Schema Based Peer-to-Peer Data Integration

Final Year Project Report

Charalambos Lazanitis
⟨ cl201@doc.ic.ac.uk ⟩

Supervised by Dr Peter M^c Brien
Second Marker: Nikos Rizopoulos

Department of Computing
Imperial College London, UK

15 June 2005

Abstract

Peer to peer data integration (*P2PDI*) is the process of integrating heterogeneous data sources in a peer to peer (*P2P*) network. The traditional approach to P2PDI achieves integration by defining pair wise mappings between schemas (of peers) something that we believe limits its scalability.

This report describes the analysis, design and implementation of a prototype P2PDI system that uses an alternative approach [MP03] based on the *Both as View (BAV) methodology* [MP03b]. In this system peers can make schemas *public* and peers in the network can provide pathways to any of the public schema. Integration between two schemas is achieved when they both of them have a pathway to some published schema.

The following are covered by the report:

- A description of the formal framework that we developed for modelling such systems.
- Analysis of how the model responds to several protocols we propose.
- Explanation of the implementation of an API that uses the AutoMed [AutoMed] infrastructure to provide capabilities for building systems of this nature.
- Demonstration of a working system that uses the API.

Acknowledgements

It is amongst the greatest clichés when writing a supervised report to say a "big thanks" to your supervisor. And as much as I despise starting my dissertation with a stereotype, I really do have to express my collective gratitude towards *Dr McBrien*. I say a "big thanks" not only for his guidance and patience during this project but also for his mentoring throughout all my brief encounters with the research world in the past.

Many thanks to *Nikos Rizopoulos* whose interest in this project was very beneficial and helped design many vital parts of the final software. Also for helping me understand many issues around AutoMed during the first stages of the project and all his helpful suggestions about the report and presentation towards the end.

Special acknowledgements to *Dr Gardner* and *Dr Drossopoulou* whose lecturing greatly influenced my formal approach towards the project and the abstract analysis on which the final design was based.

Table of Contents

1 Introduction

1.1 Motivation

We all learned things from other people who were willing to share their knowledge with us in the same way we share our knowledge with others. In this sense human society can be viewed as a network of "*knowledge sharing peers*" the power of which can be appreciated by the fact that most of the things we know actually come from our participation in this network. In the same way that humans possess knowledge about the world so do organisations (like universities, companies etc) which they store as database information. So imagine the possibilities that would arise if this knowledge could be shared in the same way as humans. However, creating a *Peer to Peer (P2P)* system where peers can share *semantic-rich* information (databases) is very difficult.

A P2P system is a network of machines where all participants act as service providers to the entire network and at the same time take advantage of the collective service provided by the other participants. The dynamic nature of the network, the lack of centralized administration and the fact that its power increases as more peers join makes the P2P networking a very attractive model and has been used successfully by popular file sharing systems.

But sharing files is easy! There is no consideration of the semantics of the data (the meaning of the information in the files) and a file is identified by a small fixed set of attributes hence locating a file can be done by simple string matching. This makes the communication between the peers feasible. So what would happen if we tried to move from sharing just plain files to sharing richer and more structured data like database information?

Typically different organisational databases will use a different *schema* (structure) for the same type of information. The problem in sharing this information is how to *integrate* the different schemas so that the participants can communicate with each other. The problem becomes even more difficult when dealing with a peer to peer system due the features mentioned above. Integration of data in a peer to peer system is referred to as Peer to Peer Data Integration (P2PDI).

1.2 Current Systems

The problem of P2PDI is currently being researched and systems that deal with it do exist [FKL+04, TIM+03]. Peers in the network have an associated schema which represents their view of the world. Integration between two peers is achieved by describing a *mapping* that defines how the schema of one relates to the schema of the other. This is used to transform the queries expressed on one schema to queries expressed on the other one.

For example consider Figure 1.1 . Since there is a mapping between peer *A* and *B* then queries expressed on the schema of A can be transformed to queries are expressed on the schema of one can be mapped to queries expressed on the schema of the other. This way the communication between the peers can occur. More interestingly, if A poses a query to B, the latter can transform the query further exploiting its logical connection with C and send the result back to A. This way the query can be distributed to all logically reachable peers. A peer can therefore join the network if it provides a mapping from its schema to a schema in the network.

This is the approach followed by the *Piazza* [TIM+03] and the *CoDB* [FKL+03] projects where working systems are being developed and tested. These projects amongst other things research the correctness of queries in such networks, optimization and termination of query distribution algorithms, caching policies and materialisation of data.

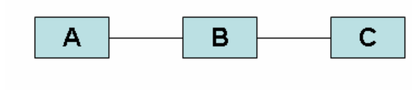


Figure 1-1 A simple P2PDI network with 3 peers

1.3 The Problem with Current Systems

There are several problems with the mentioned approach to P2PDI:

1) It is *not scalable*. Providing a mapping from one schema to another is a process that cannot be fully automated and requires human intervention and thus achieving a connection is a costly operation. This means that as the number of peers increases more and more such connections will need to be achieved by each participant which is likely to become unmanageable. For example in a fully connected network of n peers there will need to be $(n-1)/2$ such mappings per peer (one between each pair)!
2) There is no mechanism for peers to learn about transitive mappings in the network. For example in Figure 1.1, since A is mapped to B and B to C there should be a map from A to B which can be derived from the two existing mappings. This is not possible under the existing approach.
3) Because of (2), peers that are indirectly connected can only reach each other through intermediate peers. For example any query that is passed from A to C can only go through B. This increases the load on B (creating potential bottlenecks) but also makes the communication intolerant to any failure of B.
4) Achieving a connection between two peers requires at least one of the peers having knowledge of the structure and semantics of the other peer's schema. This is a very restrictive assumption for arbitrary P2P networks.

1.4 About the Project

This project takes a new approach [MP03] to P2PDI based on a data integration technique called *Both as View (BAV)* [MP03b] which is being used by the *AutoMed* project [AutoMed] to provide data integration tools (but not for P2PDI). In the BAV approach mappings between schemas are expressed using *transformation pathways*. Their main feature is that they are *bidirectional* and they can be *appended* to each other to create larger pathways.

Unlike the traditional systems, in our approach there is no direct mapping from one peer to another. Instead, peers provide pathways from their schema to one or more *public schemas*. (which is termed as "*implement*" the public schema) If two peers implement the same public schema a mapping between the two can be derived by appending the two pathways. For example the network of Figure 1.1 will look like Figure 1.2 under our approach where each pathway is represented by an arc.

We now give a brief outline of how this approach tackles the four problems mentioned above:

1) With this approach a particular public schema can be implemented by several peers and two peers are connected if they implement the same schema. In the extreme case there can be just one public schema which they all implement and therefore we could

have a fully connected network with just one pathway from each peer (compare that with $(n-1)/2$).

2) Peers can pass meta-information about pathways which can be used to create new connections. For example in Figure 1.2 peer B can pass information to A about w2 and w3 which can be appended to w1 to form a pathway to pschema2 which will connect A and C.
3) After connection to pschema$_2$ as explained above, A and B can communicate directly without depending on B
4) The structure of the peer schemas are essentially hidden from each other. Only the public schemas need to be known by the peers that implement them.

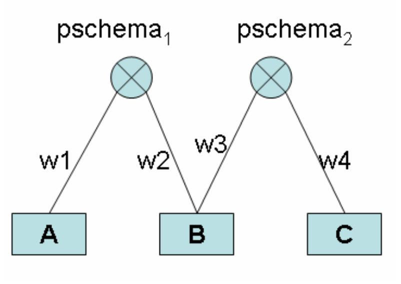


Figure 1-2 A simple network with 2 public schemas and 3 peers

1.5 Objectives

This project is the first attempt to implement a network based on this model. Its aims are:

- To provide a *formal framework* for designing and analysing such networks.
- To *extend the AutoMed API* to support P2PDI and use this API in order to develop a *working system*.

1.6 Achievements

This project claims the following achievements:

1) Defined a *mathematical model* of the network described above that provides a formal framework that is generic and implementation independent. It can be used to analyse various properties about the system, define network behaviour and analyse the evolution of the system under different communication protocols.
2) Used the formal model to define abstractly a *communication protocol* including an inter-peer *communication language*. We identified some basic properties about the protocol and evaluated its complexity.
3) Based on our formal work we developed an infrastructure that extends the AutoMed API so that AutoMed can be used in a P2PDI network. It deals with the main issues for enabling and facilitating this kind of communication and provides an API appropriate for implementing different communication protocols of varying complexity.
4) We built a *working system* that implements the abstract protocol. We used it to build small networks and tested them against simple data sets.

1.7 About this Report

- *Chapter 2* gives a background of all the main issues that relate to the P2PDI and introduces the paper [MP03] that was the starting point for this project.

- *Chapter 3* describes a mathematical abstract model of the system that we created.
- *Chapter 4* defines an abstract protocol for P2PDI and provides an evaluation based on the abstract model
- *Chapter 5* gives an overview of the design and implementation of the software developed.
- *Chapter 6* describes an infrastructure built to serialise and transfer meta-data information stored in AutoMed over the network.
- *Chapter 7* describes a framework developed that handles message passing from one peer to another and explains how it is used to define different types of messages
- *Chapter 8* describes the overall framework to define different communication protocols and illustrates how the abstract protocol of Chapter 4 was implemented.
- *Chapter 9* evaluates the achievements of this project and identifies some future work.

2 Background

This chapter introduces the reader to the general concept of data integration in section 2.1. Section 2.2 gives a description of the AutoMed framework that is was used by this project. Then it informs the reader of the current state of the art in peer-to-peer data integration viewing the matter from many different perspectives. Finally, when all the related concepts have been explained, in 2.4 we introduce the suggestion for P2P integration using BAV, which is the starting point of this project.

2.1 Data Integration

This section gives an introduction to the concept of Data Integration (**DI**).

It is a modern world axiom that "information is power". Organisations keep large databases for their own use, but it's often the case that information in one organisation can be useful to another and vice versa and therefore if the two organisations could share their information, it would mean that both their *information power* would increase benefiting both. However, the databases of the two organisations are typically heterogeneous which does not allow the data to be unified.

Generalising the above, it is not surprising that large enterprises, business organisations, e-government systems and in short any kind of internetworking community needs an integrated virtualised access to distributed information resources. [Len02]. There is a need for a way to share the information possessed by each member of the community for the benefit of the entire group but still allowing each member to have a local authority to its own resources. This is the problem that data integration attempts to tackle.

First we should give a more precise definition of what DI refers to. [Lenz02] defines DI as *"the problem of combining data residing at different sources and providing the user with a unified view of these data"*. Similarly [BKL+04] defines it as *"the process of combining several data sources such that they may be queried and updated via some common interface"*. An exhaustive description of data integration is not in the scope of this document (see [[McB03] and [Lenz02] for this) but instead we give a very brief summary of the models and techniques that are generally used.

The majority of the approaches in data integration attempt to somehow form a **mapping** between schemas (and data models) such that either the data that is defined in a schema can be *materialised* in the mapped schema or a query expressed to a schema can be *translated* to a query defined to the mapped schema. Most approaches have the notion of *local* and *global* schemas. Local schemas are the schemas of the sources and global schemas are the shared ones.

One of the most important aspects in the design of data integration systems is the specification of the correspondence between the data at the sources (local schemas) and those in the global schema [Lenz02]. There are four basic approaches.

Global as View (GAV): In the GAV approach, there is an association between each element of the global schema and a view over the sources. This means that the information held in the global schema is defined in terms of the sources

Local as View (LAV): In the LAV approach, each data source is expressed as a view over the global schema. This means that the sources are defined in terms of the global schema.

Global Local as View (GLAV): Combines the expressive power of both GAV and LAV. It can be considered as a variation of LAV in the sense that the Global schema is independent of the sources.

Both as View (BAV): BAV is an approach to data integration, where schemas are mapped to each other using a sequence of **schema transformations** that we call transformation **pathway**. An important feature is the fact that the transformations are *bi-directional* and it's therefore easy to go from the local to the global schemas and vice versa. [MP03] [MP03b].

Peer-to-peer (P2P): Unlike the other techniques, in P2P systems, data integration is *not based on a global schema*. Instead there is a network of peer nodes which can define mappings from themselves to other nodes thus forming a **logical association** between peers [Len02], [TIM+03]. This can be exploited during query evaluation, where queries can be passed along the logical associations, thus exploiting the entire network.

2.2 AutoMed and BAV

We now briefly describe the structure and semantics of **AutoMed** which is a joined project between Department Of Computing, Imperial College and Department of Computer Science in Birkbeck that provides an implementation of the **BAV** approach to data integration. This section also gives some more details about the BAV approach and places it within the AutoMed context. More details can be found in [AutoMed] [BKL+04] and [McB04].

AutoMed provides a framework for applications to implement BAV integration which as argued in [MP03b] and [BKL+04] subsumes the expressive powers of LAV, GAV and GLAV integration techniques (see section 2.1) but also has a clear methodology for handling a wide range of data models in the integration process as opposed to the other approaches that assume integration is always performed in a single common data model.

2.2.1 The BAV Methodology

In the BAV approach we map schemas by defining a pathway of **primitive transformation steps** applied in sequence to the source schema that is transformed to the target schema. These transformations are *bi-directional* and therefore you can be used to define a reverse mapping from the target to the source schema.

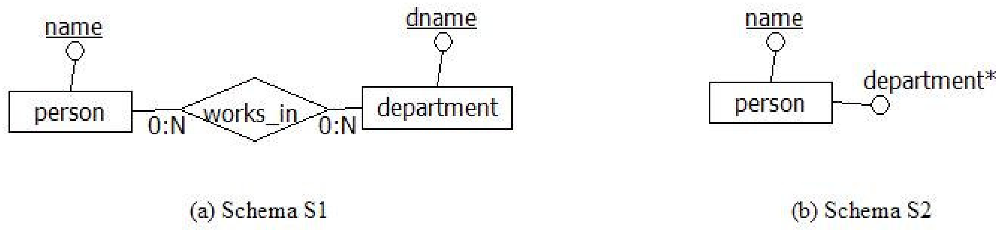


Figure 2-1: Schema S1 is equivalent to S2. A BAV pathway can transform one to the other.

For example (adjusted from [McB03]) consider **Figure 1**. If we want to map schema S1 to S2 we have to apply the following primitive transformations in the displayed order:

① `addEntity(<<department>>,Q1)`

② `addAttribute(<<department, dname, key>>, Q2)`

③ `addRelationship(<<works_in, person, department, 0:N, 0:N>>, Q3)`

④ `deleteAttribute(<<person, department, many>>, Q4)`

`Q1..Q4` are queries that select the appropriate values in the data, expressed in a language (**IQL**) that we will describe later.

As mentioned above the transformations are bi-directional. Therefore, if we wish to map S2 to S1 we just have to apply the same sequence in the reverse primitive transformations and in the reverse order. i.e.

❹ `addAttribute(<<person, department, many>>, Q4)`

❸ `deleteRelationship(<<works_in, person, department, 0:N, 0:N>>, Q3)`

❷ `deleteAttribute(<<department, dname, key>>, Q2)`

❶ `deleteEntity(<<department>>,Q1)`

Apart from the primitive transformations shown in this example (**add/delete**) AutoMed also supports the transformations: **rename, ident** and **extend/contract** For a full explanation of the usage and the semantics of the transformation please consult [BKL+04].

2.2.2 IQL

As mentioned above the queries `Q1..Q4` in the previous example are written in a language called **IQL**. IQL is described quite thoroughly in [Poul04]. It stands for **Intermediate Query Language** and it is used to query any schema (in any model) defined in AutoMed. IQL is in fact a typed functional language with sufficient power to express any query that can be expressed in relational algebra as demonstrated in [Poul04]. It can be used in AutoMed in two different ways:

Express queries within add, delete, extend and contract transformations

Express queries on schemas defined in the AutoMed repositories.

2.2.3 How AutoMed Works

We now summarise the main functions and core components of AutoMed.

This paragraph gives a simplified (probably oversimplified) view of the functions of AutoMed: AutoMed provides the facility to **wrap** any data source and express its schema using the **AutoMed constructs**. The schema definition is stored in AutoMed. Having defined the schema of the data source we can use transformations to create mappings to target schemas. Using those transformations an IQL query on the target schema can be transformed to an appropriate query on the source schema and vice versa. Transformations can also be used to materialise views from one schema to another. The following paragraphs elaborate on these concepts.

2.2.3.1 The AutoMed Repositories.

The AutoMed **meta-data repository** forms a platform for other components of the AutoMed software architecture to be implemented upon. When a data source is wrapped, a definition of the schema is added to the repository. A user can then use several tools created by AutoMed (or create her own tools using the API) in order to generate *transformations* from the source schema to any target schema.

The repository has two logical components:

2.2.3.1.1 MDR

The **Model Definition Repository (MDR)** defines how a data modelling language is represented as combination of nodes, edges and constraints in the **hypergraph data model (HDM)**. (HDM forms the low level definition of all modelling languages in AutoMed. More on HDM can be read in [PM98]). MDR is used to configure how a Modelling Language is defined in AutoMed.

2.2.3.1.2 STR

The **Schema Transformation Repository (STR)** defines schemas in terms of the data modelling concepts in MDR. It also stores all the information about the transformations defined on the schemas. **Figure 2.2** (taken from [BKL+04]) shows the overall architecture of AutoMed.

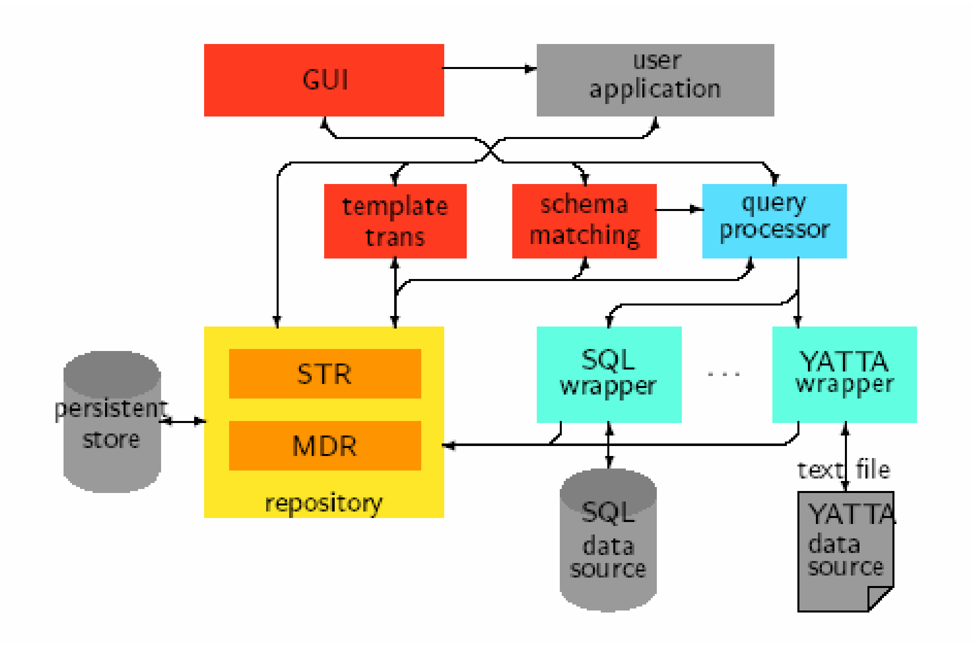


Figure 2-2 The AutoMed Architecture

2.2.3.2 AutoMed Networks

It is important to appreciate the way resulting schemas are stored in STR after transformations are applied. In a large data integration, there will be many intermediate schemas produced in the process of mapping one data source to another. If we stored all the intermediate schemas, it would probably make the whole approach very inefficient. However, AutoMed does not store the intermediate schemas.

