

vrije Universiteit *amsterdam*



Design and implementation of a secure, decentralized location service for agent platforms

Reinout van Schouwen

Aug. 31, 2006

Master's thesis

Supervisors:

Dr. Benno J. Overeinder

Dr. Michel Oey

Second reader:

Prof. Dr. Frances M.T. Brazier

The research for this thesis was supported by Stichting NLnet.

Abstract

In this thesis, the design of Foncation is presented. Foncation is an agent location service for mobile agent platforms such as AgentScape. It is designed to handle a high amount of concurrent registration and lookup requests, and to guarantee authenticity of the stored data. In contrast to many other location service designs, Foncation does not impose a hierarchical naming scheme on its clients. The underlying decentralised version of Fonkey implemented in the context of this thesis is based on the *Bunshin* Peer-to-Peer Distributed Hash Table. A prototype implementation is tested with a varying number of nodes to evaluate its scalability. The results are promising: the Foncation location service scales according to expectations and the overhead caused by the authentication algorithm is relatively small. Under simulated real-world circumstances, Foncation performs clearly better than under high load. A number of directions for future research are explored: trust models and techniques for performance improvement of queries in DHTs are the two most important.

This document was produced using the following Free software:

- The Mandriva Linux operating system
- L^AT_EX typesetting software
- The jEdit text editor
- The Gnumeric spreadsheet
- The Evince document viewer
- The Epiphany web browser

and various GNOME¹ modules.

¹See <http://www.gnome.org/>

Preface

At the end of 2004, I started looking for a MSc project with which to conclude my AI programme. My good friend David R.A. de Groot introduced me to the people in the IIDS group at the Vrije Universiteit where he had written his thesis. This first lead to a small research project I conducted with Olivier Marin, with which I earned some study credits that I still lacked. After that, I was ready for my final project. Benno Overeinder suggested that I look into a piece of research that was conducted two years earlier into a public key distribution system called Fonkey. There was an idea to build a secure location service for AgentScape with Fonkey and a Distributed Hash Table. Working out this idea was estimated to be about one month of work. Now, one and a half year later, the final result lays before you.

This thesis would not have been possible without the dedicated assistance of Michel Oey and Benno Overeinder. Thank you for your help with code debugging and your writing tips. I also wish to thank my friends, in particular David de Groot, Niels Drost, Daphne Koopmans, Tom Burger, Nadine Lamotte, Boris Huisman, Ingrid Arts, and Nienke Klomp for the good conversations, useful feedback and moral support during this period! Above all I want to thank my family for their support and never-ending confidence in my ability to complete this project successfully.

Dedicated to my grandfather Nicolaas van Schouwen (December 1, 1917 – March 2, 2006)

Contents

Preface	iii
1 Introduction	1
1.1 Multi-agent systems	1
1.1.1 Software agents	1
1.1.2 Agent platforms	2
1.2 Naming and locating entities	3
1.2.1 Security in location services	4
1.2.2 Agent location services	4
1.3 Fonkey	6
1.4 Requirements	6
1.5 Research questions	7
1.6 Approach and outline	7
2 Design of a secure location service	9
2.1 Authenticity considerations in location services	9
2.1.1 Private and public keys	9
2.1.2 The web-of-trust	10
2.2 High-level design of the Foncation agent location service	11
2.3 Fonkey overview	11
2.3.1 Fonkey packages	13
2.3.2 Possibilities and restrictions	14
2.3.3 Client interface	15
2.4 The Foncation location service	15
2.4.1 The location service interface	15
2.4.2 Initialization of Foncation	17
2.4.3 Mapping the location service on Fonkey operations	17
2.4.4 Verifying authenticity	18
2.5 Summary	19
3 A decentralized data store	21
3.1 Distributed data storage	21
3.2 Peer-to-peer networks	22
3.2.1 Unstructured overlays	22
3.2.2 Structured overlays	22
3.3 Abstractions in structured P2P networks	23
3.3.1 Tier 0: The Key-based Routing Layer	23

3.3.2	Tier 1 abstractions: DOLR, CAST, and DHT	23
3.3.3	Tier 2 abstractions: Bunshin, SCRIBE and OceanStore	24
3.4	Functional requirements and constraints	24
3.4.1	Evaluation criteria	25
3.4.2	Candidate storage systems	26
3.5	Implementation of Fonkey on Bunshin	28
3.5.1	Using keywords to support partial queries	29
3.5.2	Fonkey-Bunshin interaction	30
4	Experimental results	33
4.1	The prototype	33
4.1.1	The software architecture	33
4.1.2	Integration with AgentScape	35
4.2	Test setup	35
4.2.1	Used hardware	35
4.2.2	Goals	35
4.2.3	Experiment categories	36
4.2.4	The test application	36
4.3	Results	38
4.3.1	A single node-experiment	38
4.3.2	Experiments on the DAS-2 cluster	39
4.4	Discussion	40
5	Related research	43
5.1	Mobile agent location services	43
5.2	Distributed naming services	45
5.3	Alternative trust schemes	46
5.4	Alternative search strategies	48
6	Conclusions and Future work	51
6.1	Summary	51
6.2	Conclusions	52
6.3	Future work	52
6.4	Open issues	52
	Bibliography	54

Chapter 1

Introduction

This thesis is the result of a MSc research project in the IIDS group at the Vrije Universiteit Amsterdam. The thesis describes the design and the implementation of a prototype of a distributed, secure location service for agent platforms and the rationale behind it.

This chapter starts with an introduction to multi-agent systems and distributed systems. It then provides a short description of the AgentScape project, the context of the research presented in this thesis. It shows how current solutions for looking up agent locations in networks fail to fulfill the requirements of AgentScape and proposes a new location service, which is scalable and supports a strong security model. The final sections of this chapter are dedicated to the research questions, the main contribution, and an overview of the remaining chapters.

1.1 Multi-agent systems

“A distributed system is a collection of independent computers that appears to its users as a single coherent system.”, is the definition of a *distributed system* that Tanenbaum and Van Steen [46] give. Distributed systems have been subject of intensive research during the past decade. Examples of distributed systems include peer-to-peer networks and multi-agent systems, as explained below.

1.1.1 Software agents

Software agents are computer programs that can, to a certain extent, operate autonomously to achieve their goal. Four common characteristics of software agents are autonomy (ability to take decisions without external directions), proactivity (the agent takes initiatives to reach its goal), reactivity (ability to take appropriate action on external stimuli) and social ability (the ability to communicate with other agents or possibly humans). Together, these characteristics are defined as the *weak notion of agency* by Jennings and Wooldridge [51]. A *strong notion of agency* exists as well: among AI researchers the functionality of agents is often explained in terms of human behaviour. For example, the BDI model [19] ascribes beliefs, desires and intentions to agents.

One of the interesting aspects of an agent-based approach to solving problems in computing is that it provides an unorthodox way of tackling issues where privacy and data security are of utmost importance. In law-enforcement for example, it may be necessary to compare

reports concerning a crime suspect produced by various government agencies. However, these agencies do not always trust each other fully so they refuse each other unlimited access to their records. Also, sharing a suspect's personal data may be against privacy regulations. In extreme cases, mistrust and privacy concerns could lead to errors in the justice process caused by inavailability of information to agencies that need it. Software agents that are trusted by all parties could help prevent this when they get privileged access to all relevant records. Instead of transferring—possibly large amounts of—sensitive data over the network, a software agent could visit the servers of the involved agencies to gather the desired data. Added advantage is that the amount of used network bandwidth is reduced in the process. This trait is useful in slow networks or networks with intermittent connections.

Software agents can work alone or in groups and usually operate in a distributed environment. When multiple software agents cooperate in a group, it is called a *multi-agent system*. Multi-agent systems that exist in a distributed environment often allow software agents to **migrate** from one location to another. Such migrating agents are referred to as *mobile agents*. A multi-agent system can be small-scale and large-scale. For example, a small-scale multi-agent system is a system where software agents coordinate household electronics, and each piece of audio/video equipment in a home forms an agent location. Currently, the ultimate large-scale distributed environment where a multi-agent system could be run is, of course, the Internet.

For cooperation in a multi-agent system, inevitably communication is required. An important part of the research reported in this thesis is to support agent communication and migration by creating a dependable and efficient way of resolving the current contact address of other agents.

1.1.2 Agent platforms

Agent platforms are layers of software that facilitate the execution of agent processes and multi-agent systems. In the same way that the agent platform itself needs an operating system like GNU/Linux or Solaris to run on, they are responsible for a variety of services that agents need, such as communication with the outside world, migration from one instance of the platform to another (including instances on different geographic locations) or looking up the current location of a fellow agent.

The **AgentScape** middleware infrastructure [50] is an agent platform designed to develop and deploy open, large-scale distributed agent systems. Among other features, it supports mobility, authenticated communication channels and interaction with services, such as a location service. In AgentScape, a location is a “place” where the basic AgentScape entities, agents and services, reside. Figure 1.1 illustrates the AgentScape conceptual model. Observe that one location may host multiple instances of the AgentScape middleware.

Important to the research presented here is the fact that AgentScape does not use a predefined naming scheme for agents. A globally unique and location-independent identifier for each agent is not enforced. This characteristic, known as a **flat name space**, sets AgentScape apart from many other agent platforms that require agent names to be unique, imposing a hierarchical naming scheme.

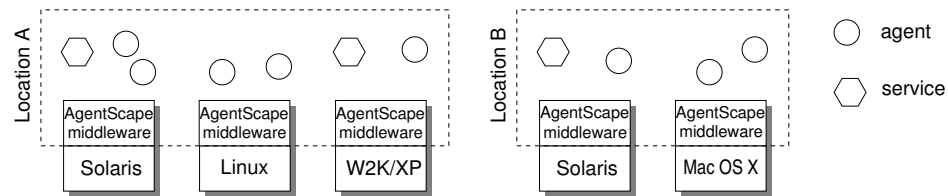


Figure 1.1: Conceptual model of the AgentScape middleware infrastructure

1.2 Naming and locating entities

On wide-area networks such as the Internet, entities on the network are usually identified by a symbolic name that is mapped to the unique (IP)-address where the entity resides. The service that performs such a mapping is called a *naming service*. Citing Ballintijn [6], p.2:

“By allowing users to refer to a resource by a name instead of its location, these naming services shield users from the problems of where a resource is located, whether it consists of more than one copy, and whether it can move.”

In this thesis, a more subtle definition of *naming service* is followed, namely, the one introduced by Van Steen [44]. A traditional *naming service* provides an immutable mapping between an object’s **symbolic name** and an **object handle**. An object handle is an immutable, globally unique identifier for an object. In contrast, a *location service* in a distributed system maps an object handle to a physical **address** of any host that contains the requested object or a replica. The assumption in this document is that there can be no more than one instance of an object (agent) in the distributed system.

The primary purpose of location services is to register object handle-to-contact address mappings in a computer network and to make these mappings available to any entity in the network that requests it. A location service implemented on a single host in the network is called **centralised**. A centralised location service will fail as the number of nodes in the network gets very large, because it will no longer be able to handle all requests. More importantly, if the network depends on the availability of a single node to handle its location lookups, it becomes an easy target for attack. A location service can be made to cope with growing numbers of requests and attacks by making it scalable. Scalability can be achieved by making the service **distributed**, in other words, to spread its functionality over multiple nodes.

The most widely used naming service today is Domain Name System (DNS) [28]. DNS is the naming service used on Internet. Domain names in DNS are organised in a hierarchical manner. It uses a number of synchronized root servers and many “subordinate” name servers that are each responsible for a part of the name hierarchy. Domain information is synchronized in regular intervals between the root name server and the subordinate servers. When information is updated on one of the name servers, it will eventually be known to all of them. The process that finds a network address belonging to a domain name is called *name resolution*. To improve the efficiency of name resolution, replication and caching techniques are used. For these techniques to work reliably, however, it is required that domain name to address mappings change infrequently [6].

The Lightweight Directory Access Protocol (LDAP) [49] can be used as a distributed location service, although it has more capabilities. The LDAP protocol offers read and update operations that require secure authentication. It is similar to DNS in that data is distributed over multiple servers in a hierarchical fashion.

A location service [44, 6] designed for mobile objects in Globe¹ is organised as a hierarchical search tree. In order to achieve scalability, higher-level nodes in the tree are partitioned into subnodes, where each subnode is responsible for a subset of the records originally stored at the actual node. The Globe location service is extended by Hu, Rodney, and Druschel [21] with the NLS naming and location service. With scalability as its key characteristic, NLS is based on a two-layer architecture with a fat-tree based topology at the global layer.

1.2.1 Security in location services

To protect the integrity of naming and location services is essential. If integrity is not guaranteed, the possible advantages of multi-agent systems with regard to handling sensitive data, are void. DNS has proven to be vulnerable to attacks such as *spoofing* and *Denial of Service*. In case of spoofing, tables of a single DNS server can be filled with incorrect information, which will propagate to other DNS servers on the Internet. Denial of Service means that the service receives such a large number of (invalid, or valid but useless) requests that it is unable to cope with the normal traffic, so the service availability is disrupted.

Some naming and location services have been extended to provide security features, so that it becomes more difficult to perform spoofing attacks. For instance, the DNSSEC [5] extensions to DNS provide data integrity and origin authentication. One approach to DNSSEC is to use a digital signature scheme based on **public-key cryptography** [16]. In this approach, each node in the DNS hierarchy certifies the *public key* of the nodes below it in the hierarchy. A public key is a public token, accompanied by a private counterpart, that, among other things, one can use to identify himself. Given that the public key of the node at the root of the hierarchy is known and trusted, this certification process makes it possible to verify whether messages that appear to originate from a certain node can be trusted. Other approaches to DNSSEC have been proposed, such as the symmetric (secret)-key cryptography based scheme proposed by Ateniese and Mangard [4]. Its advantage over a public-key based scheme is its greater speed, as secret key algorithms require much less computing time as public-key algorithms do.

The Globe location service features a number of safeguards against abuse [6], the most important of which is **access control**, also based on public-key cryptography. With the access control method, clients of the location service have to determine which update operations are allowed on their objects.

The research presented here focuses on providing location service security features using public-key based authentication schemes. A more detailed discussion of the principles of public-key cryptography and authentication will be given in Chapter 2.

1.2.2 Agent location services

Most naming and location services, such as DNS, do not take highly dynamic entities such as mobile agents into account. Information about migration of an object from one host to an-

¹Globe is a worldwide system built to support distributed applications.

other can take up to days to propagate, depending on the cache settings of a DNS server. In multi-agent systems however, software agents can autonomously decide to migrate, for instance to dynamically adapt to a changing environment[8]. As Roth and Peters state, agents may decide to migrate at any time, and a huge number of updates must be expected[39]. This makes current DNS implementations unsuitable for locating mobile agents on a large scale network. It should nevertheless be noted that Pappas *et al.* [32] argue that dynamic functionality can be introduced in DNS to make it capable of handling high rate dynamic updates, making it suitable for use in a distributed system with mobile agents.

In literature, a number of location services have been proposed that specifically address the requirements for multi-agent systems. Notable examples are [15, 22, 52, 48, 39]. The purpose of an agent location service is to track the current location of each agent in the network. It can store addresses of agents and retrieve them upon request. Being able to find the location of agents in a network is important to be able to support communication between agents, and even more so in scenarios where mobile agents often migrate from one host to another. Mobile agents still need to be able to communicate with other agents and services.

