

# Resource and Service Discovery for Mobile Agent Platforms

Ziyan Maraikar

August 9, 2006



# Contents

<b>1</b>	<b>Context and Objectives</b>	<b>1</b>
1.1	Directories . . . . .	1
1.1.1	Naming Services . . . . .	1
1.1.2	Attribute-based Directories . . . . .	2
1.2	Agents . . . . .	2
1.2.1	Agent Platforms . . . . .	2
1.3	Objectives . . . . .	3
1.4	Thesis Outline . . . . .	3
1.4.1	Terminology . . . . .	4
<b>2</b>	<b>Resource and Service Discovery</b>	<b>5</b>
2.1	Generic Discovery . . . . .	5
2.1.1	Hardware and Computational Resources . . . . .	6
2.1.2	Agents and Software Services . . . . .	6
2.1.3	Descriptive Information . . . . .	6
2.2	Query Requirements . . . . .	6
2.2.1	Syntactic Query . . . . .	7
2.2.2	Semantic Query . . . . .	7
2.3	Choice of Data Model . . . . .	7
2.3.1	Describing Ontologies . . . . .	7
2.4	Sample RDF Directory Schema and Use-case . . . . .	8
<b>3</b>	<b>Decentralised Directory Design</b>	<b>11</b>
3.1	Why a Decentralised Directory Service? . . . . .	11
3.1.1	Location Autonomy . . . . .	11
3.1.2	Scalability . . . . .	11
3.1.3	Reliability . . . . .	12
3.1.4	Data Freshness . . . . .	12
3.2	Strategies for Decentralisation . . . . .	12
3.2.1	State Replication . . . . .	12
3.2.2	Query Routing . . . . .	12
3.3	Choice of Architecture . . . . .	13
<b>4</b>	<b>Security</b>	<b>15</b>
4.1	Authorisation . . . . .	15
4.2	Authentication . . . . .	15
4.3	Integrity . . . . .	16

<b>5</b>	<b>Implementation</b>	<b>17</b>
5.1	DS Interface . . . . .	17
5.2	Routing Overlay . . . . .	19
5.2.1	Evaluation of Gnutella-based Routers . . . . .	19
5.2.2	A Generic Flood Routing Protocol . . . . .	19
5.3	Managing Directory Entry Metadata . . . . .	20
<b>6</b>	<b>Evaluation</b>	<b>23</b>
6.1	System Simulation . . . . .	23
6.2	Node Connectivity . . . . .	24
6.3	Number Nodes . . . . .	24
6.4	Effects of Query Rate . . . . .	25
6.5	Duplicate Packets . . . . .	25
6.6	Small-world Networks . . . . .	25
6.7	Discussion . . . . .	27
<b>7</b>	<b>Conclusion</b>	<b>29</b>
7.1	Related Work . . . . .	29
7.2	Future Work . . . . .	29
7.3	Concluding Remarks . . . . .	30
<b>A</b>	<b>Partial RDF Schema for Compute Resources</b>	<b>31</b>

# Chapter 1

## Context and Objectives

Resource discovery is fundamental to any distributed system. This thesis focuses on the problem of resource discovery in the context of mobile agent systems. The term “resource discovery” in agent literature is usually restricted to the problem of locating agents with a certain capability. Yet, for reasons laid out in the sequel we take the broader view, defining resource discovery to encompass the gamut of activities from finding compute cycles or storage capacity to locating a desired web service or agent.

Our thesis is threefold. Firstly, that the highly dynamic nature of mobile agent systems lends itself to peer to peer style discovery of resources. Secondly, to better support intelligent agents must be able to not just query but reason about services and resources available to them. Thirdly, in an open environment mechanisms must exist to facilitate a level of trust between agents in the directory service infrastructure.

The concepts of directory services and agent platforms which are pertinent to this thesis are introduced first. We then go on to clarify our objectives in the context of directory services and agents.

### 1.1 Directories

Resources in a distributed system may either be discovered by name or by their characteristic attributes. These two methods are analogous to looking up the phone number of a business by name in the “White Pages” or by a category index in the “Yellow Pages”. In fact in distributed systems jargon naming and attribute-based lookup services are known as white and yellow pages services, respectively.

The white pages naming service typically provides human-readable name to physical address mapping (or vice versa) while the yellow pages directory service supports storing and querying entities with multiple attributes. In essence directories are actually special-purpose (possibly distributed) databases, optimised for reads.

#### 1.1.1 Naming Services

The domain name system[21] (DNS) is perhaps the best known example of naming service. It handles the task of mapping textual names like `ftp.foo.nl` to IP addresses. DNS uses a hierarchical namespace divided among administrative *domains*, like `foo.nl`. DNS software like BIND actually performs two distinct tasks. On the one hand acts as an authoritative database for names within the domain. On the other it performs *recursive name resolution* on behalf of clients. The recursive resolution for `ftp.foo.nl` from within the VU for example would result in the VU DNS server contacting one of the well-known DNS *root servers* to find the name

server IP address of the `nl` top-level domain and then querying that server for the `foo.nl` name server. Finally, the VU nameserver would contact the `foo.nl` and retrieve the IP address of `ftp.foo.nl` and return it to the client.

The Globe[27] mobile object system uses a two level mapping from name to physical address. An object naming service maps a name to a unique, location independent *handle*, while the mapping from handle to physical network address is done by a separate *location service*.

### 1.1.2 Attribute-based Directories

Traditionally yellow pages (YP) directories have occupied a niche in systems management applications, such as Microsoft Active Directory and Sun's Network Information Service. More recently it has become an integral part of the web services and Grid infrastructure.

LDAP is the de-facto protocol for querying and modifying YP directories over TCP/IP. It implements a subset of the functionality present in the OSI X.500 directory standard. Information is structured as a tree of entities. Each entity consists of attributes that are defined by one or more *schemata* — templates that define the attributes a particular class of objects may have. One of LDAP's drawbacks is that it encodes attribute information in the complex ASN.1 notation (a binary precursor to XML) to maintain compatibility with the X.500 standard.

UDDI (Universal Description, Discovery, and Integration) is an XML based registry for web services. UDDI stores provides both name and attribute based lookup. Attributes are defined in taxonomic models (tModels) which are similar to LDAP schemata. In addition to white and yellow pages UDDI also provide so-called Green Pages containing information regarding protocol specific bindings to web services in WSDL (web service description language). UDDI directories are centralised although the UDDI version 3 standard proposes hierarchically federated registries (how query forwarding works is unclear).

Grid computing frameworks use YP directories for both resource discovery and monitoring. A core component of the popular Globus toolkit is Globus-MDS (previously the metacomputing directory service, now referred to as the Monitoring and Discovery Service). The older MDS-2 version was essentially a highly available replicated LDAP service. The latest MDS-4 incarnation stores information as XML and is queriable via XPath expressions.

## 1.2 Agents

Software agents can be defined to be autonomous, computational entities capable of operating in a dynamic environment. Agents typically interact with other agents, sometimes collaborating to achieve common goals. In an abstract sense agents are simply a design metaphor for constructing complex systems around autonomous, communicating entities. One may give a more concrete definition of an *intelligent agent* as aggregating the previously distinct AI concepts of planning, learning and coordination.

From a computer systems perspective, agents may be simply viewed as processes having code, data and state. In this respect they resemble distributed object systems. But unlike distributed object which are reactive, agents are *proactive*. Of particular interest to systems practitioners is the concept of *mobile agents*. Depending on whether agent state is preserved during migration, mobility may be classified as either *strong* or *weak*.

### 1.2.1 Agent Platforms

Agents, as alluded to in the above definition, typically exist as part of a distributed ensemble, known as *multiagent systems*. An *agent platform* (AP) or *environment* is a middleware layer that provides common services (and possibly a runtime environment) for easing the construction of

multiagent systems. The Foundation for Intelligent Physical Agents (FIPA) is a standards body working towards agent interoperability. FIPA specifications address issues from a standardised agent communication language (FIPA ACL) to agent management services[8].

JADE[4] (Java Agent Development Environment) was one of the first FIPA-compliant platform developed. JADE offers an agent runtime system and a predefined programmable agent model and of a set of management and testing tools.

AgentScape is an agent platform structured as a loose federation of autonomous *locations*—each running an instance of the AgentScape middleware. Locations manage resources compute clusters and host services offered by agents. Agents from other locations may be granted access to these services and resources, if permitted by local policy. AgentScape also offers weak migration of agents between locations.

### 1.3 Objectives

Agent environments utilise both white and yellow pages directories, although the terminology used differs. The former is referred to as an agent naming or location service while the latter is called a directory facilitator (DF) in FIPA terminology.

Our goal is to identify deficiencies in current DF implementations and present a design that rectifies them. Specifically, the agent directory service we design has the following aims:

**Discovery** The basic requirement is a generic directory for resource and service discovery using attribute-based query. To better support intelligent agents we also aim to provide semantic query capabilities beyond the simple attribute matching present in traditional YP implementations.

**Decentralisation** Support for query across multiple instances agent platform distributed across the Internet. We opt for an overlay-based broadcast that should scale moderately (circa 1000 nodes).

**Security** Since we operate in an open environment there is limited trust between two agents, and between agents and locations. Therefore mechanisms to restrict access to, and verify the integrity and authenticity of, published information are needed.

In proceeding chapters we discuss the rationale for these goals and arrive at a directory service design suited to mobile agent platforms. We implement and evaluate a subset of the design in the context of AgentScape.