What happens is that only source schemas are stored explicitly (or, as put by [BKL+04] have an **extensional** representation). Intermediate schemas are not stored but can be *derived* using the transformations from the extensional schemas. (They have an **intensional** representation). This is demonstrated in Figure 2-3

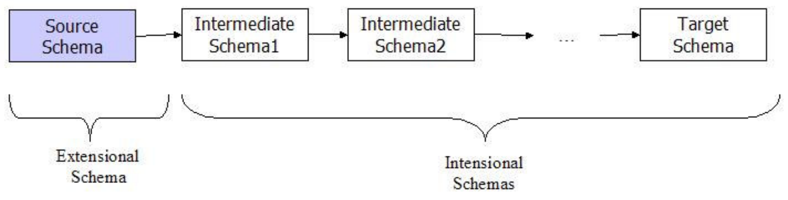


Figure 2-3: Only the source schema is explicitly stored in this AutoMed network

Two schemas that have the same structure can be connected together with an **ident** transformation (which declares that the schemas are homogeneous). A set of schemas connected together by transformations that contain at least one extensional schema is called an **AutoMed network**. If an AutoMed network does not contain ident is called a **subnet**. Note that it is possible to define an extensional (non-source) schema explicitly and form a subnet that originates from it in the same way as described in Figure 2-3.

AutoMed provides many more tools and facilities that we haven't covered in this document. All the relevant publications, technical reports, API documentation and the AutoMed source code itself can be found in [Automed].

2.3 The P2P Data Integration World Today

This section summarises the current theoretical frameworks, practices and technologies that relate to the subject. The organization of this section is as follows:

1) Talk about the main issues and challenges around the subject according to the current literature.
2) Discuss some current P2PDI systems with particular emphasis to the **Piazza** project.
3) Discuss some popular traditional P2P systems in order to focus on the networking aspect of the problem.
4) Talk about some P2P protocols focusing on the **CHORD** project which provides a framework for locating data in P2P systems.
5) Last we mention some relevant technologies that exist and were assessed during the implementation of this project.

2.3.1 The P2PDI Literature

It is generally accepted that a P2P approach to data integration can bring about invaluable benefits both for data integration and data distribution. The ultimate goal is a **decentralized** community of machines pooling their resources to benefit everyone in a way that provides **scalability, robustness, lack of need for administration** and even **anonymity and resistance to censorship**. [GHI+01]. A community whose power increases as more peers join without affecting the overall performance. Especially in the case of data integration, P2P means that there is no need for mediated schemas making peers able to share data without the need of a central authority [CGL+04], something that makes the very practice of data integration distributed.

The potential benefits however are only matched by the large challenges and practical difficulties around the subject. The lack of central administration means that it's difficult to predict or reason about the location and quality of the system resources or perform any kind of **global optimisation** to data transfers and service requests. These and many other issues resulted to all the existing and past P2P systems to be only partly successful in attaining their goals [SMK+01]. And all these without considering data semantics which adds extra problems like **query reformulations** [TH04], suitable data **materialisation** etc.

2.3.1.1 Overview of Main Issues

We now give an overview of the issues that several publications have pointed out and were considered for the purpose of this project.

In a paper titled "What Can Databases Do for Peer-to-Peer?" [GHI+01], the authors explore the limitations of current "**semantics free**" P2P systems and discuss how the database community can help the distributed systems community in designing P2P systems that incorporate data semantics. (this paper was written before P2PDI systems were fully implemented). The most interesting point about this paper is the discussion around the **data placement problem** for P2P systems.

They identify 4 key properties that a peer possesses in a P2P system:

1) **data origin**: provides original content to the system
2) **storage provider**: stores materialised views
3) **query evaluator**: uses portion of its CPU and resources to evaluate a set of queries
4) **query initiator**: clients to the system posing new queries

They argue that the overall cost of answering a query includes the **transfer cost** from the storage provider or data origin to the query evaluator, the **cost of resources** utilised at the query evaluator and other nodes, and the **cost to transfer the results** to the query initiator. They then define the data placement problem as *the problem to distribute data and work so the full query workload is answered with the lowest cost under the existing resource and bandwidth constraints*. They then identify **7 dimensions** of P2P systems that affect the data placement problem and argue that they should be compromised in order to achieve a balance between the ideal goals and approaching a solution to the data placement problem. The dimensions are the following:

1) **Scope of decision making**: The scale at which query processing and view materialisation decisions are made. A global strategy (taking into account the entire network) is more efficient but decisions are more expensive to make on the global scale.
2) **Extend of knowledge sharing**: How much should each peer know about each other peer and what is the structure of this information sharing. Possible ways of the structure are: a centralised directory, distributed knowledge (everyone knows about everyone) or a hierarchical directory service like LDAP or DNS.
3) **Heterogeneity of information sources**: Data may originate at a few authoritative sources, or alternatively, every participant might be allowed to contribute data to the community.
4) **Dynamicity of participants**: How often do peers join and leave the network?
5) **Data granularity**: Is data granularity atomic, hierarchical or value based?
6) Degrees of replication: Data items can be replicated at will, only sparingly or not at all.

7) **Freshness and update consistency**: The best way would be to use validation messages, but this would have large overheads and limit scalability. An alternative is having a time-out system.

The authors of this paper show that the data placement problem considering these dimensions is **NP-complete**. I believe that these points should be considered carefully when designing a P2P system like the one in this project.

[Len02] discusses the **basic principles** for P2P data integration systems and presents a general framework for such systems. The author also discusses possible methods for specifying the semantics for such systems. The paper argues that there are three central principles that should form the basis of P2PDI.

1) **Modularity:** how autonomous are the various peers in a P2P system with respect to the semantics. Since each peer is autonomously built and managed (this is apparently an implicit assumption that the author is making), it is argued that the concepts expressed in the peer should not radically change during interactions with other peers.
2) **Generality:** how free we are in placing connections (P2P mappings) between peers.
3) **Decidability:** are sound, complete and terminating query answering mechanisms available? If not, it becomes critical to establish quality assurance of the answers returned by the peer.

The paper then argues that a first **order logic (FOL)** approach is not sufficient to reason about these properties and defines a new framework based on **epistemic logic** and demonstrates that this approach is superior to the FOL approach.

[CGLR04] is a more mature version of [Len02] but also presents *a sound complete and terminating procedure* that result to a program that can return the **certain answers** to a query posed to a P2P system.

[CGL+04] elaborates on the concepts introduced by [Len02] and uses them to reason about a proposed infrastructure to implement P2PDI on **Grids**. It explains the benefits of Grid computing and introduces **grid-based Virtual Databases** as "*loosely-coupled database federations, which integrate heterogeneous sources, with the purpose of responding to business demands in a flexible manner*". It then tries to set up a general framework for P2P data integration in a Grid operating environment. The details are omitted here but we extract an interesting suggestion. The authors formalise their definition of a P2P system as constituted by a set of **data-peers** and **hyper-peers**. A data-peer is a system that exports data in terms of an exported schema. Hyper-peers instead do not have access to local data but are interconnected with both hyper-peers and data-peers from which they extract data. This model has similarities with the approach followed in this report where AutoMed peers (i.e. hyper-peers) are loosely associated with their data sources (i.e. data-peers) and can interact to extract data. More on this later.

An interesting point is raised in [BB04] about **trust relationships** between peers. If a peer is asked to answer a query, it might need to decide whether it should answer it alone or also consult other peers (and which peers?). Making decisions of this type will probably require to establish (i.e. define) *trust relationships between peers*. Another interesting point that was discussed in [TIM+03] is that a P2P system should *encourage users to join the network*. A user will not join if the commitments that are required in terms of resources and administration are higher than what the candidate peer is willing to provide. Therefore, we could argue that a good P2P system should allow each user to customise its commitment to the network according to its own needs. Finally, [TH04] is a very useful document that discusses **query reformulation** in P2P systems which provides some good insights on the subject including a discussion about how different paths to the same node can yield different query answers, something that looks like a potential pitfall when designing a query reformulation algorithm. We will discuss this issue in Chapter 4 when we analyse query distribution algorithms in our system.

2.3.2 Current P2PDI systems

Now we explore some existing P2PDI systems that are currently functioning, focusing on the ***Piazza Project*** *which is the best example of a pure P2P system.*

At present there are not many examples of working pure semantic-rich peer to peer systems that we can use as a suitable point reference to suit the purposes of this project. We came across two examples of such systems that are currently operational, Piazza [TIM+03] and CoDB [FLK+04]

2.3.2.1 Piazza

The best example of P2P data integration system that is currently operational is the **Piazza** Peer Data Management Project which was developed by the Department of Computer Science and Engineering of the University of Washington [GHI+01], [TH04], [TIM+03]. In the next paragraphs, we will explore the functionality of the Piazza project with references to its key components and issues that are being dealt with by its developers. There are many similarities between this project and Piazza and it's very important to evaluate the latter in order to recognise the attained knowledge and potential problems.

We now summarize the architectural design for the Piazza system. It models a **data origin** as an entity distinct from the peers in the system (though a peer can actually serve both roles) — Piazza can only guarantee availability of data while its origin is a member of the network, and only the origin may update its data. Each peer is aware of the peers it's **logically connected** to and can pass queries to them, even queries during query evaluation initiated from another peer, thus a query can transitively be answered by the entire network (or to be more exact, by all the logically reachable peers from its initiator). All peer nodes belong to **spheres of cooperation**, in which they pool their resources and make cooperative decisions. Each sphere of cooperation may in turn be **nested** within a successively larger sphere, with which it cooperates to a lesser extent. **Figure 4** summarizes the Piazza architecture. Nodes between peers indicate logical associations. With this design in mind, we take a look at how Piazza deals with some of the issues of P2P data integration.

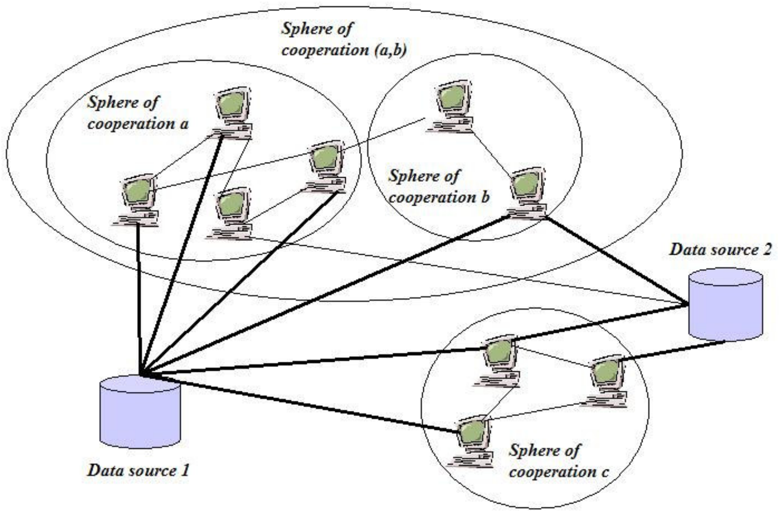


Figure 2-4: Illustration of he Piazza Architecture

2.3.2.1.1 Schema Mediation.

In contrast to a data integration environment, which has a tree based hierarchy with data sources schemas at the leaf nodes and one or more dedicated schemas as intermediate nodes piazza can support an arbitrary graph of interconnected schemas which are called **peer schemas**. Queries in the system will be posed over the relations from a specific peer schema. A peer schema represents the peer's view of the world that is unlikely to be the same at different peers. There are *two types of schema mappings* in Piazza. A mapping that relates two or more schemas is called a **peer description**, whereas a mapping that relates a stored schema to a peer schema is called a **storage description**. Peer descriptions define the correspondences between the view of the world at different peers. Storage descriptions on the other hand, map the data stored at a peer into the peer's view of the world. The set of mappings of Piazza defines its **semantic network** (or topology).

2.3.2.1.2 Querying and construction of mapping

Query reformulation is arguably the most important aspect of query processing in a PDMS. Piazza uses a **query reformulation** algorithm based on XML. The algorithm takes a set of peer mappings and storage descriptions and a query **Q** and creates a query **Q'** over the stored relations. The details of this algorithm are too lengthy and technical to be covered in this document, but you can refer to [TIM+03] and [HIST03] for more details.

[TIM+03] also describes the techniques used to form the mappings between peers. The construction of mappings includes a **schema matching** phase and a semi-automated technique to provide the precise mappings. The same paper (section 2) describes the format of the mappings which can be (over)simplified by saying that they define how queries on the source schema correspond to queries on the mapped schema. The relations are either **equality** or **subset**.

2.3.2.1.3 Searching

To avoid sending every query to the entire network (or at least to all the logically reachable nodes) in order to identify which peers have the data that is required, there is a need to have some sort of a distributed indexing that will allow the search algorithm to be more efficient. Piazza implements an indexing mechanism which at the current stage is not centralised and are based on the search engine paradigm, while the researchers at the project try move towards a more distributed implementation, possibly based on Distributed Hash Table techniques. ([TIM+03] has an interesting discussion about the difficulties in having a distributed indexing mechanism).

The Piazza index system allows peers to upload summaries of their data at different granularities. It allows every peer to specify data summaries at an appropriate granularity level. The peer also makes available to the index all its peer mappings, allowing the index engine to correlate attributes from different peers, thus supporting the simplest type of schema mappings. The system supports **exact** and **partial match queries**.

2.3.2.1.4 Future Work For Piazza

Piazza is a live research project. Main future work [TIM+03] will focus firstly on exploring richer and more expressive means of defining mappings and defining semantic information and secondly the problem of analysing the semantic network of a PDMS and developing efficient caching strategies to speed up query execution.

2.3.2.2 CoDB

CoDB [FLK+04] is a relatively new system which we only came across after the implementation of the project and therefore we haven not investigated it in depth. It uses

GLAV rules to define mappings between the peers in the network. The approach appears to be similar to piazza in terms of query distribution and termination.

The prototype system created was tested by comparing the scalability of updates under different topologies. The results were encouraging for some topologies (i.e. tree) and discouraging for others (i.e. clique).

2.3.3 Traditional peer-to-peer systems

Using P2P data management systems like Piazza as points of reference about this project is the most logical thing to do, because they match the requirements of the system we try to produce. However, Piazza was developed by a database research team and they focus more on the database related issues and less on the network details [GHI+01]. However the networking aspect is very important for the problem in hand. Therefore, it's essential to take a look at some traditional (semantics free) P2P systems and evaluate their architecture.

Many peer-to-peer systems have attracted the attention of the public and the research world from time to time. The majority of the systems, like **Napster, Gnutella, Kaaza, FreeNet** etc. allow users to share their storage resources and are mainly used for transferring media files while others like **distributed.net** [Distributed] let users share their processing resources. Earlier systems like Napster are centralized, having a central server responsible for locating documents in the network. Later systems like Gnutella have distributed this service amongst the peers themselves. We'll now analyse two of the most popular open P2P systems, Gnutella and FreeNet, and try to identify some of their strong and weak points.

2.3.3.1 FreeNet

FreeNet [CSWH01], [Cla99] is a data sharing P2P system that focuses on one key point: **Anonymity**. It claims to permit the publication, replication and retrieval of data, while protecting the anonymity of both authors and readers. Files are referred to in a location-independent manner, and are dynamically replicated in locations near requestors and deleted from locations where there is no interest.

The main design goals of the system are the following:

4) **Anonymity** for both producers and consumers of information
5) **Deniability** for stores of information
6) **Resistance** to attempts by third parties to deny access to information
7) Efficient **dynamic** storage and routing of information
8) **Decentralization** of all network functions

FreeNet enables users to **share unused disk space** and tries to provide a framework where the users view FreeNet as an *extension to their own hard drives*.

2.3.3.1.1 How it Works

The system forms a **virtual network** on the **application level**. Each node maintains its own local data-store which it makes available to the network or reading and writing, as well as a dynamic routing table containing addresses of other nodes and the keys that they are thought to hold. Files in FreeNet are identified by binary file keys obtained by applying a hash function. Requests for keys are passed along from node to node through a chain of proxy request in which each node makes a local decision about where to send the request next, in the style of IP rouging. The routing algorithm is described in section 3.1 of [CSWH01].

To retrieve a file, a user must first obtain or calculate its binary file key and then send a request message to his own node specifying that key value and hops-to-live value (similar to IP). When a node receives a request, it first checks its own store for data and returns it if found, together with a note saying it was the source of the data. If not found, it looks up the nearest key to the key requested and forwards the request to the corresponding node. If that request is ultimately successful and returns with the data, the node will pass the data back to the upstream requestor, cache the file in its own data-store, and create a new entry in its routing table associating the actual data source with the requested key. A subsequent request for the same key will be immediately satisfied from the local cache.

In FreeNet, no node is privileged over any other node, so no hierarchy or central point of failure exists. Joining the network is simply a matter of first discovering the address of one or more existing nodes and then starting to send messages. The system uses a cryptographic protocol to enhance security and privacy.

2.3.3.1.2 Evaluation

FreeNet is shown to have good **Fault-Tolerance** when random nodes fail (although it's possible to reduce this tolerance when selected peers are removed) and achieves reasonably efficient caching system avoiding excessive peer overloading and bottleneck creation. It also manages to provide a degree of anonymity, but the cost of doing that is its inability to guarantee retrieval of existing documents or from providing low bounds on retrieval costs. [SMK+01].

2.3.3.2 Gnutella

Gnutella [Gnutella], [CRB+03], [Ripe01] is an open, decentralised, P2P search protocol that is mainly used to find files. In addition, the term Gnutella designates the virtual network of Internet accessible host running Gnutella speaking application. In Gnutella network, each node maintains open TCP connections with at least one other node, thus creating virtual network of peers at the application level. Query and group maintenance messages are propagated using a flooding technique while query reply messages are back propagated.

Gnutella has a simple structure and it has the advantage over previous systems (like Napster) that it has a **distributed** file location service. However Gnutella failed because it is not scalable. The reasons for that are varying but the main ones are the overhead messages and the flooding of the network in order to monitor connectivity. This caused people starting to move towards other techniques like **distributed hashing**. Others however have suggested different modifications to the protocol (like [CRB+03], and [LS02]) that they claim should be able to increase scalability. One example of a modification is to design an agent that constantly monitors the network and intervenes by asking peers to drop or add links as necessary to keep network topology optimal. Other example is the suggestion to replace flooding with a smarter, less resource hungry routing and group communication mechanism [Ripe01]. [APHS02] proposed a Gnutella-compatible P2P system called Gridella that follows the so-called Peer-Grid approach.

2.3.4 Peer to Peer Protocols

In our discussion for the P2P systems in the previous section we covered their underlying protocols, which are good first examples for this section (In fact, technically speaking Gnutella is not a P2P system but rather a P2P protocol, however we used the convention of using this term to refer to Gnutella based systems). In this section we will focus our attention to a chord which is a protocol developed my MIT which and is used to locate documents in P2P applications. The term Chord is also used to describe the software developed in MIT to implement the protocol.

2.3.4.1 Chord

A fundamental problem that confronts P2P applications is to efficiently locate the node that stores a particular data item, a problem that chord tries to solve efficiently. Chord provides support for just one operation: *given a key, it maps the key onto a node*. **Data location** can easily be implemented on top of Chord by associating a key with each data item and *storing the key/data item pair at the node to which the key maps*. Chord tries to simplify the design of P2P systems and applications based on it by addressing the following problems:

1) Load balance
2) Decentralisation
3) Scalability
4) Availability
5) Flexible naming.