Various strategies can be followed to track agent locations. Roth [38] discusses several ways of tracking the location of agents. He divides them in four categories:

Direct response. When an agent migrates to a different location, it notifies its home base (the location where the agent first originated). Requires that the home base is permanently on line and that the starting locations of all agents are known by everyone in advance.

Buffered response. The classic name server model. A dedicated location service is notified of location updates and is used for location queries as well. The dedicated service may consist of more than one server that stays synchronized with the others. Each participating server can be made responsible for a certain part of the name space.

Searching Agent locations are queried in a systematic fashion to discover whether a certain agent is currently located there. Unsuitable for environments where agents migrate randomly (as opposed to in predictable patterns).

Forward references The agent's home base keeps a forward reference to the next location of an agent, and if it has already migrated away from there, the next forward reference is followed until the agent is reached. The disadvantage is that reference chains may become long and that they can break when a forward reference expires or when an intermediate location goes down.

Roth disregards *searching* and *direct response* because their disadvantages make them unsuitable as location strategy for the type of mobile agent system the location service is designed for. Roth compares the remaining two strategies and concludes that the classic, server-based approach called *buffered response*—also called *registering* by Milojević *et al.* [27]—is the most favourable, because the overhead related to lookup queries is placed at dedicated servers, instead of burdening the agent servers with it. In this approach, the associative mapping of agent names to agent locations is performed by one or more dedicated servers.

Overeinder, Rozendaal, and Brazier [31] propose a location service for mobile agents with similar goals and a similar approach as taken by Roth and Peters [39]. However, their proposal combines a strong security model with scalability and data integrity verification. The security and authentication features are based on the Fonkey specification [42].

1.3 Fonkey

Fonkey provides an infrastructure to publish public keys with optionally some additional information. All published information can be digitally signed by the publisher or by other entities in the system, thus enabling verification of data integrity. The signed information in Fonkey can be anything from a cooking recipe to whereabouts of mobile agents. Thus, the Fonkey infrastructure enables the creation of a location service which can verify the authenticity of the location information contained in its database. Chapter 2 treats the details of Fonkey extensively.

1.4 Requirements

Van Steen *et al.* [44] formulate four generic requirements for location services: scalability, locality, stability and fault tolerance. The dynamic nature of wide-area distributed agent systems calls for more requirements: mobility, dynamicness and security. The unique requirement that is addressed in this research is the *flat name space*. All requirements are briefly explained below.

- Flat name space: users of an agent location service should not be forced to use a certain naming scheme. If so desired, users should be able to define their own name space.
- Mobility: agent addresses may change at a high rate. The service must be prepared to handle many updates concurrently.
- Scalability: the ability to gradually scale to increasing demand. This translates to being able to serve many concurrent requests as well as the ability to store huge numbers of objects.
- Dynamicness: the service should be designed to support easy adding and removing of hosts without violating the integrity of the data store.
- Security: it should be possible to validate the authenticity of received information by using cryptographic techniques, such as digital signatures.
- Locality: the principle that data from a client of the location service is stored in nearby places, so that it is retrievable with low latencies (thus, low costs) to the client.
- Stability: When movements of an object through the network are predictable, then the object is **stable** with respect to the region in the network where it exists. The desired property of the location service is that lookup or update operations for a stable object are less costly than for unstable objects.
- Fault tolerance: the service should continue operation when a part of the network goes down. The performance and functionality should degrade gracefully when failures occur.

The requirements addressed in this thesis are security, flat name space, scalability, dynamicness and mobility. The other requirements are not addressed directly. Instead, the underlying system that has been selected for handling the data storage (discussed in Chapter 3) is

relied upon for its locality and fault tolerance features. The other location services for wide area- or distributed systems discussed above do not address both data integrity concerns as well as the flat namespace requirement. The research in this thesis is aimed at satisfying both demands.

1.5 Research questions

The central question in this thesis is the following:

Can we build a scalable, secure location service that deals with highly dynamic information?

The next, derived question is:

To which extent is it possible to design the required infrastructure using the Fonkey specification?

Our contribution to the field of research is that it is the first attempt to build such a scalable agent location service with a strong security model.

1.6 Approach and outline

The basic approach to the research questions is the design and implementation of a location service on top of the distributed Fonkey infrastructure. The Fonkey infrastructure is built using Peer-to-Peer (P2P) technology to store the location data. The resulting design is called **Foncation**.

The *buffered response* approach, introduced in Section 1.2.2 is loosely followed in the Foncation lookup service design, because it is sufficiently scalable and fault tolerant, supports mobility and it is not dependent on the availability of agent home bases. In order to meet the *flat namespace* requirement, no name space hierarchy is imposed on the participating servers. The remaining chapters of this thesis are classified as follows. Chapter 2 gives an introduction to the agent location services, public key cryptography and the Fonkey specification. Chapter 3 explains the basic concepts of Peer-to-Peer networks and their abstractions, and elaborates on the choice of a decentralized Fonkey storage back-end. Chapter 4 discusses the implementation and test results. The fifth chapter gives a discussion of related work. Finally, Chapter 6 discusses future work and presents conclusions.

Somewhere your fingerprints remain
concrete
And it's your face I'm looking for on every
street

*On every street
Dire Straits*

Chapter 2

Design of a secure location service

This chapter describes the design of the Foncation secure location service. Foncation is based on the Fonkey infrastructure. Public-key authentication mechanisms are used to implement the required security extensions. A brief introduction to public-key cryptography will be given, followed by a description of the location service functionality. A closer examination of the Fonkey architecture is given, followed by the design of an agent location service harnessing the security features that Fonkey has to offer.

2.1 Authenticity considerations in location services

As stated in the requirements formulated in Chapter 1, security and trust in a location service is important to counter attacks that could bring down the location service or alter location data in malicious ways. Another aspect is accountability of software agents. Actions and movements of agents need to be traceable. A principal¹ should be able to prove that information he received was sent by another party, even if that other party later denies having sent the information at all. This ability is known as *non-repudiation*. The Foncation design addresses these trust considerations.

2.1.1 Private and public keys

A common problem in cryptography is that, to securely communicate information to one and only one receiver, one needs to have a “shared secret”, like a password that is known to both parties. But this shared secret has to be communicated over a secure channel first! A solution to this “chicken and egg” problem is a scheme where each participant has a public key that everyone may see and a uniquely matching private key only known to himself. With current computing techniques, the private key that belongs to a public key cannot be derived within reasonable time. Private keys can be used to either **decrypt** information that was encrypted by its public key counterpart, or to **sign** data so that the authenticity of the data can be established later, again using the corresponding public key. Many authentication and encryption mechanisms nowadays require the use of public-private key pairs. In Fonkey, signing and authentication functionality is used, but not encryption.

¹A principal is an entity that can either authenticate itself or verify the identity of other principals.

Signing, in this context, means: to create and attach a digital signature over a piece of data d using your private key. Because cryptographic calculations are expensive, often a checksum (using an algorithm such as SHA-1[17]) is first generated from the signable data to speed up the process. The signature is subsequently calculated over the checksum. An arbitrary principal B can verify that principal A signed the data by generating a checksum over d and using A 's public key to see if the signature matches.

Using private and public keys to sign and authenticate, however, raises a new problem: how to be sure that a public key, claimed to belong to another person or entity, really does belong to that person or entity? This issue will be addressed later in this chapter.

2.1.2 The web-of-trust

An infrastructure that offers all facilities to administer name to public key mappings and to issue and maintain public-private key pairs, is called a Public Key Infrastructure (PKI). Many PKI's support key revocation procedures, in case a private key is compromised. A number of trust models in PKI's exist. A trust model basically defines how users of the PKI decide when to trust each other's public key.

Many trust models employ one or more Certification Authorities (CA): a trusted node that digitally signs data structures known as *certificates* that state the mapping between names and public keys. Perlman [33] lists the most common ones, including the Single-CA model and the Anarchy model. In Single-CA, there is one universally trusted certification authority on which all clients rely. The Anarchy model, employed by the well-known PGP software², implies that all participants start out with a set of public keys, securely acquired by a means other than the PKI software. Then, a participant with its own private key can declare that he trusts other participant's keys by signing a certificate for them. These certificates are stored in publicly accessible databases or exchanged in other ways. When at some point a principal wants to know if he can trust someone else's public key, he checks whether a link exists between himself and the other participant, either directly or via already trusted people that have certified the other person's key. This way, a *web-of-trust* can be formed. The *web-of-trust* model does not require one or more central CA's; that is why it is called a **decentralised model**. By its nature, this aspect of decentralisation makes the Anarchy model well-suited for a distributed secure location service. Other, more complex, trust models are beyond the scope of this research and are not further discussed here.

Formal definition of a web-of-trust

Aberer *et al.*[3] express the web-of-trust idea formally as a transitive trust rule: "If [Principal] P_A trusts that [Key] K_B is P_B 's public key, and also relies (personally determined) on P_B to certify a third party's public key, then P_A will believe that K_C is P_C 's public key if P_B certifies it."

The above definition can be interpreted as "Assume that Alice receives a document signed by Charles, whom she does not know. She may decide to trust that the information contained in the document really originated from Charles, based on the fact that she trusts Bob who signed a declaration stating that he recognizes Charles' autograph on the document."

²Pretty Good Privacy, see <http://www.pgp.com/>.

Disadvantages

The main drawback of the web-of-trust model is basically that “a chain can only be as strong as its weakest link”. In the example above, Bob might be trying to mislead Alice, and have forged Charles’ signature himself. In that case, Alice will mistakenly decide to trust the signed document because there exists a trust link between her and Charles. In summary, the web-of-trust depends on the benevolence of all peers. Aberer *et al.* [3] cite several more problems in using a *web-of-trust* to base an identity authentication infrastructure on and recommend a statistical approach instead. More discussion on alternative trust models can be found in Chapter 5.

The Foncation proof-of-concept secure location service has been built using the straightforward web-of-trust model. Applying the statistical approach in a future implementation of Foncation is definitely worth considering.

2.2 High-level design of the Foncation agent location service

In the previous sections, a security model is proposed that fulfills the security requirement stated in Chapter 1. The **Fonkey** infrastructure was chosen to implement the signing and authentication mechanisms in the location service design, as discussed in section 2.3. The requirements that Fonkey addresses are briefly discussed below.

To support a flat name space, the input that Foncation takes for agent identifiers is a free-form string. If two identifiers clash, references to multiple agents can be returned, and the agent platform software can decide which is the right one.

Scalability, dynamicness and mobility are addressed by choosing a distributed storage system. This distributed system has to enable the security model discussed previously and be able to scale up to increasing demand (the scalability requirement), support easy adding and removing of participating nodes (the dynamicness requirement), be able to handle many updates concurrently (the mobility requirement), allow low-latency lookups (the locality requirement), and be resilient to network failures (the fault tolerance requirement). Peer-to-Peer (P2P) overlay networks are very suitable to fulfill these requirements. Lua *et al.* [26] give a survey of P2P overlay network schemes. This survey establishes that these schemes can handle thousands of participating nodes, do not require a central administration, can handle continuous incoming and leaving of nodes, and allow replication and caching to speed up information retrieval. Chapter 3 gives an extensive discussion of the distributed storage alternatives considered, and the rationale for selecting a particular one.

The resulting high-level design for Foncation is displayed in Figure 2.1. The nodes in the distributed storage network are theoretically distinct from the nodes on which the agent platforms run. In practice however, each agent platform with the Foncation location service is a node in the distributed storage network as well. The interrupted line indicates that the agent platforms themselves may communicate over other channels than the Fonkey storage nodes.

2.3 Fonkey overview

Fonkey is a system to globally distribute public keys with additional information. The additional information can be data referring to objects that exist outside of the Fonkey system.

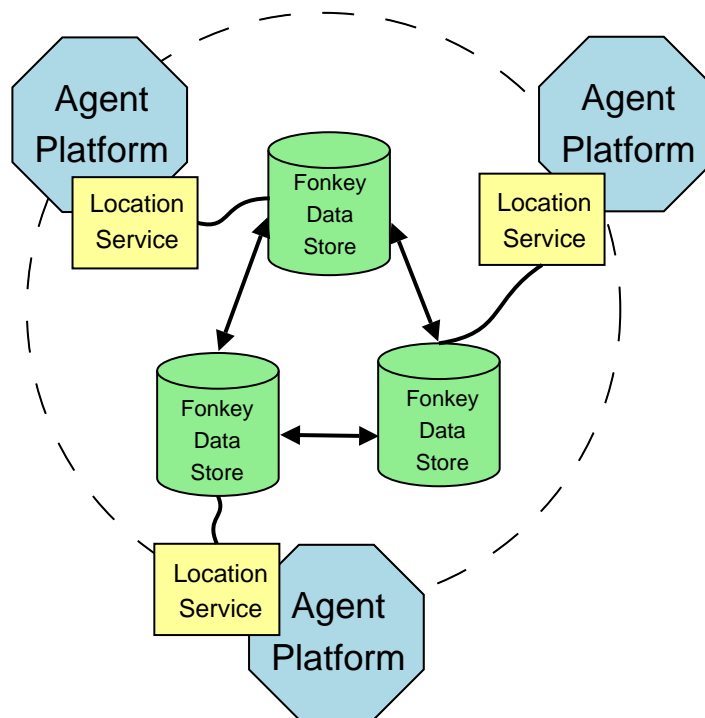


Figure 2.1: High-level design for an agent location service with a Fonkey-based distributed storage.

Fonkey is not a public key infrastructure in itself, since it does not support Certificate Authorities, key revocation mechanisms, or other features that make up a typical PKI. However, more complex systems such as PKIs could be built on top of the Fonkey infrastructure [42]. Participants in Fonkey each generate their own public-private key pair. When a participant publishes information, his public key serves as identification of the publisher. Of course, this identification is rather weak by itself, considering that no cryptographic link exists between the public key and the participant that publishes it. This is the reason that keys in Fonkey can be signed with other keys.

2.3.1 Fonkey packages

Fonkey defines three data types to publish and retrieve public keys: **keys**, **named data** and **signatures**. These three types correspond directly to the data structures supported by Fonkey. These data structures are called *packages*. There are three kinds of packages in Fonkey: the *Key Package*, the *Named Package*, and the *Signature Package*, providing all necessary building blocks to support applications using public-key mechanisms for authentication and validation of data. Key Packages can be employed to publish public keys, Named Packages serve to store arbitrary data that is to be signed automatically, and Signature Packages are utilized to express trust in the authenticity of arbitrary other packages [42]. More details of Fonkey packages are discussed below.

Basic package structure All Fonkey packages share the common properties shown in Table 2.1. Of these fields, `Properties` and `Payload` are open fields. The remaining ones are not: `Public Key` must be the public key of the user that published it, `Version` is always raised by one whenever the package is re-published. `Signature` must be a cryptographic signature calculated over the package contents, using the publisher's private key. When the contents of a Package are altered by a party that does not have the disposition of the private key that produced the Package signature, it is easy to detect because checking the signature and the package contents against the public key will fail in that case.

Field	Description
Type	Type of the package is either <code>Key</code> , <code>Named</code> , or <code>Signature</code> .
Public Key	The public key component of the key pair.
Version	A strictly increasing integer version number to ensure older packages do not accidentally overwrite newer packages.
Properties	A set of name/value pairs
Payload	Application specific payload
Signature	The signature used to ensure the integrity of the package.

Table 2.1: Fonkey Basic Package. Adapted from Overeinder *et al.* [31].

Key Package Key Packages are used to hold public keys. The structure of a Key Package is identical to the basic package. A Key Package can be identified by its `Public Key` field. The `Type` field of the package is set to `Key`.

