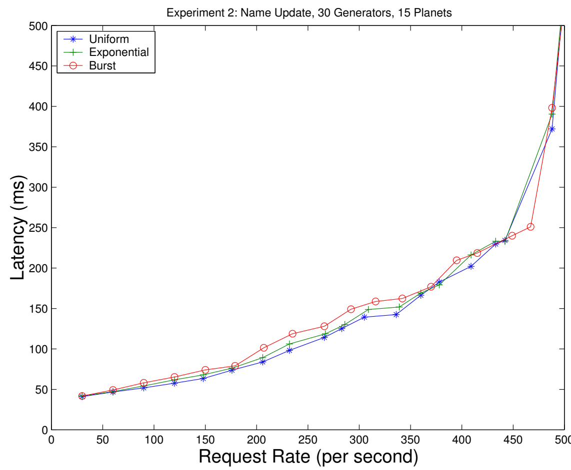
All three plots look similar, and in each the difference between distributions is more apparent when the request rate was the in medium range, about from 200 to 350 per second. On the other two ends, the difference decreases, which is not surprising. When the request rate was low, Solar was not congested and the latency remained low, because any bursts were easily absorbed. When the request rate was high, the queues were congested regardless of the distribution; indeed, the generators sent requests one after another without any pause and thus all distributions were effectively the same. Only a moderate request rate displayed the effect between distributions.

4.5 Number of strands

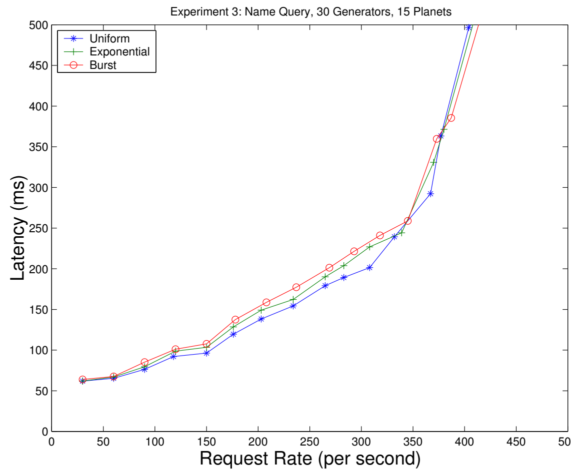
Because Solar several duplicate copies of each name, according to the number of name strands, the number of name strands may affect the overall performance. We expect that



(a)



(b)

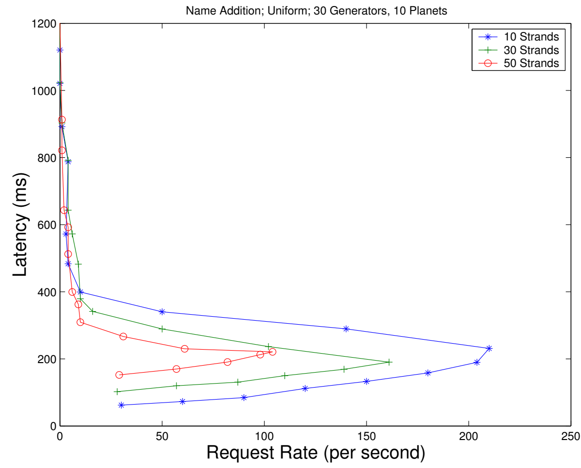


(c)

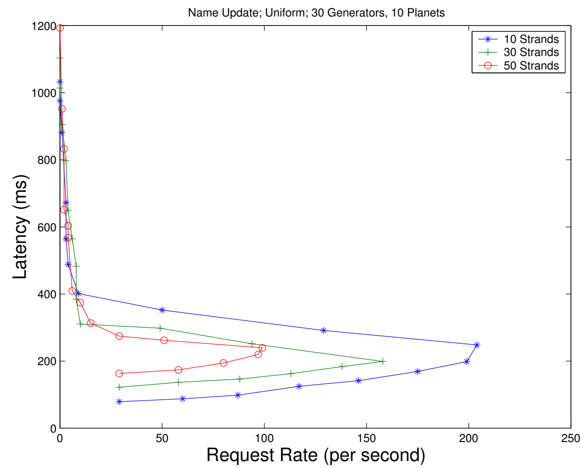
Figure 4.10: The results for different request distributions (focused on the lower latency portion of the results). The blue star line is for uniform, the green plus sign line is for exponential, and the red circle line is for burst. The x-axis is request rate with the unit of per second; the y-axis is latency with the unit of millisecond.