A software that will implement chord (and in particular "**the chord software**" implemented by MIT.) takes the form of a library to be linked with the applications that use it. The application interacts with chord in two main ways: First, chord provides a `lookup(key)` algorithm that yields the IP address of the node responsible for the key. Second, the chord software on each node notifies the application of changes in the set of keys that the node is responsible for. The application that uses chord is responsible for providing any desired authentication, caching, replication and user-friendly naming of data.

2.3.4.1.1 The base Chord Protocol

The chord protocol specifies how to find the locations of keys, how new nodes join the system, and how to recover from the failure of existing codes. (Here we'll use the chord software to illustrate some of the features of the protocol)

At its heart, chord provides a fast distributed computation of a hash function, mapping keys to nodes responsible for them. With high probability the hashing function used balances load (all nodes receive the same number of keys). Chord improves the scalability of consistent hashing by avoiding the requirement that every node know about every other node. A chord node needs only a small amount of routing information about other nodes. In an `N-Node` network, each node maintains information for only about $O(\log N)$ other nodes, and a lookup requires $O(\log N)$ messages. Chord must update the routing information when a node joins or leaves the network. A join or leave requires $O(\log^2 N)$ messages.

2.3.4.1.2 How it works

A hashing function assigns each node and key an **m-bit identifier** using a base hash function. A node's identifier is chosen by hashing the node's IP address while a key identifier is produced by hashing the key. The identifier length *m* must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible. Now we need an algorithm to assign the keys to nodes (i.e. distribute the lookup service)

We try to give an intuition of the algorithm used but not fully describe the entire algorithm (for this see [SMK+01] section 4). If we have an *m-bit identifier space*, it means that there are 2^m available slots for the networks peers to be mapped in. The slots are ordered **cyclically** (from zero to 2^m-1 and then zero again). The following is an example of a *3 bit* (i.e. 8 slots) identifier system that happens to (currently) have 4 nodes in the network which are arranged as shown in **figure 5**.

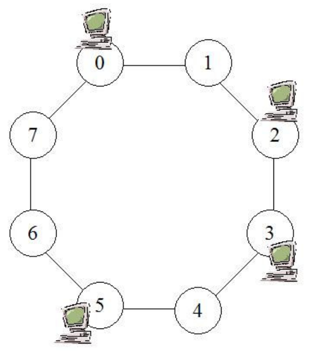


Figure 2-5: An illustration of a 3 bit chord network with 4 peers

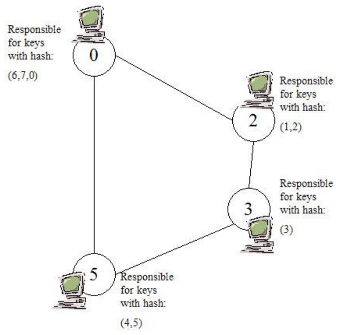

Responsible for keys with hash: (6,7,0)
0
Responsible for keys with hash: (1,2)
2
Responsible for keys with hash: (3)
3
5
Responsible for keys with hash: (4,5)

Figure 2-6: A more abstract view of the network in figure 5

Each node only needs to know about its neighbours (i.e. 0 only knows about 5 and 2. Now assume that there is a file with *key k* and we need to find the node responsible for it. First we have to perform the **hash function** in order to find the *id* of the key. Assume the id happens to be *6*. What we need to do is start from *slot 6* and move around the ring (clockwise) until we find a slot that maps to a host. In this case it's the peer at position **zero** which is responsible for the key 6. Similarly it's responsible for keys 7 and zero. We term that as i.e. `successor(6)=0`. Another example is `successor(3)=3`. Therefore, conceptually this instance of the network looks like **figure 2.6**

Now assume that *node 3* wants to locate a data that maps to *id 7*. It only has knowledge of *2* and *5* so let's say it asks *5* whether it holds data with *id 7*. It doesn't but it can pass the information about the location of node zero. So *node 3* can ask 0 that luckily happens to hold the key. (Note that this iterative method can be replaced by a recursive approach). Using this naïve algorithm however means that a typical search would require a **linear search** across the network, which is very inefficient making the system not scalable. The way chord overcomes this problem is by having each peer hold some extra information about the some other parts of the network in a clever way (a network with 2^m slots would require a host to have at most *m* entries to a **finger table** that holds information about other slots in the network). This greatly increases the search algorithm, making it from O`(N)` to O`(logN)` time.

2.3.5 Other Technologies

We now discuss some other relevant technologies that are good candidates to provide part of the infrastructure of the project.

2.3.5.1 Message Passing

The way the messages are passed around a peer to peer network is an important decision to be made. Typically network applications like P2P systems are multi-tiered such that the top layers are unaware of the network details, which should be handled by the lower tiers.

The most crucial choice in this context is the **message format**. Peers should understand what each other is saying when different messages are sent. One way to achieve this is by defining our application specific format and create a custom parser at each peer to interpret incoming messages. This has the advantage of being a very flexible approach but it would mean that something would need to be written from scratch and not take advantage of techniques already developed to interpret well know formats.

The dominant message format is **XML**. XML [McB03],[McB03b][DOM] is a format of encoding structured information which can be used to represent complex structures and it's well understood by the computing community. This means that using XML would allow us to exploit its expressiveness to communicate semantics-rich information, but also to use existing techniques and software to handle messages.

WebServices [GSB+01], [UDDI] are defined on top of XML and are used to request and receive services (for example information) over a network. The most popular format is **SOAP**, which is becoming a standard for communications between nodes. Using the WebServices instead of plain XML has the extra advantage that it is possible to make the system compatible with other systems if something like this is required. Furthermore there are many APIs (for example the `javax.xml.soap` package) that provide a higher-level abstraction making this easier to use.

2.3.5.2 Agent Technologies

It was not our initial intention to look into the Multi-agent systems literature for the purpose of this report until we realised that for the interactions between the peers should not be bounded by a fixed number of message types but should have reach and extensible semantics. Furthermore the peers should not be viewed as simply reactive entities that either react to request like servers or initiate request like clients. Instead they should be pro-active entities (or at least they should be developed under a model that allows them to evolve into proactive entities), take their own initiatives and can incorporate intelligence as required by the particular application.

In this aspect, a peer might be viewed as an agent in a multi-agent system. The urge to develop the system purely under the agent paradigm was not followed in this project, but we decided to "steal" some ideas from the multi-agent community especially for issues relating to agent interaction protocols and negotiations.

Many areas of multi-agent systems appear to be relevant to this project. For example, [Smith88] describes the Contract-Net protocol which tries to match task to task solvers efficiently (or in our model, match queries to query evaluators) in a distributed environment where each node can be both a client and a server at the same time (i.e. it's a network of peers). [Wool02, Ch. 6&7] describes how ideas from welfare economics and game theory can be used to implement negotiation mechanisms between nodes in a network. This can be used by a P2P network to negotiate when a query should be answered by a particular node.

The paper that was mostly influential for this project was [FFM+94]. It describes an agent communication language called *KQML* (Knowledge Query and Manipulation Language). The language was designed to be used for building large-scale knowledge bases which are sharable and reusable. KQML refers to both a message format and a message-handling protocol to support run-time knowledge sharing among agents. The language was developed taking into account research that studied the way humans interact.

Simplifying what the paper describes, the information enclosed in a message is split into 4 main parts:

1) performative, which describes the mode of communication
2) context, which describes what the message relates to
3) routing information
4) content, which is the information in the message

We elaborate on the message format in Chapter 4 where we adjust it and use it for inter-peer communication.

The same paper also describes how mediating agents can be used to facilitate communication. These are agents that might not have any data associated with them but have knowledge about the services that different agents can provide therefore link service requesters with service providers. The linking can be done in different modes like recommending a service provider, forwarding or brokering a query on behalf of the provider etc.

I found the ideas and the technology presented by this paper very relevant for the purposes of this project. The extensible and modular structure of the messages and the richness in semantics seems to be ideal to be used for peer communication. Furthermore, the ideas about mediators are very useful for a peer to peer network where peers can not only respond to requests regarding their own knowledge base but can also carry information regarding other appropriate peers. Last, this is a very well researched and mature area and it's likely to provide a high quality framework for inter-peer communication. Section 4.1 describes how KQML was adjusted and was used to provide a framework for the inter-peer communication in the network.

2.3.5.3 Directory service

A directory service could be used to locate nodes and documents in a P2P network. Very popular directory services like **DNS** and **LDAP** [Losh02] are built as **Application** layer protocols on the **TCP/IP** model and have the advantage that they are well understood and hierarchical (which means we don't introduce a potential bottleneck). However there is the issue of flexibility and whether their expressiveness is enough for the purpose of our application.

For systems implemented using WebServices, UDDI [UDDI], [GSB+01] is a good candidate for implementing a directory service. However this approach is not very popular at the current time. Furthermore it seems to be designed to deal with businesses and business services which they provide (i.e. `<businessEntity>` is a primitive in UDDI [UDDI]) which means there might be a problem with expressing non-business related concepts using this protocol.

For this project we used a simple directory service that provides the basic functionality but using a more sophisticated version as the ones described here should be considered for real applications.

2.4 Peer to Peer Data Integration using BAV rules

Having read this chapter up to this point you should already have some understanding of the concepts of schema integration, the BAV approach and the AutoMed project, P2P database integration, general P2P systems and relevant technologies. This section describes a proposal that intends to marry all the above concepts and provide a P2PDI system with a different model than the ones developed so far. This was the inspiration for this project.

The **BAV** and P2P approaches in data integration could be considered as two distinct methodologies for data integration. However, as argued in [MP03], BAV can provide a framework of implementing P2PDI and in particular the AutoMed API can be extended in such a way as to enable different AutoMed users to form a P2P network and use a variation of the P2PDI techniques to share data. This section gives a summary of this paper and of [McB04b], highlighting some important points, although it doesn't claim to be exhaustive.

As discussed previously an advantage of the BAV approach is that it supports the evolution of both global and local schemas, including the addition or removal of local schemas. Such evolutions can be expressed as extensions to the existing pathways. New view definitions can then be regenerated from the new pathways as needed for query processing. This feature makes BAV very well suited for the needs of P2P data integration where peers may join or leave the network at any time, or even may change their set of local schemas, published schemas or pathways between schemas.

[MP03] argues that the traditional P2P approach of defining pairwise mappings between peer schemas does not scale as the number of peers grows and suggests using super-peers to implement the mappings. Roughly, the technique argued by the paper is the following: Any peer can advertise a public schema (through a directory service like UDDI) and then any peer can provide a mapping from its local schema to any number of public schemas. Therefore, since BAV pathways are **bi-directional**, we can establish pathways from any local schema to any other local schema through the public schemas. This means that it's possible for one peer to either **bind** to another peer's data source or **query** another peer thought the global schemas. Additionally, the nature of the BAV approach allows us to indirectly deduce the logical network, which is formed in a traditional P2P data integration system such that a peer's query can traverse the entire network and not just the peers with which it shares a public schema. We demonstrate this with reference to **figure 2.7**.

Peer *A* can access *Sb* using the pathway `Sa->PS1->Sb`. This connects *A* with *B,* something which is expected since they both have a pathway to *PS1*. However, *A* can also access *Sc* using the pathway `A->PS1->Sb->PS2->Sc`, even though *A* and *C* do not have a common public schema. This **transitive** nature of the relations means that a query can crawl across the entire network. Selecting the reachable peers with the required information is something potentially very challenging. If we allow a query to traverse the network in this manner, we would need a mechanism of tracing the information source. An obvious reason for this is to avoid cycles.

This report describes an analysis and implementation of a P2PDI system based on the suggestions stated above.

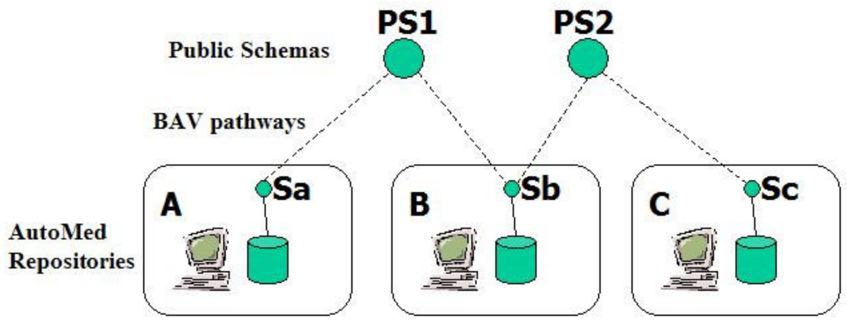


Figure 2-7 An example sturcture of the suggested system

3 Abstract Model

In this chapter we identify the main characteristics of the system we are developing and try to define an abstract model that describes the network and the interactions between the different nodes. This model will provide a solid formal framework for the rest of the report, will explain the rationale behind some of our design decisions and will form the basis for assessing the results of the implementation and talking about future work.

3.1 From the Real World to the Abstract World

Before describing the model we will explain informally the features we desire to model and thus place the things that follow in some context.

Our target system is comprised of set of peers $Peer$ which are linked by a set of pathways PW. Two peers are logically linked together if they both provide mappings to the same schema. For example if $pr1 \in Peer$ provides an AutoMed pathway $w1 \in PW$ to schema $s1 \in Schema$ and $pr2 \in Peer$ provides a pathway $w2 \in PW$ to the same schema, then a pathway from $pr1$ to $pr2$ can be inferred by appending $w1$ to $w2$, since the pathways in AutoMed are bidirectional. This is a very important assumption for the soundness of this approach. Note that the schemas that are common amongst the different peers are termed as *Super Peer Schemas* in [MP03] but for the purposes of this report we will call them *Public Schemas*.

From the previous paragraph we note perhaps the most fundamental difference between the BAV approach and traditional P2PDI models. In our approach, peers are not linked directly to each other. Instead, the mapping is always through public schemas and it is implicit. Therefore, the model described in [TIM+03], where the peers are connected by direct mappings (arcs) would not be valid in this case.

At this point, it is best to make clear what is meant by the notion of a peer and in particular what we mean when we say mapping a peer to a schema. A peer in our context is an active entity able to perform several actions and collaborate with other peers in order to achieve its goals and the goals of other participants in the network. By the term goal we mainly mean the best possible evaluation to posed queries. In our network, a peer is associated with a schema. We call this the *exported schema, (*$es \in Schema$)of the peer. Each peer has a unique exported schema and when we say a peer provides a mapping to a public schema, we mean it provides a pathway from its exported schema to the public schema.

An association between two peers can be inferred if they map on the same public schema. This association can be defined to be transitive because if there is a pathway from $pr1$ to $pr2$ and one from $pr2$ to $pr3$ we can deduce a pathway from $pr1$ to $pr3$ by merging the two pathways together. This notion of transitivity has to be somehow incorporated in the model.

So far we have described just one aspect of the story: the logical connections between the peers. The other side is the pathways from export schemas to data sources and query evaluations on different schemas. In other words, a peer can (or should) provide pathways from its exported $es \in Schema$ schema to a set of data $DS \subseteq Schema$. A query

$q \in Query$ expressed on the export schema can then be evaluated by transforming it to the data source schemas and then transforming the result back on the export schema. (All this is part of the existing AutoMed functionality.)

The model should therefore incorporate the notion of data source schemas and pathways from peer schemas to datasource schemas. We also need to model query evaluations on the datasource schemas but also on non-datasource schemas (i.e. export schemas, public schemas) with pathways to data sources.

Note that in the model that will follow we will assume that querying a datasource schema can give you results from some data source. It assumes that a data source schema is associated to a physical datasource from where it can extract data as appropriate.

3.2 The Model

In this section we will describe the model in detail. We will start by defining a simple intuitive model of the network by recognising its basic features of the . We identify some key properties of the network as well as several inefficiencies. We then refine the network by introducing several components to model things that the initial model could not. We use the new definitions to model the network as simply a set of peers and illustrate that the new definition is at least as expressive as our initial (more intuitive) approach.

Having defined the network, we attempt to model network evolution. We create a framework where operations that will affect the state of the network can be defined, and suggest an initial set of such operations. We describe the operational semantics of the initial set regarding their effect on the network state.

We then model inter-peer communication and define the full framework that can model how peers interact and how the operations that we defined previously are generated. Using this, we define a define how to model the Network evolution over time through peer interaction and peer operations.

3.2.1 Basic Network Definition

The network is defined as:

$$Network : \langle Peer, Schema, DSchema, PSchema, PW, peerAssoc, peerSource \rangle$$

where

$Peer \subseteq Peer^*$

is a set of peer nodes

$Schema \subseteq Schema^*$

is a set of Schemas

$DSchema \subseteq DSchema^* \subseteq Schema^*$

is set of Datasource Schemas

$PSchema \subseteq Schema$

is a Set of public schemas

$PW \subseteq PW^*$

is a set of pathways

$peerAssoc : (Peer \times PSchema) \to PW$

is a **partial** function that expresses the logical connections between the peers.

$peerSource : (Peer \times DSchema) \to PW$

is a **partial** function that expresses the connections of peers to data sources.

Note the convention we are using. *Peer** describes all the peers in the world while *Peer* refers to just the peers in the network. Same for the other structures.

Note also that the $peerAssoc$ and $peerSource$ functions are **partial** and therefore an absence of a pathway for example from a peer pr to a public schema ps is expressed by having $(pr, ps) \in (Peer \times PSchema)$ being **undefined** for $peerAssoc$. Similarly for $peerSource$ undefined pathways indicate a datasource is not connected to a peer.

We will start with these basic data structures at this point and define a very simple model and then identify its weaknesses which will help evolve the model to increase its expressiveness.

3.2.1.1 Example

We first illustrate how these structures are used to define the topology of the network. Consider the network illustrated in Figure 3.1

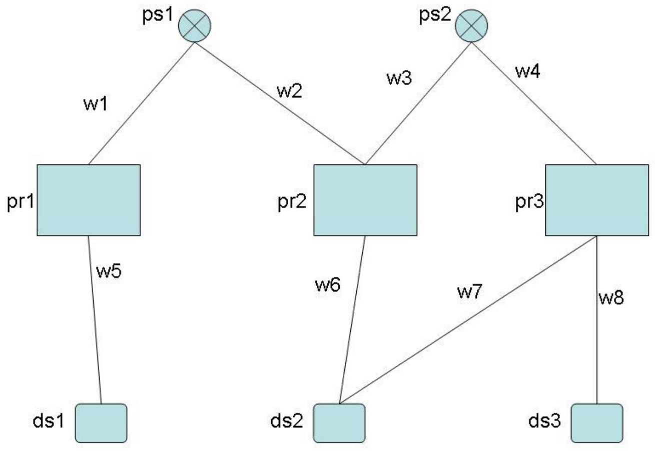


Figure 3-1 An example network with 2 public schemas 3 peers and 3 data source schemas

The network has 3 peers, *pr1, pr2* and *pr3*. There are two public schemas in the network, *ps1* and *ps2*. Peers *pr1* and *pr2* provide mappings to *ps1* while *ps2* and *ps3* provide mappings to *ps2*. There are also three datasource schemas, *ds1*, *ds2* and *ds3* and the peers provide mappings to them as illustrated by the arcs in the diagram. The labels in the arcs are the pathways from the peer (specifically to peer's export schema) to the target schema.