### 1.4 Thesis Outline

The rest of this report is organised as follows. Chapter two discusses the types of queries our system should be able to field and a data model that supports these queries. Chapter three presents the rationale for a decentralised architecture and a design that overcomes the limitations of centralised systems. The security requirements and a general outline of how they may be designed into the system is presented in chapter four. The details of our implementation are discussed in chapter five. An evaluation of the scalability of the system based on various parameters is presented in chapter six. Chapter seven concludes with a review of related literature and a critical appraisal of our work.

### 1.4.1 Terminology

We use the terms directory service (DS) to refer to a conventional yellow pages directory while the term directory facilitator (DF) refers to the FIPA's notion of a YP service. We favour the former over the term directory facilitator because the design goes beyond typical DF functionality. We borrow the term *location* from AgentScape nomenclature to refer to an instance of the agent environment.

## Chapter 2

# Resource and Service Discovery

Having reviewed agent platforms and directories this chapter ascertains what is required of an agent directory service or directory facilitator. Yellow pages directories in current agent platforms with reference to the relevant FIPA specifications are investigated to crystallise query requirements of the DS. We then arrive at data model that is able to support these query requirements.

The directory facilitator as described by FIPA in their abstract architecture document [8] is primarily targeted towards agent discovery.

Agents may register their services with the DF or query the DF to find out what services are offered by which agents. At least one DF must be resident on each agent platform (the default DF). However an AP may support any number of DFs.

The FIPA specification makes no mention of resource discovery. This is possibly because in most agent environments agents do not *need* direct access to specific hardware resources, as they are purely concerned with agent interactions. In fact resource management in system like JADE is relegated to the underlying platform<sup>1</sup>, hidden from the view of agents. Yet when it comes to agent mobility this approach is limiting, as the ability to search for specific hardware and software resources is crucial. An agent wanting to migrate to another location must first be able find a host with a compatible architecture, operating system and associated libraries, adequate network bandwidth etc. It is clear the service versus resource discovery dichotomy is both artificial and unnecessary in the context of mobile agent systems.

### 2.1 Generic Discovery

Following this line of argumentation leads to directory service that is sufficiently generic to handle both service and resource discovery within a unified framework. Agents should be able to query the service for:

**Hardware and computational resources** based on hardware architecture, memory, disc and bandwidth requirements.

**Agents and software services** such as web services and functionality offered by other agents running at other locations, possibly on different on FIPA-compliant platforms.

**Descriptive information** like security certificates, policies etc.

---

<sup>1</sup>e.g. Operating system or JVM

Note that this does not necessarily imply that a single *instance* of the DS will handle all these entities. It may well be that the differing policies on publishing resource versus service information means that each is handled by a separate instance of the DS.

### 2.1.1 Hardware and Computational Resources

Hardware resource discovery in an agent platform has much in common with resource discovery in computational Grids [9]. In Grids — just as in agent environments, resources like compute clusters, owned by organisations are aggregated and published in a DS such as the Globus MDS.

It should be emphasised that a DS is commonly meant to store relatively static data, and is therefore optimised for reads. Short-lived data e.g. monitoring information like current CPU load at a location, does not belong within a directory service. The DS should merely hold pointers to providers of such information e.g. a separate load monitoring service.

### 2.1.2 Agents and Software Services

The FIPA definition of a DF simply says that it should allow agents to register their services and query for services offered by other agents. At its most basic this no different than the role UDDI plays in web services. In fact in agent applications and platforms primarily targeted at web services UDDI plays the role of a DF.

However, in the context of intelligent agents, richer query facility that goes beyond attribute based lookup is desirable. Take for example, the case of registering a route-planner web service that covers all of The Netherlands. A search for a route planner service for Amsterdam using a keyword-based search engine is not guaranteed to return the above service. Registering this service in a UDDI registry is equally cumbersome, and would require explicitly listing out each and every city covered by the service.

If instead our data model could represent the notion that Amsterdam is a city in the Netherlands, then we may simply register the route-planner as a service for The Netherlands. An intelligent agent could then infer that this service can also fulfil a route planning request within Amsterdam. A richer data model lets us support applications like *service matchmaking* and automated runtime composition of web services[22].

### 2.1.3 Descriptive Information

A YP service can store miscellaneous information relating to agents, the canonical example being X.509 security certificates. Another example is policy information. In AgentScape for example a mediator service called the location manager[16] aggregates resources on individual hosts at a location into “agreement templates”. The DS will serve as a repository for publishing such information.

## 2.2 Query Requirements

The discovery scenarios above basically require support for two different types of query. At its most basic agents must be able to query for entities using arbitrary attributes they possess. However, as illustrated, attribute based query alone is limiting for intelligent agents. Overcoming this limitation requires an *ontology* — a representation of semantic relationships between the entries stored in the DS.

### 2.2.1 Syntactic Query

The most basic query any yellow pages service needs to provide is exact matching of attributes of the form `attr1=value1, attr2=value2, . . .`. The next level of sophisticating is the ability to perform range query, i.e. use comparison operators in conjunction with numerical attributes. Another type of query is pattern matching on string attributes.

### 2.2.2 Semantic Query

Our semantic query needs can be clarified using a contrived example. Suppose we have the facts “Agent X is running on host Y” and “Host Y runs operating system Z” we should be able to infer that Agent X is able to run on OS Z. Essentially we need the ability to store *statements* describing Semantic relationships of the form `<subject> <predicate> <object>`. While it is possible to do this by defining attributes for every possible relationship between two objects in a directory such an ad-hock solution is unwieldy. A standard ontology on the other hand would let us use available reasoner for making inferences.

## 2.3 Choice of Data Model

Following from these query types is a fundamental design choice: the data model DS will use to store information. Our choice of data model completely determines the types query that the DS is capable of answering. Given that a yellow pages directory is actually specialised database, at first glance a relational database seems a reasonable choice for a data store. Indeed SQL (structured query language) is able to support all form of syntactic query discussed above.

However, as noted earlier data in yellow pages directories do not conform to a fixed set of schemata. Unlike the fixed tables in the relational model directory data is *semistructured*. In LDAP for instance, one may define a new schema and let existing objects inherit its attributes. It is quite possible to transform semistructured data to store it in a relational database<sup>2</sup>. Nevertheless the relational model is wrong abstraction to directly deal with semistructured data.

The de-facto standard semistructured data is XML (eXtensible Markup Language). A single XML document may have attributes that are defined by any number of XML schemata (using XML namespaces). There are numerous data stores for XML as well as libraries for XML validation, parsing and transformation. Declarative query language for XML such as XPath or XQuery FLWOR use *path expressions* which can refer to arbitrary nodes in the XML document object model. XML satisfies all our needs for data storage and syntactic query, but something additional is needed to represent an ontology.

### 2.3.1 Describing Ontologies

Standard tools and techniques for representing ontologies have been already developed by the Semantic Web community. There is a range of languages from the Resource Description Framework (RDF) to the Web Ontology language variants — OWL-lite, OWL-S and OWL-Full, that support incrementally more powerful definition of relationships and constraints. This in turn supports inferencing of increasing complexity, but higher sophistication comes at the cost of higher processing overhead.

The RDF standard [13] supports expressing binary predicates much as in logic languages like Prolog. RDF defines a *data model* based on a directed graph. The graph’s nodes represent *resources*, i.e. subjects and objects. The directed arc from subject to object represents a predicate or *property* in RDF terminology. RDF does *not* define a separate format. Instead it

---

<sup>2</sup>Most XML and RDF datastores do in fact support relational database backends

is expressible in a variety of syntaxes including XML using the RDF/XML encoding. For a comprehensive tutorial-style introduction to RDF the reader is referred to [15].

RDF has an associated vocabulary description language—RDF Schema (RDFS). RDFS allows Resource and property types to be grouped into classes and supports (multiple) inheritance, much as in object oriented programming. Resources roughly correspond to objects while properties correspond to attributes. However there are several differences. Defining a class a resource belongs to, is purely optional. Also, unlike attributes in OOP, which belong to a specific class properties are *global* in the sense that they describe a relationship between any two classes (although this can be constrained). An example showing the use of RDF(S) is given in the next section.

## 2.4 Sample RDF Directory Schema and Use-case

Figure 2.1 which was automatically generated using the RDFS/XML schema listed in Appendix A illustrates a partial RDF directory schema for storing information about compute resources. Note however that RDF *does not* require a predefined schema. Schemata only serve to group resources into classes so relationships can be defined between groups resources of similar type. However, RDF resources are not limited to only the properties and resource types defined by a schema.

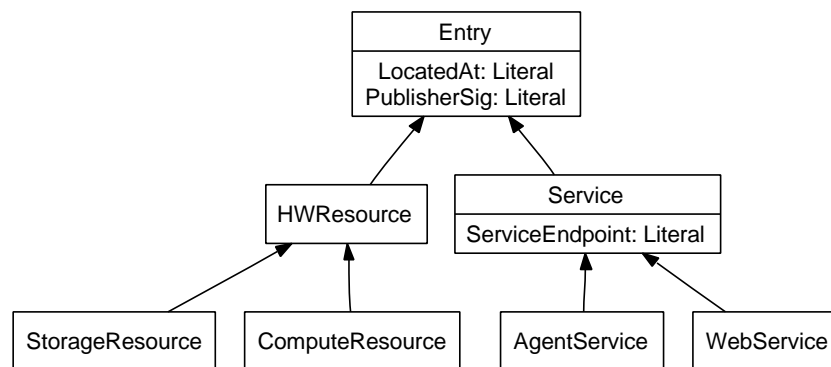


Figure 2.1: Example RDF directory schema.