Named Package The Named Package extends the basic package with a `Name` property, as defined in Table 2.2. Apart from a public key, it holds a name (this can be an arbitrary word or sentence). The identifier of a Named Package is the combination of its `Public Key` and its `Name`.

Signature Package A Signature Package adds two fields to the basic package structure, as shown in Table 2.3. When a principal vouches for the authenticity of a certain package p , the rest of the world can be made aware of this information by publishing a digital signature calculated over p 's unique identifier in a Signature Package. The trusted package p is known as the *subject* of the Signature Package.

Field	Description
Name	The name of this package. The name can be an arbitrary string, used to locate this package.

Table 2.2: Fonkey Named Package. Adapted from Overeinder, Rozendaal and Brazier [31].

Field	Description
Subject	The unique identifier of the package that is signed by this signature package
References	The parts of the subject package being signed. For each part a SHA-1 hash is stored.

Table 2.3: Fonkey Signature Package. Adapted from Overeinder, Rozendaal and Brazier [31].

2.3.2 Possibilities and restrictions

Applications can use Fonkey to search for information identified by public keys, names, signatures, or subjects. Also, the payload information of all package types depends on the application using Fonkey. For instance, it could refer to an e-mail address or a web site address. Due to the fact that there is no limitation imposed on either the type of query, or the contents of the payload, flexible and generic usage of Fonkey for disseminating information in a secure manner is made possible.

Notwithstanding the flexibility of Fonkey, it should be emphasized that it is not designed to store large volumes of information. Whenever a piece of information is stored in Fonkey, a fingerprint (*hash*) is calculated over it and a digital signature pertaining to the hash is stored together with it. These cryptographic operations are expensive, so when distributing for example a large music or movie file requiring a lot of processing power to analyze, it would be better to place it on a web server and put its URL in the payload than to use the file itself as payload.

Operation	Description
<pre>update(Package p, Long duration): boolean</pre>	<p>Publishes (any type of) package, under condition that is self-signature is valid. If package <i>p</i> already exists in the store, its contents are replaced by the new contents. The <i>duration</i> parameter specifies the number of seconds until <i>p</i> expires and must be re-published.</p>
<pre>lookup(PackageId pid): Package</pre>	<p><i>pid</i> is the unique identifier of a package. This call finds the corresponding package in the store if it exists, and returns <code>null</code> otherwise.</p>
<pre>findPackagesByKey(PublicKey pk): Set</pre>	<p>This call finds all packages that have been published by the principal with public key <i>pk</i> and returns their unique identifiers as a Set.</p>
<pre>findPackagesByName(String n): Set</pre>	<p>This call finds all Named packages that have been published with name <i>n</i> and returns their unique identifiers as a Set.</p>
<pre>findPackagesBySubject(PackageId pid): Set</pre>	<p>This call finds all Signature packages that have been published for a Key Package with ID <i>pid</i> and returns their unique identifiers as a Set.</p>

Table 2.4: Fonkey client interface.

2.3.3 Client interface

The basic interface available to Fonkey users is the *Fonkey client interface*, specified in Table 2.4. To publish new information, the `update` operation must be used. To retrieve information, the `lookup` operation is available. A `PackageId` is the unique identifier of any information package that can be calculated with either just the package's public key (for Key Packages), the public key and the name (for Named Packages) or the public key combined with the subject's public key (for Signature Packages). A `Key` is an instance of a public key, not to be confused with a Key Package.

2.4 The Foncation location service

Foncation is a service that offers lookup and registration operations. It is designed to be used in agent platforms. It translates the location service primitives `lookup`, `register`, and `deregister` to Fonkey calls (detailed in section 2.3.3). The user of the location service does not have to be concerned with the way Fonkey handles data internally.

2.4.1 The location service interface

Table 2.5 shows the interface for the Foncation location service. The *duration* registration parameter is added to prevent the data store from slowly filling up with outdated information. It provides a primitive garbage collection mechanism that prevents that registrations stay

Operation	Description
lookup(String identifier): String []	Looks up location information of the agent specified by the identifier. A String array containing the corresponding value at position 0 is returned. Higher indices are reserved for the case that <i>identifier</i> matches more than one agent.
register(String identifier, String data, Long duration): Long	Registers the new location of the agent specified by the identifier. The <i>data</i> parameter contains the actual location data and the <i>duration</i> parameter specifies the time-to-live(TTL for this registration. A Long indicating the time (in seconds) when the registration information validity runs out is returned.
deregister(String identifier): void	Removes the entry and the location data of the agent with name <i>identifier</i> from the location service.

Table 2.5: Foncation location service interface.

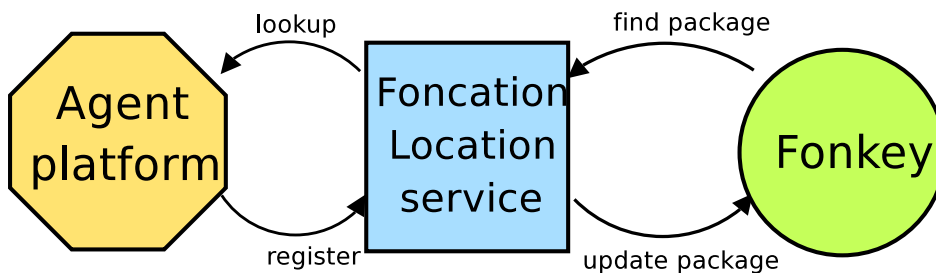


Figure 2.2: Design of a Fonkey-based agent location service

around forever (this is possible for instance in case of a failure of the location service node that registered the agent). When the indicated time has passed, an agent's registration is removed from the system. A beneficial side-effect is that when it is known in advance that an agent will shut down or otherwise leave the system after a certain period of time i , then no explicit deregistration is required if i is given as *duration* parameter at registration time.

Note that the interface in the current form does not mention any operations related to authentication, whereas Foncation is claimed to be a secure location service. All cryptographic and storage/retrieval operations are performed transparently and the authentication issues are handled on a lower level. The Anarchy trust model, where administrators are responsible for specifying which other public keys to trust, is not supported with this interface. In order to support the Anarchy model, it would be required to add operations for adding and removing trusted public keys to the interface. The next paragraphs explain how Fonkey is used to implement the agent location service.

2.4.2 Initialization of Foncation

Since Fonkey is designed as an infrastructural service for other applications to use, it is launched separately from the location service. The location service talks with Fonkey using remote method invocation.

Foncation itself is designed to be instantiated as a service specific to one AgentScape location. Each AgentScape location is assumed to have its own public-private keypair. A Foncation instance then publishes location information through Fonkey, on behalf of its AgentScape location, using the public-private keypair belonging to that location (the *platform key*). The advantage of this approach, rather than using one Foncation service for multiple AgentScape locations, is that there is no need to worry about which public-private keypair to use for publishing new Fonkey packages. Also, scalability of the location service comes naturally when the number of nodes in the distributed storage grows with the number of agent locations.

To implement the trust model, a commonly trusted key (the *root key*) exists, that is used to sign the public key of each AgentScape location. When launched, Foncation will load the root key. Then, a new platform key is generated with which the AgentScape location will identify itself. Using the platform key, a Fonkey Key Package is created, signed, and subsequently published using the Fonkey `update` operation. In the same vein, a Signature Package certifying the platform key is published, signed by the root key. This procedure is performed on every AgentScape location. As a result, a trust link exists from each platform key to the root key. In other words, a limited web-of-trust is created.

Note that in real-world settings, it is likely that agent location administrators will not just want a shared root key to certify location public keys, but will also want to certify other location's public keys using their own. They can do so by publishing Signature Packages that take Key Packages representing the others' public keys as subject. Also, in such settings, there will not likely be a root key shared by all agent locations using the location service, so agent locations certifying each others's keys becomes necessary for the trust model to work.

2.4.3 Mapping the location service on Fonkey operations

The natural way to translate an agent identifier with its corresponding location to a Fonkey package is to use a Named Package. Fonkey can identify Named Packages by the `Name` field, so if that field is used to store an agent identifier, and the `Payload` field for the location data, it is easy to retrieve the information again at a later time.

Once the location service is up and running, it can be accessed by the agent platform. Below are three scenarios illustrating how Foncation maps location service calls on Fonkey operations.

Registering an agent

Consider the following scenario. Agent P is created on location L . L has a public-private keypair K_L . To notify the rest of the world about its existence, the location service is called like this: `register(P, L, 300)` (register identifier P on location L , for the duration of 300 seconds). Foncation receives the call and proceeds to check if this agent is already registered with the location service. It does so by calling `findPackagesByName(P)` on Fonkey. If the agent has already been registered, Fonkey returns a Named Package referring to it. But

P is a newly created agent so Fonkey returns an empty result. This means a new Fonkey Named Package NP_P is instantiated identified by both the agent identifier P and the public key part of K_L , with L as payload. A signature is generated over L using the private part of K_L and added to the Signature field of NP_P . The version number field (a simple integer ≥ 1) is set to 1. The exact time t of registration as well as the package's expiration time, in this case $t + 300$, is placed in the Properties. Finally, NP_P is published with the Fonkey operation `update(NP_P)`.

Locating an agent

Another typical scenario is where the location of another agent is looked up. Suppose the agent platform on location M hosts an agent Q that wishes to communicate with agent P . M needs to find out where P is located, so it has to invoke the location service. In order to fulfill Q 's request, the platform calls the location service as follows: `lookup(P)`. After receiving this call, Foncation looks up whether the agent is already registered in Fonkey with `findPackagesByName(P)`, like in the registering scenario. This time, Fonkey returns a Named Package NP_P . Foncation verifies that the signature over the payload in NP_P is valid, using the public key K_L contained in the package. The most important step comes next, namely the check on whether K_L is trusted. The procedure followed for this check is described below, in section 2.4.4.

If the K_L is trusted, Foncation extracts the location information L from the payload field and returns it to M . The agent platform on M is now aware that it should relay P 's communication destined for Q to location L .

Deregistering an agent

Agent P has successfully fulfilled its goals and no longer serves a purpose. Therefore, it is terminated and must be removed from the location service data store. The agent platform on the location where P resided last, sends a `deregister(P)` call to Foncation. As Fonkey has no explicit package removal operation, Foncation publishes an updated and newly signed Named Package NP'_P with a Payload that states that P no longer exists, by calling `update(NP'_P)`. The version number of NP'_P is higher than the number of NP_P , to indicate that the information in it supersedes the information in the previous Named Package. Both packages disappear over time due to the *duration* parameter of the `register` operation.

2.4.4 Verifying authenticity

As Foncation utilizes public-key mechanisms but does not rely on an external PKI, an internal trust model has to be developed. For the sake of simplicity, the Foncation location service prototype works with a single commonly trusted public key, which resembles the Single-CA model discussed in section 2.1.2. It also implements a basic web-of-trust concept, similar to the Anarchy model. For real-world applications this does not suffice, because when the number of participants gets very large, there will always be a percentage of them that, for whatever reason, do not trust the certification authority. It is, however, easy to switch to an Anarchy model by taking away the commonly trusted public key. Administrators of agent platforms may then decide themselves which public keys to trust. The location service interface (section 2.4.1) will have to be adapted to allow this, though.

Signatures can be published over any type of package. But when the subject of a Signature Package is a Key Package, it has a special meaning: the issuer of the Signature Package vouches for the authenticity of the public key contained in the Key Package. The web-of-trust forms itself.

The algorithm that determines whether a publisher can be trusted is recursive. The stop condition is reached when the public key from a publisher of a Key Package matches a trusted public key (possibly a universally trusted “root” key), or when a Key Package matching the previous public key cannot be found. Otherwise, the procedure calls itself with the Signature Package that signed the Key Package for the current package’s public key. Algorithm 1 clarifies this in pseudo-code. The algorithm does not take into account the possibility of cycles occurring in the “trust graph”.

Algorithm 1 Simple procedure to decide whether Package p is trusted

```

procedure isTrusted(Package  $p$ ) : boolean
if  $p$ .publicKey = rootkey.publicKey then
    return true {Stop when the root key is encountered}
end if
if  $p$  is-not-a KeyPackage then
    keyPkg  $\leftarrow$  Fonkey.findKeyPackage( $p$ .key)
    if keyPkg = None then
        return false {No Key Package could be found for  $p$ .key}
    end if
    packageId  $\leftarrow$  keyPkg.packageId
    {Retrieve packageId from KeyPackage matching  $p$ 's public key}
else { $p$  is a KeyPackage}
    packageId  $\leftarrow$   $p$ .packageId
end if
signatures  $\leftarrow$  Fonkey.findPackagesBySubject(packageId)
for sig in signatures do
    if isTrusted(sig) then
        return true
    end if{Recursively determine if the publisher's key from Signature Package sig is trusted}
end for
return false {No trusted Signature Package for  $p$  can be found, the package is untrusted}
end procedure

```

2.5 Summary

This chapter introduces the design of a secure, distributed location service called Fonca-tion. The security model as well as the way it builds upon the Fonkey infrastructure was discussed. It has become clear that Fonkey provides the building blocks required to create a location service that uses public-key based identity verification techniques in order to return authentic location information.

The next chapter elaborates on how Fonkey can be implemented using decentralized data structures based on Peer-to-Peer overlay networks.

And you go dancing through doorways
just to see what you will find
leaving nothing to interfere
with the crazy balance of your mind

Love over Gold
Dire Straits

Chapter 3

A decentralized data store

Peer-to-peer *overlay networks* have been gaining popularity as a substrate for distributed data storage systems. The term “overlay network” refers to the fact that a network structure is “overlaid” on top of an existing network, in this case IP (Internet Protocol)-networks. Peer-to-Peer overlay networks support tens of thousands of participating computers at a time, and by using various techniques they can be made resilient against partial network failures. By using an overlay network as basis for the storage back-end for Fonkey, it will gain the scalability and fault tolerance properties that are needed to base an agent location service upon.

Various types of overlay networks, and abstractions upon them, exist. This chapter summarizes a number of available P2P overlay networks, and describes the selection of a suitable distributed storage back-end for Fonkey.

3.1 Distributed data storage

Storing important data in a central location has a few obvious disadvantages: it creates a single point of failure for availability of information, and it is hard for a single node to scale up with ever growing number of requests and bandwidth requirements. One of the requirements formulated in Chapter 1 is scalability, so it is logical to find a distributed solution for the storage needed.

The terms *decentralized* and *distributed* have a slight difference in meaning. *Distributed* means that multiple computers are involved (see the definition by Tanenbaum and Van Steen given in Chapter 1), but there can still be a central point of coordination. *Decentralized* means distributed, with all nodes being equally important.

The problem of storing data in a distributed manner has been researched extensively. A number of distributed data storage solutions have been created that have proven to work well. Many of these databases—DNS is a prime example—require some form of central coordination. Other solutions such as OceanStore [23], that are based on a P2P-network, do not require central coordination and are thus decentralized. Not all P2P networks are completely decentralized however, for instance the well-known Kazaa [24] P2P file-sharing network gives some nodes coordination tasks, introducing a hierarchy.

The Fonkey location service, that features a flat name space and utilizes a decentralized trust model, does not require any form of hierarchy. Consequently, a decentralized storage

system is a natural fit. Because it is Fonkey that will be responsible for the actual storage, this chapter is dedicated to finding a suitable P2P decentralized storage solution for Fonkey.