So the instance of this Network is characterized by the following:

$$Peer = \{pr1, pr2, pr3\}$$
$$Schema = \{ps1, ps2, ds1, ds2, ds3, ...\}\ (possibly\ more\ local\ ones)$$
$$DSchema = \{ds1, ds2, ds3\}$$
$$PSchema = \{ps1, ps2\}$$
$$paths = \{w1, ..., w8\}$$
$$peerAssoc = \{(pr1, ps1, w1), (pr2, ps1, w2), (pr2, ps2, w3), (pr3, ps2, w4)$$
$$peerSource = \{(pr1, ds1, w5), (pr2, ds2, w6), (pr3, ds2, w7), (pr3, ds3, w8)\}$$

3.2.2 Connectivity and Reachability

Now we need to express the fact that a peer is reachable from another peer. For instance, in our initial example, *pr2* is reachable from *pr1* through pathway $(w1 \mid w2)$ (where "|" means append). But also *pr3* is reachable from *pr1* via $(w1 \mid w2 \mid w3 \mid w4)$. To define this we first formalize the notion of appending pathways:

$$append : (Path \times Path) \rightarrow Path$$

where *append* can be abbreviated by the "|" symbol and be an infix operator. You should notice that not all paths can be appended to each other. For example $w1 \mid w2$ is only a valid path if the last schema of *w1* is the same as the first of *w2*. For now we ignore this fact and will come back to it later.

We now define some auxiliary relations in terms of the existing structures.

We say that a peer $pr \in Peer$ *implements* a public schema $ps \in PSchema$ iff *(pr,ps)* is defined in *peerAssoc* (i.e. when there is an arc from the peer to the schema), and we therefore define the relation *Imp* as:

$$Imp : (Peer \times PSchema)$$
$$Imp(pr, ps)\ iff\ peerAssoc(pr, ps) \neq undefined$$

We say that a peer $pr \in Peer$ is associated to datasource schema $ds \in DSchema$ iff (*pr,ds)* is defined in *peerSource* (i.e. they are linked with an arc), and we therefore define the relation *Src* as:

$$Src : (Peer \times DSchema)$$
$$Src(pr, ds)\ iff\ peerSource(pr, ds) \neq undefined$$

We can now define notions of *connectivity* and *reachability*.

We say that $pr1 \in Peer$ is *logically connected* to $pr2 \in Peer$ when they both implement the same public schema. We denote this as $pr1 \Leftrightarrow pr2$ and we define the $\Leftrightarrow$ relation as:

$$\Leftrightarrow : (Peer \times Peer)$$
$$(pr1 \Leftrightarrow pr2)\ iff\ \exists ps \in PSchema.\ Imp(pr1, ps) \wedge Imp(pr2, ps)$$

by definition $\Leftrightarrow$ is symmetric.

We say that $pr1 \in Peer$ is *logically reachable* from $pr2 \in Peer$ when they are *transitively* logically connected. We denote this as $pr1 \leftrightarrow pr2$ and we define the $\leftrightarrow$ relation as:

$$\leftrightarrow : (Peer \times Peer)$$
$$(pr1 \leftrightarrow pr2)\ iff\ \exists pr' \in Peer.\ (pr1 \Leftrightarrow pr') \wedge (pr' \leftrightarrow pr2)$$

by definition $\leftrightarrow$ is symmetric.

We say that $ds \in DSchema$ is reachable from $pr \in Peer$ when $pr' \in Peer$ is reachable from *pr* and is associated to *ds2*. We define this as $pr \multimap ds$ and we define the $\multimap$ relation as:

$\multimap : (Peer \times DSchema)$

$(pr \multimap ds)\, iff\ \exists pr' \in Peer.\ (pr \leftrightarrow pr') \wedge Src(pr', ds)$

Similarly, for public schemas we use the $\hookrightarrow$ operator

$\hookrightarrow : (Peer \times PSchema)$

$(pr \hookrightarrow ps)\, iff\ \exists pr' \in Peer.\ (pr \leftrightarrow pr') \wedge Imp(pr', ps)$

Referring back to our example, for the *Network* in this case, the new relations are:

$Imp = \{(pr1, ps1), (pr2, ps1), (pr2, ps2), (pr3, ps2)\}$

$Src = \{(pr1, ds1), (pr2, ds2), (pr3, ds2), (pr3, ds3)\}$

$\Leftrightarrow = \{(pr1, pr2), (pr2, pr1), (pr2, pr3), (pr3, pr2)\}$

$\leftrightarrow = \mathcal{P}(Peer \times Peer)$

$\multimap = (Peer \times DSchema)$

$\hookrightarrow = (Peer \times PSchema)$

3.2.3 Refinement of Network Model

The model so far allows us to express some useful properties about the Network, including topological information connectivity between peers and connectivity between peers and data sources.

However, there are many things that the constructs defined so far cannot express. We have not given any definition of *pathways* and the *quality* of the pathways. We treated all the pathways from one peer to another peer or data source schema as equal and have not given any definition of the *complexity* that is required to achieve a connection between two peers. Furthermore we haven't defined the notion of a *query* or *query evaluation* over a pathway. Last, we have only viewed the network from a *global point of view* and not from the point of view of a peer.

We now try to extend the model to be able to express all these properties. We will refine the existing structures and add more constructs in order to achieve this.

3.2.3.1 Pathways

A pathway (in AutoMed) is a list of *transformations* that link one schema to another. It is comprised of an *initial schema* a *final schema* and a set of pathways that link the two together. However, the pathway is bidirectional and therefore it is not correct to talk about initial and final schemas since the initial can become final when the pathway is seen from the opposite direction and vice versa. Therefore defining the pathway as:

$PW = (Schema \times Schema \times [Transformation])$

would be a wrong approach. Instead we should look for a structure without direction. This will make our definition slightly more complex.

We define the set of transformations as:

$Trans : (\{Schema, Schema\} \times Action)$

In other words, a transformation involves **exactly two** schemas and an action that describes how one schema is transformed to the other. The same action should be applicable in both directions and should have the reverse effect depending on which direction it's applied. For example if it is an *add* action when going from $s1 \in Schema$ to $s2 \in Schema$ it should be a *delete* action when going from *s2* to *s1*. We will be vague about the definition of *Action* but we believe that it is intuitive.

We initially define a pathway simply as a *finite* set of transformations. We will later refine this definition.

$$PW : \mathcal{P}(Trans)$$

We need to express that a pathway is valid *iff* the transformations form a *chain*. For example $w1 \in PW = \{\{s1, s2\}, \{s2, s3\}\}$ is a valid pathway since it forms the (undirectional) chain: $s1 - s2 - s3$. However, $w2 \in PW = \{\{s1, s2\}, \{s3, s4\}\}$ is not a valid pathway. Other examples of valid and invalid pathways are:

$w3 \in PW = \{\{s1, s2\}, \{s2, s3\}, \{s3, s1\}\}$ invalid, forms a cycle

$w4 \in PW = \{\{s1, s2\}, \{s3, s4\}, \{s2, s4\}\}$ valid

$w3 \in PW = \{\{s1, s2\}, \{s2, s3\}, \{s2, s4\}\}$ invalid

(For simplicity we are excluding the *Action* part of the transformation)

To formalise this *chain* property, we need to first define some auxiliary functions.

We first define the function *times* that counts how many times a schema is present in a pathway. For example, in example pathways $times(w1, s1) = 1$, $times(w1, s2) = 2$ etc.

$$times : PW \to Schema \to \mathbb{N}$$
$$times(w, s) = | X |$$
$$where$$
$$X = \{(S, a) \mid (S, a) \in PW \wedge s \in S\}$$

We define the function *extremal* that defines the schemas in a pathway that are on the edges of the chains. For example, in *w1* above, *s1* and *s3* are the *extremal* schemas.

$$extremal : PW \to \wp(Schema)$$
$$extremal(w) = \{s \mid times(w, s) = 1\}$$

We can now give a complete description of a pathway as a set of transformations with two extremal schemas and where each schema appears at most twice.

$$PW : \mathcal{P}(Trans)$$
$$where$$
$$\forall w \in PW :$$
$$\quad \neg \exists s \in Schema : times(w, s) \leq 2$$
$$\quad | extremal(w) | = 2$$

Note that the advantage of using this rather long definition of a pathway instead of using some simpler list structure is that this has no notion of direction anywhere. This means that *two pathways are the same if and only if they have the same structure*. One the other hand, if we used a list structure, two pathways might be semantically identical but have different structure. For example the forward list and the backward list of transformations might describe the same pathway.

3.2.3.1.1 Append Pathways

We now give a full definition of the *append* function we defined earlier on. It now needs to be defined as a partial function since obviously not all pathways can be *append*ed to each other.

$$append : PW \to PW \to PW$$

$$\forall w1, w2 \in PW :$$

$$append(w1, w2) = \begin{cases} w1 \cup w2 & if\ \ w1 \cup w2 \in PW \\ undefined & otherwise \end{cases}$$

3.2.3.1.2 Final Pathways

A final has one extremal schema which is a datasource.

$$FPW = \{w \mid w \in PW \wedge extremal(w) \cap DSchema \neq \varnothing\}$$

3.2.3.2 Quality

The quality of a pathway is a measure of how well it maps the initial schema to the target schema. For example a pathway that contains many *contract* and *extend* transformations is probably of lower quality that one that only contains *add* and *delete*. It is not within the scope of this project to try and find ways of measuring the quality; this is discussed in the Future Work section. We are assuming however that the quality can be derived looking *only* at the structure of the pathway.

We define the function quality as:

$$quality : PW \to Quality$$

where *Quality* is a (possibly infinite) totally ordered set.

Since quality is a comparable property we can use it to compare pathways. We define the partial order $\sqsubseteq$ as:

$$\sqsubseteq : PW \times PW$$

$$\forall w1, w2 \in PW : w1 \sqsubseteq w2\, iff$$

$$extremal(w1) = extremal(w2)\ \wedge$$

$$quality(w1) \leq quality(w2)$$

In other words, two pathways are only comparable if they link the same two schemas the one with the higher quality is considered to be the better one.

3.2.3.2.1 Property of Pathway Quality

The following property states that if you *append* $w1 \in PW$ to $w2 \in PW$ the resulting pathway will be of no better quality than w1 or w2. Based on our assumption that the quality of the pathway can be derived from its structure only, the property is intuitive because there is only a finite number of transformations and none of them *increases* the quality of the pathway.

$$\forall w, w' \in PW : w \subseteq w' \to quality(w') \leq quality(w)$$

3.2.3.3 Queries

We now attempt to model queries and query evaluation.

A query always relates to a particular schema. So we define the set of queries as:

$$Query : \langle Question \times Schema \rangle$$

Where *Question* can be thought of as a subset of the set of character lists (Strings) or it could be thought of something with structure. We are ignoring these details for now.

a query can be transformed to another query by a pathway. We define the partial function *transQ*, which illustrates how :

$$transQ : Query \rightarrow PW \rightarrow Query$$
$$\forall qn \in Question, s, s' \in Schema, w \in Pathway, \exists qn' \in Question :$$
$$\text{if } \{s, s'\} = extremal(w) \text{ then } transQ((qn, s), w) = (qn', s')$$

This definition does not state how the transformation occurs, just that it is always possible to transform a query expressed on one of the *extremal* schemas on a query expressed to the other *extremal* schema (although the resulting query might be void, and always return null results).

3.2.3.3.1 Query evaluation.

We define a query result in a similar way as a query:

$$Result : \langle Answer \times Schema \rangle$$

where *Answer* can be defined in a similar way as a Question.

A query expressed on a datasource schema can be immediately evaluated. For this we define the sets *SrcQuery* and *SrcResult* to denote queries and results relating to source schemas:

$$SrcQuery = ((q, ds) \in Query \mid ds \in DSource)$$
$$SrcResult = ((a, ds) \in Query \mid ds \in DSource)$$

We now express query evaluations on data source schemas defined by the function *evalSrc*.

$$evalSrc : SrcQuery \rightarrow SrcResult$$
$$\forall qn \in Question, ds \in DSchema, \exists a \in Answer : evalSrc(qn, ds) = (a, ds)$$

Note that *evalSrc* contains information about the contents of the datasource. Therefore if a datasource is updated this will be reflected by the change of the *evalSrc* results for queries on the datasource schema.

A *Result* can be transformed across a pathway in a similar way as the query.

$$transR : Result \rightarrow PW \rightarrow Result$$
$$\forall a \in Answer, s, s' \in Schema, w \in Pathway, \exists a' \in Answer :$$
$$\text{if } \{s, s'\} = extremal(w) \text{ then } transQ((qn, s), w) = (qn', s')$$

3.2.3.3.1.1 Evaluation Across Pathways

We now define query evaluation across a pathway using the function *eval*.

$$eval : Query \rightarrow FPW \rightarrow Result$$

$$\frac{\begin{array}{l} transQ(q, w) = q' \\ evalSrc(q') = r' \\ transR(r', w) = r \end{array}}{eval(q, w) = r}$$

Sometimes we need to express the evaluation of a query from different data sources, for example when the schema we are querying is connected to more than one DSchema. We formalize this with the function *eval**.

$$eval^*: Query \rightarrow \mathcal{P}(FPW) \rightarrow Result$$

$$\frac{\begin{array}{l} rs = \{eval(q, w_i) \mid w_i \in ws\} \\ r = merge(rs) \end{array}}{eval^*(q, ws) = r}$$

The function *merge* defines a merging algorithm that combines results from different sources. Such algorithms are already implemented in AutoMed.

3.2.3.4 Peers

So far have built a framework to analyze pathways and query evaluations in our network. We now proceed to formalize the notion of a Peer.

We characterize a peer by 3 attributes:

- its the *export schema*
- its *local knowledge*
- its *global knowledge*.

By local knowledge we mean the pathways that originate from its export schema, that is, its connections to public schemas and its connections to data sources. By global knowledge we refer to its knowledge concerning the rest of the network.

3.2.3.4.1 Local versus Global Knowledge

Global knowledge is in essence the peer's *cache*. Unlike its local knowledge, the cache is typically incomplete and possibly unsound. That is, it does not have a full view of the network (incomplete) and sometimes its knowledge of the network is either wrong or out of date (unsound). The exact structure of the cache, its components and the way it is being updated cannot be fully defined in a generic manner. It is typically application specific and depends on the specific characteristics of the network including the 7 dimensions identified in [GHI+01] and explained in Section 2.3.1.1 Here we will give an example of a simple cache and later suggest ways of using cache to optimize the network.

Irrespective of the structure of the cache though, its advantage is that as the network evolves the peers gain more global knowledge which optimizes the network. The disadvantage is that there needs to be an efficient mechanism to keep the cache updated properly as the network changes.

3.2.3.4.2 Peer Definition

More formally a peer is defined as:

$$Peer: \langle Schema \times Local \times Cache \rangle$$
$$where$$
$$Local: \langle publicPaths \times dataPaths \rangle$$
$$publicPaths: PSchema \rightarrow PW \quad (partial\ function)$$
$$dataPaths: DSchema \rightarrow FPW \quad (partial\ function)$$

We do not give any definition of *Cache* yet. We define the following obvious constraints on the structure of *Peer*.

$$\forall (s,(pp,dp),c) \in Peer, \forall ps \in PSchema, \forall ds \in DSchema:$$
$$pp(ps) = w \quad \rightarrow \quad extremal(w) = \{ps, s\}$$
$$dp(ds) = w \quad \rightarrow \quad extremal(w) = \{ds, s\}$$

In other the pathway that a schema map to contains the schema itself in one end of the path.

3.2.4 New Network Definition

A desirable feature of a P2P network is the ability to express the global network only in terms of the local knowledge of its peers. We now demonstrate how the network described in the beginning of this chapter can be inferred only by the structure of the peers.

$$Network : \langle Peer, Schema, DSchema, PSchema, PW, peerAssoc, peerSource \rangle$$
$$where$$
$$DSchema = \{ds \mid (s, pp, dp, c) \in Peer \wedge dp(ds) \neq undefined\}$$
$$PSchema = \{ps \mid (s, pp, dp, c) \in Peer \wedge pp(ps) \neq undefined\}$$
$$Schema = DSchema \cup PSchema \cup \{s \mid (s, pp, dp, c) \in Peer\}$$
$$PW1 = \{dp(ds) \mid (s, pp, dp, c) \in Peer \wedge ds \in DSchema\}$$
$$PW2 = \{pp(ps) \mid (s, pp, dp, c) \in Peer \wedge ps \in PSchema\}$$
$$PW = all\, the\, pathways\, derivable\, from\, PW1 \cup PW2$$
$$peerAssoc = \{(pr, ps, w) \mid pr = (ss, pp, dp, c) \wedge pp(ps) = w\}$$
$$peerSource = \{(pr, ds, w) \mid pr = (ss, pp, dp, c) \wedge dp(ds) = w\}$$

The network can therefore be defined only in terms of the set of peers that it contains since this is at least as expressive as demonstrated above.

$$Network : Peer$$

3.2.4.1 Directory Service

A pure P2P network should be defined in this way. However, for our model we assume the existence of a *directory service* in accordance to [MP03]. We could investigate ways in which the directory service can be distributed amongst the peers in the spirit of P2P networking (This is covered in the Future Work section).

The main purpose of the directory service is to facilitate the communication between the peers. The main scenario is when a peer wants to know all the other peers that it is logically connected to. Even if we assume if there is a mechanism of locating all the peers in the network (i.e. broadcasting) this would require the requesting peer to ask all the peers in the network (in the worst case) for the public schemas they are implementing (unless a sophisticated algorithm is used to locate such information, for example something along the lines of *Chord* discussed in Chapter 2.

3.2.4.1.1 Definition

We define the directory service *Dir* as a **special peer** in the network with **no local knowledge** (no pathways) . Its **global knowledge** (cache) a function *dir*. *dir* can match public schemas to the set of peers that implement them. Since all peers can contact the directory service all of them can access the same *dir* function. We will give a definition of *dir* and *Dir* in section 3.2.4.3

Semantically, *dir* can be though as equivalent to the *Imp* relation defined earlier with a very importance difference: *dir* can be *incomplete* and *unsound.* Incompleteness comes from the fact that it might not know about a path to a public schema because it was not informed by the peer. Unsoundness comes from the fact that it's possible to contain information about peers that have left the network or pathways to public schemas that have been revoked.

3.2.4.2 Schema and Peer Naming

At this point we should make the distinction between objects and names of objects in our model. In particular we want to differentiate between:

- Schemas and schema names
- Peers and peer names.

For that we define two sets, $Peer_{\&}*$ and $Schema_{\&}*$ to mean peer name and schema name respectively. We organise these sets in the same ways as their real object counterparts:

$$Peer_{\&} \subseteq Peer_{\&}*$$
$$DSchema_{\&} \subseteq Schema_{\&} \subseteq Schema_{\&}*$$
$$DSchema_{\&} \subseteq DSchema_{\&}* \subseteq Schema_{\&}*$$
$$PSchema_{\&} \subseteq Schema_{\&} \subseteq Schema_{\&}*$$
$$PSchema_{\&} \subseteq PSchema_{\&}* \subseteq Schema_{\&}*$$

We define two functions that are assumed to be *bijections*

$$PeerName : Peer_{\&}* \rightarrow Peer*$$
$$SchemaName : Schema_{\&}* \rightarrow Schema*$$

The fact that these functions are bijective is a very important assumption for the model. We are assuming that *schema names and peer names are unique in the model*. This assumption might seem restrictive especially with regards to schemas. In fact it is possible to relax the rule to only include public schemas however for simplicity of the model we assume this for the entire schema domain.

We will consistently use the lower case "&" to distinguish variables that refer to name from ones that refer to real objects.