the more name strands, the longer latency, because Solar needs to do more operations on each name. We measured the performance of Solar when names had 10 strands, 30 strands, and 50 strands on 10 Planets, 30 generators and a uniform distribution with different name actions. Because we chose 10 Planets, on average each Planet had 1, 3 or 5 copies for each name. The results are shown in Figure 4.11.

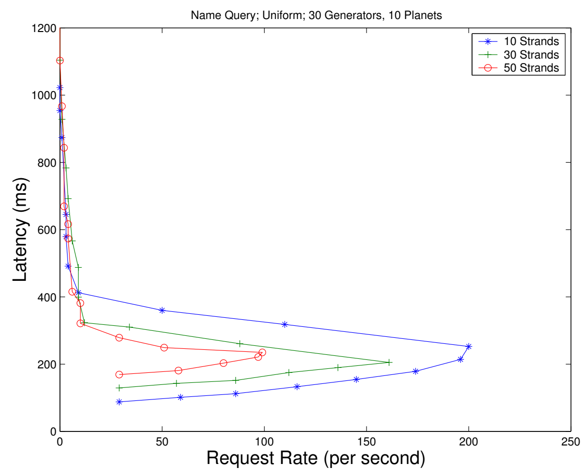
We used a lower desired request rate in this experiment, varied from 1 to 17 requests per second per generator, because names with more strands tended to use more Planets when processing, resulting in a more complex cooperative processing procedure. So we focus on the cases with the low or medium desired request rate. Compared with the 3-strand case, the larger names have lower scalability: the 10-strand case could be scaled to about 200 requests per second; the 30-strand case could be scaled to about 160 requests per second; and the 50-strand case was even lower, 100 requests per second. Not surprisingly, the name with more strands did require the processing of more requests, and thus longer queues, and thus latency was more affected by the many causes discussed in Section 4.2



(a)



(b)



(c)

Figure 4.11: The results for different numbers of name strands, for name addition (a), update (b), and query (c).

Chapter 5

Related Work

In this chapter, we describe two resource-discovery systems that are quite related to Solar. The Intentional Naming System (INS) is a resource-discovery and service-location system for mobile devices. INS/Twine is a scalable resource-discovery system, an upgraded version of INS. They both let *resolvers* collaborate as peers to distribute resource information and to resolve queries. In our description, we focus on their performance evaluation methods and results.

5.1 INS's Evaluation

The Intentional Naming System (INS) [1] is a distributed resource-discovery and service location-system for dynamic and mobile networks of devices and computers. INS uses a simple language based on attributes and values for its names. Applications use the language to describe what they are looking for, i.e., their *intent*. Thus, names are *intentional*; they describe application intent in the form of properties and attributes of resources and data, rather than simply network locations of objects, which is the way most traditional naming systems like the DNS work today. To learn and share information about names, INRs

(Intentional Name Resolvers) communicate via a name discovery protocol. The protocol uses periodic updates to disseminate name changes, and uses triggered updates for fast changes. INS saves names in the directory, a copy of which is on every resolver of INS. Although INS has many interesting characteristics, here we consider their performance evaluation method, rather than the resource discovery service itself.