The following RDF fragment uses this schema to store information about two compute resources. It defines Linux and Solaris as instances of POSIX operating systems and Opteron and Sun-Fire as instances of the “i386” and ‘sun4u” architectures respectively.

```

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns= "http://iids.org/agentscape#"
  xml:base="http://iids.org/agentscape/resources/compute">

  <as:POSIX rdf:ID="Linux"/>
  <as:POSIX rdf:ID="Solaris"/>

  <as:I386 rdf:ID="Opteron"/>
  <as:Sun4u rdf:ID="Sun-Fire"/>

  <as:ComputeResource rdf:ID="keg.cs.vu.nl">
    <as:runsOS rdf:resource="#Solaris">
  
```

```

    <as:hasArchitecture rdf:resource="&as;#Sun-Fire">
  </as:ComputeResource>

  <as:ComputeResource rdf:ID="rattler.cs.vu.nl">
    <as:runsOS rdf:resource="&as;#Linux">
      <as:hasArchitecture rdf:resource="&as;#Opteron">
    </as:ComputeResource>
  </rdf:RDF>

```

Such statements may be stored, queried using RDF databases like Jena and Sesame. For example, a query to find all i386-based machines in SPARQL looks like this:

```

PREFIX as: <http://iids.org/agentscape>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?name
WHERE {
  ?x rdf:ID ?name .
  ?x rdf:type as:ComputeResource .
  ?x as:hasArchitecture ?y .
  ?y rdf:type as:I386 .
}

```

This of course very similar to an SQL or LDAP query. What sets apart RDF databases is their support for making inferences. By enabling the inference capabilities of the RDF data store<sup>3</sup> we could, for example, request a list of all machines running a POSIX-compliant operating system as follows:

```

PREFIX as: <http://iids.org/agentscape>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?name
WHERE {
  ?x rdf:ID ?name .
  ?x rdf:type as:ComputeResource .
  ?x as:runsOS ?y .
  ?y rdf:type as:POSIX .
}

```

Note that this query is no different or more complex from the one above. Yet, by merely turning on inferencing we are able to query for an attribute (i.e. machines running POSIX-compliant OSs) not listed explicitly anywhere in the above machine descriptions.

---

<sup>3</sup>A platform-dependent operation



## Chapter 3

# Decentralised Directory Design

Having decided on a data model to support the the query need of intelligent agents we turn our attention to the issue of decentralisation. The next section gives the reasons against choosing a centralised system and goes on to list the characteristics desired of a decentralised solution. An overview of possible decentralisation mechanisms is given followed by the choices that our design makes.

### 3.1 Why a Decentralised Directory Service?

In our introduction we claimed that a key objective of this work is to explore a scalable and robust distributed DS design. But what exactly is the rationale to preclude a simple centralised DS for serving mobile agents?

The primary virtue of a centralised system is its simplicity in both design and administration. On the downside it introduces a single point of failure and a potential scalability bottleneck. It may be argued that scalability concerns are offset by the relentless advances in hardware. For systems of moderate size like our directory service this may well be the case.

What this line of reasoning ignores is that hardware advances do not help a system scale in terms geography and administrative trust. Adding hardware to a centralised service does not hid network latency when accessing the service from the other side of the world. More serious is the problem of centralised trust. To wit, if Google suddenly decided to unilaterally lower the page rank of a popular website there is very little the owner could do about it.

Based on the limitations of a centralised DS we identify three characteristics that a decentralised design should possess.

#### 3.1.1 Location Autonomy

In an open agent environment it is unrealistic to designate a centrally trusted authority for resource and service discovery. A centralised DS introduces a single point of trust because it must be considered *the* authoritative source for resources and services across all locations. In other words, whoever is responsible for the DS can arbitrarily deny registration or query facility to any agent or location. A centralised DS is thus in conflict with the idea of an *open* platform. Thus we require that each location be able to run a local directory service instance.

#### 3.1.2 Scalability

FIPA's specification quoted above mentions that an agent platform may host multiple directory services. Yet it makes no mention of the ability to perform federated (i.e. global) query across

all DF instances. We specifically require the ability to perform federated query in a reasonably scalable fashion in order to support agent migration.

### 3.1.3 Reliability

A frequent argument against distributed design is that it replaces a single point of failure with multiple points of failure. This is best exemplified by Leslie Lamport's quip that "a distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." All too often this is actually the case with poorly designed systems with inter-dependencies between components, having no redundancy to compensate for partial failure.

The failure of a DS at one location should not affect the proper functioning of DSs at other locations. A DS should have a degree fault-tolerance as it is central to the proper functioning of the location. We allow for the possibly a short recovery period, during which the service operates at a degraded level.

### 3.1.4 Data Freshness

Over time the DS could fill up with data that is out of date. Registrations by agents that have crashed for example, should be expunged. The DS needs maintain some indication of the freshness of data objects, and should purge "stale" data.

## 3.2 Strategies for Decentralisation

Next we turn to techniques that can be employed to achieve the above goals. There are basically two complementary techniques to achieve federated query. State replication can be used to have a copy of data at remote DSs giving a "global" view of data at all DSs available locally. Alternatively queries may route be routed to DSs at other locations that could potentially satisfy the request.

### 3.2.1 State Replication

Replication mechanisms can be classified in many different ways. Depending on how much state is replicated across nodes replicas may either be partial or full. Generally full replication is only feasible when replicas are located on a single LAN because of the high network overhead incurred in keeping them synchronised.

Replication can also be characterised as push versus pull depending on which party initiates the action. Push protocols work best when the set of replicas is fixed in advance. Caching may be classed a form of pull-based replication.

The majority of replication techniques (excluding caching) have been designed for LAN environments and assume a static set of replicas. Epidemic or gossip protocols are an interesting class of protocols for wide-area replication across a dynamically changing set of nodes. These work by replicas periodically selecting another at random and synchronising their state by exchanging updates. Probabilistic guarantees may be given that updates will disseminate to a certain fraction of nodes within a given period.

### 3.2.2 Query Routing

Query routing is a concept popularised by peer to peer overlays. The basic idea is to route packets at the application layer not based on a network address but based on their (application-

specific) *content*. All P2P overlays may be broadly classified as being either structured or unstructured.

**Structured overlays** such as Pastry[23] and Chord[25] are mostly based on a *distributed hash table* abstraction. Peers are assigned an integer identifier out of a large address space. To store an object, it is run through a hash function such as MD-5 or SHA-1 and stored at the peer(s) with the closet matching identifier. The routing algorithm ensures a hash-value query is routed to peer(s) where a potential matches exist.

The distributed hash table abstraction natively supports only exact match searches on hash value. These systems have been extended with facilities for regular expression matching and range queries to better support higher level query functionality. PIER[7] is one such system that sits atop Chord or CAN which provides database query engine. However it has the limitation of imposing a standard set of global schemata.

**Unstructured overlays** exemplified by Gnutella use broadcast to locate objects. The original Gnutella v0.4 protocol performs flooding constrained by a packet's maximum hop count. In the newer Gnutella v0.6, only well connected nodes called *super-peers* flood queries amongst themselves. So called leaf nodes simply forward any query to its parent super peer.

Although Gnutella was intended for filename searches it could be used to route any type of query. It has been extended by servants like LimeWire to support arbitrary attributes encoded as an XML document. LimeWire's proprietary PeerDB system uses this technique to connect various data stores like relational databases to Gnutella.

There are trade-offs in both these types of overlays. Structured overlays provide automatic load balancing and in some cases transparent replication while unstructured overlays provide unconstrained data placement and better locality.

### 3.3 Choice of Architecture

Our DS's overall structure is shown in Fig. 3.1. Each location runs its own DS which uses an RDF data store such as Jena or Sesame. A DS will attempt to satisfy queries by first looking them up in its local data store. To achieve both location autonomy and scalability we chose a to link individual directories via a simple broadcast overlay for query routing with caching as a possible optimisation. In the current prototype implementation we use a naive flood routing algorithm. However, the architecture could just as well use a more efficient broadcast such as Structella or even random walk.

Gossip based replication is another viable option but most queries for generic resources and services can be satisfied by a location's local DS. Wide-area query would be the exception, usually triggered by an agent requesting a specialised resource not available locally. Thus spatial locality of related data is important for efficient functioning of the DS. In [11] the authors of TerraDir argue that although structured overlays offer provable latency bounds and load-balancing, it sacrifices spatial locality particularly in the context of attribute-based search. This argument further strengthens our choice of an unstructured overlay.

For reliability our DS uses the very same master-slave mechanism as DNS. The master DS periodically checkpoints its RDF database state to disc in RDF/XML. Master and slave DSs belonging to a location periodically synchronise their state. If a synchronisation, which is initiated by the slave fails, it takes over query answer and overlay routing functions. When the master resumes operation it must explicitly inform the slave to return to passive synchronisation mode.