3.2.4.3 Definition of Dir

We can now give a formal definition of *Dir* and *dir* using the new notations.

$$dir : PSchema_{\&}* \rightarrow \mathcal{P}(Peer_{\&}*)$$

$$Dir \in Peer = (schema, local, cache)$$
$$where$$
$$schema = undefined$$
$$local = (\emptyset, \emptyset)$$
$$cache = dir$$

Note how the range of *dir* is $\mathcal{P}(Peer_{\&}*)$ and not $\mathcal{P}(Peer_{\&})$. This is because it's possible to contain peers that are no longer in the Network. Similar argument goes for the domain of the function.

3.2.4.4 Basic Definition of Cache

We now revisit the *Cache* component of a *Peer*. In order to illustrate some of the points that follow we will give a "demo" definition of Cache. In this definition, *Cache* only contains a local version of *dir*. The rationale is that accessing the global directory is a costly operation and caching parts of its information can restrict its access.

$Cache: PSchema_{\&} \rightarrow \mathcal{P}(Peer_{\&}*)$

3.2.4.5 Final Network Definition

We can now can now define the Network as a set of peers and a directory service.

$Network: \langle Peer, Dir \rangle$

(remember that $Dir \in Peer$)

3.2.5 Network evolution

We will now give a framework to define how a Network can change its state.

The Network is evolved when any of the following occurs:

- A peer joins the network
- A peer leaves the network
- The local knowledge of a peer changes (*Local*,changes).
- The global knowledge of a peer changes (*Cache*, changes).
- The knowledge of the directory service changes (*dir* changes).
- A data source schema's information is updated changes (*evalSrc* changes)

3.2.5.1 Operations

We call any actions that evolve the network *Operations.* The set of operations is defined as follows:

$Operation: Peer* \times Update$

As it can be inferred from the structure, an operation is defined in the context of a peer. So an operation describes an update (defined later) and the peer that the update relates to. We use *Peer** instead of *Peer* to account for operations that involve peers that are not part of the network, for example when a new peer enters.

3.2.5.1.1 Initial Set of Operations

The set *Update* is meant to be extensible and we now define an initial proposed *Update* set and then the operational semantics of each operation.

The *Update* set:

$$Update =$$
$$enter() \mid exit() \mid public(\mathcal{P}(PSchema_{\&})) \mid remove(Peer_{\&}*) \mid$$
$$new\_path(PW) \mid impl(PSchema_{\&}, \mathcal{P}(Peer_{\&}*))$$

We now give an informal description of each *Operation* and then describe the operational semantics of each :

Operation	**Description**
(pr, enter())	Peer pr enters the network

(pr, exit())	Peer pr exits the network
(pr, public(pss$_{\&}$)	Peer pr is informed that the set pss$_{\&}$ are public schemas.
(pr, remove(pr1$_{\&}$))	Peer pr is informed that peer named pr1$_{\&}$ was removed from the network
(pr, new_path(w))	Peer pr is informed of a pathway w
(pr, impl(ps$_{\&}$, prs$_{\&}$))	Peer pr is informed that the set of peers prs implements public schema ps.

Note that it's possible in the above examples that pr=Dir. So if the directory is informed of a fact, all the peers will have access to it.

3.2.5.2 The Operational Semantics

We are assuming that the network is evolved one operation at a time. We are not concerned with message passing or the format of communications at this point. *Operations* are typically the consequence of a series of message exchanges and other peer actions. For example if a peer gets a pathway from another peer and stores it in its repository, the change in the network will be reflected by the new pathway in the network. To achieve this, possibly several messages were sent which do not concern us at this point. We are concerned with the *effects* of operations.

The entities of the network that change state with operations are:

- The set of Peers (peers can enter and leave the network)
- The elements of the set of peers(i.e. the individual peers) (Their local knowledge can change if they get new pathways etc. and their global knowledge can change if their cache is updated)
- *Dir* and *dir*. (Since the directory is just another peer sand *dir* is just another cache, this point is redundant since it is covered by the previous one. We are making it explicit here)
- The elements of DSchema. (This happens when the data that is contained in the datasource is updated. This will change the *evalSrc* function for queries expressed on the updated DSchemas.)

Note:

- Other only schemas in DSchema change.
- The set of schemas changes but the change is implicit when changing the state of the other dynamic entities (i.e. since the set of schemas in the network is implied by the local knowledge of peers, changing the state of the peer will be reflected in a change in the set of schemas)

3.2.5.2.1 Conventions Used

We use several conventions when defining the operational semantics and it is therefore important that you understand what is described in this section to be able to follow the notation.

- We use *pr,pr1,pr2…* to denote peers. A peer is defined as $pr = (es,\ pp,\ dp,\ cache)$ (see definition above). We use *pr.es* to denote the first element in the tuple, *pr.pp* for the second one ect. (i.e. similar to the OO programming notation). So when we say *pr.cache* we mean the cache component of the peer, when we say pr.pp we mean the *publicPath* component of the peer and so on.

- For a particular rule only the entities that change state are defined. Whatever is not mentioned it's assumed that it is unaffected by the operation. For instance, any peer that has not been mentioned in a rule has not changed state.
- If for a particular network there is no appropriate operational rule for an operation it's assumed that the event leaves the network unchanged. For example if there is an operational semantic rule that removes a peer from the network and it is applied to a network where the peer is not present, there will be no change in the state.
- We use the convention used by many formalisms that describe change of state: If K describes an entity before the operation, K' is the same entity after the operation. For instance, certain formalisms (for example the *Z Language*) that describe the effect on objects before and after a method invocation, use *myObject'* to denote *myObject* after the method has returned. We use a similar notation. For example, for a peer *pr* to add a pathway *w* to its repository, a *(pr,newPath(w))* operation should take place. We denote peer *pr* before the operation as simply *pr* and the **SAME** peer after the operation as *pr'*. So it is important to understand that in the rules that will follow, **any hyphened *x'* is the variable *x* after the operation** and that **all variables after the operation are hyphened, even the ones that do not change state.**

3.2.5.2.2 Format

The format of the operational semantics is:

$$Operation \times Network \rightarrow Network'$$

and it's denoted by:

$$Update, Network \triangleright_{Peer*} Network'$$

This is because *Operation* is defined as $(Peer \times Update)$ and we put the first element on the arrow $\triangleright$ since this makes the notation more natural. In other words, saying: $enter(), (Peer, Dir) \triangleright_{pr} (Peer', Dir')$ is equivalent to saying:

$(pr, enter()), (Peer, Dir) \triangleright (Peer', Dir')$.

3.2.5.2.3 The Rules

The derivation system for operational semantics that we are defining follows the notation which is widely used for defining operational semantics for programming languages for example as used in [NN92] . The shape of the rules is as follows:

$$\frac{Conditions}{Operations, State\ Before \triangleright State\ After}$$

The conditions defined above the line define constraints on the state before and state after. The way to interpret such rule is: If the operations are applied on the *state before*, it will be evolved to the *state after* if and only if the conditions hold. **Otherwise the rule does not apply.** We will give intuitive interpretation for each rule to make things clear. Note that some authors prefer to put the Conditions at the botton and the operation at the top.

The way you should read the conditions is by going from top to bottom and for each line you add an existential quantifier for variables that have not been defined by the rule or the lines above. For example the new_public_path rule that follows should be read as the following. Notice that we are not (mentally) quantifying variables (like pr or w) that are met at the result of the rule or the hyphened version of quantified variables (like pr')

$$\frac{\begin{array}{l}\exists \mathbf{ps} : extremal(w) = \{pr.es, ps) \\ ps \in PSchema \\ (pr.pp(ps) = undefined\ OR \\ \ quality(pr.pp(ps)) < quality(quality(w))) \\ pr'.pp(ps) = (w)\end{array}}{new\_path(w), (Peer, Dir) \rhd_{pr} (Peer', Dir')} \text{(new\_public\_path)}$$

We will be using the operator $\uplus$ defined as:

$$\uplus : (\mathcal{P}(Set) \cup undefined) \to \mathcal{P}(Set) \to \mathcal{P}(Set)$$

$$Y \uplus S = \begin{cases} S\ if\ Y = undefined \\ Y \cup S\ otherwise \end{cases}$$

Note that the rules that follow are just a suggestion. One can modify them to suit the particular needs of the application, especially concerning the update of the cache and the update of local knowledge of a peer.

$$\frac{Peer' = Peer \cup \{pr'\}}{enter(), (Peer, Dir) \rhd_{pr} (Peer', Dir')} \text{(enter)}$$

Intuition: If an enter() update for peer pr is applied on a network, it will evolve to the same network where the peer (its evolved state pr') is added to the set of peers.

$$\frac{Peer' = Peer \setminus \{pr'\}}{exit(), (Peer, Dir) \rhd_{pr} (Peer', Dir')} \text{(exit)}$$

Intuition: If an exit() update for peer pr is applied on a network, it will evolve to the same network where the peer is removed.

$$\frac{\forall pr_{\&} \in pss_{\&} : pr'.cache(ps) = pr.cache(ps_{\&}) \uplus \varnothing}{public(pss_{\&}), (Peer, Dir) \rhd_{pr} (Peer', Dir')} \text{(public)}$$

Intuition: If a peer pr is informed that the set $pss_{\&}$ are public schema names, then in the evolved network the peer pr (i.e. pr') will add those schemas in its cache.

$$\frac{\forall ps_{\&} \in PSchema_{\&}* : pr1_{\&} \notin pr'.cache(ps_{\&})}{remove(pr1_{\&}), (Peer, Dir) \rhd_{pr} (Peer', Dir')} \text{(remove)}$$

Intuition: If a peer pr is informed that the peer with name $pr1_{\&}$ was removed from the network, then the evolved network will be such that $pr1_{\&}$ does not appear anywhere in the cache of pr.

$$\frac{pr'.cache(ps_{\&}) = pr.cache(ps_{\&}) \uplus prs_{\&}}{impl(ps_{\&}, prs_{\&}), (Peer, Dir) \rhd_{pr} (Peer', Dir')} \text{ (impl)}$$

Intuition: If a peer pr is informed that the schema with name $ps_{\&}$ is implemented by the peers with names $prs_{\&}$ then the cache of pr will map the schema name to the peer names in the evolved network.

$$\frac{\begin{array}{l} w \in FPW \\ ps \in PSchema \\ pr.pp(ps) = w1 \\ extremal(w) = \{ps, ds\} \\ (pr.dp(ds) = undefined\ OR \\ \ quality(pr.dp(ds)) < quality(w1 \mid w)) \\ pr'.dp(ds) = (w1 \mid w) \end{array}}{new\_path(w), (Peer, Dir) \rhd_{pr} (Peer', Dir')} \ (new\_source\_path)$$

Intuition: If a new pathway w is added in the local knowledge of peer pr, and the pathway leads from a public schema ps that pr implements to some data source schema ds then pr derives a pathway that leads to ds (w1|w) and adds it to its local knowledge. The addition is only implemented if no other pathway exists or if the existing pathway is of lower quality, in which case it is replaced.

$$\frac{\begin{array}{l} extremal(w) = \{pr.es, ps) \\ ps \in PSchema \\ (pr.pp(ps) = undefined\ OR \\ \ quality(pr.pp(ps)) < quality(quality(w))) \\ pr'.pp(ps) = (w) \end{array}}{new\_path(w), (Peer, Dir) \rhd_{pr} (Peer', Dir')} \text{ (new\_public\_path)}$$

Intuition: In a similar way as the previous rule, but for cases where the pathway links the peer's export schema to some public schema and the public schema then the peer implements the public schema using the new pathway unless there is an existing pathway to the schema that is of better quality.

$$\frac{\begin{array}{l} extremal(w) = \{ps1, ps2\} \\ \{ps1, ps2\} \subseteq PSchema \\ pr.pp(ps1) = w1 \\ (pr.pp(ps2) = undefined\ OR \\ quality(pr.pp(s2)) < quality(w \mid w1)) \\ pr'.pp(ps2) = (w \mid w1) \end{array}}{new\_path(w), (Peer, Dir) \rhd_{pr} (Peer', Dir')} \quad \text{(new\_link\_path)}$$

Intuition: In a similar way as new_source_path, if the new pathway links two public schemas and the peer has a pathway to one of them then it now has a pathway to the second one. If it already has a pathway to the existing one it uses the one with the better quality.

3.2.5.2.4 Set Semantics

We can define the effect of a **set** of operations on a network. We are using the two rules are the following:

$$\frac{}{\varnothing, Network \rhd\rhd Network} \quad (empty)$$

$$\frac{\begin{array}{l} op = (pr, update) = select(ops) \\ update, Network \rhd_{pr} Network' \\ ops \setminus \{op\}, Network' \rhd\rhd Network'' \end{array}}{ops, Network \rhd\rhd Network''} \quad (non-empty)$$

The *select* function used above deterministically selects an appropriate operation from a set of operations. We use this function to ensure that the operational semantics are deterministic.

3.2.6 Framework for Peer Interaction

3.2.6.1 Peer Communication

We now give a formal framework of the communication between the peers.

In this framework, a message is encapsulated within a mail which contains the sender's *"address"*(i.e. the name of the peer)

$$Mail : Peer_{\&} \times Message$$

MailBox is a partial function that maps mail receivers (peers) to their mail.

$$MailBox : Peer_{\&} \to Mail$$

i.e. if $MailBox(pr_{\&}) = \{(pr1_{\&}, msg1), (pr2_{\&}, msg2)\}$ it means *pr* received msg1 from pr1 and msg2 from pr2.

we do not give a definition of a Message at this stage. Messages are application specific and an example of a message set along with its semantics will be described in Chapter 4. We now fully describe the evolution of the network and the assumptions of our communication model.

3.2.6.2 NetworkState

A *NetworkState* is characterized by a Network, a list of pending *Operations* and a set of pending *Mail*.

$$NetworkState : \langle Network \times \mathcal{P}(Operation) \times MailBox \rangle$$

3.2.6.3 NetworkHistory

A NetworkHistory is a function from natural numbers to *NetworkState*.

$$NetworkHistory : \mathbb{N} \rightarrow NetworkState$$

we denote *NetworkHistory(n)* as $NetworkState^n$ and it denotes the state of the network at time n. We say "NetworkState at time n" or "network state at round n" to mean $NetworkState^n$

we denote $NetworkState^n$ as $(Network^n, Operation^n, MailBox^n)$

we denote $Network^n$ as $(Peer^n, Dir^n)$

etc. In other words when we meet a superscript *n* on the top of any variable we refer to the state of the variable tat time *n*.

Note that history domain is natural numbers which means that the Network evolves in discrete steps. This is a very important assumption of our model and it is discussed later.

3.2.6.4 Defining Peer Behaviour

At this point there is no mechanism for a peer to respond to messages or have any processing state. We would like the peer to be able to send messages to other peers, handle messages sent to it and have a framework on which we can develop various algorithms that define peer behaviour. We therefore need extend the notion of network state to add this new functionality. We wish to distinguish between a peer and the *processing state* of the peer.

3.2.6.4.1 Processing State of Peers

A processing state is meant to be associated with the peer's *"intelligence"* and should (amongst other things) be keeping track of the messages that have been sent the messages received and act as *memory* for any information that has to be remembered to implement the algorithm that defines its behaviour. We chose not to include the processing state as part of the Peer and instead decouple the two concepts mainly because the peer's processing state does not affect the structure of the network.

Therefore we include it in *NetworkState*.

3.2.6.4.2 Refined Definition of NetworkState

We extend the definition of *NetworkState* as follows:

$$NetworkState = (Network \times \mathcal{P}(Operation) \times MailBox \times peerState \times msgHanlder)$$
$$where$$
$$peerState : Peer \rightarrow State$$
$$msgHandler : Peer^* \rightarrow (\mathcal{P}(Mail) \times State) \rightarrow (\mathcal{P}(Update) \times Mailbox \times State)$$

peerState maps the peers in the network to their current processing state as explained above. State is application specific.

The function msgHandler defines the logic of each peer. Each peer, for a set of received mails and a peer state, generates:

- the mail that the peer should send at the next round
- the updates that relate to the peer which will be used to evolve the network
- its next state.

3.2.7 Evolution of NetworkState

We now define how each of the components of *NetworkState* evolves. Remember that the components of *NetworkState*^{n} are:

$$(Network^n, Operation^n, MailBox^n, peerState^n, messageHandler^n)$$

3.2.7.1 Evolution of Components

$$let\ (ud_{pr}^n, mb_{pr}^n, st_{pr}^n) = messageHandler(pr)(MailBox^n(pr_{\&}), peerState(pr)^n)$$

$$Operation^n = \left(\bigcup_{pr_i \in Peer^{n-1}} \{(pr_i, update) \mid update \in ud_{pr_i}^n)\} \right) \cup ExtOp^n$$

intuition: The Operations at round n are the set of updates generated by the MessageHanlder function of each peer in the network (note that $(pr_i, update)$ is an Operation) in the same round plus the external operations.

$$\forall pr_{i\&} \in Peer^* : MailBox^n(pr_{i\&}) = \begin{cases} \varnothing & if\ n = 0 \\ \bigcup\limits_{pr_j \in Peer^{n-1}} mb_{pr_j}^{n-1}(pr_{j\&}) & otherwise \end{cases}$$

intuition: The messages received at round n are the ones sent at the previous round. Note that according to the definition any mail sent to peers that left the network get lost.

$$\forall pr_i \in Peer^n\ peerState^n(pr_i) = \begin{cases} initState & if\ (n = 0\ OR\ pr_i \notin Peer^{n-1}) \\ st_{pr_i}^{n-1} & otherwise \end{cases}$$

intuition: The processing state of a peer at round n is the one returned by the messageHandler function at round n-1. If a peer was not part of the network at round n, its processing state is reset.

$$messageHandler^n = messageHandler^0 = messageHandler$$

intuition: The message handler function does not change with time. The function defines the protocol that the peers in the network use and since the protocol should not change with time, neither should this function.

We have introduced the notion of *ExtOp* to refer to actions that involve peers that are not part of the network. For example when a peer enters the network for the first time at round n, there has to first be an *enter()* operation.

initState is a pre-defined initial state for each peer. This is application specific.

3.2.7.1.1 Definiton of NetworkState Evolution

$Network^{n+1}$ depends only on $Network^{n}$ and $Operation^{n}$

More formally:

$$Operation^{n}, Network^{n} \rhd\rhd Network^{n+1}$$

In other words the Operations generated at round *n* are define the next state of the network. We could define $Network^0$ as:

$$Network^{0} = \langle Peer^{0}, Dir^{0} \rangle$$
$$Peers^{0} = \{Dir\}$$
$$Dir^{0} = \langle undefined, (\varnothing, \varnothing), \varnothing \rangle$$

3.3 Overview of Model

We now note some important points that are implied by the model but might not been stated explicitly:

1) Computation proceeds in rounds.
2) Each peer $pr \in Peer$ has a message handling function $messageHandler(pr)$ and a State $peerState(pr)$ associated with it. Those determine the peer's functionality.
3) The peer's functionality and the peer itself are modelled as two different things.
4) The Network state (the peers in the network) are only updated by Operations.
5) The peer's functionality can use the peer's local and global knowledge to determine the required actions but it cannot change the peer's state. It can indirectly change the state by generating Operations that will be implemented when updating the network at the next round.
6) Only peers that are part of $Network^{n}$ can generate messages in $MailBox^{n+1}$.
7) Any mail at round n can be sent to anyone, even to peers that are not part of the network at round n, but will only be received by peers that are part of the network at round n+1.
8) A message sent at round n is received at round n+1.
9) $Operation^{n}$ is updated in round n and it's used to evolve $Network^{n}$ to $Network^{n+1}$.