The paper [1] analyzes the performance of the INS name-lookup (query) algorithm and presents the results of their experiments with the lookup algorithm and name discovery protocol. They first define some key parameters in their experiments: d, one-half the depth of name-specifiers, which are the intentional name expressions used by INS; r_a, the range of possible attributes in name-specifiers; r_v, the range of possible values in name-specifiers; and n_a, the actual number of attributes in name-specifiers. Second, they constructed a large random name tree for name space, and timed how long it took to perform 1000 random lookup operations on the tree. The name specifiers and the name trees were uniformly chosen with the same parameters as in the above analysis. Then they varied n, the number of distinct names in the tree and measured the lookup times. They fixed the parameters at r_a = 3, r_v = 3, n_a = 2, and d = 3, and varied n from 100 to 14300 in increments of 100. After experiments, they found that for this name-tree and name-specifier structure, their performance went from a maximum of about 900 lookups per second to a minimum of about 700 lookups per second.

For the name discovery protocol, they measured the performance of INS in discovering new service providers. In their system, when an INR observes a new name-specifier from a service advertisement, it processes the update message and performs a lookup operation on the name-tree to see if a name-specifier with the same AnnouncerID already exists. If it does not find it, it grafts the name-specifier onto its name-tree and propagates a triggered update to its neighbors. Thus the name discovery time, $T_d(n) = n \times (T_l + T_g + T_{up} + d)$, where T_l is the lookup time, T_g is the graft time, T_{up} is the updating processing time, d

is the one-way network delay between any two INR nodes and n is the number of hops when propagating the name. They found that typical discovery times are only a few tens of milliseconds, and dominated by network transmission delays. And they concluded that $T_d(n)$ is indeed linear in n, with a slope of less than 10ms/hop.

They also measured the performance of the overall system when both name lookup and name discovery occur simultaneously. They introduced a *virtual space*, a partition mechanism, to reduce the volume of periodic advertisements needed to maintain state in the system and to isolate disjoint parts of the namespace. There are three cases: local destination, in which sender and receiver are on the same node; same virtual space, in which sender and receiver are not on the same node but in the same virtual space; and different virtual space, in which sender and receiver are in the different virtual space. They sent a burst of messages at 15-second intervals. For the first case, the processing time varies from 3.1 ms per packet with 250 names to 19 ms per packet with 5000 names. For the second case, the next-hop processing time is about 9.8 ms per packet during the burst. For the last case, they found a time of 381 ms to resolve and route the burst of 100 messages.

5.2 INS/Twine

With an increasingly large and dynamic computing environment comes the challenge of a scalable resource discovery service. INS/Twine was designed to update INS by achieving a large scalability, although INS itself has some scalability mechanisms, such as Domain Space Resolvers (DSRs) and Partitioned Virtual Spaces [22]. The INS/Twine discovery scheme was the inspiration for Solar, although they use different substrate routing protocols. As with Solar, INS/Twine splits each name into stands and saves the name on each node identified by the key resulting from hashing each strand of the name. Since INS/Twine discovers names in a similar way, the evaluation of INS/Twine is definitely related with our

work.

INS/Twine [2] evaluates its strand-splitting algorithm by examining the distribution of strands from splitting real resource descriptions, then examining how data and queries are distributed among resolvers, and finally evaluating the query success rate in the presence of failures.

For the strand-splitting algorithm, their goal was to determine how many strands are produced by such descriptions and how often the same strands come up. They use two data sets: the first one is bibliographical entries from some Latex files and the second one is a directory of tagged MP3 files. They found that bibliographical entries contain 12.9 strands on average whereas MP3 tags produce an average of 8.7 strands. Also they found that some strands are more popular than others. Nine strands appear in over 10% of bibliographical descriptions and three strands come up in over 10% of all MP3 files. Thus they concluded that their splitting algorithm splits real descriptions into a reasonable number of strands and that most descriptions are composed of unique strands.

For the resource distribution, their goal was to evaluate the quality of data distribution in INS/Twine. They ran some Intentional Name Resolvers (INR), and connected some client applications to the resolvers. Each client application inserted some resource descriptions in different numbers. They found that over half of the resolvers hold information about less than 15% of resources for both data sets. Based on this discovery, they said increasing the proportion of number of strands to number of resolvers increases the fraction of resources known by each node. They compute the expected value for the fraction of resource information stored at each resolver to be $(SK)/N$, where S is the average number of strands in resource description, K is the configurable replication level, and N is the number of resolvers in the network. After the experiments, they concluded that data is evenly distributed in INS/Twine with each resolver holding only a small subset of resource descriptions.