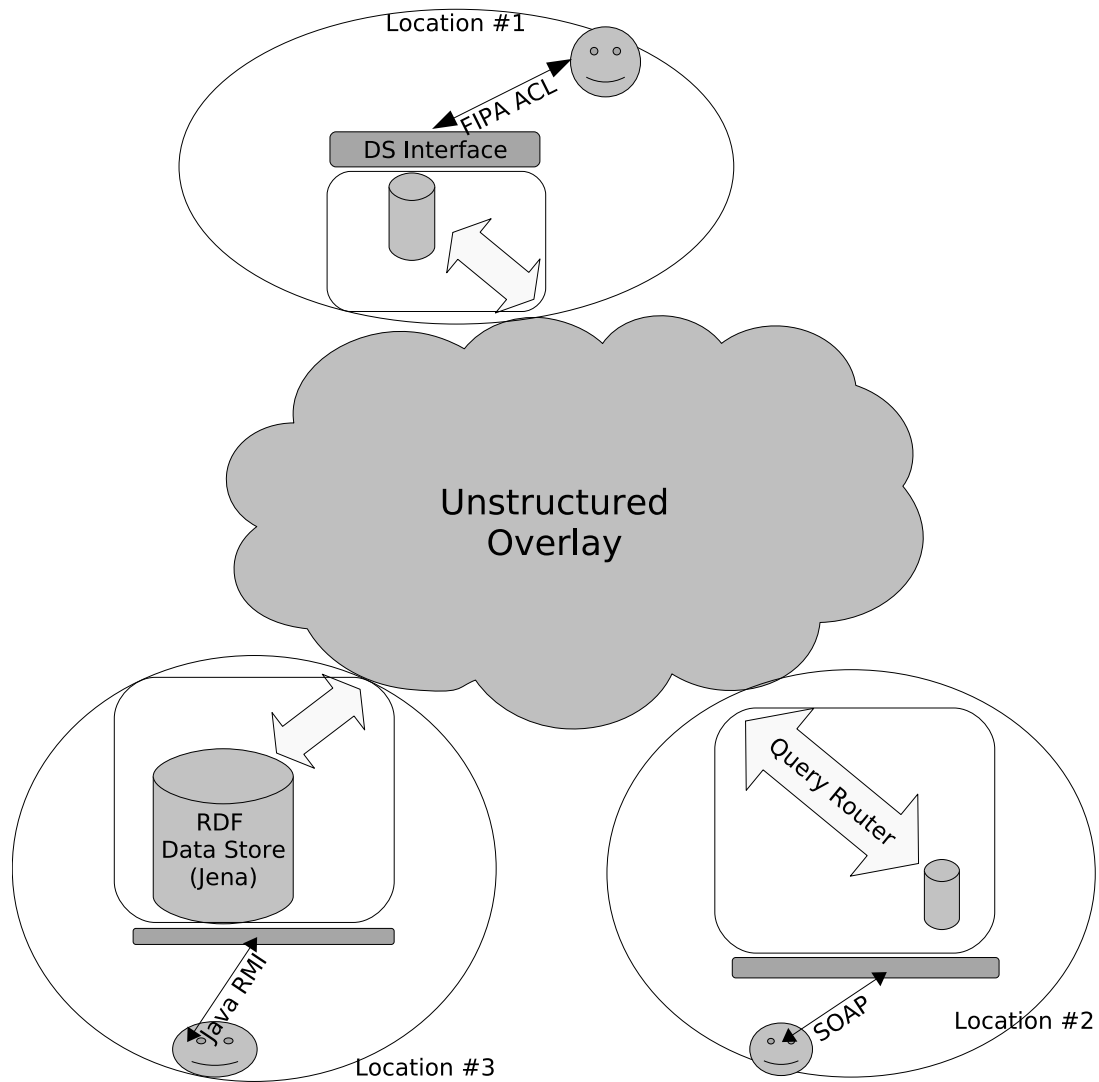


Figure 3.1: Directory service architecture.

To automatically clean up stale objects we use *leasing*, a concept used pervasively by the Jini architecture[29] for ubiquitous computing. When an agent registers an object with the DS it returns a lease to the agent giving the duration for which the information will be held. If the registration is not explicitly extended before the lease expires the object is expunged.

## Chapter 4

# Security

A number of agent platforms such as JADE assume a closed environment where there is a relatively high degree of trust between agents and absolute trust in the agent platform. The minimal security considerations in the FIPA specification relating to the directory facilitator reflect this assumption[8]:

The DF may restrict access to information in its directory, and will verify all access permissions for agents which attempt to inform it of Agent state changes.

A good overview of the security infrastructure requirements of agent middleware as a whole may be found in [28], which describes the AgentOS middleware underlying AgentScape. We limit ourselves here to security issues pertaining to the DS, namely controlling access to data (authorisation) and data integrity. We also discuss authentication which the DS requires for implementing access control, and the reasons for omitting encryption.

### 4.1 Authorisation

To conform to FIPA guidelines the DS should have some form of authorisation to restrict reading and writing objects. A typical implementation of DS authorisation is LDAP's access control lists, which specify who is allowed read, write, and *authenticate* permissions on a per-attribute basis.

The access control mechanism for RDF described in [10] is suitable for our needs. It uses RDF protection objects mapped to RDF and RDFS statements and describes methods to detect unauthorized inferences.

### 4.2 Authentication

Authorisation necessarily depends on access to the DS being authenticated. DSs like LDAP are sometimes deployed in the role of an authentication service. However, this abuse of DS functionality is poor security practice for several reasons:

1. A DS functions as a repository for publicly accessible information. Authentication secrets such as password hashes on the other hand, should never be exposed even to the authenticating entity.
2. A dedicated security service like Kerberos[14] could provide single sign-on, which is difficult to do in a general-purpose DS.

3. An authentication mechanism that is independent of the DS could be reused across other agent platform services

To discourage this practice we choose not to provide encryption of data within the DS. We assume the availability of an external authentication service such as a public key infrastructure or Kerberos. SASL (Simple authentication and security layer)[18] is useful for decoupling the actual authentication mechanism from the protocol. Unauthenticated (anonymous) access to the DS, as in LDAP, is of course also made available.

### 4.3 Integrity

An oft ignored security consideration is the integrity of data contained within a DS. In an open agent environment, our DS will accept data for publication from potentially untrusted agents. There is a risk of an agent repudiating a service level agreement for example, that it published in the DS. Furthermore, agents migrating to foreign locations may not completely trust the middleware (specifically the DS). Again a malicious DS may manipulate information to its own advantage. Thus a mechanism to ensure data integrity and data origin authentication is highly desirable.

To ensure the integrity and verify the origin of information, publishers will have the option of attaching either a message digest like MD-5 or the publisher's digital signature, to the published data[26].

An interesting extension to publishers signing data is to permit third parties to sign objects, establishing chains of trust between agents. AgentScape's Foncation secure location service provides this facility.

## Chapter 5

# Implementation

For the purpose of exposition the implementation may be divided into several logical components. Access to the DS is via Java remote method invocation and provides local query and modification operations on the RDF data store. Recursive query forwarding is handled by a broadcast overlay that forwards to the local DSs at each location and routes back replies. A third component manages metadata associated with directory entries like leases and access control lists. An overview of the important classes in the implementation are shown in Fig. 5.1. The system is implemented in Java and uses HP Labs' Jena RDF data store. The entire service is self-contained and not dependent on any external services, thus easily deployable as part of an agent platform.

### 5.1 DS Interface

Adding and modifying directory entries and querying the local DS is handled by proxying the methods Jena provides for RDF manipulation as Java remote interfaces. This is not ideal as RMI is Java specific, but was chosen as the most expedient way to making the Jena's RDF API available remotely. Additional language-neutral interfaces like SOAP and FIPA-ACL are planned for addition.

Entries to be added can be specified as either both RDF/XML fragments or the Jena API's `Resource` and `Property` classes. Modifications and deletions however, can only be specified using the latter form, because RDF/XML has no explicit syntax to express modification. New entries added to the directory are tagged with metadata such as lease durations and access control lists (ACLs). The RMI interface includes methods to query and set leases and ACLs.

If a query fails on the local DS it is transparently forwarded to other locations via the overlay. A shim layer (cf. `Main` class in Fig. 5.1) acts as the glue between the broadcast overlay and the RDF data store. It receives incoming packets from overlay, extracts the message body, and hands them to the Jena query processor in a separate thread of control. If the query returns a non-empty result set, a new reply packet is constructed to send the results back to the query initiator.

Although it would be possible to define SQL-like modification operations<sup>1</sup> to be sent over the routing overlay there is little point in doing so. An agent wanting to modify data could always locate it by issuing a query for it first. Hence broadcast of modification operations is unnecessary.