10) Any peer can update Operation^n (even ones outside Network^n Using ExtOp)
11) Only peers that are part of the network can update their State. When (re)entering the network the state is initialised
12) Peers can only update *Dir* (directory) or get information from it by sending mail to it.
13) Dir is just another peer in the network.
14) It is assumed that any local actions can occur within a single round. So for example reading the local cache can be done in the same round but requesting the same information from another peer can take several rounds for the communication to occur.
15) There is no assumption that a received request should be handled in a single round.

3.3.1 Simple Metaphor

You can think of the model using the following simple metaphor.

NetworkState is a room that contains a number of robots (peers). There is also a number of robots that are outside the room (peers outside the network). Robots cannot move or talk to each other but contain information for various issues which can be updated. In the room there is also an administrator. (What that updates the state of the network).

With each robot, there is one associated human agent (peer functionality). Agents can be very smart individuals but they have amnesia, they cannot remember anything. Luckily they have one notebook each (State) on which they can take notes.

Inside the room we find the agents that belong to the robots of the room. Agents of other peers are outside. The room is also equipped with, an array of pigeon holes (MailBox) one for each agent (even for agents outside the room) and a request bag (Operations).

When the administrator hits the buzzer, each agent in the room goes to her pigeon hole, gets her **mail**, takes her **robot** and **notebook** (which are the three arguments of the *messageHandler* function) and goes to a corner and starts thinking about what needs to be done. She makes decisions by reading her mail, her notebook and asking the robot several questions (the messageHandler function can access the local and global knowledge of the peer). She does not know anything else about the network, just that there is a robot called *Dir* (Directory service) to which she can ask information about the network (All peers know the address of the directory service). She can write mail to other agents, and update her notebook as she pleases (messageHanlder generates new State and Mailbox for peer). Unfortunately she cannot update the robot in any way but she can put requests to the request bag about changes that need to be made to her robot, including leaving the network (messageHanlder generates operations). Agents of robots outside can put requests in the bag if they want to join the network (generate ExtOp with *enter()* operations) but cannot send any mail.

At some point (when everyone is finished) the administrator hits the buzzer again and everyone stops. Agents go back to their seats and the administrator first empties the pigeon holes (*undelivered messages are lost)* and then places the mail written by the agents to the correct holes (at the next round each peer gets the messages that were addressed to it). Then she takes the request bag and updates the robots according to the requests (The network evolves according to the Operations generated at this round). Some new robots might enter the network and some might leave. The agents of the robots enter and leave the network accordingly.

The administrator hits the buzzer and the next round proceeds in the same way (next round)

3.4 Modelling Network Features

It is not very clear how to model several characteristics of real networks in this model. Here we give some suggestions of how to model some of these things.

3.4.1 Modelling failure

Failure can be generally modelled by introducing non-determinism in the *messageHanler* function. For example failure of a message to arrive (i.e. communication failure) can be modelled by having the possibility in the function to fail to consider a particular message.

i.e.`foreach(mail in Mail): consider(mail) OR ignore()`

Process failure can be modelled in a similar way. For example by non deterministically adding *exit()* updates in the function.

i.e. `Update = Update OR Update = {exit()}`

3.4.2 Modelling Timeouts

timeout can be implemented by having the State of the peer keeping count of the number of rounds it waits to receive a message and then assuming timeout after some rounds.

3.4.3 Modelling Long Messages

One of the obvious weaknesses of the model is its synchrony. The fact that the computation proceeds in rounds and that any message sent will be received in the next round means that we cannot model the delay in the reception of longer messages compared to smaller ones.

One way to overcome this problem is by having larger messages being delivered in more than one rounds. A round can be very fine grained, such that only the shortest messages can be delivered in a single round. To achieve this we only need to make a minor adjustment in the model. When sending a message one will need to define the number of rounds the message will take to be delivered (depending on its size). The definition of the mailbox will need to be extended to:

$$MailBox : \mathbb{N} \to Peer_{\&} \to Mail$$

The first argument is the intended delay of the message. The operational semantics of the network evolution need to be also extended so that at each round only $MailBox(0)$ is delivered and the rest is carried to the next round with the first argument decremented by one.

It is also possible to implement this functionality without changing the model, by having the sender delay the sending of the message in the messageHandler function. In this case it should store the message in the State along with an indication of when it should be sent.

3.4.4 Modelling Long Updates

Following a similar argument as above, some updates typically take much longer than others, while our model assumes that all updates can take place in a single round. This can be tackled in a similar way as long messages. That is, by adding a delay on the Operation part of the *NetworState*. Again, similar to the above case a long update can be implemented without changing the model, just by accounting for this fact in the messageHandler function.

3.4.5 Modelling Network Partition

A network partition can be modelled by modelling communication failures for any message sent from one part of the network to the other. The two parts of the network will consequently have independent histories, although the evolution of the two parts of the network will be synchronized.

3.4.6 Modelling Changes in Data Sources

An update to a data source is something that changes the state of the network since the *eval* function is different after the update for the datasource schemas that correspond to the data source. An update should therefore be expressed as an Operation that whose effect is to change the *evaSrcl* function for the modified schema.

3.5 Discussion about Synchrony

How does the assumption of synchronous evolution of the system affect the semantics of the network?

The synchrony assumption might seem oversimplified and potentially unsafe as a basis of a formal system. The assumption can be abused by people using the model and use it to achieve synchronization between peers and implement algorithms that assume perfectly synchronized clocks amongst the peers. However the nature of the applications that will be developed in this model do not require synchronization. For example no algorithm will require two peers to synchronise so that they both do a certain action after exactly *n* rounds.

It is safe for an algorithm to assume that for a particular peer the evolution proceeds in round but we discourage the assumption that all the peers evolve at the same time. Note that, as it can be inferred from the discussion regarding modelling long messages and long updates, it is possible that we can model the rounds as being so frequent that at most one peer is evolving at any round. In any case, we recognise the fact that the assumption, although it helps make the system easier to formalize, it might have some side effects. We believe however that for any practical applications of the model this assumption is not a problem as long as it is not used to achieve synchronisation between two peers.

In the future work section we discuss how the model can be extended so that the evolution of the system is not synchronized, using dense time instead of discrete time (i.e. real numbers instead of natural numbers) which will solve this problem completely).

3.6 Using the Model

The model is a formal framework that can be used as a basis to achieve several things.

3.6.1 Describing a protocol

This model is a formal framework on which several protocols can be described. The parts of the model that need to be defined in order to define a protocol are the following:

- Define the structure of messages
- Define the messageHandler function and the structure of peer State.

3.6.1.1.1 Extending features of model

You can extend the suggested features of the model to allow more powerful algorithms:

- Extend the structure of the cache
- Extend the set of Updates and their operational semantics
- Extend the structure of *Dir* so that it contains more useful information.

Chapter 4 illustrates how such a protocol is defined.

3.6.2 Analysing a protocol

The structures defined in the model allows to talk about things like connectivity of the peers, reachability of the peers, quality of a pathway and hence quality of peer connection and quality of the pathway in general. We could analyse the network in different topologies. For example we could compare how a protocol would work in a network with a small number of public schemas compared to the number of schemas compared to one with a large number of public schemas.

We can also talk about the global knowledge of peers and how the protocol helps the peers learn about the global structure in an efficient way. We identified the peer cache as the component that makes the network become more efficient while it evolves. We could examine if the evolution of the network increases or decreases the quality of the network under various assumptions.

We can also talk about communication complexity, the number of rounds required for some request to be answered under different situations. We could identify overloaded nodes by looking at the Mailbox function (if there are peers that get more messages than others).

Last we can analyze the scalability of the network in a formal way. How an increase in the number of peers increases the load on each peer and the communication complexity and how much it increases or decreases the communication.

3.6.3 Simulating a protocol

I strongly believe that this model can be a basis to build simulation programs that simulate different protocols. Having recognised the things that need to be defined to implement different algorithms it is possible to build a system where a user can define those parameters and then the simulation will run using the operational semantics defined giving statistics and useful information about scalability etc. A simulation program can simulate networks of millions of peers, something allowing for testing things that are impossible in the real world. More discussion about this is covered on the Future Work section.

3.6.4 Implementing a System

This model can be used to guide the implementation of a real system and then used again to evaluate parts of the system. This was done for this project as described in the chapters that follow.

4 Protocol Definition

In this chapter we use the model described in Chapter 3 to describe a protocol for the peer interactions. We start by suggesting a message format that will be the used by the protocol as an a communication language between peers. The format is based on a language used by multi-agent systems and we illustrate its expressive power and flexibility. We then use the framework developed by the abstract model of the Chapter 3 to formally define a basic protocol that describes the interactions and behaviour of peers for a set of communication types. We discuss some properties of the protocol and then suggest several extensions to the basic communication type to enhance the protocol. Finally, we provide some complexity analysis that is used to evaluate the effect response of the model to the protocol. In chapters 7 and 8..

4.1 Message Format

The main specifications that drove the design of the message format were extensibility flexibility and expressiveness. We borrowed ideas from Agent communication languages that already address these issues and especially a language called KQML [FFM+94] which is discussed in the background section. Although the language was adjusted to fit our requirements better the overall structure is very similar.

4.1.1 Syntax

We now describe the syntax of our messages using a BNF style notation. Do not worry about the meaning of each part as this will be described later.

$$
\begin{aligned}
&Message ::= Performative\,\mathbf{:(context:}\,String\mathbf{,sender:}\,String\mathbf{,receiver:}\,String\mathbf{,}\\
&\qquad\qquad\qquad\qquad \mathbf{messageId:}\,String\mathbf{,}[\mathbf{inReplyTo}:String,]^{?}\,\mathbf{content:}\,MapPairs\mathbf{)}\\
&Performative ::= String\\
&Map ::= String:\mathbf{(}MapPairs\mathbf{)}\\
&MapPair ::= String\,\mathbf{:}\,Value \mid Map\\
&MapPairs ::= [MapPair]^{*}\\
&Value ::= String \mid MapVal \mid List\\
&List ::= \mathbf{list(}[Value]\mathbf{*)}\\
&String ::= list\ of\ characters
\end{aligned}
$$

One example of a message is the following:

$$
\begin{aligned}
& ask - all : (context : pathway, \\
& \qquad sender : peer1, \\
& \qquad receiver : peer2, \\
& \qquad message\_id : labelX, \\
& \qquad content : (fromSchema : er\_s1, \\
& \qquad\qquad toSchema : list(er\_s2, er\_s3, er\_s4) \\
& \qquad\qquad constraints : (minimum\_quality : 0.52, \\
& \qquad\qquad\qquad maximum\_length : 15) \\
& \qquad\qquad) \\
& \qquad)
\end{aligned}
$$

Figure 4-1 An example of a message using the syntax

Note that the way the message syntax is defined, allows us to include a message as part of the content field of the containing message for instance notice how the constraints field has similar structure as the message itself. This feature is also supported in KQML and enhances expressiveness.

4.1.2 Semantics

We now take a look on the intended semantics of the different components of the message. The semantics are along the lines of the KQML approach (discussed in section 2.3.5.2) although we implemented several modifications to adapt the KQML structure to our system. We will examine the semantics of each field of the message and suggest how to define new types of messages using the notation.

4.1.2.1 Overview

Throughout the design of the messages we use the speech act theory paradigm [Aust69] which describes how language is used to "achieve something" for the speaker. We are using the metaphor that a message sent by a peer to another is like a sentence in a conversation between peers that tries to achieve something for the sending and the receiving peer. Below we describe how the different components achieve this.

4.1.2.2 Performative:

There is a certain amount of information we can extract from the performative field alone without looking at the content of the message. The performative, or more correctly the performative verb of a sentence makes explicit the purpose of the sentence. Borrowing an example from the literature, in the sentence "*I name this ship Elisabeth*", the performative verb "*name*" makes explicit that the sentence intents to give a name to the ship.

So without considering the context we can identify the purpose of the message and therefore predict the type of reaction the receiver is expected to have. An *ask* performative is likely to have the receiver reply with a single message of a *tell* performative, while a *stream_all* performative will result in a stream of messages by the sender. In this sense, the performative defines the *mode of communication* and using an explicit performative allows for a generic design of the protocol without considering the content. In other words it allows for a layered and modular design of the communication.

4.1.2.3 Context:

The context field describes what the content relates to and is analogous to the “Ontology” field in a KQML message.

An important assumption of our semantics is that the *performative* and *context* should:

- unambiguously identify the exact purpose of the message
- define how the required structure of the content.

For instance, an *ask* performative with a *schema context* should clearly identify that the message asks for the structure of a schema and the content should include the schema name.

4.1.2.4 sender, receiver

These fields should include the names of the sending and receiving peers. For practical reasons, one might argue that the receiving peer field might not be required since the receiver will be the process that listens to the port where the message was sent to. We believe that this assumption about the underlying communication should not be made at this level, however we can allow for the field to be left empty when not required.

4.1.2.5 Message Id

This should be an identifier that uniquely identifies the message. The uniqueness of the id might be a hard thing to implement but probabilistic solutions (like generating a very large random integer) should work for practical purposes. Furthermore the uniqueness of the messageId is sometimes dependent on the semantics of the communication. For example if a sender does the same request to several users (for example when evaluating a query from different peers) it might chose to use the same messageId for all requests (i.e. broker-all requests described in Section 4.3.2).

4.1.2.6 InReplyTo

This is an optional field which should be included in reply messages to request. The value should be the messageId of the request message to help the receiver manage concurrent replies to multiple messages.

4.1.2.7 Content

Content is an application specific field. As it can be inferred from the syntax definition, it was designed such that in can include complex structures using lists and maps with arbitrary nesting. One advantage of being able to do that, is the ability to encode data structures within the message. In this way complex information will not require application specific formatting but the message format can be used instead allowing for more high level and implementation independent communication.

As mentioned above, the *shape* and required fields of the content field of a particular message is determined by the message performative and context field. However the *order* of the fields is not important and also a message can contain more fields for the required ones. For example if the content should include the structure of a pathway, it could be possible to also include the quality of the pathway. This means that different classes of peers can extend the semantics of their communication while maintaining compatibility with other peers.

4.2 Basic Protocol

We now use the abstract model defined in Chapter 3 and the message format described above to describe a suggested basic protocol for inter-peer communication. We later discuss ways to extend the protocol to perform more complex operations and to increase the performance of the network.

We first define an initial set of performatives and their intended meaning. We then define the initial protocol in terms of the performatives and later describe the concrete communication protocol by defining the *messageHandler* function as described in Chapter 3.

4.2.1 Protocol Performatives

The initial performatives describe a basic request-reply protocol and therefore the communication semantics are very simple involving only two messages, the request and the reply. We later define more complex communication patterns as possible extensions to the initial protocol.

Performative	**Description**
ask	The sender wants to know some information about the receiver. The receiver should reply with a tell message
ask-if	The sender wants to know whether some fact holds. The receiver should reply with either a confirm or a deny message.
recommend	The sender want the receiver to recommend some information relating to its global knowledge of the network. The receiver should reply with a tell message.
advertise	The sender wants the receiver to know some information about the sender's capabilities. The receiver should reply with a confirm message if the advertisement was taken into account.
tell	This is a reply to a request as described above.
confirm	A "yes" reply as described above
deny	Send by the receiver of the request to the sender when it either denies a fact or denies to attend to a particular request
fail	Sent by the receiver of a request when it tries to attend to the request but failed.
error	Sent by the receiver of the request when it is not able to understand the format or the content of the request.

We now summarise the communication using a simple finite state machine notation. The communication always involves peers A and B, where A always initiates the communication.

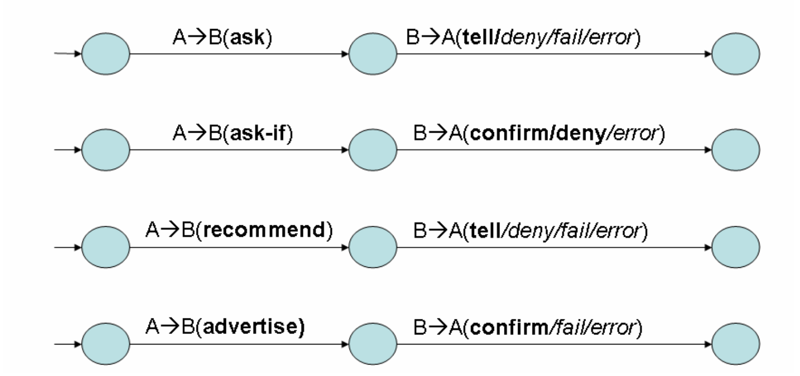

A→B(**ask**)
B→A(**tell**/*deny/fail/error*)
A→B(**ask-if**)
B→A(**confirm/deny**/*error*)
A→B(**recommend**)
B→A(**tell**/*deny/fail/error*)
A→B(**advertise**)
B→A(**confirm**/*fail/error*)

Figure 4-2 The semantics of the initial set of performatives

We now move on to describe:

- a set of concrete message types
- the message handling function of the peers
- its effect in the local and global knowledge of the peers.

We are always using the abstract model described in Chapter 3 as a basis for our design. The reader must be familiar with the concepts introduced in the same chapter to properly follow the analysis of this section.

4.2.2 Assumptions

There are three basic assumptions for our protocol

- A public schema is uniquely identified by the schema name
- A peer is uniquely identified by the peer name
- There is a mechanism for determining a peer's location from the peer name.

Remember we identified the things that need to be defined in order to build a protocol using the model described in the previous chapter. That was done in Section 3.6.1

- Define the structure of messages
- Define the msgHandler function and the structure of peer State.

To extend the suggested features of the model:

- Extend the structure of the cache
- Extend the set of Updates and their operational semantics
- Extend the structure of Dir

We are not extending the suggested features so we will only define the first two points.The reason for not extending the basic features is that we are implementing a very basic protocol which makes no assumptions about the application that uses it. The extensions to the basic features should be done by protocols that are aware of the underlying application and can make assumptions about the requirements. For example, extending the cache and updates could be used to define a caching policy something we are not attempting because caching is generally application dependent and is not addressed by this project.

The generic structure of the messages has already been defined, and we are now describing a set of concrete message types and define the *msgHandler* function to handle these messages.

4.2.3 MessageHandler Function

Remember that the message handler function is defined as:

$$msgHandler : Peer^* \rightarrow (\mathcal{P}(Mail) \times State) \rightarrow (\mathcal{P}(Update) \times Mailbox \times State)$$

A peer can be handling different types of messages at the same time and also several messages of the same type (because there might be several pending communications with other peers). At any time, different types of requests might be pending and the messages received can be both *requests* or *replies* to requests. Consequently, defining the *msgHandler* function properly is a rather complex thing. We need to define it in modular way, where different functions can handle messages of different types and can be defined *independently* from each other.

The way we chose to achieve this, is by defining a set of *handler functions* , where each handler function is called according to the type of message. As mentioned above, the information contained in the message should suffice to determine unambiguously the type of message and thus the appropriate handler. (notice how our lack of imagination made us name the two functions *handler* and *messageHandler* although they do two different things)

State information is passed around to all handlers which can use and update it accordingly. State can contain both *global* information (relevant to all handlers) and *local* information (relevant to one handler).

State keeps count of the *round* (a local count, there shouldn't be any assumption that this count is synchronized across the peers). At the end of the round the count is incremented and all handlers are called once more with a special *NULL* mail as parameter in order to handle messages that *timed out*.