3.2 Peer-to-peer networks

In peer-to-peer (P2P) overlay networks, multiple nodes, usually on multiple machines on geographically distant locations, connect with each other. Between the nodes in the network, information of any kind can be exchanged. Each node has its own unique NodeID, calculated in a way that differs per overlay network type. Messages between nodes are passed on progressively towards the NodeID of the message destination. Besides sending simple messages needed for basic network communication, the P2P nodes can of course also send messages that encapsulate complex data.

A detailed discussion of the different types of P2P overlay networks can be found in the survey of Lua *et al.* [26]. In this section, the two classes of P2P overlay networks that can be distinguished are discussed: the structured and the unstructured overlay network. Furthermore, the abstractions that exist upon structured overlays are examined.

3.2.1 Unstructured overlays

Structured and unstructured P2P overlay networks both have their strengths and weaknesses. One has better data retrieval guarantees, the other offers better search facilities. These properties are elaborated upon in this section.

Lua *et al.* [26] define unstructured networks as “overlay networks [that] organize peers in a random graph in flat or hierarchical manners [...] and use flooding or random walks or expanding-ring Time-To-Live (TTL) search, etc. on the graph to query content stored by overlay peers”. Two popular examples of unstructured networks are Gnutella [47] and Overnet [2], which are mainly used in file-sharing systems. Unstructured networks are able to retrieve content based on keywords and partial query strings. It does so by using flooding: a query is forwarded to a large number of other peers and the other peers send a list of matching items back to the peer where the query originated. It should be clear that this querying method does not scale because the query load will grow linearly with the number of queries and the network size. Flooding is efficient for locating highly-replicated items. In large networks however, not every other peer can be queried, so the chance of locating rare objects is slim. There is no guarantee that an object that exists somewhere in an unstructured network will be found when a query on it is performed [25].

3.2.2 Structured overlays

In structured networks, “the overlay network assigns keys to data items and organizes its peers into a graph that maps each data key to a peer” [26]. The primary property of a structured overlay network is that a deterministic hash function is used for allocating data, that is to say, each value or data object is consistently mapped to a unique peer. This is achieved by giving each peer a unique NodeID and using a hash function to determine the mapping of the key to a NodeID. On average, locating any object in a structured overlay network can be done in $O(\log N)$ “hops”, where N stands for the number of nodes in the network. The number of hops needed to locate an object is called *routing performance*.

3.3 Abstractions in structured P2P networks

Dabek *et al.* [14] distinguish three **tiers** in structured overlay network abstractions, illustrated in Figure 3.1. This section discusses the three tiers and the services they provide to distributed applications.

3.3.1 Tier 0: The Key-based Routing Layer

The lowest tier (tier 0) is the *Key-Based Routing* (KBR) layer. This layer, common to all structured overlay networks, is responsible for routing application objects to a unique node. The primary node where an object is stored is the one whose NodeID is numerically closest to the hash code belonging to the application object.

Examples of P2P protocols that implement a KBR layer are Pastry [40] and Chord [45]. On higher level tiers, abstractions on top of the KBR layer allow applications to use efficient APIs for purposes such as group messaging and data storage.

3.3.2 Tier 1 abstractions: DOLR, CAST, and DHT

Three higher level abstractions in structured overlays can be identified: Distributed Hash Tables (DHT), Distributed Object Location and Routing (DOLR) and group anycast/multicast (CAST). These three abstractions are located at Tier 1 in the three-layer architecture. Dabek *et al.* [14] propose a **Common API** for each of these abstractions.

The DOLR abstraction is built around the concept of publishing object replicas (*endpoints*), where each replica has an identical *objectId* and messages addressed to a certain *objectId* will be routed to the nearest available endpoint with this ID. For this purpose, DOLR offers operations to publish or unpublish (revoke) the availability of an object, and an operation to send a message to a certain object. In the Common API for DOLR, the *publish* operation makes an object available at the physical node where this operation was performed. The *unpublish* operation removes node-object mappings. Finally, the *sendToObj* operation delivers a message to *n* nearby replicas of an object [23]. It is possible to use DOLR to build a decentralized storage system.

CAST is aimed towards scalable group communication and coordination. It provides applications with multicast and anycast capabilities. The Common API operations supported in CAST are *join*, *leave*, *anycast* and *multicast*. With these operations, peers can join or leave a group, send a message to all members of a certain group of nodes (multicast), and finally, anycast forwards messages down to group members, but not necessarily to all of them.

The distributed hash table (DHT) abstraction offers a store- and retrieve functionality, similar to a traditional hash table. The operations it provides are *put*, *get*, and *remove*. These three operations form the Common API for distributed hash tables, depicted in Figure 3.2. Due to the one-to-one mapping of data to peers, queries on partial keys are—unlike unstructured P2P networks—not supported by strictly Common API-compliant DHTs. More advanced functionality that is found in the Java Collections API [7] such as methods to list all keys in the table, or to retrieve an inverse index, are not supported either by the Common API.

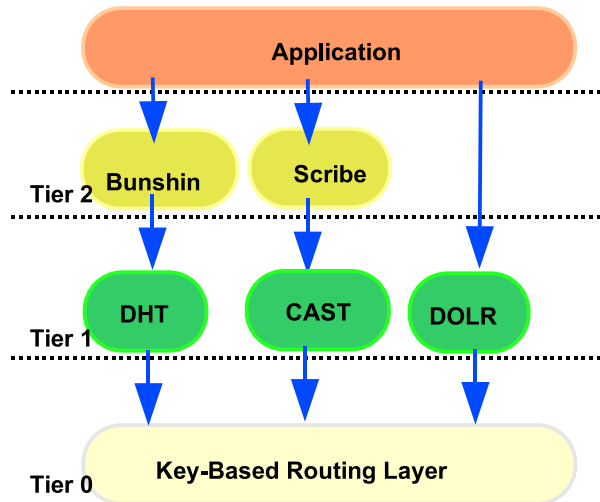


Figure 3.1: Three-layer architecture in overlay networks. Adapted from Mondéjar *et al.* [29]

3.3.3 Tier 2 abstractions: Bunshin, SCRIBE and OceanStore

Some utilities built upon the tier-1 abstractions discussed above, are Bunshin, PAST, SCRIBE [10] and OceanStore. They extend the tier-1 abstractions with features that facilitate the creation of distributed applications. Bunshin and PAST extend a DHT, SCRIBE extends CAST and OceanStore builds on DOLR. Situated between a tier-1 abstraction and the actual application, these utilities are seen as tier-2 abstractions.

3.4 Functional requirements and constraints

From the above sections it should be clear that the scalability requirement formulated in Chapter 1 can be fulfilled by using a decentralized P2P storage infrastructure. The Fonkey public key distribution infrastructure introduced in Chapter 2 provides the Foncation location service with identity verification functionality. Because Fonkey can be used as an infrastructure for authentication as well as a data storage medium, the decentralized version of Fonkey should be implemented using a distributed storage back-end such as a DHT to build the distributed, secure location service.

It is essential for Fonkey to support searches based not only on unique package identifiers, but also based on *name*, *signature*, or *subject* (see also the Fonkey description given in Chapter 2). Since unstructured P2P networks support searching for (partial) keys, and structured ones do not, an unstructured storage solution would seem to be the best solutions. However, unstructured networks do not guarantee that every object stored in it can be retrieved at any moment at any place, even if it is available (see Section 3.2.1). Since the Fonkey specification is not designed to cope with this fact, the distributed back-end to be used will have to be based on a structured overlay network instead. Many abstractions on structured overlay networks use methods such as replication to make sure that even when the primary peer for an object fails, it will still be available on other peers.

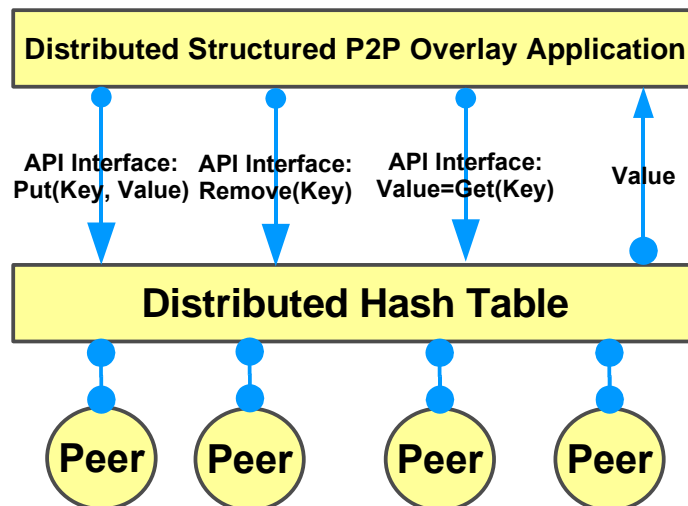


Figure 3.2: Application Interface for structured DHT-based P2P overlay systems. Adapted from Lua *et al.* [26]

Looking at the tier-1 structured overlay abstractions discussed above, it is clear that CAST is not suitable as a back-end for a decentralized Fonkey implementation. It has no operations for storing data, and there is no special need for group communication or for distributed objects that need to be able to receive messages because the data objects in Fonkey are passive in nature. That leaves the DOLR and DHT abstractions to choose from.

3.4.1 Evaluation criteria

The following evaluation criteria for a Fonkey decentralized storage system have been defined, some of them practical in nature:

- The DHT must support complex queries for (partial) values. This is needed to look for Fonkey packages with (a combination of) specific properties. For example, a Fonkey query could look like: *“Retrieve all packages of type SIGNATURE which have package P as subject”*.
- Preferably written in Java. The AgentScape middleware, introduced in Chapter 1, has been written in Java and C. Implementing the Fonkey infrastructure with Java, too, makes it easy to leverage Java-specific features like RMI in the distributed location service prototype. A demand that naturally follows from this, is that the API advertised by the DHT should match the AgentScape programming model and that it should be reasonably well-documented.
- An implementation should be available under a Free and Open Source license. The AgentScape agent middleware is published under a Free/Open Source license, so any new libraries or components that are to be part of the standard distribution should have a compatible license.

Range and keyword queries in DHTs

A number of proposals have been done to surmount the limited query possibilities in DHT overlay networks. Theoretically, most DHTs that are evaluated could meet the complex query criterion if a search system such as the Prefix Hash Tree (PHT) [35], *mSearch* [20], a structured-unstructured hybrid scheme [9], or a query-to-query index [18] is implemented upon it. A brief discussion of these schemes can be found in Chapter 6. As no implementations of these systems were available at the time this research was conducted, time constraints forced further exploration of them to be deferred to the future.

3.4.2 Candidate storage systems

A number of tier-2 structured P2P overlay networks have been considered as back-end for the distributed Fonkey storage. Below, a summary is given of the considered overlays.

- **OceanStore** [23] is a high-level storage system of the DOLR-type that has an interface comparable to a UNIX file system. It does not offer any query functionality beyond that of a traditional file system. OceanStore is designed for storage of very large amounts of data (in the order of 10^{10} users with each 10,000 files). It is based on Tapestry [53], a scalable, fault-tolerant object location and routing architecture with a strong notion of locality and load spreading features. The owner of an object published in Tapestry can decide on what peer the object will be stored. The routing method in Tapestry guarantees that objects will be routed to a peer in at most $\log_B N$ logical hops, where B stands for the NodeID base and N stands for the total number of nodes. An open source Java implementation of Tapestry is available, as well as of OceanStore. According to Rhea [36], the OceanStore API is quite unintuitive and could be improved.
- **DHash++** [13] is a DHT built on the Chord KBR layer. It is designed for high throughput and low latency. Nodes in a Chord P2P overlay network are organised in a ring with max. 2^{160} nodes. Nodes keep a *finger table* of NodeIDs that follow them at power of 2 distances in the identifier space, and a *successor list* of immediate successors. The basic lookup process in Chord is iterative: when a lookup on a key is performed from one node, it questions other nodes, that will reply with the IDs from their finger table that are still lower than the desired ID. The predecessor of the desired node will thus be reached within $O(\log N)$ steps. The desired item will then be present in the successor list of the predecessor node. DHash++ supports load balancing techniques a number of optimizations to the iterative lookup process, in order to reduce lookup latency. A C++ implementation of DHash++ and Chord is available under an open source license.
- **PAST** [41] is a persistent storage utility built on an open source implementation of the Pastry [40] KBR layer, called FreePastry. Similar to Chord, Pastry has a circular NodeID space, which ranges from 0 to $2^{128} - 1$. Message routing is performed by peers forwarding messages to another peer with a NodeID that is always numerically closer to a given key than the current NodeID. Routing performance in Pastry is of $O(\log_{2^b} N)$, where b is a parameter with a typical value of 4, relating to the number of bits that the ID of a node in the message forwarding chain has to have in common with a given key, before the node is selected as the next one to forward a message to.

	Implementation	Documentation and API	Partial queries
DHash++	C++	–	o
Bamboo	Java	–	o
OceanStore	Java	+	–
PAST	Java	+	o
Bunshin	Java	+	+

Table 3.1: Comparison table of structured P2P overlay networks

The FreePastry implementation of Pastry, as well as the PAST abstraction, are written in Java and published under an open source license.

- **Bamboo** [37] uses the same routing algorithm that Pastry uses. Bamboo’s main contribution is its resilience to performance problems caused by churn¹, for example compared to PAST. It uses the “staged event-driven architecture” (SEDA), just as Tapestry. This is a programming model that differs substantially from the one used in Agent-Scape implementations. Furthermore, the Bamboo code appears to be quite opaque and sparsely documented. These two factors would complicate the prototype implementation effort. A Java implementation of Bamboo is available under an open source license.
- **Bunshin** [29] is, just like PAST, built upon the FreePastry KBR layer but features additional functionality such as inverted indices that map keywords to keys. Bunshin supports complex queries by extending the basic DHT interface with an operation to associate keywords with key-value pairs and an operation to retrieve a list of keys that match one or more keywords. Initially, Bunshin was created out of dissatisfaction with PAST, that was found to be too unreliable. The services Bunshin offers on top of a generic tier-1 DHT are a system of adaptive caching and replication, providing load balancing and robustness in case of node failures, respectively, and a facility to mark key-value pairs with context tokens (also called *keywords*) that open up the possibility for complex queries.

The Bunshin source code and API are not very well documented but the API is fairly straightforward. Applications using it, in contrast to OceanStore and Bamboo, do not have to adapt to an event-driven model. Lookup performance is of $O(\log N)$. Tests by Bunshin’s authors have shown that Bunshin’s performance is at least comparable to the Chord-based CFS distributed file system [12, 1]. A Java implementation of Bunshin is distributed under the GNU Lesser General Public License.

A comparison in table form is given in Table 3.1. A ‘+’ means a good evaluation, a ‘o’ indicates average and ‘–’ denotes a bad evaluation on a criterion. The average mark given to DHash++ and Bamboo in the Partial queries column, is caused by the fact that although it would be possible to implement a complex query system on them, it is not possible by default, and time did not permit to explore this avenue further.

Generally speaking, Fonkey packages contain only small amounts of data. The Fonkey conceptual model does not specify any requirements with regard to the way of storing data,

¹Churn is the process of nodes continuously leaving and joining the network.

Operation	Description
<code>insert(String tokens, Id idValue, Serializable value): void</code>	Inserts a key-value pair with key <i>idValue</i> , value <i>value</i> and a string of <i>tokens</i> (keywords, separated by spaces).
<code>query(String tokens): ResultSortedQueue</code>	Queries the DHT for key-value pairs annotated with one or more of the <i>tokens</i> . Returns the Id's of the found keys in a queue sorted on number of matches per keyword token.
<code>removeObject(Id idValue): void</code>	Removes the value belonging to the key <i>idValue</i> from the DHT.
<code>retrieve(Id idValue): Serializable</code>	Gets the value from the DHT that belongs to key <i>idValue</i> .