---

<sup>1</sup>The SPARQL draft standard defines only query operations

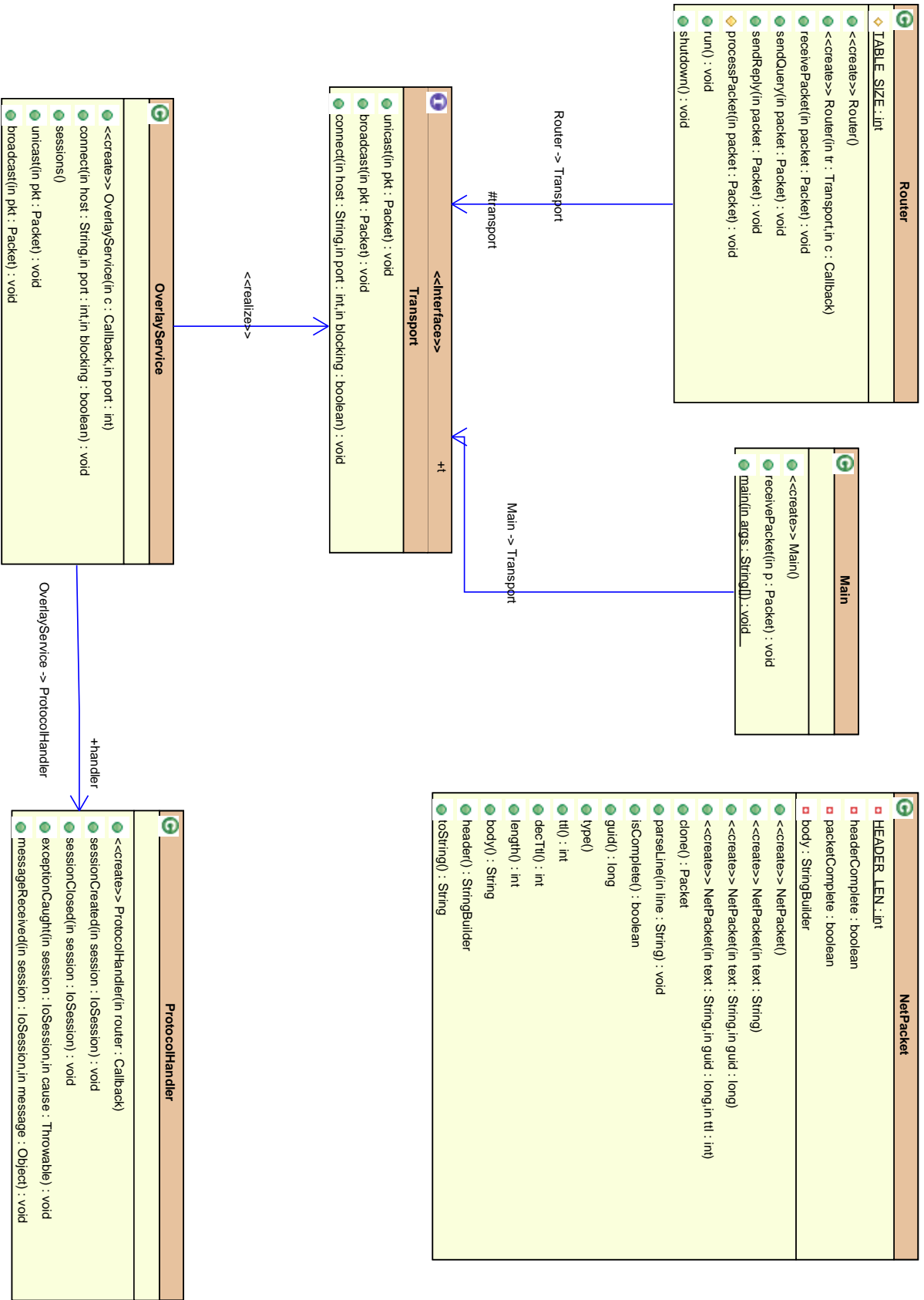


Figure 5.1: UML class diagram.

## 5.2 Routing Overlay

We evaluated several options for the broadcast overlay design but chose an unstructured overlay motivated by the ready availability of Gnutella implementations. However, there were unexpected caveats in reusing them.

### 5.2.1 Evaluation of Gnutella-based Routers

JTella a bare-bones Java Gnutella routing library was our first choice for routing. Unlike other clients JTella makes few assumptions that the queries are file name searches, so it seemed suitable for our purpose. Unfortunately it had several performance and stability issues, such as its overly aggressive connection management slowing down query routing.

Next we evaluated LimeWire, one of the most popular and mature Gnutella clients available. The Gnutella router implementation in the `com.limewire.gnutella.core` package, along with a custom implementation of the LimeWire `FileManager` interface was used for query forwarding. LimeWire implements the Gnutella 0.6 protocol where only super-peers broadcast queries amongst themselves. This requires leaf nodes to forward hashed indices of files to its parent super-peer. Indexing the RDF graph data model detracts from our broadcast based design. So we circumvented LimeWire’s complex super-peer election protocol to make each node a super-peer.

Both JTella and LimeWire made several implicit assumptions about bootstrapping using a relatively large number of peers on an existing Gnutella network. This made bootstrapping private network between DSs impossible, without major changes. We did attempt to use the modified LimeWire client on the public Gnutella network but were faced with a massive number of spamming peers.

### 5.2.2 A Generic Flood Routing Protocol

We finally chose to implement our own flooding broadcast overlay. The initial version used a binary protocol that used the same headers listed in Table 5.1 and our own networking code which used Java’s nonblocking I/O. Subsequently, we opted to use Apache-MINA (Multipurpose Infrastructure for Network Applications) for network management, which coincidentally is part of the Apache Directory Server project. MINA provides a convenient event-based abstraction based on Java’s high-performance non-blocking I/O APIs. MINA also has an I/O processing pipeline consisting of “filters” that perform tasks like SSL and compression to be added easily. As part of the transition, the previous binary protocol was converted to the more extensible and easy to debug text protocol described in Table 5.1.

Header	Description
Type	Query, Reply, Ping or Pong (the latter 2 are for peer discover)
TTL	Time to live: number of hops the packet is forwarded before being dropped
PacketID	A 128 bit random number identifying the packet for routing
Length	Length of packet body (query or reply) in bytes

Table 5.1: Protocol header descriptions

As illustrated in the UML class diagram Fig. 5.1 the routing overlay consists of four classes. Class `OverlayService` is a façade to the overlay router. It provides client applications methods for registering a callback to handle incoming queries, send replies and originate queries. The `ProtocolHandler` class handles callbacks from MINA triggered by incoming data and opening or closing connections with other peers. The protocol wire format is implemented by the `Packet`.

It is an HTTP inspired text-based protocol consisting of the headers given below, followed by an arbitrary byte stream in the body. These headers are parsed and stored in a hash table by the `Packet` class. The packet format can be extended with arbitrary headers for use in other applications.

The routing algorithm is implemented by class `Router` is basically same as the original Gnutella 0.4 flood routing protocol. Query packets are broadcast to all peers except the one it was received from while packets of type `Reply` are routed back to the originator on their `GUID`.

```
//Check the type of packet we received
switch (packet.type()) {
  case Query:
    //Keep a record to route back replies
    routeTable.put(packet.id(), packet.peer);
    //Send it to all neighbours who haven't seen it
    transport.broadcast(packet);
    //Forward it to the local RDF DB
    callback.receivePacket(packet);
    break;

  case Reply:
    id = packet.id();
    //Is this a packet I originated?
    if (forMe.contains(id))
      callback.receivePacket(packet);
    //Forward it to whoever sent me this query
    else if (routeTable.containsKey(id)) {
      packet.peer = routeTable.get(id);
      transport.unicast(packet);
    }
}
```

The virtue of such a simplistic algorithm is that it only incurs a hash table lookup or insertion per packet which has constant  $O(1)$  cost. However, the simulations we performed (described in the next chapter) showed flood routing results in an excessive number of duplicate packets being sent. To mitigate this problem we added a `visited` list to each query packet. Before each broadcast, a node excludes any neighbours in the `visited` list of the packet, from the list of recipients. It then adds all new recipients of the packet to the `visited` list. This ensures that packets do not loop back a node it has visited and that neighbours who receive a particular packet do not redundantly exchange it amongst themselves. This does not completely eliminate duplicates except in the case where the overlay is a fully connected mesh.

### 5.3 Managing Directory Entry Metadata

Metadata about directory entries such as ACLs and leases are stored as RDF properties, no different from an entries' regular properties. This means that queries return metadata elements along with an entry's regular properties. This is easily prevented either by rewriting the query to exclude for example subtypes of the `Metadata` RDF class or by filtering out the metadata elements from query results. Although not currently done, the access control lists of the results must be checked for proper permissions before returning the results.

As explained previously, leasing is used as a means to prevent stale directory entries from polluting the directory. A lease management thread periodically goes through all directory entries and removes those whose lease duration has expired. It does this by issuing a query for the `lease` property and deleting all subject resources that point to expired leases. Any orphaned object resources (i.e. those that have no properties pointing to them) are also removed.



# Chapter 6

## Evaluation

Our overall aim in evaluating our architecture is to verify our claims that it scales to a moderate size of 100-1000 nodes. The query broadcast in our system incurs a query at each intermediate node. This could cause a potential bottleneck if the RDF data store is unable to handle the load.

We are particularly interested in knowing conditions under which congestion sets in. Several benchmarks of the Jena RDF data store have been done. A large-scale bibliography database application of Jena [20] gives an average query time of 385ms for a two million record database, while authors of Jena report a value of 0.84ms for a database of 1,000 objects. As our directory service is distributed the number of records an individual nodes holds will be quite modest, so the latter is closer to our use case.

### 6.1 System Simulation

Since the parameter of primary interest is the query load on RDF data stores at nodes, we opted for a simulation over actual deployment of the DS. As always, a simulation runs the risk of ignoring important real-world parameters, but it does allow for larger scale experiments under controlled conditions.

We constructed a discrete time simulator that mimics query broadcast and reply routing of the system. The simulated nodes run the same routing algorithm presented in 5.2.2. Running hundreds of RDF store instances on a single JVM running the simulation was simply not practicable, so we assign a fixed query cost to the database query that packet propagation incurs at each node. Based the Jena benchmarks cited above a conservative value of 2ms per query was chosen as the basis of our estimates.

During a discrete time step of the simulation each node generate a zero or more queries with Poisson rate distribution. Nodes reply to each query received with a preassigned uniform probability. By convention we take each time step to represent one second and set query and reply rates accordingly.

The number of queries generated, queries received, replies generated, replies received and the number of duplicate packets queued (due to flood routing) is measured at each node per time step. The metric we are most interested in is the number of queries received per second, as this determines the load on the RDF data store. This is graphed as a frequency (i.e. per-node per time step counts) histogram.

We began by conducting some basic experiments both to verify the correctness of our simulation and establish some basic results. We use the JUNG (Java Universal Network/Graph) Framework to construct networks with various topologies.