We will define the global information in the state as the round count and a function that maps *pending requests* to the round they timeout.

$$State : (\mathbb{N} \times pendingRequests)$$
$$pendingRequests : ID \rightarrow \mathbb{N}$$

We will implicitly assume that each handler can add its own components to the state tuple.

A schematic illustration of the msgHanlder function is presented in Figure 4.3.

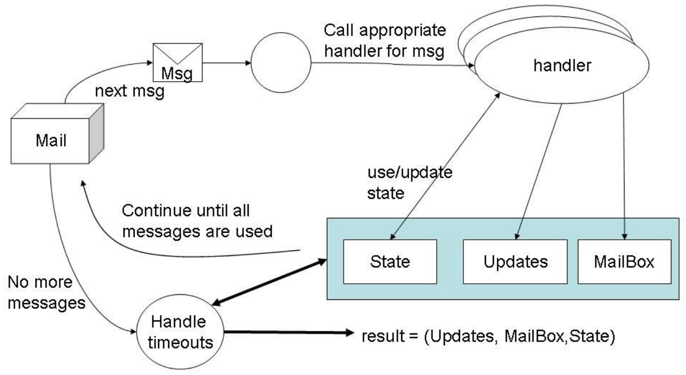

Call appropriate handler for msg
handler
Msg
next msg
Mail
use/update state
Continue until all messages are used
State
Updates
MailBox
No more messages
Handle timeouts
result = (Updates, MailBox,State)

Figure 4-3 Overiview of the msgHandler function

Figure 4.4 gives a detailed definition of the msgHanlder function given in some functional language. After defining the overall structure we can define the protocol just by considering one handler at a time and assuming that the handler will be called when the appropriate message is received and will be called once at the end of each round to handle timed out pending requests.

Given this definition, it means that we can define the protocol one handler function at a time with out worrying about the interactions with other handlers that are managed by the main function.

```
==========
msgHandler
=========
//We use {X} to mean powerset of X in the declaration of methods.
msgHandler::Peer->({Mail}×State)->({Update}×MailBox×State)

msgHandler(peer,state,{})=timeOuts(peer,state);


msgHandler(peer,state,msg:mails)=
                  ((ud1++updates), (mb1++mailbox), finalState)

  where
    handler=getHandler(msg,handlers())
    (ud1 mb1, st1)=handler(peer,msg,state)
    (updates, mailbox, finalState)=msgHandler(peer,mails,st1)



========
timeOuts
========
timeOuts::(Peer×State)->({Update}×MailBox×State)
timeouts(peer,state)=handleTimeouts(handlers(),peer,state’)
  where state’=state:round=round+1



================
handleTimeouts
================
hanldeTimeouts::(List×Peer×State)->({Update}×MailBox×State)
handleTimeouts([],peer,state)=({},{},state)
handleTimeouts(handler:hs,peer,state)=
                  ((ud1++updates,(mb1++mailbox),finalState)
  where
    (ud1,mb1,st1)=handler(peer,NULL,state)
    (updates,mailbox,finalState)=handleTimeouts(hs,peer,state)
```

```
===========
getHandler
===========
getHandler:Mail->[Peer->(Mail×State)->({Update}×Mailbox×State)]->
                      Peer->(Mail×State)->({Update}×Mailbox×State)


No code given. It should find the appropriate hanler for a message by
looking at its performative, its context and possible its in_reply_to
field (for fail messages). If no appropriate hanlder is found it
should return an ERROR_HANDLER that will send an error message to the
sender.



=========
handlers
========
handlers::[Peer->(Mail×State)->({Update}×Mailbox×State)]


No code given. Returns a static list of hanlder functions.
```

Figure 4-4 The messageHanlder function

4.2.4 The Communication Handlers

In Appendix A we describe the different types of interactions between peers. For each type we will define:

- the structure of the messages
- the appropriate handlers (we will define three handlers, one that makes the request one that receives the request and one that receives the reply)
- the extra state information that the communication type requires (if any)

We do not use the functional programming language notation to define the handlers but a more imperative style pseudo code. When defining the required message structure we will define the *performative*, the *context* and the structure of the *content* since the other fields can be trivially defined.

When defining the handlers we will need to describe a message using a compact notation. To illustrate the notation we will be using, the message of figure 4.1 will be defined as:

ask-all[pathway]{fromSchema=er_s1, toSchema=[er_s2,er_s3,er_s4]}

The rest of the information will be implicit from the context at the place of the definition.

Important!

The description of the protocol was initially part of this chapter but was demoted to an appendix because of its length that affected the readability of the chapter. Although you can follow the chapter by reading the summary of of messages in the next section (given that your intuition is correct about what its message does by reading its description) it is important that you read the full protocol definition. When defining the handlers of the communication types we illustrate how the different components of the abstract model defined in the previous chapter are used. It is important that you read the code of the handlers carefully to understand some important details about the intended use different features of the model.

4.2.5 Summary of messages

Performative	**Context**	**Description**
ask	*schema*	requests the definition of a schema
tell	*schema*	tells the definition of a schema
ask	*query*	requests the answer of a query
tell	*query*	tells the answer of a query
ask-all	*final-pathway*	request a set of pathways from a public schema to a set of data sources
tell-all	*final-pathway*	tells a set of pathways to data sources
recommend	*pathway*	asks for a pathway that links two schemas
tell	*pathway*	tells a pathway that links two schemas
recommend-all	*peer*	asks for the names of the peers that implement a schema
tell-all	*peer*	tells the names of the peers that implement a schema
ask-all	*schema-list*	asks for the set of public pathways
tell-all	*schema-list*	tells the set of public pathways
advertise	*pathway*	informs that the sender provides a pathway to a public schema
confirm	*pathway*	confirms that the update for advertise pathway was successful
deny	*pathway*	informs that the update for advertise pathway was unsuccessful

4.2.6 Example

We now illustrate the protocol described so far using an example. Consider the example network described in Figure 3-1 An example network with 2 public schemas 3 peers and 3 data source schemas. We will give a series of messages that will result to pr1 having a pathway to ps2 and two more final pathways to ds2 and ds3 respectively. It first gets a pathway to ps2 by requesting a path that links ps1 to ps2 from pr2. It then gets pathways to ds2 and ds3 by requesting pr3 to give a set of final pathways from ps2 and appends its pathway to ps2 to the returned final pathways.

In order to achieve this, it needs to gain global knowledge about the network. It needs to know two things:

- which are the public schemas that are not within its local knowledge
- which peers implement the public schemas it is interested in.

It gains this knowledge by querying the directory service. The exchanged messages are shown in Figure 4.5 and explained in Table 4.1 The resulting updates that evolve the network are also shown in Figure 4.5 and are explained in Table 4.2. The resulting network is shown in Figure 4.6

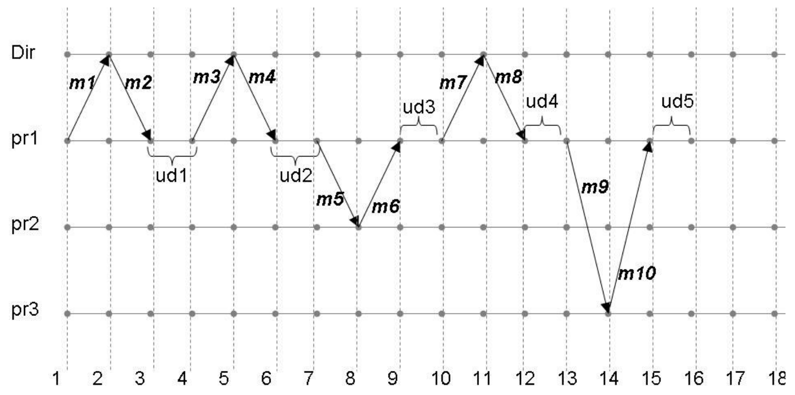


Figure 4-5 The message exchanged in the example

Table 4-1 The messages of the example

Messages	
m1	recommend-all[peer]{schema:ps1}
m2	tell-all[peer]{peers:[pr1,pr2]}
m3	ask-all[schema]{}
m4	tell-all[schema]{schemas:[ps1,ps2]}
m5	recommend[pathway]{fromSchema:ps1, toSchema:ps2}
m6	tell[pathway]{path:(w2|w3)}

m7	recommend-all[peer]{schema:ps2}
m8	tell-all[peer]{peers:[pr2,pr3]}
m9	ask-all[final-pathway]{schema:ps2}
m10	tell-all[final-pathway]{paths:[(w4|w7),(w4|w8)]}

Table 4-2 The updates of the example

Updates		
	Updates	Effect
ud1	impl(ps1,{pr1,pr2})	The fact that ps1 and pr2 impliment ps1 are added in the global knowledge of ps1 (cache)
ud2	public({ps1,ps2})	The fact that ps1 and ps2 are public schema names are added in the global knowledge of ps1
ud3	new_path(w2|w3)	There is a new path to ps2 from pr1, which is (w1|w2|w3)
ud4	impl(ps2,{pr2,pr3})	The fact that pr2 implements ps2 is added in the global knowledge of ps1.
ud5	new_path(w4|w4), new_path(w4|w8)	There are two final paths from pr1 to ds2 and ds3 which are (w1|w2|w3|w4|w7) and (w1|w2|w3|w4|w8) respectively.

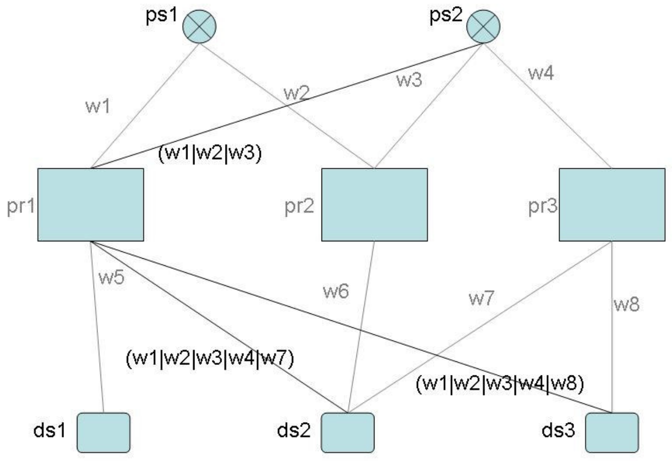


Figure 4-6 The resulting network after the communication

4.2.7 Summary of protocol functionality

The protocol described so far provides the basic functionality for P2P communication. In particular it provides the following capabilities:

- A peer can locate other peers in the network, and in particular the peers that are logically connected to it.
- A peer can locate public schemas in the network
- A pathway to a reachable public schema can be derived by appending the pathways that lead to the schema in question. This can be achieved after a series of message exchanges. ie. $if \ \ (pr \hookrightarrow ps) \ \ then \ \ \Diamond Imp(pr, ps)$ where the $\Diamond$ modality is taken to mean "after a possible sequence of message exchange"
- connected peer. i.e. $if \ \ (pr1 \Leftrightarrow pr2) \ \ then \ \ canQuery(pr1, pr2)$ (canQuery has the obvious meaning)
- A reachable peer can become a connected peer after a series of message exchanges. i.e. $if \ \ (peer1 \leftrightarrow peer2) \ \ then \ \ \Diamond(peer1 \Leftrightarrow peer2)$
- The above two points imply that: $if \ \ (pr1 \leftrightarrow pr2) \ \ then \ \ \Diamond canQuery(pr1, pr2)$
- Similarly, a peer can connect to a reachable data source schema as illustrated in the above example. i.e. $if \ \ (pr \multimap ds) \ \ then \ \ \Diamond Src(pr, ds)$

4.2.7.1 The Basic Capabilities

The properties described in the previous section are summarized by Table 4.3. We call these 5 properties the "*basic capabilities*" of the network. We illustrated that the basic protocol has basic capabilities. We isolate these properties and give them a name because we believe that they can be a measure of the functionality of a protocol. They can be a benchmark that can be used to assess any protocol built using the abstract model.

They are not a perfect measure since they do not talk about quality of pathways or how efficient the communication is. However they cover the concepts of reachability and connectivity and *how reachablity can be evolved to connectivity.*

Table 4-3 The Basic Capabilities

$if \ \ (pr \hookrightarrow ps) \ \ then \ \ \Diamond Imp(pr, ps)$
$if \ \ (pr1 \Leftrightarrow pr2) \ \ then \ \ canQuery(pr1, pr2)$
$if \ \ (peer1 \leftrightarrow peer2) \ \ then \ \ \Diamond(peer1 \Leftrightarrow peer2)$
$if \ \ (pr1 \leftrightarrow pr2) \ \ then \ \ \Diamond canQuery(pr1, pr2)$
$if \ \ (pr \multimap ds) \ \ then \ \ \Diamond Src(pr, ds)$

4.3 Extending the protocol

The description of the protocol in Appendix A has demonstrated how to define message types and how to describe in a formal way the reaction of peers to different messages using the model described in Chapter 3. It should be obvious how to extend the set of message types and the peer behaviour in order to extend the protocol. Therefore, the description of the extensions in this section will not be as formal and thorough; instead we will give an informal description, assuming that the reader has already realised how to map it into a more formal definition similar to Appendix A.

4.3.1 Streaming Messages

The communication patterns described so far are all based on the client server model. We now introduce a new message type to illustrate how one can use the existing infrastructure for more complex communications. We introduce the performative *stream-all*. The sender of a *stream-all* message requests some information from the receiver which will be sent in a stream. The receiver replies with a *ready* message (message with *ready* performative) and waits for instructions from the sender. The sender can send a next message to get the *next* stream or a *stop* message to stop the streaming. The receiver replies to a next message with a *tell* message at a time until the end of the stream is reached, on which case he will reply with a eof message. The communication is illustrated in Figure 4.7.

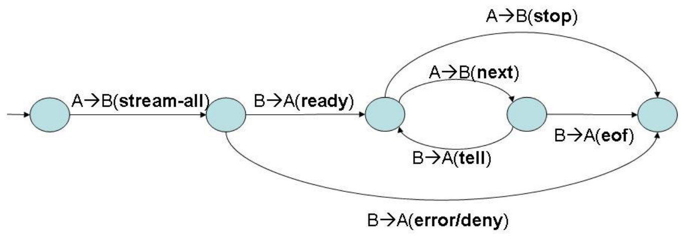


Figure 4-7: The communication semantics of the stram-all peformative

The advantage of using stream-all queries is that you can avoid using very large messages and that the receiver can stop the streaming when required. In our protocol, all ask-all/stream-all messages can have their stream-all twin. For example, the ask-all schemas message that is sent to the directory service to get all public schemas can be replaced by stream-all schemas in order to get the answer in a stream, one schema at a time. A query evaluation can also be sent back in a stream especially when the result is likely to be large.

Variations of the stream-all communication defined above can also be implemented. For example we can define the communication so that the stream arrives without using ready and next messages as illustrated in Figure 4.8. This has the advantage that it is more efficient in terms of communication complexity but there is no synchronization between the sender and the receiver.

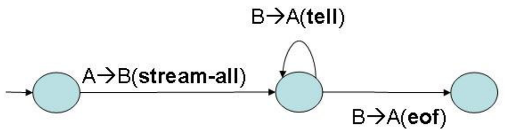


Figure 4-8 Alternative semantics for the stream-all performative

4.3.2 Query Distribution

We will now discuss a query distribution algorithm using "*query brokers*". The difference between posing a simple query to another peer (as demonstrated with the "ask query" message) and querying a broker, is that the brokers use not only their own data sources to answer the query, but also propagate the query to other peers connected to them. A peer receiving a broker query expressed on a public schema ps, will transform the query such that it is expressed in all the other public schemas that it implements. For each such public schema it will locate all the peers that are connected to it, and it will send a broker query request to all those peers, who will act in the same way. When all the peers have replied the broker will transform all the results to be expressed back in ps (the original public schema), will use its own data sources to answer the query as well and will send the merged result back to the requesting peer (which might be another broker itself). The procedure is illustrated in Figure 4.9.

Note that the techniques for locating peers that implement a public schema, evaluating queries and transforming queries and query results from one schema to another have all been covered when describing the basic protocol. In this sense this type of communication does not introduce any new concepts, and it should be obvious how to implement the proper hanlder functions although they will be much more complicated that the ones described so far.

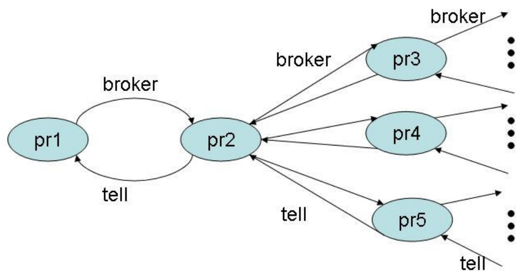


Figure 4-9 Outline of the broker-all performative

This incomplete of the broker query description has three problems:

- How to avoid requests going into cycles and as a result, how to terminate the algorithm.
- How to keep the algorithm from travelling too deep into the network and accessing bad data through very low quality pathways.
- How to avoid answering duplicate evaluation of the same query.

4.3.2.1 Avoiding Cycles

We first describe two alternative solutions to this projec.

4.3.2.1.1 Using Message Id

The first solution is rather simple and it merely requires that peers remember broker requests that they have attended to. To implement this, all the sub-broker queries should have the same messageId as the original broker message. In other words, referring to the example at Figure 4.9, broker queries sent by pr2 to pr3..pr5 should have the same messageId as the broker message sent by pr1 to pr2. This way, when someone sends the query back to pr2, pr2 will refuse to answer it (send a deny message), and therefore cycles will be avoided. It should be easy to see that after this adjustment the algorithm will terminate in a network with finite number of peers.

A major disadvantage of this approach is that it does not take full advantage of all the pathways. We illustrate this point with an example. Consider Figure 4.10, peer pr1 and peer pr3 are connected through pathways w=(w5|w4) and w'=(w1|w2|w3|w4). There is no guarantee which of the two pathways is of better quality. If pr1 brokers a query expressed on its export schema the query will reach pr3 from both of these pathways. However, with this algorithm, only one of the two queries will be answered, the one that arrives first, which is not necessarily the one with of the best quality. Even when the one with the better quality is answered, it is possible that when using the other pathway you can obtain information is not available with the first query.

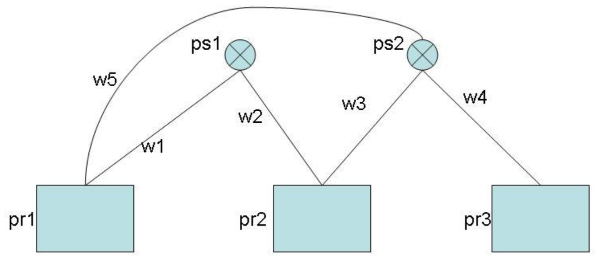


Figure 4-10 A simple network to demonstrate broker queries

4.3.2.1.2 Using Message Path

We now describe an algorithm that does not cycle and takes advantage of all the pathways. The algorithm is summarised as follows:

- Each message contains a list of peers called "*message_path*" which describes the peers that the message has travelled through.
- The initiator of the broker message puts only itself in the message_path and sends the message as before
- Any peer that receives a broker message checks whether itself is already in message path. If it is, it refuses to answer the query. If it is not, it adds itself in the path and propagates the message as before (this can be optimized so that the sender checks the path and does not send it to peers that are already there)
- Everything else works as described earlier.