Table 3.2: Relevant part of the Bunshin keyword-extended DHT interface.

but it does state that packages may be identified by a combination of fields, such as the public key and the name property, that make a package unique. When such an identifying combination of fields is seen as the *key*, and the package itself as the *value*, a *key,value* mapping is obtained, which is exactly the format that a DHT supports. Combined with the fact that the interface of most of the tier-2 DHT abstractions evaluated is far simpler than that of OceanStore, the only DOLR abstraction available, the conclusion can be drawn that, for a decentralized Fonkey implementation, a DHT is preferable to a DOLR-based system.

The DHT that best satisfies the given non-functional requirements, is Bunshin, because of its support for keyword-to-key indices so that complex queries are possible. Hence, the choice of using Bunshin as the DHT back-end for the distributed Fonkey prototype was a fairly obvious one.

3.5 Implementation of Fonkey on Bunshin

To be able to implement the Fonkey infrastructure using the Bunshin DHT, a design has to be made to map Fonkey packages to *key,value* pairs. The design should take the requirement for partial queries into account (see the criteria specified in section 3.4.1). It should take advantage of Bunshin's keyword support to support partial queries.

Table 3.2 shows the part of the Bunshin DHT interface that is relevant to the implementation of the Fonkey distributed data store. The methods it offers that regular DHTs do not, are `insert` and `query`. These are responsible for inserting a *key,value* pair into the DHT together with associated keywords, and retrieving a list of *keys* that are associated to one or more keywords, respectively. Note that in Bunshin, the equivalents to the Common API `put` and `get` operations, are called `storeObject` (not used directly in the distributed Fonkey design) and `retrieve`.

A high-level design of how Fonkey communication with the DHT and the outside world happens, is shown in Figure 3.3. The Fonkey process and the distributed storage utility, displayed as a single entity in Figure 2.1 for clarity, are drawn separately here. As has become

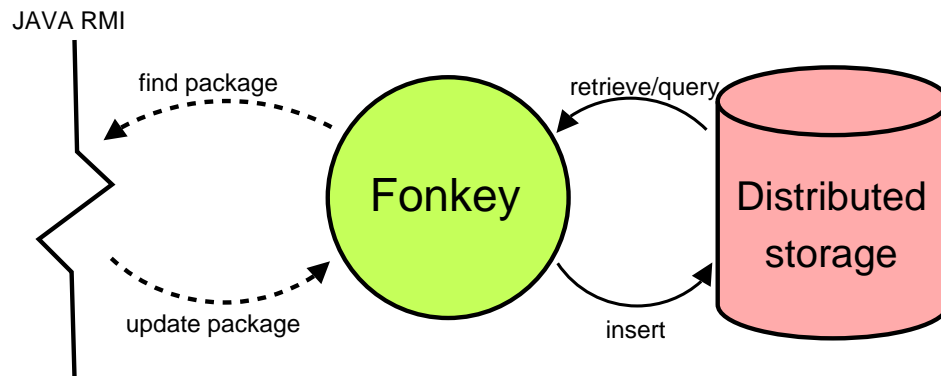


Figure 3.3: Translation of Fonkey operations to DHT operations.

clear in this chapter, they are in fact separate. The Fonkey client is conceptually separated from the location service (not drawn in Figure 3.3) as well. Because of this separation, a Remote Method Invocation (RMI) interface is used. Using RMI, it will be able to communicate with other processes. The assumption made in this design is that an appropriate RMI security policy is in place. Fonkey does not perform any additional identity checks on the applications that call it on its RMI interface.

Note that multiple instances of the location service could theoretically share the same Fonkey instance. That is not advisable, however, because it would mean less participating peers in the DHT, which leads to reduced scalability.

3.5.1 Using keywords to support partial queries

In the case of Fonkey packages, their public key, combined with the name or subject (depending on the package type), are unique identifying properties. For performance reasons, the public key's *fingerprint* can be used instead of the full key. A public key fingerprint is a short cryptographic checksum over a public key. It is very hard to calculate a public key that produces a given fingerprint, so using the fingerprint instead of the public key itself is not a security problem. As Fonkey needs to be able to retrieve packages based on these identifying properties, the logical consequence is that those properties should be added as keywords, so that an inverted index that maps keywords to hash table keys, can be kept for them.

An overview of the keywords that are necessary to support proper Fonkey operation, is shown in Table 3.3. The keywords are dependent on the package type. A Named Package is associated with the Name property of the package; a Signature Package gets the Id (see section 3.5.2) of its subject as keyword.

With these keyword associations, it is possible to support the type of queries that are required in Fonkey. When the location service has to look up data concerning an agent location, it will be translated to a query for a Named Package with the agent's Id as the Name property. Bunshin is then used to retrieve the Id of the Named Package that is associated with the agent's name. If the agent location has indeed been registered with the location service before, then the desired Id will be found. The package can subsequently be retrieved

Package type	Keywords	Description
Key Package	none.	
Signature Package	<i>Subject</i>	<i>Subject</i> stands for the Id of the subject package.
Named Package	<i>Name</i>	<i>Name</i> stands for the <code>Name</code> property.

Table 3.3: Keywords used to support queries for partial values in Bunshin

using this Id and the requested package will be returned to the location service. The process is analogous for looking up Key and Signature Packages.

3.5.2 Fonkey-Bunshin interaction

In this section, which concludes the Chapter, the mapping of Fonkey packages to keys and values suitable for a DHT will be discussed. Also, three scenarios will be given that illustrate how Fonkey interacts with the Bunshin DHT.

Mapping Fonkey packages to *key,value* pairs

The hash code belonging to a *key* is also known as the **Id**. In much the same way as a hash code in a regular hash table, an Id gives no information about the type or content of the value it belongs to. The hash calculation function is located in the Pastry layer, but can be accessed from higher layers. This hash function returns values that account for a uniform distribution of data items over the P2P network.

The design presented here simply uses the uniquely identifying properties of a package, such as the publisher's public key fingerprint, as input for the hash calculation. Using these properties, it will be possible to reconstruct the Id of a package later by calling the hash function. To maintain a conceptual separation between Fonkey and the storage implementation, Fonkey treats Id objects as generic Java Objects.

The *values* passed to Bunshin are, obviously, Fonkey packages. The only condition Bunshin imposes on values is that they implement the Java `Serializable` interface. The distributed Fonkey prototype based on the design presented in this section, fulfills this condition.

Scenario: Inserting a Named package

When a Named Package *np* is inserted by the location service, because the service needs to publish the location of an agent, the Fonkey `update` method is called with *np*, the package to be stored, as parameter. The package is then stored in the DHT with the *Name* of *np* as keyword, by calling the `insert` method on Bunshin, with the package's Id as *key*, the package object itself as *value*, and the keyword as *tokens* parameter. How exactly the pair is stored in the P2P network is left to the Pastry Key-Based Routing layer.

Scenario: Retrieving a Signature package

When the authenticity of a public key K must be established, the Foncation location service requests from Fonkey all Signature Packages that have as subject Id_K , the Id for the Key Package corresponding to K . The Bunshin DHT accesses the keyword index it keeps for Id_K and returns the set of corresponding Signature Package Ids. These Signature Packages are subsequently retrieved one by one using the `retrieve` method (equivalent to `get` in the Common API) with their Id as parameter and returned to the location service.

Scenario: Removing an expired package

The semantics in Fonkey are such that one cannot directly remove a package from the package store. Instead, the package should be re-published with a higher version number than the old one, with a payload that states that the old package no longer exists.

However, if information was never removed from the database, the database would never stop growing, which would obviously lead to problems. For this reason, Fonkey must remove a package once its time-to-live has expired. Using the Bunshin `removeObject` method the expired package is removed and the keyword indices pointing to its Id are updated.

The automatic removal functionality described in the previous paragraph has not yet been implemented in the prototype built during this research, but it could be done by making each participant in the distributed Fonkey system responsible for cleaning up his own expired packages. A scheme that would take into account publishers that do not take their responsibility and fail to remove their own expired packages, would require more integration of the DHT and the Fonkey infrastructure, so that each peer could inspect the local storage and clean up expired Fonkey packages.

And when you finally reappear
at the place where you came in
you've thrown your love to all the strangers
and caution to the wind

Love over Gold
Dire Straits

Chapter 4

Experimental results

In this chapter, the implementation of the distributed and decentralized Fonkey prototype will be discussed, as well as the results of the tests that have been performed with the Fonkey-based location service. The main objective of these tests is to assess the scalability of the prototype. Secondly, the responsiveness of the location service under average and under heavy load is measured.

4.1 The prototype

A centralized implementation of Fonkey written in Python existed before the development of the decentralized prototype was started. The architecture of the centralized Fonkey prototype, with the exception of the data storage part, was the basis of the decentralized Java prototype. The data storage and retrieval functionality has been written from the ground up to build on the Bunshin distributed hash table. How this was done is described in the previous chapter.

4.1.1 The software architecture

Figure 4.1 shows the UML model of Foncation. The three Fonkey package types all inherit from a base Package class. The `FonkeyService` class in the diagram is the class that provides Fonkey operations to the outside world. When the location service needs to store or retrieve a Fonkey package, it talks to the `FonkeyService`. The `FonkeyService`, then, uses a `PackageStore` object to manage the storage and retrieval of packages.

A `PackageStore` object in Fonkey separates the program logic in the `FonkeyService` from the storage facilities. The `PackageStore` interface hides the actual implementation from the `FonkeyService`, so that the `FonkeyService` could be provided with different (distributed or centralized) storage systems without having to touch the `FonkeyService` implementation. The implementation of the `PackageStore` interface, in this case `DHTPackageStore`, is in turn separated from the actual DHT implementation. This means that different DHT implementations could be used. Note, however, that the prototype implementation created in the course of this research, requires that support for keywords is available in the DHT interface. In order to map Fonkey packages to an object with associated context tokens (keywords) that the Bunshin DHT accepts, the `AnnotatedSerializable` interface was introduced.

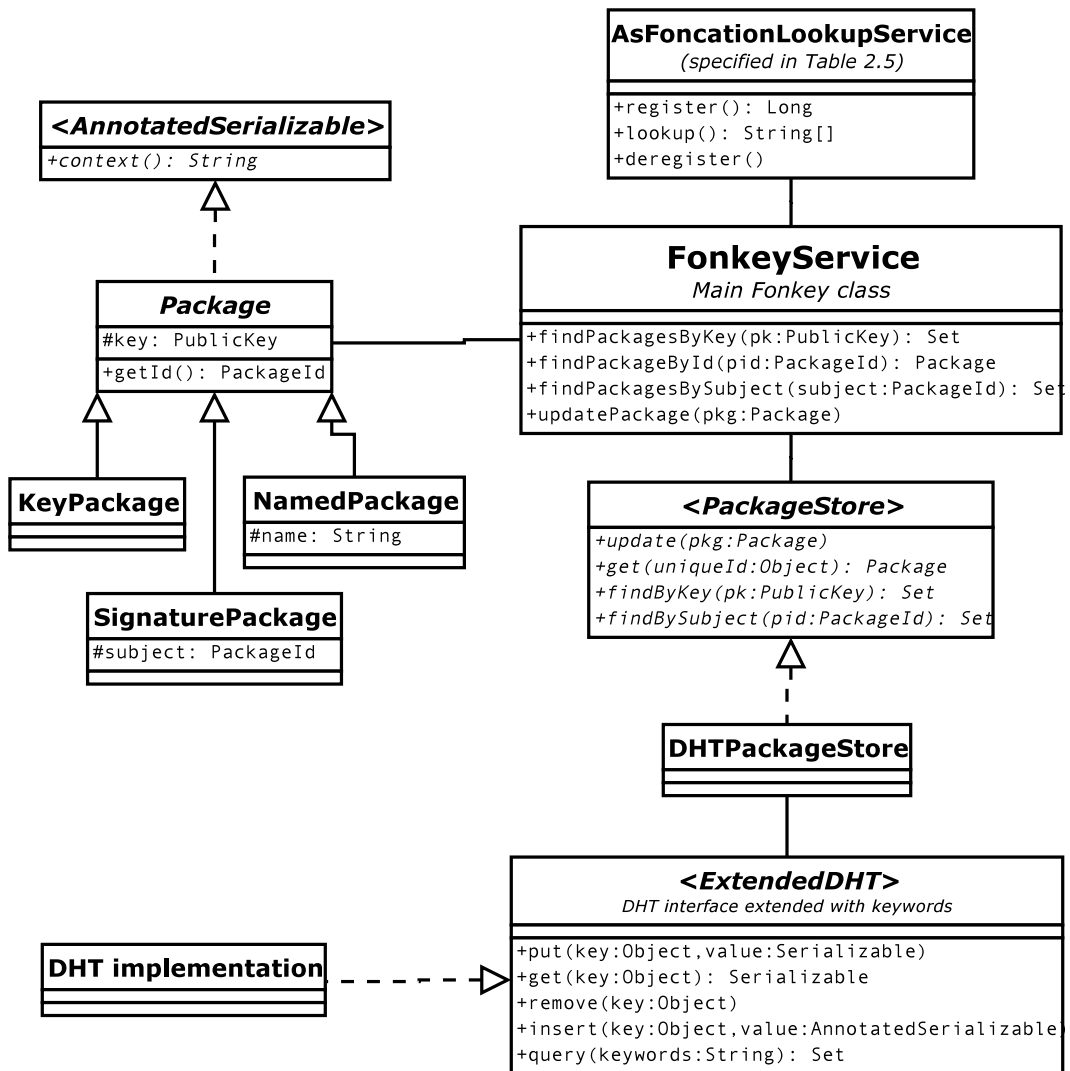


Figure 4.1: UML conceptual model of the distributed Fonkey infrastructure. Interfaces are placed between pointy brackets.

This interface extends `Serializable` by specifying the method `context()` that returns a `String` containing one or more context tokens pertaining to the object being serialized. All Fonkey packages implement the `AnnotatedSerializable` interface. The DHT implementation will call the `context` method of the object to store to learn about its keywords. This way, a Fonkey package can be located after insertion by performing a keyword query.

4.1.2 Integration with AgentScape

The AgentScape middleware contains a simple, centralized location service that communicates over XML-RPC and does not have any of the security and authentication features that Foncation has. A class `AsFoncationLookupService` has been created that can be used by the AgentScape middleware without much modification. Its interface is discussed in section 2.4.1. The similarity of that interface with the simple AgentScape location service makes it possible to perform a proper performance comparison of both.

Fonkey has been implemented such that it can communicate with another process. This way, the distributed storage and retrieval functionality is separated from the location service so that starting and stopping the location service does not require shutting down all of the AgentScape middleware. The `AsFoncationLookupService` class used in the tests is implemented using the RMI method to communicate with Fonkey. The choice for RMI is arbitrary, other communication protocols are also possible.

4.2 Test setup

This section describes the goals of the experiments that have been performed, as well as the way the experiments have been performed and the hardware that was used to do it.