For a given query distribution and success rate, their goal was to find the query success

rate as a function of the fraction of failed resolvers. They selected some descriptions as queries and submitted them to randomly selected resolvers. Then they shut down some resolvers in different numbers and counted the success of queries. In the worst scenario, in which only one strand was randomly selected from each resource description to serve as query. They calculated the query success rate, using the formula $(F/N)^k$ where F is the number of failed nodes, N is the number of total number, and k is the replicating level.

5.3 Comparison

INS, INS/Twine and Solar have many similarities. They are all attribute-based distributed resource-discovery systems, with directory registration and de-registration. The INS paper discusses name discovery performance, using a latency metric similar to the name update latency metric we used in our evaluation (Chapter 3). Our name-strand number experiments are inspired by the name-strand performance experiment in INS/Twine.

However, there are some differences between these evaluations. Considering the scalability experiments first, the INS evaluation includes a name-lookup performance experiment, in which the researchers build a name tree (name directory) on a local node and scale the name tree size from 100 to 14300 names. We also focus on scalability, by scaling the name request rate other than scaling the name directory volume as INS did. So the INS experiments involved only local operations in their scalability experiments, but we evaluated Solar's resource discovery using network operations. Then INS measured name discovery performance by using the latency metric counting time from the name begins to be processed to name is ready to send out to the applications, which is similar with our name update latency metric. The last INS experiment, involves virtual spaces and is thus not closely related to our evaluation.

In INS/Twine, the name-split algorithm performance is tested by inputting two kinds of

files, the data distribution experiment focused on the fraction of resource information stored at each node, and query-success measured how many queries can be correctly completed with some node failures. That is, INS/Twine did not measure scalability directly although it was designed to increase scalability. Indeed, they did not do any speed testing at all. Thus it is hard to compare our methodology with theirs and we would say these evaluations are complementary.

5.4 Others

Jini is a Java-based infrastructure that provides a directory protocol for service registration and a discovery protocol for service binding [38]. The registration is attribute-based and the directories can be federated. Domain Name System (DNS) is used to locate Internet services and it uses a hierarchical naming structure [26]. DNS is not designed for pervasive-computing environment, thus it can not tolerate frequent name updates. Like DNS, Jini is designed for Internet-based services originally (requires multicast and full-blown JVM) [16], though it has potential to be used for pervasive-computing small devices, using recently proposed Jini Surrogate Specification (see [20]). Since neither systems uses an overlay-based directory distribution, our evaluation framework and results may not apply directly to Jini and DNS.

Chapter 6

Conclusions and Future Work

In this thesis, we evaluate the performance of Solar's resource-discovery service. In the previous chapters, we define the problem scope, introduce the pertinent background information, described the methodology, discuss the experimental results and present some related work. In this chapter, we summarize our contributions, discuss future work and some limitations of our current approach, and conclude.

6.1 Contributions

This thesis makes several contributions.

- We evaluated the performance of a distributed resource-discovery service with a focus on scalability, using a large number of requests. We also tested how the service handles a dynamic namespace, which is unusual in existing literature. The scalability and naming dynamics are important in a pervasive-computing environment.
- We built a configurable test framework that generates common naming operation requests. We believe that framework, with some adaptation, could be used to eval-

uate other resource-discovery services, allowing a performance comparison across different services.

- During our performance study, we discovered an implementation issue in Solar's resource-discovery service, and offered a possible solution. We believe that this may be a common issue for distributed cooperative systems.

6.2 Future work

There are some opportunities for improving or extending our work.