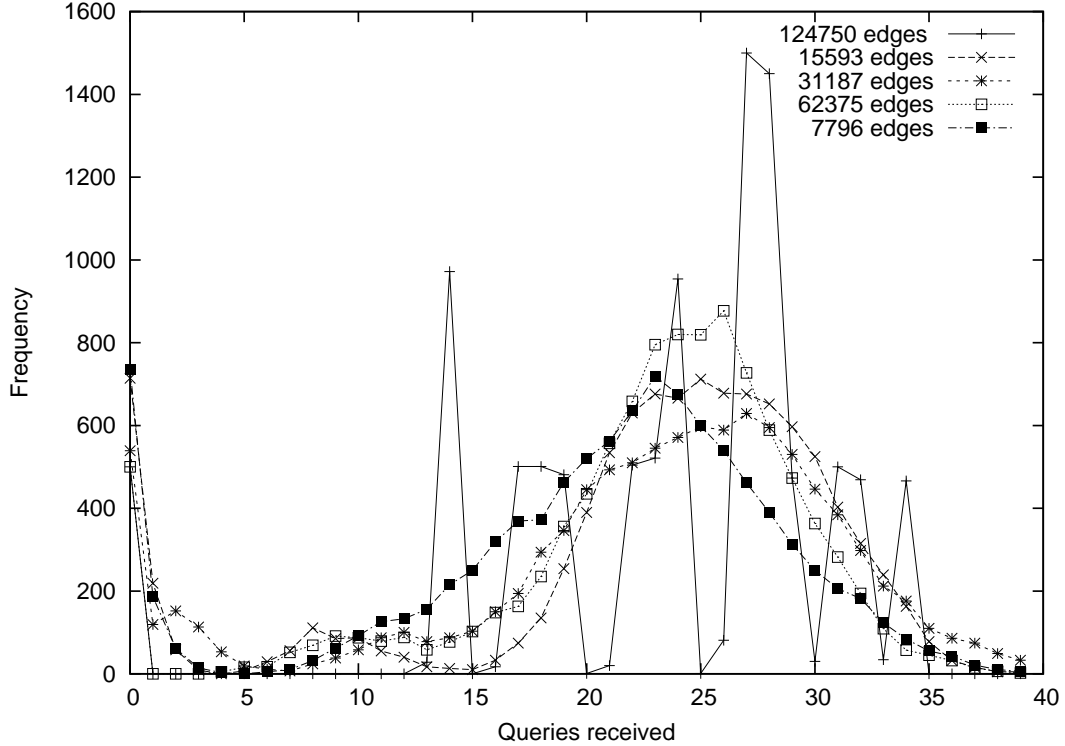


Figure 6.1: Overlay connectivity versus queries received per time step.

## 6.2 Node Connectivity

Our first experiment ascertains the effect of connectivity of the query load at a node. We used JUNG’s simple random graph generator which generates graphs with a given number of edges and vertices. We constructed random overlay networks of 500 nodes and measured queries received by each node while the total number of links to construct overlays of varying diameter. We fix query rate at 0.05 (1 query per 20 seconds) and reply probability at 5% and set the query packet time-to-live at 4 (i.e. propagate a packet a maximum of four hops).

The number of queries received per node in a time step is between 25 and 27 according to Fig. 6.1. That this is not effected by connectivity is to be expected as the diameters of all the networks do not exceed the TTL 4. Thus all queries should reach the entire network regardless of node connectivity, barring any partitions in the network. Using our estimate of 2ms per query and the upper bound of 40 queries/second gives an acceptable overhead of 80ms.

## 6.3 Number Nodes

To see how well the system scales the number of nodes was varied from 200 to 1000 in increments of 200. Graphs used were generated by JUNG’s Erdos-Renyi generator which connects each vertex with a given probability  $p$ . We set  $p$  slightly higher than  $\frac{\log(n)}{n}$ , where  $n$  is the number of nodes. This “phase transition value” is the minimum  $p$  value at which at a random graph is likely to be connected.

Figure 6.2 shows the average query rate varying from 4 for 200 nodes to 15 for a 1,000 node network. It never exceeds 40 giving a maximum RDF processing time of 80ms at each node.

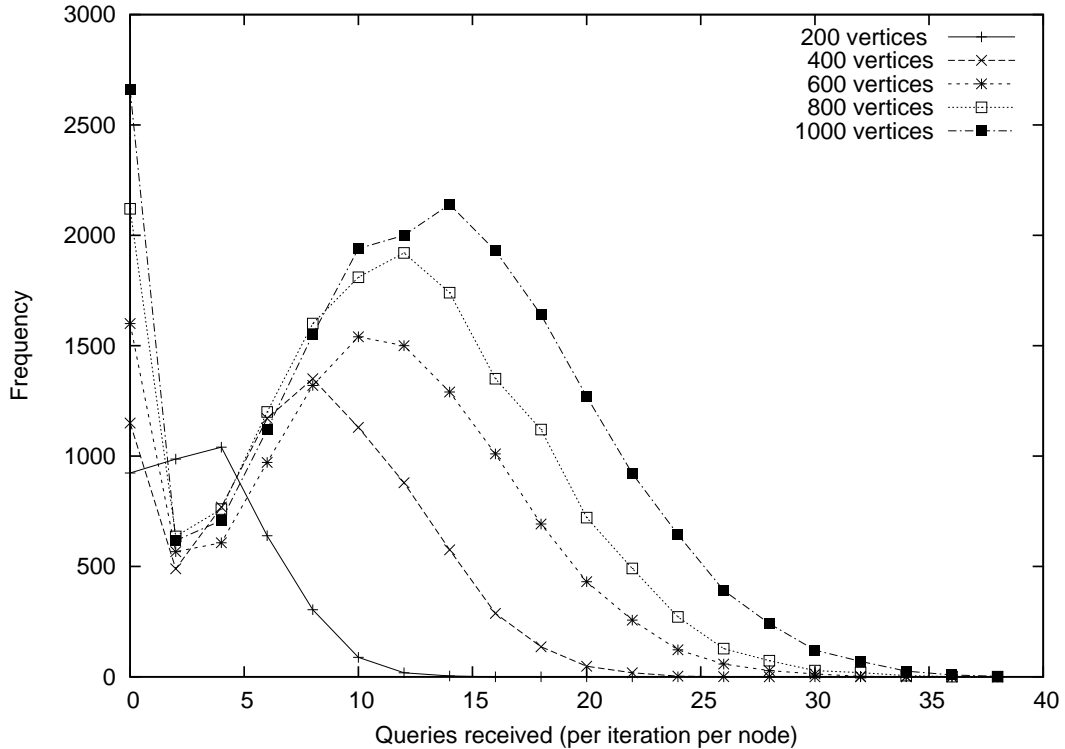


Figure 6.2: Number of nodes versus queries received per time step.

## 6.4 Effects of Query Rate

The previous experiments a low query rate setting of 0.05. As explained in Section 3.3 the rationale for this is that, in typical use we expect wide area query to be relatively infrequent.

Figure 6.3 shows the expected strong effect query generation rate has on the queries received per second. Increasing the query rate to one query every two seconds sees the maximum number of queries received rise to 180, costing 360ms of processing at the RDF data store.

## 6.5 Duplicate Packets

Given that our choice of flood routing is somewhat naive, the number of duplicate packets is of concern. We counted duplicate packet on the same 500 node overlay configurations used to test the effect of node connectivity.

As seen in Fig. 6.4 the number of duplicate packets rises sharply as the density of the network increases. This is in spite of adding a visited nodes header to which a node adds all neighbours it send the packet to. Flood routing is thus unsatisfactory except in sparsely connected overlays, but this is the common case in practice.

## 6.6 Small-world Networks

Studies[12] show that most real-world networks exhibit the *small world* property. Unlike random graphs where an edge between any two nodes is equiprobable, small-world graphs exhibits densely connected subgraphs, and a number of highly connected *hub* nodes. A node can often reach any other node via a hub node, giving these networks a low diameter. In social networks this manifests itself as the “six degrees of separation” principle.

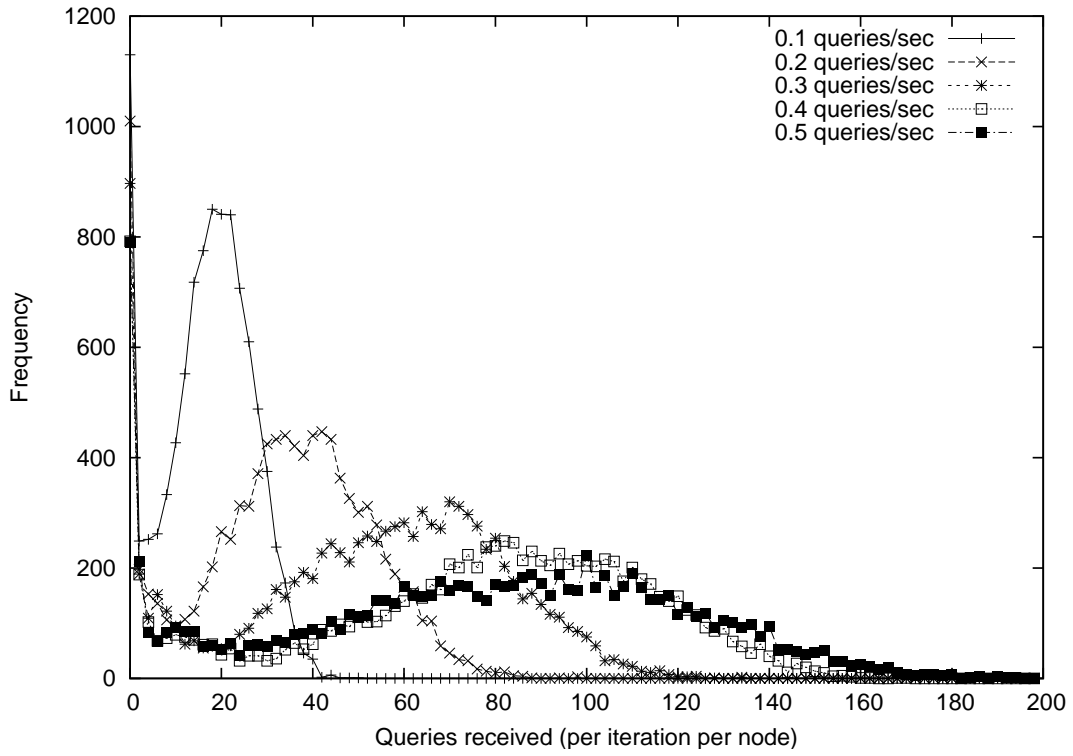


Figure 6.3: Poisson query generation rate versus queries received per time step.