4.2.1 Used hardware

The experiments have been performed on the DAS-2¹ system, consisting of 200 1GHz Dual Pentium III processors with at least 1 GB of RAM each. The DAS-2 consists of five clusters of nodes, located at five universities, connected by the Dutch university Internet backbone. The operating system is Red Hat Enterprise Linux AS 3 (kernel version 2.4.21). In this thesis, only the cluster at the Vrije Universiteit (72 nodes) was used. Only for the single-node tests a different machine was used; a system built around an AMD Athlon 64 3500+ processor (running at 2.2 GHz) with 1 GB of RAM. The operating system for that machine is Debian GNU/Linux with kernel version 2.6.15. Bunshin version 2.2 was used, based on FreePastry version 1.4.2.

4.2.2 Goals

The scalability requirement formulated in Chapter 1 states that the location service be able to store a huge number of objects as well as to serve many concurrent requests. To verify this claim for Foncation, a number of experiments have been performed that fall into two categories. The first category is a comparison with the existing centralized AgentScape location service, to assess how Foncation stacks up to it when run on a single computer. The second,

¹<http://www.cs.vu.nl/das2>

divided in a number of sub-experiments, is a test of the performance and scalability aspects of the decentralized location service prototype. Additionally, a baseline measurement was performed on one of the DAS-2 nodes, to find the minimum computing time required for Foncation without network communication delays. The experiment categories are described in the following section.

4.2.3 Experiment categories

All experiments are centered around registering and looking up identifier – agent location pairs (hereafter referred to as *items*) in the location service, as fast as possible. Knowing the maximum performance on a varying number of nodes is key to validating the scalability and mobility requirements formulated in Chapter 1. Also, all experiments were run three times. Consequentially, the values shown in the figures in this chapter are averages from three runs.

The centralized experiment, run on the Athlon 64 machine, is designed to show the relative performance hit that the authentication and decentralization features cause, compared to the old centralized location service from AgentScape. Foncation is run unconnected to any other nodes, as if it were a centralized location service, so that network communication delays cannot dilute the timings. The centralized location service is started without any special modifications. A test application (described in the next section) is then executed on it.

The experiments in the decentralized category have been run on 2, 4, 8, 16, and 32 nodes concurrently. These experiments measure the average time it takes for a high number of lookup and register operations to complete when either some or all of the participating nodes are performing operations at the same time. The goal is to find if even under heavy load, the decentralized location service performs adequately. The difficulty here is that there is no definition in literature of what is adequate performance for a decentralized agent location service, but the author proposes that a location service with acceptable performance will at least be able to handle requests quick enough to ensure that the number of communication errors between migrating agents in a large-scale distributed multi-agent system caused by inaccurate or delayed information from the location service, is negligible.

The decentralized experiments have been divided as follows: One experiment exclusively registers new items; one exclusively looks up items that had been inserted before; the previous two experiments are repeated with only four nodes actively participating; and in a “mixed” experiment 80% lookups and 20% registrations are performed, resembling a real-world situation a bit closer than the other two decentralized experiments.

4.2.4 The test application

As stated earlier, the Foncation test application that was written inserts and/or looks up a set of items in the data store as fast as it possibly can, on each node where it is started. Each item in the test set (described in the next paragraph) is inserted into the location service, and it is retrieved again as part of the lookup timing experiments. This means that each node which is part of Foncation is continuously handling lookup or register requests, as well as responding to messages from other nodes and storing the data it is responsible for according to the DHT routing algorithm.

The number of milliseconds it takes before a register or lookup operation is completed, is measured and summary statistics of the timings are calculated. The test application breaks down the timings into time spent in the authentication layer of Foncation and the time spent in Bunshin, and records lookup and register operations separately.

The test set

To test the functionality of the location service, it is not required that actual agent identifiers and locations are used, any similar data will suffice. Agent locations are similar to URLs, and since it is straightforward to quickly retrieve a lot of URLs from web browser history and similar sources, the test set used was built out of 2,000 unique URLs. For mock agent identifiers it was convenient to use the hostname of the node combined with a counter; this ensures uniqueness of the identifiers, and makes it easy to verify if expected items are in fact present in the data store. Most importantly, this approach prevents the same *name* keyword from being used for packages from different nodes, which would have a bad effect on lookup performance: the location service then has to perform more lookups to find the correct Named Package if it gets more than one result from Bunshin on one agent name that was registered as a keyword.

A shared trusted key

For proper operation, the Foncation authentication algorithm needs a trusted public key (see section 2.4.4). Such a key was provided by distributing a “root key” over a separate channel to the participating nodes when the test application initializes. Also, each node has its own public-private key pair which is used to sign published Fonkey packages, and each node’s public key in turn is signed by the root key. This way, the authenticity of information in the data store in this experiment is always established in two verification steps: one to check the Fonkey Key Package that signed a Named Package representing an agent location, and one to check the Signature Package that expresses the trust from a trusted key (the root key in this case) in the Key Package from the first step.

Bunshin initialization

To ensure that Fonkey, the Foncation location service, the Bunshin DHT, and the test application are launched in the correct order, a shell script was written that starts up the components as required. The experimental application creates one instance of the lookup service per participating node. Foncation takes care of initializing the Pastry and Bunshin DHT layers by itself.

When a Bunshin DHT node comes up, it needs to know about another node (a peer) to connect to, for it to be able to join the ring of nodes. If no peer is made known to this node, it will assume that it is the first node in a new ring. To facilitate the experiment, one computer was designated as a “bootstrap node”. The nodes participating in the timing experiments all connected to it during the startup phase. During the experiment, the bootstrap node was a “passive” member of the DHT, which means that it could be used to store or retrieve data from by other nodes, but it did not initiate such operations itself. Of course, even with a “passive” DHT member, network communication is still involved. To be able to compare the

	Register (ms)						Lookup (ms)					
	DHT		Foncation		Total		DHT		Foncation		Total	
	\bar{x}	σ	\bar{x}	σ	\bar{x}	σ	\bar{x}	σ	\bar{x}	σ	\bar{x}	σ
Old					1.6						3.8	
New	17.0	15.0	13.4	7.0	30.4	17.2	5.4	12.7	0.6	1.1	6.0	12.8

Table 4.1: Registration and lookup measurement table for the centralized Agentscape location service and Foncation, averaged over three test runs of 2,000 items.

old AgentScape location service to Foncation, the “single node” experiment was performed on a separate machine where Bunshin was not connected to any other DHT nodes.

In order to execute the experiments, the Bunshin replication feature had to be switched off because it lead to an explosion of messages between the nodes. It appears that this is a software flaw that can be repaired, but for this experiment, it was decided not to use replication. Lack of replication means that items are not stored on multiple nodes. The theoretical downside is that data will be lost when a node fails, but in the experimental setup this was not an issue. Still, it is important to understand that the measurements represent a “theoretical maximum” with regard to replication.

A modification to the experiment that had to be done to stay within the 15-minute time limit for DAS-2 experiments was to reduce the number of items registered per node from 2,000 to 1,000 for the experiments involving 16 and 32 nodes. Thus, the maximum amount of items that was contained in the DHT during all experiments, was $32 \times 1,000 = 32,000$.

4.3 Results

This section discusses the results that have been retrieved from both the single-node and the multiple-node experiments.

4.3.1 A single node-experiment

The latest release of the AgentScape middleware at the time of this writing has a simple, centralized, single node location service. As stated before, the comparison of the “old” location service with the decentralized one has been performed on a different machine than the multiple nodes-experiment, so the time measurements resulting from this experiment are not comparable to the multiple nodes-experiments. What can be learned from the comparison results is how big the performance impact is of the operations related to authentication and decentralized storage.

Table 4.1 shows the results of the comparison. The **DHT** column indicates the amount of time spent in the Bunshin subsystem; the **Foncation** column indicates the time spent within the Foncation code responsible for the authentication operations. If the average time per operation for the old location service is multiplied by the total number of items and the result divided by the duration of one experiment ($(3.8 \times 2000) \div 7600$), a theoretical lookup throughput (locations per second) of about 263 is found.

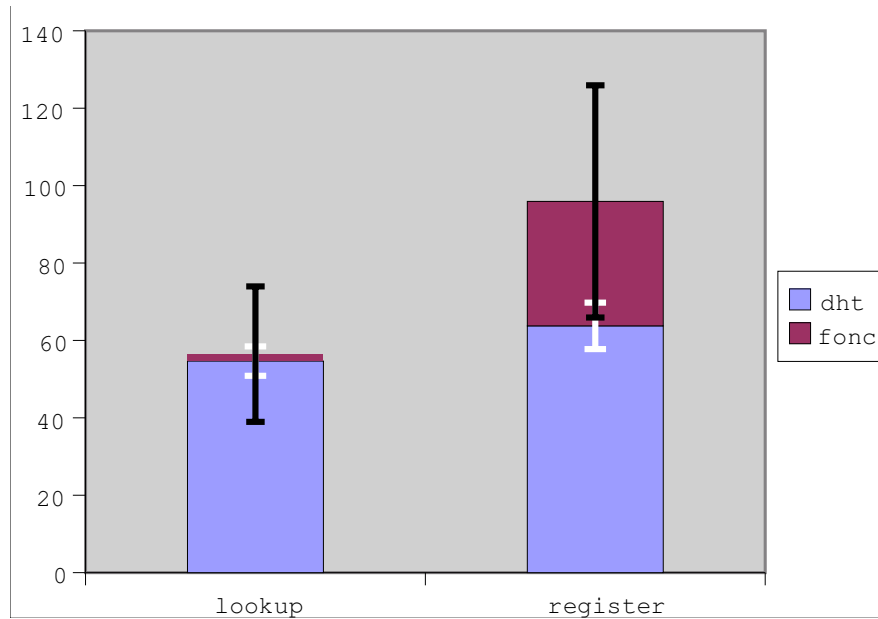


Figure 4.2: This graph indicates the ratio of time spent within Bunshin compared to the time spent in foncation authentication operations in a decentralized setting.

	Lookup	Register	$\sigma(\text{lookup})$	$\sigma(\text{register})$
dht	54.7	63.8	3.8	6.0
fonc	1.8	32.2	17.5	30.0
sum	56.5	95.9		

Table 4.2: The data that Figure 4.2 is based on.

4.3.2 Experiments on the DAS-2 cluster

Figure 4.2 shows the baseline measurement on a single DAS-2 node. The decentralized location service needs about 96 ms per register operation and about 56 ms for a single lookup operation. The decomposition of these timings is displayed in Table 4.2. These timings are the sum of the time spent in the Bunshin DHT and the time spent in the Foncation location service layer, where the authentication calculations are performed. The bars are split in two corresponding surfaces: the bottom part stands for the time spent in Bunshin, the top part for the time spent in Foncation. The black and white error bars indicate the standard deviation for the Foncation timings and the standard deviation for the Bunshin timings, respectively. Figure 4.3 shows the average time needed by Foncation for pure register and lookup operations on an ascending number of nodes. The error bars stand for the standard deviations. The timings that are shown have not been broken down in two surfaces like in Figure 4.2. Such a breakdown would not have added any new information because the time spent in the authentication layer stays constant; the increase in latency when the node pool grows is solely due to the DHT layer.

Since every additional node provides more “capacity” for handling location service opera-

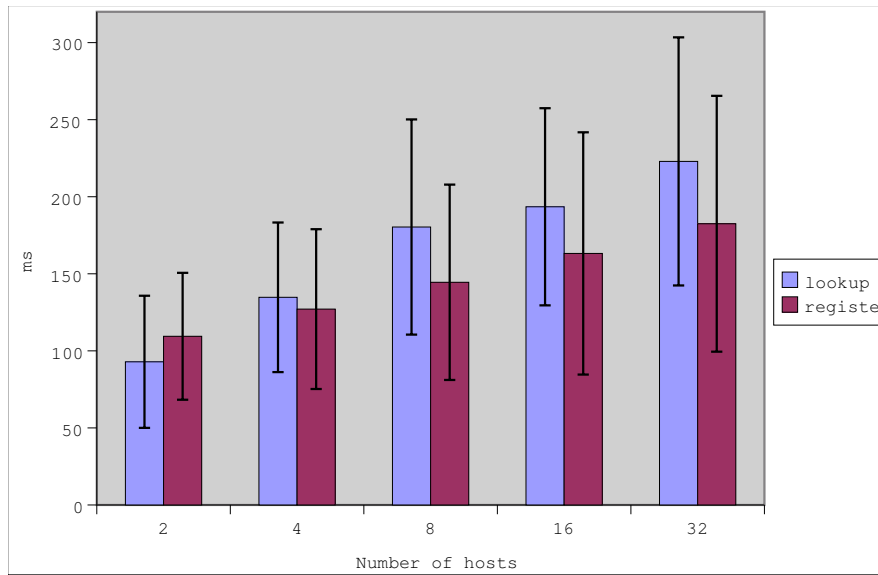


Figure 4.3: Timings for pure register and lookup operations.

tions, the throughput grows with the number of nodes involved. With the 32 nodes experiment, the throughput comes at 3.75 lookup operations per second per node. Multiplied by 32 this gives about 120 retrieved items per second.

Figure 4.4 depicts the average measurements for mixed lookup and register operations, in a 80% lookup / 20% register ratio. This ratio approaches the expected real-world usage. The order in which lookup and register operations were performed, was randomized. Again, the error bars indicate the standard deviations.

Finally, Figure 4.5 is the result of mixed measurements performed with only four active nodes. That is to say, in a pool of location service nodes of size 8, 16, and 32, the performance of the decentralized location service is assessed when just four of them are participating in actively registering items or looking them up. The other nodes are participating passively: they do store and return items when the DHT routing algorithm requires them to, but they do not perform register or lookup operations themselves.

4.4 Discussion

At first sight, it seems a bit strange that lookup operations in node pools larger than two nodes take longer to complete than register operations. This can be clarified by the fact that one lookup operation in the location service translates to multiple lookup operations at the DHT level. After all, as is explained in the previous chapter, the authentication mechanism must perform lookups of key and signature packages as well as lookups of the actual location data.

The time needed for a DHT operation far outweighs the time needed to process data in Foundation. For the most part, the timing results follow the $\log N$ curve (discussed in section 3.2.2) which is characteristic for DHTs. One interesting result visible in Figure 4.3 is that the lookup

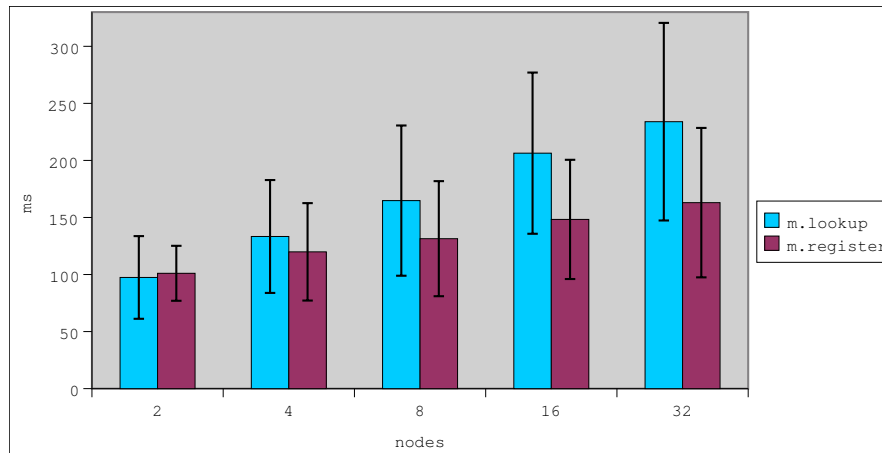


Figure 4.4: Timings for 80/20 mixed register and lookup operations.