- In our experiments, we used multiple generators and multiple Planets, which run on multiple hosts. These processes did not start up at the same time, which may have led to some noise in the data. For example, the Planet may not be tested with the desired request distribution as the generators may have started slightly late.
- Our generators could not achieve a high desired request rate; the best was only about 25 requests per second for each generator process. One reason is that the granularity of Java's timer is about 20ms, and the generator can not sleep shorter than the interval. In the future work, we may refine the generators to improve the generating efficiency.
- In our experiments, we ran one Planet per host although several Planet instances can run on one host at the same time. We did not do any experiments on multiple Planets per host. It may be interesting to explore.
- Planets are involved in other business, e.g., operators and event flows. These other functions of Planets may affect the resource-discovery service. In the future work, we may design some experiments to figure out do these modules affect resource-discovery or vice versa.

- It would be useful to evaluate Solar under a real workload, rather than the synthetic workload presented by our generators.

6.3 Conclusion

Many pervasive-computing applications need to locate resources (sources of information, services, or local devices) and thus require a resource-discovery service with sufficient flexibility and speed. A detailed evaluation of a resource-discovery service reveals the overall performance and helps to discover its limitations. We evaluated Solar's resource-discovery module with a focus on scalability and naming dynamics. We explored four aspects: different numbers of Planets, different request distributions, different name actions, and different numbers of name strands. The results from these experiments show that Solar's resource discovery performed generally well in a dynamic environment, and that it can be scaled to different degrees by starting up different numbers of Planets or using names with different numbers of strands. On the other hand, we also discovered some issues in Solar's resource-discovery service, which led to high request latency, and offered some potential improvement suggestions.

Bibliography

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 186–201, Kiawah Island Resort, South Carolina, December 1999. ACM Press.

[2] Magdalena Balazinska, Hari Balakrishnan, and David Karger. INS/Twine: A scalable peer-to-peer architecture for intentional resource discovery. In *Proceedings of the First International Conference on Pervasive Computing*, pages 195–210, Zurich, Switzerland, August 2002. Springer-Verlag.

[3] Guanling Chen. *Solar: Building a Large-scale Context fusion Network for Pervasive Computing Environments*. PhD thesis, Department of Computer Science, Dartmouth College, July 2004.

[4] Guanling Chen and David Kotz. A survey of context-aware mobile computing research. Technical Report TR2000-381, Department of Computer Science, Dartmouth College, November 2000.

[5] Guanling Chen and David Kotz. Solar: Towards a flexible and scalable data-fusion infrastructure and ubiquitous computing. In *Proceedings of the Workshop on Application Models and Programming Tools for Ubiquitous Computing at the Third Interna-*

tional Conference on Ubiquitous Computing (UbiComp 2001), Atlanta, GA, October 2001.

[6] Guanling Chen and David Kotz. Context-sensitive resource discovery. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, pages 243–252, Fort Worth,Texas, March 2003.

[7] Guanling Chen and David Kotz. A case study of four location traces. Technical Report TR2004-490, Department of Computer Science, Dartmouth College, February 2004.

[8] Guanling Chen, Ming Li, and David Kotz. Design and implementation of a large-scale context fusion network. In *Proceedings of the First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous 2004)*, Boston, MA, August 2004.

[9] Norman H. Cohen, Hui Lei, Paul Castro, John S. David II, and Apratim Purakayastha. Composing pervasive data using iQL. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 94–104, New York, June 2002. IEEE Computer Society Press.

[10] James C. Corbett. Evaluating deadlock detecting methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, March 1996.

[11] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Pearson Education, 2001.

[12] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An architecture for a secure service discovery service. In *Proceedings of Mobile Computing and Networking (MobiCom 1999)*, pages 24–35, Seattle, WA, August 1999.

[13] Anind K. Dey. *Providing Architectural Support for building context-aware applications*. PhD thesis, College of Computing, Georgia Institute of Technology, December 2000.

[14] Maria R. Ebling, Guerney D. H. Hunt, and Hui Lei. Issues for context services for pervasive computing. In *Workshop on Middleware for Mobile Computing*, Heidelberg, Germany, 2001.

[15] Erik Guttman. Service location protocol: Automatic discovery of IP network services. *Internet Computing*, 3(4):71–80, July 1999.