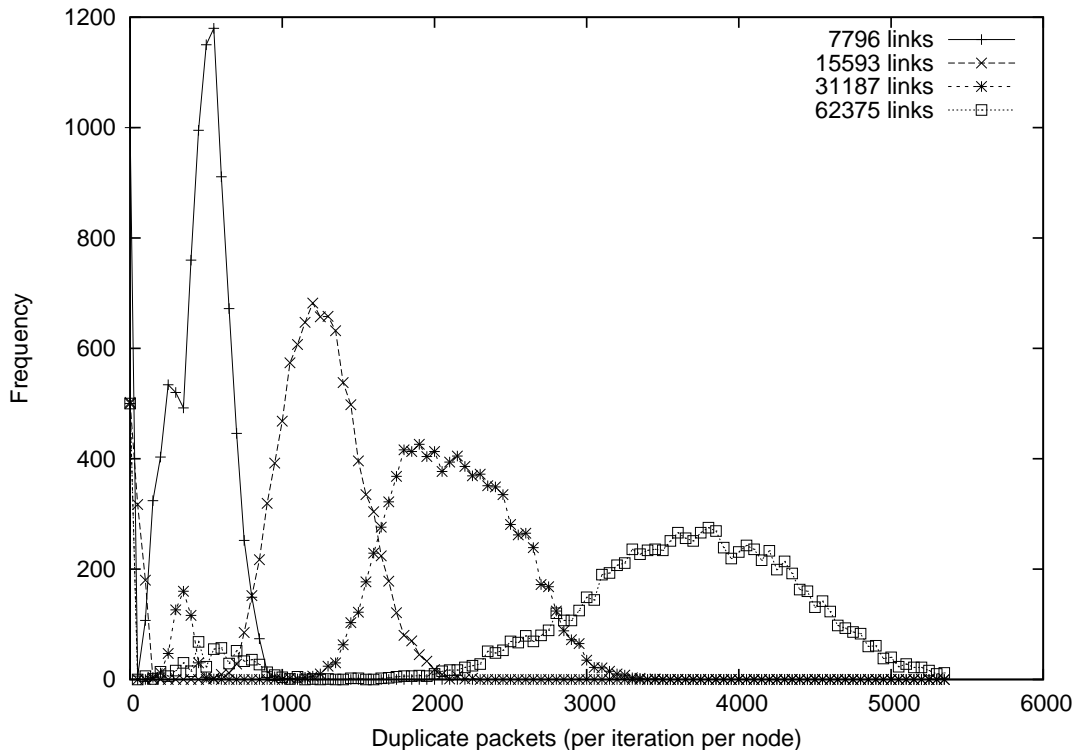


Figure 6.4: Total number of overlay links versus duplicates received with packet TTL=3.

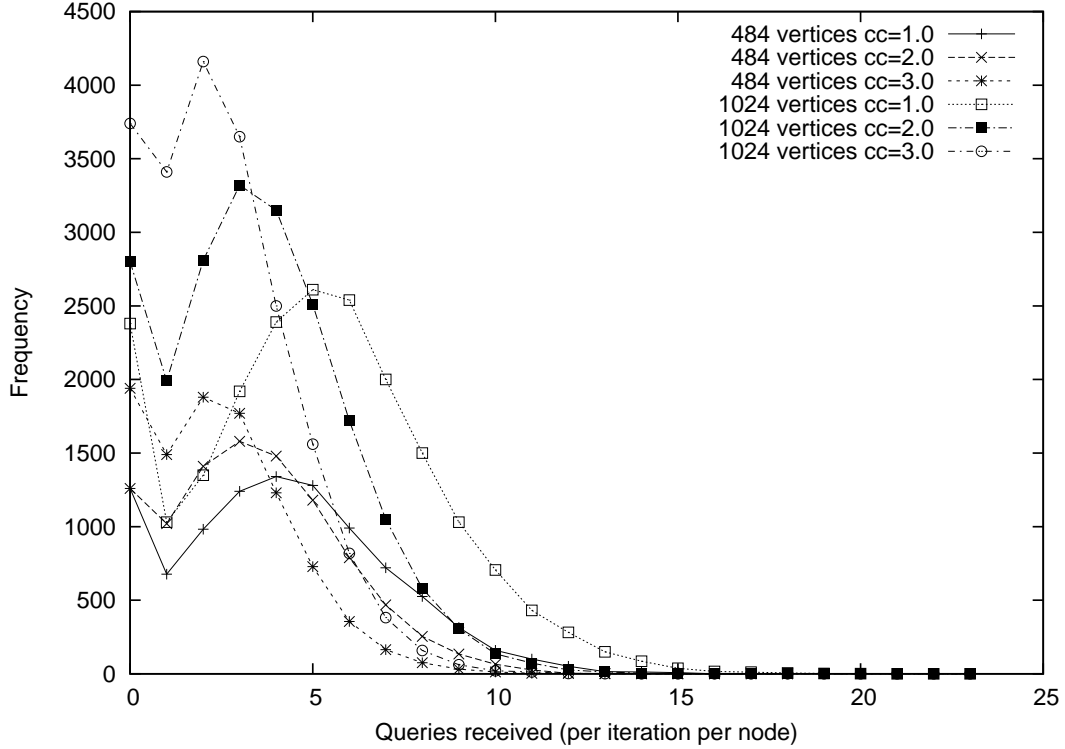


Figure 6.5: Number of nodes and clustering coefficient versus queries received per time step.

We would like to know how a small world network structure affects our system. Figure 6.5 shows the effect of varying the number of nodes and the *clustering coefficient*, which controls how strongly clustered the network is (i.e a lower *cc* tends towards random networks). The query rate and TTL values were set at 0.05 and 3 as before.

The graph shows some rather surprising results. First notice that the mean query rates for 1024 and 448 nodes with equal clustering coefficients are situated very close together. For instance the means for  $cc = 1$  are approximately 4 and 5. The second surprise is that the query load of 4 for a 1024 node small-world network is about a quarter of the load of 15 for a 1,000 node random network. This runs counter to our intuition that query broadcasts would cause hot-spots at hub nodes in small-world networks overlays. We do not have an adequate explanation for these results currently.

## 6.7 Discussion

The results from varying the number of nodes shows that using a broadcast-based approach does indeed scale to around 1000 nodes, provided that the rate of query remains at a low value of about one in twenty seconds. As expected we see little effect of the network connectivity on the query load at a node. Also unsurprising is the fact that the query rate directly impacts the load at an individual node. By far the most interesting result is that the systems performs slightly *better* on small-world networks, compared to random networks. Finally it is clear that our naive flood broadcast does not well scale due to the prohibitively high number of duplicate packets, in comparison to say, Structella[6].



# Chapter 7

## Conclusion

This chapter presents a brief review of related work. We conclude with a critical appraisal of our work and future work that would improve the system.

### 7.1 Related Work

A significant body of work exists on decentralised naming (i.e. white pages) implementations for mobile agents. Wright [30] describes a white pages directory loosely based on DNS that supports rapid updates using lease-based name expiry for cache and replica invalidation. Alternatives to supporting mobility include the *home node* approach, where the entity's original location tracks its movement and forwards messages to its current location. A variation on this technique that alleviates the bottleneck at the home node is presented in [17].

Decentralised yellow pages directories have also been proposed. Banerjee [3] describes using a DHT to hash search terms and route queries between UDDI registries. TerraDir [24] is a distributed naming service with support for aliasing (symlinks), regular-expression search, based on a structured overlay. It claims to also support attribute based query using “dynamic view materialisation” that creates a new overlay tree hierarchy.

Related to our design is the area of distributed RDF query. RDFPeers [5] relies on an enhanced Chord overlay to store RDF triples hashed by subject, predicate and object. However, it lacks support for RDF's inference capabilities. Gridvine [1] is another attempt at using a structured overlay for RDF storage and query. Gridvine uses the Trie-structured P-Grid as the basis for a yellow pages directory targeted at Grid computing. Our own query overlay architecture is very similar to Edutella[19] a RDF based P2P e-learning system.

Our overall goals are most similar in spirit to MIT's intentional naming service (INS)[2]. INS is an attribute based directory that uses an application level overlay for decentralisation. INS however is targeted at pervasive computing and thus features location aware discovery for instance, but lacks ontology or inference support importance in an agent context.