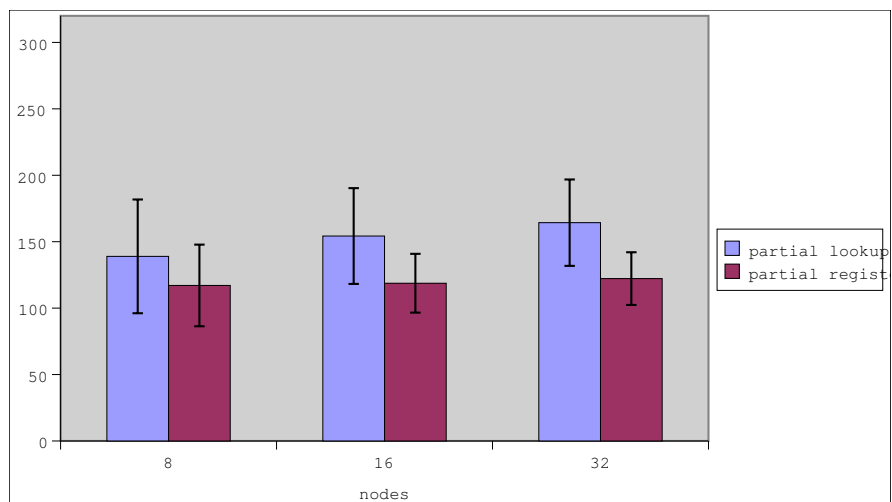


Figure 4.5: Timings for 80/20 mixed lookup and register operations with only 4 out of all nodes active.

timings for pure lookup operations seem to flatten a bit between 8 and 16 nodes. No definite explanation has been found for this phenomenon, but one possible explanation is that when the number of nodes reaches a certain threshold (around 8 nodes), the percentage of nodes that has to be contacted to find the destination node lowers. Thereby, the overall average lookup time is brought down. This effect is not clearly visible in the register-timings, which could be explained by the fact that for inserting an item into the location service, less actual *put* and *get* operations are required on the DHT level than for looking up an item. Neither is the described effect visible in the “mixed” and “partial” graphs (see below), so it appears to occur only when the location service is under a heavy load.

The measurements that have been performed with only 4 active nodes (the “partial” experiment) and the “mixed” experiment where 80% of the operations were lookups and 20% of the operations were registrations, are very encouraging. In a real-world setting, not every node in a decentralized location service will be performing operations at maximum speed, and lookup operations will be far more frequent than register operations. The measurements show that in the “mixed” experiment, the operations complete slightly faster than in the “pure” experiment, whereas in the “partial” experiment, the operations (especially the lookups) are not only completed significantly faster than in the other experiments, but also the increase in latency with a growing pool of nodes is hardly noticeable.

Still, the performance of the centralized location service is clearly better than that of the decentralized one with 263 lookups per second versus 120. This is to be expected given the security and decentralization features that Foncation has. However, the performance difference between the centralized and the decentralized location service grows smaller as the number of nodes in Foncation increases. At 32 nodes, the Foncation location service gives only about half the throughput of the old centralized location service. If the graphs are extrapolated, at a certain number of nodes Foncation will be able to handle more concurrent requests than the centralized version is able to.

Chapter 5

Related research

The previous chapters have shown how Foncation, a scalable and secure location service, can be built using the Fonkey public key distribution infrastructure and the Bunshin keyword-enabled distributed hash table. This chapter describes research by others which is related to the research done for this thesis. It can be categorized into the main topics mobile agent location services, distributed naming services, distributed PKIs and trust schemes, and search strategies.

5.1 Mobile agent location services

In this section, a number of other (agent) location services are compared with Foncation. The focus in this section is on agent location services that can be compared on characteristics such as security and scalability.

Kastidou *et al.* [22] have designed a hash-based scalable mobile agent tracking mechanism that uses special *Information Agents* to keep location information of the regular agents that they serve. The emphasis of this tracking mechanism is on scalability: the number of Information Agents can vary with the workload and a Chord-like hash function is employed to achieve constant lookup times. Security questions are not addressed in the design of this tracking mechanism. Unique about this approach is the fact that the location service itself is implemented using software agents. Using software agents to implement a location service is an example of a “buffered response” model (see Section 1.2.2).

The problem of locating mobile agents in wide area- or distributed systems has also been addressed by Di Stefano and Santoro [15], who propose a naming scheme and a location finding protocol called *Search-by-Path-Chase* that takes into account factors such as availability, scalability, migration overhead, and interaction overhead. It improves upon the “forward references” technique described in Chapter 1 by, among other aspects, dividing the distributed space in regions and sites where local agent location registries are kept. As a result, the chain of forward references is less likely to break. This protocol does not address security issues.

Roth and Peters [39] propose a global tracking service for mobile agents that is able to scale and takes security and migration issues into account. Their design minimizes the use of public key cryptography to achieve a good tradeoff between performance and security. Agents in this scheme have an *alias* (a human-readable name, such as “Harry”), and a machine-

readable *implicit name* (such as “5A:77:1E..”) which is mapped to the actual agent location. Aliases and implicit names are concepts very similar to the *symbolic names* and *object handles* described in Section 1.2. *Implicit* names are created by first calculating the cryptographic signature from the agent’s owner over the identifying parts of the agent and then applying the the Secure Hash Algorithm [17] to generate a one-way hash from it. The advantage over just using aliases (also referred to as *explicit names*), is that it is next to impossible for agents to impersonate other agents, because the owner’s private key is unknown to other parties. Data integrity is assured by encrypting the initial update request with the tracking server’s public key; all subsequent requests are protected by using cookies. Scalability is achieved by making multiple tracking servers responsible for a part of the name space.

FHNS (Fault-tolerant Home-based Naming Service) by Tolman [48] is a variant on the “forward references” category of agent location services where agents have a home base. In a home-based location service, the location of each agent in the multi-agent system is stored by one host (the home base) of the agent. Thus, there is no longer a single point of failure like centralized location services have. The impact of the failure of a home base is limited to the agents served by that home base. FHNS appoints home bases to agents using a consistent hashing function (see also Section 3.2.2) provided by a DHT, rather than using the originating host of the agent as home base.

FHNS supports both *location-dependent* and *location-independent* naming. Location-dependent naming requires that an agent’s home base has a list of other home bases, where the home base addresses are resolved using a regular lookup service such as DNS. With location-independent naming, the home base lookup is performed using a DHT such as Chord. The agent–location mappings from each home base are replicated on a number of different home bases with DHT identifiers near to their own, to preserve this information in case of a home base failure.

Directly querying an agent’s home base before querying the DHT, as the location-dependent method does, drastically cuts down on the number of messages that have to be sent over the network, between the sending of a query and receiving its result. Still the location-dependent method benefits from the presence of a DHT because it provides a fallback in case a home base fails.

This method differs from the “forward references” technique mentioned in Chapter 1 in that there is no chain of references along the path a migrating agent has traveled. Instead, either the agent’s home base is queried directly, or when the home base name is unknown, a DHT and its consistent hash function are used to find it. Agents are responsible of notifying their home base about their current location. Taking advantage of the DHT, FHNS is able to perform lookups even in the case of failure of all but one agent home base. This is possible because each home base also acts as a node in the DHT.

The two-layered technique, where the home base lookup is done first (with DNS when possible, and with a DHT otherwise), yields better performance than using a DHT for everything. Like Foncation, scalability is achieved as a result of the scalable properties of a DHT. Security has not been incorporated in FHNS, but its design does allow for extensions such as a PKI-based authentication mechanism.

Comparison

When comparing the location services discussed in the previous paragraphs to Foncation, a number of differences become clear. First and foremost, with the exception of Roth and Peters' system [39], none of the discussed agent location services has an authenticity verification mechanism. The mechanism of Roth and Peters is not as strong as that of Foncation: whereas Foncation signs all data packages using public key technology, only the initial registration request is signed by Roth and Peters' system. All subsequent requests are protected by cookies with weak encryption.

Roth and Peters argue that the performance hit caused by a full PKI authentication system is too big. This thesis shows that while the performance hit is significant, it does not have to be prohibitively high. From the experimental results in Chapter 4 it becomes clear that a decentralized location service that uses a PKI-based authentication mechanism can respond to lookup requests within 300 ms under a high load. Only for mobile agents that migrate multiple times per second, does this not suffice. It has to be noted that these results were gained in a setup where the recursive lookup of a trusted key in the web-of-trust always took exactly one step, so in a setting with longer trust chains, the Foncation lookup performance will be proportionally worse than 300 ms. FHNS shows that it is possible to benefit from the scalability properties of a DHT without using it for every lookup action, thereby making response times even better. It is not possible to apply this technique to Foncation because that would require a drastic change in design. However, if Bunshin would use a more optimized way of reversed indexing, it would result in better performance.

Another deviation in the Foncation approach in comparison to the related research discussed in the previous section is that Foncation does not require human-readable names or a fixed naming scheme upon agents. Neither does Foncation partition the name space in regions. The price to pay for this is a higher latency before a lookup or register operation has completed. That is caused by the number of messages that have to be sent over the network when there is no regional partitioning of the search space. However, Foncation gives a greater freedom to agent owners to name their agents as they see fit, without having to take the peculiarities of the used naming scheme into account.

5.2 Distributed naming services

There are a number of researchers who have investigated secure replacements for DNS. DNS is not designed to handle highly mobile objects such as software agents, but the subject is similar enough to be relevant to this section. Below, two of the secure DNS designs are discussed.

DDNS, by Cox *et al.* [11] is geared towards a new implementation of DNSSEC, forcing a hierarchical naming scheme. It uses DHash [13], the DHT built on top of Chord, to store and retrieve DNS record sets. DHash also provides load balancing and robustness. DDNS uses public key cryptography to sign and verify record sets. The measured median response time of DDNS on a simulated network with 1000 nodes is 350ms, which is in the same league as Foncation. Nevertheless, Cox *et al.* conclude that DDNS is a worse solution for serving DNS data than the current DNS. The main reason for this conclusion is that using DHTs implies that advanced DNS features such as dynamically generated records for local domains, have to be implemented on the client side.

Hu *et al.* have designed NLS [21], a scalable naming and location service inspired on Globe [44]. It can resolve textual names (or *URIs*) to the nearest cached or replicated objects that provide the requested content. It has a two-layer distributed search tree architecture. To improve scalability, NLS aggregates location information for names with a common prefix (although there is no fixed naming scheme) and stores hash codes of names in the interior nodes of the search tree. Using these hash codes, the system can always find a nearby replica in a lookup operation.

Comparison

Both DDNS and NLS are not designed specifically with mobile agents in mind. In contrast to Foncation, DDNS is designed to be a secure DHT-based DNS replacement. Obviously, this means that it adheres to the same fixed naming scheme that DNS uses, making it unsuitable to use for agent lookup with a flat naming scheme. DDNS fails as a proper DNS replacement, but the drawbacks that make it unsuitable for that purpose are hardly relevant to a mobile agent setting. The notion of a “local domain” in a distributed system with migrating agents, is not as unequivocally clear as in traditional settings where DNS is used. In security and scalability aspects, DDNS has characteristics that are remarkably similar to those of Foncation.

NLS assumes that the objects to be located can be replicated. This is uncommon in the case of mobile agent systems. Agents are unique entities that cannot be replicated at will, unless special precautions are made to keep the internal states of the replicas in sync. The AgentScape platform, introduced in Chapter 1, does not support replication of agents at the time of this writing.

In contrast to the ring organization found in structured P2P networks, NLS is organized as a search tree. It scales very well. Security aspects are not covered in the NLS design, but the naming scheme, like Foncation, is open.

5.3 Alternative trust schemes

This section describes alternative trust schemes that can be used for authentication in a peer-to-peer network. These are relevant to describe here because a classic public key infrastructure, as used in Foncation, is not the only possible way of verifying authenticity. In fact, there are other models that may be more suitable for use in a peer-to-peer environment.

Ooi *et al.* [30] give an overview of existing *peer-to-peer* reputation-based trust systems and propose one themselves. In a reputation-based system, the trustworthiness of peers is kept track of using a reputation metric. The reputation information is available to every participant, and reputation can be improved (or worsened) by peers who record their experience in dealing with another peer. The RCertPX reputation scheme proposed by Ooi *et al.* combines efficient retrieval of reputation information with protection of information integrity. Their scheme presumes the availability of an outside PKI to address issues like authentication and non-repudiation.

Aberer *et al.* [3] describe a statistical, or *quorum-based* decentralized PKI that can be used in combination with structured P2P systems. This approach is an example of a reputation-based trust system. The statistical approach entails obtaining public key information from many peers to form a quorum. In such a decentralized quorum-based PKI, when a certain

number of random and presumably independent peers replicate the same public key information, it is considered to be authentic. There can be various levels that determine when the trust quorum is reached, to deal with different situations. For instance in the startup phase when the number of participants is low, the rules are different than in a fully populated P2P network.

Advantages of the statistical approach to the web-of-trust based approach include:

- A public key or its fingerprint does not have to be communicated over a separate, secure channel first, to be trusted.
- Web-of-trust models cannot guarantee that the identity of strange peers can be established since there might not always be a transitive path that leads to a certification of the strange peer's public key. The statistical model does not require a transitive path but rather builds on the collective knowledge available in the system.
- It does not depend on the benevolence of each peer in the transitive trust chain.

A disadvantage of the statistical approach is that in a P2P community with a significant population of collaborating malicious peers, impersonation of one peer by another cannot be ruled out.

The quorum-based PKI is built to enable P2P e-commerce, for example an online marketplace like eBay. There are, however, no reasons why a decentralized quorum-based PKI such as the one described above could not be used for establishing authenticity in P2P systems built for other purposes, including a secure agent location service.

Popescu *et al.* [34] propose Turtle, a system for securely sharing sensitive data among friends, which does not assume transitive trust relationships. Turtle heavily relies on a-priori human friendship relations. Each node has an owner that establishes a cryptographically secure connection with the nodes of his friends in real life. Data items may be stored in the resulting network with a number of attributes of the form *attribute=value*. Users can query the network for information using (a logical combination of) these attributes. Exchanging data items and forwarding user queries is performed with "hops" from one friend to another. Eventually when a data item is found, it moves back to the requester following the same series of hops in reverse order, which is a slow, but safe, process.

Comparison

The web-of-trust PKI used in Foncation is a minimal implementation created with the building blocks provided by the Fonkey architecture (described in Chapter 2). As such, it suffers from the disadvantages of a web-of-trust listed by Aberer *et al.* [3] referred to earlier in this section. For example, to set up the experiments described in Chapter 4, first there had to be a system in place so that participating nodes were guaranteed to find a signature from a trusted source. To this end, the choice was made to distribute a universally trusted "root" key among the participating nodes. In a real-world situation however, such a "root" key might not be available. In that case, a Certification Authority would be needed. It appears that a secure, decentralized agent location service such as Foncation could benefit from using a quorum-based PKI since it would no longer depend on a web-of-trust.

Other reputation-based trust schemes are less likely to provide benefit since they assume the availability of an outside PKI. This requirement re-introduces all disadvantages related to a PKI that an alternative to the web-of-trust model should negate!

Turtle overcomes a weakness in the web-of-trust by not assuming transitive trust relationships. However, because Foncation should be able to connect agent locations whose administrators do not necessarily have to know each other, and because its retrieval process is slow, the Turtle approach to trust is not likely to provide benefit to Foncation.

5.4 Alternative search strategies

A number of researchers have studied the problem of how to efficiently perform queries in a distributed hash table. This section describes two approaches that are different from the one currently employed by the Bunshin DHT used by Foncation.