[16] Sumi Helal. Standards for Service Discovery and Delivery. *IEEE Pervasive Computing*, 1(3):95–100, July 2002.

[17] Jeffrey Hightower, Barry Brumitt, and Gaetano Borriello. The location stack: A layered model for location in ubiquitous computing. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing systems and Applications*, pages 22–28, New York, June 2002. IEEE Computer Society Press.

[18] Richard C. Holt. Some deadlock properties of computer system. *Computing Surveys*, 4(3):179–196, September 1980.

[19] Jason I. Hong and James A. Landay. An infrastructure approach to context-aware computing. *Human-Computer Interaction*, 16(2), 2001.

[20] S. Landis and V. Vasudevan. Reaching out to the cell phone with Jini. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, pages 294–303, Big Island, Hawaii, January 2002.

[21] Henry Lieberman and Ted Selker. Out of context: Computer systems that adapt to and learn from context. *IBM Systems Journal*, 39:3–4, 2000.

[22] Jeremy Lilley. *Scalability in an Intentional Naming System*. PhD thesis, Department of EECS, MIT, May 2002.

[23] Christopher P. Masone. Role definition language (RDL). Technical Report TR2002-426, Department of Computer Science, Dartmouth College, May 2002.

[24] Goff K. Max. *Network Distributed Computing: Fitscapes and Fallacies*. Prentice Hall, 2003.

[25] Kazuhiro Minami and David Kotz. Controlling access to pervasive information in the "solar" system. Technical Report TR2002-422, Department of Computer Science, Dartmouth College, February 2002.

[26] P. Mockapetris and K. J. Dunlap. Development of the domain name system. In *Proceedings of the 1988 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 123–133, Stanford, CA, August 1988.

[27] P. V. Mockapetris and K. J. Dunlap. Development of the domain name system. In *Proceedings of the Symposium on Communications architectures and protocols*, pages 123–133, Stanford, California, 1988. ACM Press.

[28] Antony Rowstron and Peter Druschel. Pastry: Scalability, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the 2001 International Middleware Conference*, pages 329–350, Heidelberg, Germany, November 2001.

[29] Umar Saif and Justin Mazzola Paluska. Service-oriented network sockets. In *Proceedings of the First International Conference on Mobile Systems, Applications and Services (Mobisys 2003)*, San Francisco, USA, 2003.

[30] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Proceedings of Symposium on Principles of Distributed Computing*, pages 1–7, 1996.

[31] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, 2001.

[32] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *Proceedings of IEEE Workshop of Mobile Computing Systmes and Applications*, pages 85–90, Santa Cruz, California, December 1994. IEEE Computer Society Press.

[33] William Noah Schilit. *A System Architecture for context-aware mobile computing*. PhD thesis, Columbia University, May 1995.

[34] Albrecht Schmidt, Kofi Asante Aidoo, Antti Takaluoma, Urpo Turmela, Kristof Van Laerhoven, and Walter Van de Velde. Advanced interaction in context. In *Proceedings of the First International Symposium on Handheld and Ubiquitous Computing*, pages 89–101, Karlsruhe, Germany, September 1999. Springer-Verlag.

[35] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[36] Mike Spreitzer and Marvin Theimer. Providing location information in a ubiquitous computing environment. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 270–283, Asheville, NC, 1993. ACM Press.

[37] S.S. Isloor and T.A. Marsland. The Deadlock Problem: An Overview. *IEEE Computer*, pages 58–78, September 1980.

[38] Jim Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, July 1999.

[39] Jue Wang, Guanling Chen, and David Kotz. A sensor-fusion approach for meeting detection. In *Proceedings of the Workshop on Context Awareness at the Second International Conference on Mobile System, Applications, and Service (Mobisys 2004)*, Boston, MA, June 2002.

[40] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, January 1991.

[41] Abram A. White. Performance and interoperability. Technical Report TR2002-427, Department of Computer Science, Dartmouth College, May 2002.

[42] Abram A. White. XSLT and XQuery as operator languages. Technical Report TR2002-426, Department of Computer Science, Dartmouth College, May 2002.