### 7.2 Future Work

A weakness of our architecture is our flooding broadcast overlay. As noted previously this could quite easily be replaced by a more efficient broadcast built atop structured overlays. Structella[6] a proposal to build Gnutella atop the Pastry structured overlay, guarantees broadcast using  $O(n)$  messages in  $O(\log(n))$  hops. The RDF data store query load measurements based on our simulation still hold regardless of actual broadcast technique. The security and leasing aspects of our design also need to be fully fleshed out.

### 7.3 Concluding Remarks

Our stated goal was to build a decentralised directory service for mobile agents. The design presented supports both attribute based syntactic query as well as annotating entities with an ontology for semantic query within a unified framework. Based on the requirements for a distributed DS we chose a broadcast-based decentralised query architecture. The system's scalability along various parameters was examined using a simulation.

In hindsight the scope of the work turned out to be somewhat overly ambitious. Our energies were spent designing three very different aspects: RDF-based rich query, query routing and security. Possibly due to this certain design compromises were made that impacted overall system performance. Studying just one of these three in-depth might have yielded better results.

# Appendix A

## Partial RDF Schema for Compute Resources

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
  <!ENTITY as "http://iids.org/agentscape">
]>
<rdf:RDF xml:lang="en"
  xmlns:rdf= "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs= "http://www.w3.org/2000/01/rdf-schema#"
  xmlns= "http://iids.org/agentscape#"
  xml:base= "&as;">

  <!--Standard Resource classes-->
  <rdfs:Class rdf:ID="Entry">
    <rdfs:comment>superclass for all DS entries</rdfs:comment>
  </rdfs:Class>

  <rdfs:Class rdf:ID="HWResource">
    <rdfs:comment>Hardware resources</rdfs:comment>
    <rdfs:subClassOf rdf:resource="#Entry"/>
  </rdfs:Class>

  <rdfs:Class rdf:ID="Service">
    <rdfs:comment>Software services</rdfs:comment>
    <rdfs:subClassOf rdf:resource="#Entry"/>
  </rdfs:Class>

  <rdfs:Class rdf:ID="ComputeResource">
    <rdfs:subClassOf rdf:resource="#HWResource"/>
    <rdfs:comment>Computational resources</rdfs:comment>
  </rdfs:Class>

  <rdfs:Class rdf:ID="StorageResource">
    <rdfs:subClassOf rdf:resource="#HWResource"/>
    <rdfs:comment>Storage resources</rdfs:comment>
  </rdfs:Class>

  <rdfs:Class rdf:ID="WebService">
    <rdfs:comment>A SOAP web service</rdfs:comment>
```

```

    <rdfs:subClassOf rdf:resource="#Service"/>
</rdfs:Class>

<rdfs:Class rdf:ID="AgentService">
  <rdfs:comment>A service hosted by an Agent</rdfs:comment>
  <rdfs:subClassOf rdf:resource="#Service"/>
</rdfs:Class>

<rdfs:Class rdf:ID="OS">
  <rdfs:comment>Families of OSs</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:ID="POSIX">
  <rdfs:comment>Nix-like OSs</rdfs:comment>
  <rdfs:subClassOf rdf:resource="#OS"/>
</rdfs:Class>

<rdfs:Class rdf:ID="WIN32">
  <rdfs:comment>MS OSs</rdfs:comment>
  <rdfs:subClassOf rdf:resource="#OS"/>
</rdfs:Class>

<!--Properties for standard resources-->
<rdf:Property rdf:ID="LocatedAt">
  <rdfs:comment>AgentScape Location identifier</rdfs:comment>
  <rdfs:domain rdf:resource="#as;#Entry"/>
</rdf:Property>

<rdf:Property rdf:ID="PublisherSig">
  <rdfs:comment>Publisher's digital signature</rdfs:comment>
  <rdfs:domain rdf:resource="#as;#Entry"/>
</rdf:Property>

<rdf:Property rdf:ID="ServiceEndpoint">
  <rdfs:comment>Service-specific connection information</rdfs:comment>
  <rdfs:domain rdf:resource="#as;#Service"/>
</rdf:Property>

<!--Properties for hardware resources-->
<rdf:Property rdf:ID="runsOS">
  <rdfs:comment>Operating system</rdfs:comment>
  <rdfs:domain rdf:resource="#as;#ComputeResource"/>
  <rdfs:range rdf:resource="#as;#OS"/>
</rdf:Property>
</rdf:RDF>

```

# Bibliography

- [1] K. Aberer, P. Cudre-Mauroux, M. Hauswirth, and T. van Pelt. GridVine: Building Internet-scale semantic overlay networks. In *Proceedings of the International Semantic Web Conference (ISWC)*, volume 3298 of *Lecture Notes in Computer Science*, pages 107–121. Springer, Berlin, Germany, 2004.
- [2] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. *ACM SIGOPS Operating Systems Review*, 33(5):186–201, 1999.
- [3] S. Banerjee, S. Basu, S. Garg, S. Garg, S.-J. Lee, P. Mullan, and P. Sharma. Scalable Grid service discovery based on UDDI. In *Proceedings of the 3rd International Workshop on Middleware for Grid Computing (MGC'05)*, pages 1–6, Grenoble, France, 2005.
- [4] F. Bellifemine, A. Poggi, and G. Rimassa. JADE: A FIPA2000 compliant agent development environment. In *Proceedings of the Fifth International Conference on Autonomous Agents (AGENTS'01)*, pages 216–217, 2001.
- [5] M. Cai and M. Frank. RDFPeers: A scalable distributed RDF repository based on a structured peer-to-peer network. In *Proceedings of the 13th International Conference on World Wide Web*, pages 650–657, 2004.
- [6] M. Castro, M. Costa, and A. Rowstron. Should we build Gnutella on a structured overlay? *ACM SIGCOMM Computer Communication Review*, 34(1):131–136, 2004.
- [7] B. Chun et al. Querying at Internet scale. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 935–936, Paris, France, 2004.
- [8] FIPA abstract architecture specification. <http://fipa.org/specs/fipa00001/SC00001L.html>, 2002.
- [9] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proceedings of the 6th IEEE Symposium on High Performance Distributed Computing*, pages 365–375. IEEE Computer Society Press, 1997.
- [10] A. Jain and C. Farkas. Secure resource description framework: An access control model. In *Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies (SACMAT'06)*, pages 121–129, Lake Tahoe, CA, 2006.
- [11] P. Keleher, B. Bhattacharjee, and B. Silaghi. Are virtualized overlay networks too much of a good thing? In *Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 225–231. Springer, Berlin, Germany, 2002.

- [12] J. Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the 32nd ACM Symposium on Theory of Computing*, 2000.
- [13] G. Klyne and J.J. Carroll. Resource description framework (RDF): Concepts and abstract syntax. W3C recommendation, W3C, February 2004. <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>.
- [14] J. Kohl and C. Neuman. The Kerberos network authentication service (v5), 1993.
- [15] F. Manola and E. Miller. RDF primer. W3C recommendation, W3C, February 2004. <http://www.w3.org/TR/2004/REC-rdf-primer-20040210/>.
- [16] D.G.A. Mobach, B.J. Overeinder, and F.M.T. Brazier. A resource negotiation infrastructure for self-managing applications. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC 2005)*, pages 381–382, Seattle, WA, 2005.
- [17] L. Moreau. A fault-tolerant directory service for mobile agents based on forwarding pointers. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02)*, pages 93–100, Madrid, Spain, 2002.
- [18] J. Myers. RFC 2222: Simple Authentication and Security Layer (SASL), October 1997.
- [19] W. Nejdl et al. Edutella: A P2P networking infrastructure based on RDF. In *Proceedings of the Eleventh International World Wide Web Conference (WWW2002)*, Honolulu, Hawaii, May 2002.
- [20] K. Portwin and P. Parvatikar. Scaling Jena in a commercial environment: The Ingenta MetaStore project. In *Proceedings of the 2006 Jena User Conference*, Bristol, UK, May 2006.
- [21] J. Postel. RFC 1591: Domain name system structure and delegation, March 1994. Status: INFORMATIONAL.
- [22] D. Richards, S. van Splunter, F.M.T. Brazier, and M. Sabou. Composing web services using an agent factory. In L. Cavedon, editor, *Extending Web Services Technologies*, volume 13 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 229–252. Springer, Berlin, Germany, April 2005.
- [23] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, January 2001.
- [24] B. Silaghi, S. Bhattacharjee, and P. Keleher. Query routing in the TerraDir distributed directory. In *Scalability and Traffic Control in IP Networks II*, volume 4868 of *Proceedings of SPIE*, pages 299–309, Boston, MA, July 2002.
- [25] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*, pages 149–160, San Diego, CA, 2001.
- [26] G. Tummarello, C. Morbidoni, P. Puliti, and F. Piazza. Signing individual fragments of an RDF graph. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web (WWW'05)*, pages 1020–1021, Chiba, Japan, 2005.

- [27] M. van Steen, F.J. Hauck, P. Homburg, and A.S. Tanenbaum. Locating objects in wide-area systems. *IEEE Communications Magazine*, 36(1):104–109, January 1998.
- [28] G. van 't Noordende, F.M.T. Brazier, and A.S. Tanenbaum. Security in a mobile agent system. In *Proceedings of the First IEEE Symposium on Multi-Agent Security and Survivability*, Philadelphia, PA, August 2004.
- [29] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, 1999.
- [30] T. Wright. Naming services in multi-agent systems: A design for agent-based white pages. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, pages 1478–1479, New York, NY, August 2004.