The Prefix Hash Tree (PHT) as proposed by Ramabhadran *et al.* [35] is one of these different approaches. The PHT is a trie¹-like distributed data structure that can be implemented on top of any DHT as long as it supports the common `put` and `get` operations. The PHT allows for one-dimensional range queries on the data set. The basic idea is that a trie is built over the data set to be indexed, and all leaf nodes of the trie get a certain prefix. The prefix labels of the trie nodes are hashed over the DHT identifier space. A binary search-based algorithm makes it possible to look up values within a specified range with a performance of $\log D$, where D indicates the size of the domain.

The *mSearch* system proposed by Gulati and Ranjan [20] uses Pastry and SCRIBE to build multicast trees on top of a P2P network. *mSearch* uses a hybrid indexing system: it keeps a reverse index locally for the keyword-to-document mapping and subscribes to multicast groups related to each of these keywords for the keyword-to-node mapping. The root of the multicast tree responsible for one keyword is the peer with the NodeID numerically closest to the Pastry hash for that keyword. It offers a load-balancing feature to prevent that the root of the multicast tree gets overloaded by requests. *mSearch* is also capable of extracting keywords from documents by itself, and it can perform conjunctive / disjunctive searches. The number of messages needed to resolve a query is an order-of-magnitude lower than the size of the network.

Comparison

The search-by-keyword functionality in Bunshin is implemented by keeping a reverse index of keywords coupled to objects, in the DHT itself. This could lead to a performance hit when there are many keywords to keep a reverse index of, or when it has to be updated frequently. After all, for each keyword, a hash table object is kept in the DHT. Resolving a keyword-value mapping is not any faster than locating a value directly with its Pastry hash. The Prefix Hash Tree is probably not the right solution to improve the performance of Foncation lookups. A range query is useful for queries like “give me all values in the range from Elton to Eric” but cannot improve the response time for the query “give me all Fonkey Named Packages with the keyword Elvis” since it specifies no searchable range.

mSearch, on the other hand, cleverly integrates the CAST and DHT abstraction to create a low-cost and scalable search infrastructure for structured P2P networks. It performs better than the traditional search methods such as the one currently used in Bunshin, and therefore Foncation could benefit from the *mSearch* approach if it were used in Foncation’s underlying

¹A **trie** (from **retrieval**) is a data structure similar to a tree. It is often used for efficient text lookups.

DHT. It has not been used in the Foncation prototype, as there was no implementation of *mSearch* available at the time the research was conducted.

I just about managed to forget you
when you appear in a dream
and you're even more beautiful there than I
remember you being

If the world ends
Guillemots

Chapter 6

Conclusions and Future work

In this thesis, a design and prototype of a scalable and secure agent location service is described, using the Bunshin distributed hash table and the Fonkey key distribution infrastructure. This chapter first presents a brief overview of this thesis, secondly the conclusions that can be drawn, followed by possibilities for future research.

6.1 Summary

This thesis focuses on Foncation: a secure, decentralized location service for agent platforms, in particular platforms that support agent mobility. Location services provide lookup, register and deregister operations for identifier–location pairs. A new approach, combining public key-based authentication and storage of data in a distributed hash table, brings security and scalability to agent platform location services.

Chapter 1 gives an introduction to distributed systems, software agents and agent platforms. The second chapter explains how a public key distribution infrastructure called Fonkey makes it possible to distribute public keys along with related information. The Fonkey infrastructure can also be used to store and cryptographically sign agent name and location data, enabling a location service model that performs secure authentication to establish if the data can be trusted.

The Bunshin distributed hash table, described in Chapter 3, is based on a structured peer-to-peer key based routing layer, enabling decentralized storage of Fonkey data. Bunshin is scalable, fault-tolerant and it supports search by keyword. Keyword search is an essential feature, because, as Foncation does not operate on the DHT directly, but indirectly through Fonkey, it cannot use agent names as keys in retrieval requests. Therefore, a way to execute searches on agent names and key signatures is required. Bunshin was the only DHT available at the time of this research that fulfills this requirement.

The fourth chapter describes the experiments performed with Foncation. These experiments have shown that Foncation is considerably slower than a simple, centralized location service without authentication when running on one or a low number of nodes. However, with a growing number of nodes, the number of concurrent requests that can be handled grows, too. The response times under a high load are comparable to those of similar DHT-based systems such as DDNS. Preliminary experiments suggest that the Foncation location service performs better under simulated real-world circumstances where lookup and insert opera-

tions are mixed and where not all participating nodes are actively performing operations at the same time.

The most important performance bottlenecks in Foncation are the DHT lookup latency, and the recursive key verification algorithm which is required for the *web-of-trust* trust model. Solutions to these bottlenecks should be searched in the direction of alternative trust models and alternative DHT search strategies that require less messages over the network. These alternatives are discussed in Chapter 5.

6.2 Conclusions

Returning to the research questions formulated in Section 1.5, it can be concluded that building a scalable, secure location service that deals with highly dynamic information is possible, under the condition that updates to a specific object–location mapping do not occur with a frequency higher than the frequency determined by the maximum speed at which cryptographic calculations are performed. In the experimental setup described in Chapter 4, this maximum frequency was about three updates per second per host.

Distributed hash tables are suitable for the purpose of providing decentralized storage for an agent location service. A mechanism has to be present to perform complex queries on the stored data, as simple *put* and *get* operations do not suffice.

Fonkey can be used in the design of the required infrastructure for a secure and scalable location service to the extent that the building blocks it provides are sufficient to verify authenticity of information published in such a location service. It has been used to design a simple *web-of-trust* trust model in Foncation, but it might well be used in a different trust model that would fit the decentralized nature of Foncation better, such as the statistical model.

6.3 Future work

During the implementation of the Foncation prototype, inevitably issues have come up that are subject to further research. Some of them were already mentioned in Chapter 5.

As the *web-of-trust* model has limitations such as the dependency on the benevolence of all entities in the trust chain, the trust model for authentication needs further research. In particular, it is worth developing a statistical trust model for Foncation. A statistical trust model would allow an arbitrary agent platform to join the location service and earn trust from the other participants, instead of having to have many other participants to sign its public key.

Another avenue to explore is the question whether information about agent location can be used to optimize the choice of nodes in the network where data is stored. Currently, the hash function that selects the node to store key-value pairs on, is unaware of such information. One technique to investigate in this context, is the Semantic Overlay Network [43] that clusters DHT nodes based on semantic information.

6.4 Open issues

An issue that has not been explored is the security of the communication between nodes in the DHT itself. Currently, there are no provisions to prevent new nodes from joining the

Foncation Pastry ring that do not participate in the location service. This is not a problem in itself, since the data in Foncation is signed and any forged information will be detected immediately. However, it is not known what will happen when malevolent nodes in the Pastry ring purposely try to disturb the working of Foncation, for instance by publishing faked agent locations, or when they do not respond to requests.

Lastly, the Foncation API is subject to improvement. Operations could be defined to support a directory service besides a location service. With a directory service it is possible for agents to advertise its services and capabilities to the world. Agent platforms participating in the Foncation location service could take advantage of this to find agents that fulfill specific tasks.

Bibliography

- [1] The Bunshin website [online]. Available from: <http://planet.urv.es/bunshin/>.
- [2] The Overnet website [online]. Available from: <http://www.overnet.com/>.
- [3] K. Aberer, A. Datta, and M. Hauswirth. A decentralized public key infrastructure for customer-to-customer e-commerce. *International Journal of Business Process Integration and Management (Inderscience Publishers)*, 2005.
- [4] G. Ateniese and S. Mangard. A New Approach to DNS Security (DNSSEC). In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 86–95. ACM, Nov. 2001.
- [5] D. Atkins and R. Austein. Threat analysis of the Domain Name System [online]. Available from: <http://www.ietf.org/rfc/rfc3833.txt>.
- [6] G. Ballintijn. *Locating Objects in a Wide-area System*. PhD thesis, Vrije Universiteit Amsterdam, Oct. 2003.
- [7] J. Bloch. The Java Tutorial [online]. Available from: <http://java.sun.com/docs/books/tutorial/collections/>.
- [8] F. M. T. Brazier, B. J. Overeinder, M. v. Steen, and N. J. E. Wijngaards. Agent factory: Generative migration of mobile agents in heterogeneous environments. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC 2002)*, pages 101–106, Madrid, Spain, 2002. Available from: http://www.iids.org/publications/aims_sac2002.pdf.
- [9] M. Castro, M. Costa, and A. Rowstron. Peer-to-peer overlays: structured, unstructured, or both? Technical Report MSR-TR-2004-73, Microsoft Research, Cambridge, UK, 2004.
- [10] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 20(8):1489–1499, 2002. Available from: citeseer.ist.psu.edu/castro02scribe.html.
- [11] R. Cox, A. Muthitacharoen, and R. Morris. Serving DNS using a peer-to-peer lookup service. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 155–165, London, UK, 2002. Springer-Verlag.

- [12] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, 2001.
- [13] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, California, March 2004.
- [14] F. Dabek, B. Y. Zhao, P. Druschel, and I. Stoica. Towards a common API for structured peer-to-peer overlays. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003.
- [15] A. Di Stefano and C. Santoro. Locating mobile agents in a wide distributed environment. *IEEE Trans. Parallel Distrib. Syst.*, 13(8):844–864, 2002.
- [16] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.
- [17] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1) [online]. Available from: <http://www.ietf.org/rfc/rfc3174.txt>.
- [18] L. Garcés-Erice, P. Felber, E. W. Biersack, G. Urvoy-Keller, and K. W. Ross. Data indexing in peer-to-peer dht networks. In *ICDCS*, pages 200–208. IEEE Computer Society, 2004.
- [19] M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge. The belief-desire-intention model of agency. In J. Müller, M. P. Singh, and A. S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 1–10. Springer-Verlag: Heidelberg, Germany, 1999. Available from: citeseer.ist.psu.edu/georgeff99beliefdesireintention.html.
- [20] A. Gulati and S. Ranjan. Efficient Keyword Search using Multicast Trees in Structured P2P Networks. Under preparation, 2005.
- [21] Y. Hu, D. Rodney, and P. Druschel. Design and scalability of NLS, a scalable naming and location service. In *Proceedings of IEEE INFOCOM 2002*, New York, June 2002. IEEE.
- [22] G. Kastidou, E. Pitoura, and G. Samaras. A scalable hash-based mobile agent location mechanism. In *ICDCSW '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 472, Washington, DC, USA, 2003. IEEE Computer Society.
- [23] J. Kubiawicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: an architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pages 190–201. ACM Press, November 2000.
- [24] J. Liang, R. Kumar, and K. Ross. Understanding Kazaa. 2004. Available from: citeseer.ist.psu.edu/liang04understanding.html.

- [25] B. T. Loo, R. Huebsch, I. Stoica, and J. M. Hellerstein. The Case for a Hybrid P2P Search Infrastructure. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 141–150. UC Berkeley, Springer Verlag, 2004.
- [26] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. *IEEE Communications Surveys and Tutorials*, 7(2), 2005.
- [27] D. S. Milojicic, W. LaForge, and D. Chauhan. Mobile objects and agents (MOA). In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 179–194. USENIX, Apr. 1998.
- [28] P. Mockapetris. Domain names - Concepts and facilities, Nov. 1987. RFC 1034. Available from: <http://www.ietf.org/rfc/rfc1034.txt>.
- [29] R. Mondéjar, P. García, and C. Pairet. Bunshin: DHT para aplicaciones distribuidas. *Jornadas de Concurrencia y Sistemas Distribuidos (JCSD)*, XIII, Sept. 2005. I Congreso Español de Informática (CEDI 2005). Granada, Spain.
- [30] B. C. Ooi, C. Y. Liau, and K.-L. Tan. Managing trust in peer-to-peer systems using reputation-based techniques. In G. Dong, C. Tang, and W. Wang, editors, *WAIM*, volume 2762 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 2003.
- [31] B. J. Overeinder, E. Rozendaal, and F. M. T. Brazier. Secure and Decentralized Location Service for Agent Platforms. (Unpublished), 2005. Available from: <http://www.iids.org/>.
- [32] A. Pappas, S. Hailes, and R. Giaffreda. Mobile host location tracking through DNS. In *London Communications Symposium 2002*, 2002.
- [33] R. Perlman. An overview of PKI trust models. *IEEE Network*, 13(6):38–43, Nov. 1999.
- [34] B. C. Popescu, B. Crispo, and A. S. Tanenbaum. Safe and Private Data Sharing with Turtle: Friends Team-Up and Beat the System. In *Proc. 12th Cambridge International Workshop on Security Protocols*. Springer-Verlag, April 2004.
- [35] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Brief announcement: prefix hash tree. In *PODC '04: proceedings of the twenty-third annual ACM symposium on principles of distributed computing*, pages 368–368, New York, NY, USA, 2004. ACM Press.
- [36] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *Proceedings of the Conference on File and Storage Technologies*. USENIX, 2003. Available from: <http://www.usenix.org/publications/library/proceedings/fast03/tech/rhea.html>.
- [37] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, June 2004. Available from: <http://www.bamboo-dht.org/>.

- [38] V. Roth. Scalable and secure global name services for mobile agents. In *6th ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages*, Cannes, France, June 2000. Available from: citeseer.ist.psu.edu/article/roth00scalable.html.
- [39] V. Roth and J. Peters. A Scalable and Secure Global Tracking Service for Mobile Agents. In *Proc. Mobile Agents 2001*, Lecture Notes in Computer Science. 5th IEEE International Conference Mobile Agents (MA), Springer Verlag, December 2001.
- [40] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218(0):329–350, 2001. Available from: citeseer.ist.psu.edu/rowstron01pastry.html.
- [41] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 188–201, New York, NY, USA, 2001. ACM Press.
- [42] E. Rozendaal. Fonkey Overview - DRAFT. Technical Report TR-2003-01, NLnet Labs, 2003.
- [43] R. Siebes. pNear - combining Content Clustering and Distributed Hash Tables. In *Proceedings of the IEEE'05 Workshop on Peer-to-Peer Knowledge Management*, San Diego, CA, USA, 17 July 2005.
- [44] M. v. Steen, F. J. Hauck, P. Homburg, and A. S. Tanenbaum. Locating Objects in Wide-Area Systems. *IEEE Communications Magazine*, pages 104–109, Jan. 1998.
- [45] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001. Available from: citeseer.ist.psu.edu/stoica01chord.html.
- [46] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
- [47] The Gnutella development forum. The Gnutella Protocol Specification v0.4, 2001. Available from: http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf.
- [48] C. Tolman. A fault-tolerant home-based naming service for mobile agents. Master's thesis, Rensselaer Polytechnic Institute, Troy, New York, Apr. 2003.
- [49] M. Wahl, T. Howes, and S. Kille. Lightweight directory access protocol (v3). RFC 2251, IETF, Dec. 1997.
- [50] N. J. E. Wijngaards, B. J. Overeinder, M. v. Steen, and F. M. T. Brazier. Supporting internet-scale multi-agent systems. *Data and Knowledge Engineering*, 41(2-3):229–245, June 2002. Available from: <http://www.iids.org/publications/dke-2002.pdf>.

- [51] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, June 1995.
- [52] T. Wright. Naming services in multi-agent systems: A design for agent-based white pages. In *Proceedings of the Third International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1478–1479. IEEE Computer Society, 2004.
- [53] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001. Available from: citeseer.ist.psu.edu/zhao01tapestry.html.