4.3.2.1.2.1 Example

- Using example at figure 4.10 for illustrating this algorithm, assume that pr1 is the initiator of the broker-all request:
- pr1 sends message with message_path=[pr1] to pr2 and another message with the same message path to pr3
- pr2 sends message with message_path=[pr1,pr2] to pr3 and pr3 sends message with message_path=[pr1,pr3] to pr2
- pr3 sends message with message_path=[pr1,pr2,pr3] to pr1. pr2 sends message with message_path=[pr1,pr3,pr2] to pr1.
- pr1 refuses to propagate any of these messages and sends a deny message to both pr2 and pr3.
- The algorithm then enters is retracting phase where all the request are being replied until they fall back to pr1.
- As we can see the query has reached pr3 through (w5||w4) and (w1|w2|w3|w4). Similarly it reached pr2 through (w1|w2) and (w5|w3) and all the queries have been answered.

4.3.2.1.3 Avoiding Duplicate Evaluations

A potential problem with this approach is that two different pathways might be the exactly the same. For example w5 might be the same pathway as (w1|w2|w3|w4). This will mean that the same query will be answered twice by pr3.

A way to improve on this problem is to keep track of the queries that have been answered for a particular broker-all request. So if pr3 sees that for request with *id=x* a query q expressed on ps2 has already been answered and it arrives again through a different message_path, then it will refuse to reply.

4.3.2.1.4 Avoiding Long Paths

We now address the issue how to keep the algorithm from travelling too deep into the network and accessing bad data through very low quality pathways.

4.3.2.1.4.1 Using Time-to-Live

We will describe two simple techniques for doing that, which can be used in parallel to the techniques used above and also in parallel to each other. The first is using a time to live (ttl) field in the message, which denotes the number of peers that it is allowed to travel through. When propagating the broker request a peer should decrement the field. A peer receiving a request with ttl=0 should refuse to answer it.

4.3.2.1.4.2 Using Quality

The second technique is by defining a minimum quality field which denotes the minimum quality of the pathway that the query should travel, and another field measuring the quality so far. When ever a query is transformed from one schema to another the quality so far should be adjusted to account for the extra transformations which should decrease the quality so far (by assuming that adding more transformations will never increase the quality). When a peer receives a request with quality so far being less than the minimum quality, it should refuse to propagate it or evaluate it.

4.3.3 Discovering Pathways Considering Quality

We will now describe an algorithm based on broker requests that can be used to discover pathways from one public schema to another. We say that two schemas are directly linked if there is a peer which implements both of them. Two peers are indirectly linked if there is pathway from one to the other in the network. For example, in Figure 4.10 ps1 and ps2 are directly linked while ps1 and ps3 are indirectly linked.

We can define the notion of indirect link recursively: Schema ps is indirectly linked to schema ps' if they are directly linked, or if there is a schema ps'' which is directly linked to ps and indirectly linked to ps'.

We will use this definition to describe a protocol for discovering pathways between two public schemas. A scenario for this protocol is the following: Assume that peer pr3 in Figure 4.11 wants to implement public schema ps1. It will want to know that pr2 implements both ps3 and ps2 such that it can get a linking pathway from ps3 to ps3 and then that pr1 implements both ps1 and ps2 in order to get a linking pathway from ps2 to ps1. The desirable result of the request $[pr2(ps3, ps2), pr1(ps2, ps3)]$. After this it can send the appropriate recommend[pathway] requests to pr2 and pr1 in order to get the actual linking pathways. The point here is that we are not interested in the actual pathways but to the chains of schemas and peers that lead to the pathways.

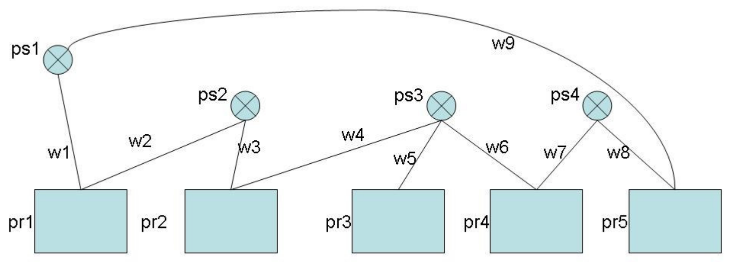


Figure 4-11 Example network

The algorithm uses the broker message infrastructure described in the previous section including the techniques for avoiding cycles etc. The result to a request will typically be a list of pathway descriptions as illustrated above. As an extension, for each pair of schemas we will also add the *quality* of the pathway such that the interested peer can use this fact to decide which is the best pathway to use. For example, if pr3 does such a request for linking pathway from ps3 to ps1, the result will be:

$$\{[pr2(ps3, ps2, ql1), pr1(ps2, ps3, ql2)], [pr4(ps3, ps4, ql3), pr5(ps4, ps1, ql4)]\}$$

We define the protocol using a pseudo code as follows:

```
for peer pr:
when receiving a broker[pathway]ps1,ps2:
  //assuming that the peer is implementing ps1

  reply={} //empty list
  if(pr2 implements ps2)
      reply = {[pr(ps1,ps2,ql)]}
      where ql = quality of path from ps1 to ps2

  foreach(ps3 which pr implements except ps1){
    forech(pr' which implements ps3){
      send message broker[pathway]ps3,ps2
    }
  }

  foreach(reply from peer pr' from some schema ps3 in the form
                                        {path1,path2,...}){
     //append path ps1,ps3
    add to reply {pr(ps1,ps3,ql):path1, pr(ps1,ps3,ql):path2...}
        where ql=quality from ps1 to ps3
  }

  when all replies received or timed out:
  SEND BACK REPLY
```

This algorithm, when implemented using the message_path approach (which explores all possible paths) and without using the techniques for limiting the query path using the ttl approach described above, will return all possible paths to the target path and since the paths contain the qualities as well, the peer that made the request can choose the path with the best quality. Finding the best path between two nodes in the network is a known $O(n^2)$ complexity problem using some known algorithms like the Dijkstra's Algorithm. We will not complicate our definition by implementing such an algorithm. Instead we mention thre heuristics for speeding up the algorithm, which however do not guarantee the best quality:

- Use the ttl field to reduce the depth that the message will travel and the approach that does not explore all the pathways of broker queries as described above.
- When a peer has a direct link between the schema (i.e. when the first if statement of the algorithm is true), it should not propagate the request.

4.3.3.1 Iterative algorithm

An iterative version of this recursive algorithm can also be implemented. We will describe an iterative algorithm that finds a pathway (not the one with the best quality) between two peers. It should be obvious how to extend it to include quality as well.

The requester, instead of sending a broker request to a peer, it only requests the set of schemas that the peer implements The requester then proceeds to request further paths itself from peers implementing those schemas until it finds a linking path. We can choose to implement a depth first or breadth first algorithm. A depth first algorithm should be more efficient in terms of message complexity for finding some path (i.e. for finding the first path), while a depth first will generally generate more messages but finish in less number of rounds because in every round it will be traversing more paths.

4.3.4 Anonymity of Data sources

The protocol described so far does not provide any way of peers locating particular data sources. In some applications this might be necessary, for example when peers want to retrieve information from particular data sources. There are several ways of reavealing the identity of a data source. A possible way is the following:

- Extend the definition of Dir (directory service) such that it provides a mapping from data source schemas to peers that are connected to it.
- Add a new message type which queries a peer for data only from a particular data source. i.e. ask[query-source]
- Add a new message type which asks for a final pathway from a particular data source.
- A peer that wants to query a data source schema can get the peers that are connected to it, and ask for the public schemas that they implement.
- It will then try to connect to one of those public schemas using one of the techniques described above.
- It can either query the peer on the required data source schema or ask for a final pathway that links to the data source.

4.4 Evaluation of the protocol

We evaluate the protocol with reference to time and communication complexity for different scenarios, considering the worst case. We have to consider the worst case because we do not know what the average case will be as this depends on the application. For instance, it will be a different analysis for a network where all the peers are connected to a single public schema and a different one for a network where at most two peers are connected to a public schema.

We will analyse the following cases:

- Discovering all the peers that implement a public schema
- Querying a peer that implements a particular public schema
- Discovering a path between two public schemas
- Discovering the best quality path between two public schemas
- Executing a broker-all query

4.4.1 Discovering all the peers that implement a public schema

The requesting peer will need to send a recommend-all[peer] message to Dir and Dir should send a reply back. This has communication complexity 2 messages and time complexity of 2 rounds

4.4.2 Querying a peer that implements a particular public schema

This requires finding out a peer that implements the schema and then sending a query and getting the result. This requires 2 messages to get all the peers that implement the schema and two messages to send the query and get the reply. (assuming that the peer implements the schema itself). If this request is repeated there only need to be two messages since the fact that the particular peer implements the particular schema is already within the global knowledge of the requesting peer.

When instead of querying the peer directly you get a pathway to the data sources and then query the data sources there will need to be 4 messages the first time (2 for discovering the peer and 2 for getting the data sources). If the peer needs to be queried again no message transfers are needed since the pathway is already part of the local knowledge of the peer!

4.4.3 Discovering a path (any path) between two public schemas

This depends which algorithm will be used. We will avoid considering algorithms that only involve Dir since they would cause too much load on Dir and will not be scalable. Algorithms that guarantee correct results are inefficient in the worst case.

We consider the iterative breadth fist and depth first algorithms described in Section 4.3.3.1

Both algorithms in the worst case will require to traverse all peers (and get all the schemas from each) and will therefore require $O(n)$ messages, where n is the number of peers.

The depth first algorithm is considering only one peer at a time so it is easy to see that the time complexity is also $O(n)$ rounds.

The breadth first algorithm will be typically sending more than one messages at a one round but in the worst case the topology will be such that only one peer can be considered (i.e. when a peer is only connected to at most two peers). Therefore the complexity at the worst case is again $O(n)$.

4.4.4 Discovering the best quality path between two public schemas

The algorithm described in section 4.3.3 which uses *message_path* and guarantees a correct result will need to consider all possible pathways. Consider the case where there are n peers and m public schemas. The worst case is when every peer is connected to every schema. In this case there will be there will be $O(m)$ rounds (until all the public peers are traversed).

At the first round the initiator will send $O(n)$ messages. In the next round every peer will send $O(n)$ messages to every other peer, that is $O(n^2)$, so in the next round each peer will need to broker $O(n^2)$ request, therefore in the next round there will be $O(n^3)$ messages. So you can see that the total number of messages will be $O(n^m)$. As mentioned earlier the problem of finding the best path between two nodes is a known $O(n^2)$ problem so a more efficient algorithm should exist (i.e. based on Dijkstra's algorithm) but it is not within our scope to analyse such complex algorithms.

Instead we are considering heuristics. Using the broker protocol that uses time to live field as described above, where ttl=c there will be $O(c)$ rounds and $O(n^c)$ which is a much better result but it does not guarantee a correct result.

Using an algorithm that does not traverse all the paths (where remember the msgId of the broker queries they attended to), each peer will send at most $O(n)$ messages during the algorithm (because they can only propagate one broker request). Therefore communication complexity is $O(n^2)$.

4.4.5 Executing a broker-all query

This is a similar analysis as the one for finding the best possible pathway since the structure of the protocol is the same. The results are identical to the ones above.

4.4.6 Conclusion

Analysing the complexity we get some encouraging results regarding the interactions between two peers and the way that the knowledge of the peer is updated such that in subsequent requests the number of messages that need to be transferred are less. This is demonstrated in the first two scenarios.

However the results are not encouraging for requests that need a peer to gain global view of the network for example for discovering connections between arbitrary schemas and distributing queries. The communication complexity in these cases is discouragingly high in the worst case.

However the worst case is very rarely met in real applications. More importantly, we have seen how some simple heuristics can significantly decrease the complexity. We believe that the problem of decreasing the complexity can only be addressed when considering particular applications that are using the infrastructure that we are defining. This is because only at this level we can take advantage of the special characteristics of the network. This project does not assume any particular application and therefore any application specific heuristics and algorithms are not within the scope. Furthermore we are not considering any caching policy and how this will help in increasing efficiency. Again caching policies are mostly application specific and were not addressed by this project. Such issues are discussed in the future work section of this report.

5 Implementation Overview

5.1 The Problem in Hand

The implementation aspect of this project requires the development of an infrastructure which can be used to run a network of peers. Each peer should wrap an AutoMed repository including its data sources and part of the repository should be made available to other peers in a data integration environment. We should introduce the notion of public schemas on which different peers can provide pathways from their local schemas. Two peers will be logically connected to each other if they implement (i.e. provide a pathway to) the same public schema. There should be a mechanism for peers to make schemas public and locate public schemas from other peers.

There should also be a mechanism for a peer to locate other peers in the network and communicate with them in some specified language. Through this communication peers should be able to pose queries to each other in order to obtain information about the data sources of other peers. Furthermore, peers should be able to derive pathways to public schemas by using existing pathways from other peers in the Network and therefore logically connect to other peers.

These problems were addressed formally using an abstract model as described in Chapter 3 and an abstract protocol as described in Chapter 4. The problem is now how to translate the model into a real implementation. This chapter identifies the main issues that were tackled while translating the model into the real world and gives an overview of the software architecture of the system. Finally it gives an overview of the capabilities of the final system and identifies some issues that were left open by this implementation.

5.2 From the model to the real world

There was a two way association between the model and the implementation. The basic features of the model were used to drive the implementation, while the lessons learned during the implementation were used to refine the model. It is therefore expected that there is a direct mapping between the model and the system.

Similar to the model the real system is made up of a set of peers and a directory service which behaves as described in the model. Peers can join and leave the network when they wish. The assumptions made in the model are also made in the system:

- Any peer can send messages to any other peer unless there is a network failure
- The peer name uniquely identifies a peer in the network
- A public schema name uniquely identifies the public peer
- The location of the directory service is known by all peers.

Now we state the main issues that needed to be addressed when translating the model on the real system and give an overview of the way the problems were solved.

5.2.1 Peer Description

In the model a peer is characterized by an export schema, its local knowledge and its global knowledge. This was not difficult to map to the real world. We associate the peer with an AutoMed repository and in particular one of the schemas in the repository (export schema). The local knowledge is the pathways that already exist in the repository while the global knowledge is a cache that maintains information that was retrieved from the directory service. The cache was implemented in such a way that it is easy to extend to add extra features as suggested by the model. See overview of software architecture section.

5.2.2 Directory Service

For the purpose of this project, the directory service was implemented as a server that runs on a well known address and uses information stored in an SQL database. More details are covered in Section 5.4.4

5.2.3 Publishing Schemas

A peer can send the structure of a public schema in text format (see Chapter 7) in order to publish a schema. A schema publication also includes the name of the schema which needs to be unique and a description so that other peers can know what the schema is used for.

5.2.4 Locating Peers

Besides the name of a peer, the directory service maintains the IP of the host that the peer is running on. Therefore the assumption of the model that a peer can be located by knowing only its name can be implemented by appending the IP as part of the peer's name. (see discussion later about peers moving to different hosts and having more than one peer on one host).

5.2.5 Sending Messages

The model talks about message transfer in the abstract without making any assumptions about the underlying network properties. For this project we chose to implement the communication on a TCP transfer protocol because of its reliability. The price to pay is of course the speed which in not as good as other approaches like UDP. This project did not focus into this aspect of the communication.

5.2.6 Transferring Pathways and Schemas from One Repository to Another.

The model simply assumes that a pathway can be included in a message and be transferred from one peer to another. However, serialisation of AutoMed objects from one repository and desirialisation to another one was one of the biggest issues faced in this project and it is discussed extensively in Chapter 6.

5.2.7 Datasource schemas

The model assumes that a data source schema can be simply queried and give a result. However in AutoMed, this can only happen by using an access method to define how the schema maps to a physical data source [McB04]. Consequently, when transferring a datasource schema from one peer to another, the relevant access method have to be transferred as well. The way this is done is explained in chapter Chapter 6.

5.2.8 Query and Query Results

A query in AutoMed is expressed in IQL [Poul04] and it is represented as an Abstract Syntax Graph (ASG). Once the query is expressed as an ASG, there are tools that enables users to transform it from one schema to another, according to the *transQ* function in the Model. The problem of representing the abstract syntax tree as and sending it across from one peer to another (something that the model assumes you can do) can be done through *write* methods that are part of the AutoMed API. On the receiving end, the peer can parse the text back to an ASG using predefined *read* methods. Therefore, the assumption of the model that you can send a query from one peer to another can be implemented quite easily.

The treatment of the result of a query is the same as the query, and therefore the model assumptions that you can send the results across peers and transform them from one schema to another can also be implemented.

5.2.9 Message handler function

The model assumes that there is a message handler function that for each round at every peer it describes what messages should be sent and what updates should occur at the peer. In Chapter 4 we saw how this function can be defined in order to describe a protocol. The implementation of the protocol on the real system is described in Chapter 8.

5.2.10 Messages

One of the main challenges of the project was to implement the message format described in Chapter 4 and provide an infrastructure for parsing those messages and easily extending the language by defining new message types based on the format on which the network can easily adapt. The way this was done is covered in Chapter 7.

5.2.11 Quality

In the model we assumed the existence of the quality function that can find the quality of the pathway given only the structure of the pathway. Finding the correct measure for comparing pathways in not within the scope of this project. However, for demonstration and testing purposes we have extended the definition of the Pathway in AutoMed so that it gives as a measure of quality the number of extend and contract transformations (where the larger the number the worse the quality). This is some indication of the pathway quality but it might not be accurate. In the Future Work section we discuss how this measure can be implemented.

5.2.12 Evolution of the network through operations

All the operations described in the operational semantics of the network evolution in Chapter 3 can be implemented in the real system. They all involve adding or removing a peer in the network, adding a pathway or a schema in the local knowledge of the peer and adding simple information in the global knowledge (cache) of each peer. Some of these take the quality measure into account, and as we have seen above, a quality measure has been implemented.

5.2.13 What is a "Network"?

When defining a model, we described the network as a set of peers and a directory service. We said that peers that are outside the network may also exist. So when going to the real world the question is: which peers are inside a network and which peers are outside the network? One approach could be to define the network as a set of peers with communication links from one to the other. This means that in case of physical network partition the network would be split into two networks. However we have seen that in the model we can model network partition within a single network. Furthermore the links between the peers are not physical in our network but logical (through public schemas).

So instead we define the network as: *"A directory service and a set of live peers that are aware of the location of the directory service"*. This makes no assumptions about any physical links, not even reliable links between a peer and the directory service. This definition is closely linked to the abstract model which can represent arbitrary communication failures as we have seen in Chapter 3.

5.3 Other issues

There were several other practical issues that needed to be dealt with while implementing the system.

5.3.1 Running two peers on the same repository

It is often possible that we will want more than peers to run on the same AutoMed repository. A potential problem this has is the reaction of the system to concurrent updates by the different peers. One problem we noticed was that the graphical editor provided by AutoMed did not get notified of changes that were implemented by a different peer. However when restarting the editor the changes appeared. We identify the refreshing of object cache for changes done by different processes as a thing that the AutoMed designers should address.

5.3.2 Moving the same peer to a different host

In our initial approach in naming peers we were using the peer's URL as its name in the network. This would however mean that when a peer moves to a different host the network would identify it as a different peer. Still, we want a way to associate the peer's name with its location.

The solution was to introduce the notion of *PeerDescription* which includes the peer's name (a name chosen by the peer which needs to be unique in the network) and IP address. The directory service maintains the *PeerDescription* for each peer. When a peer logs from another host it logs on the directory service and notifies it that the IP has changed. The *PeerDescription* for that peer gets updated to include the new IP.

5.3.3 Several Peers in the same Host

One of the things that we wanted to achieve in our network is the ability to run several peers in the same host. Consequently each peer will need to be at a different port and therefore the approach of having a well known port when every peer will be mapped on is not a valid option. Still we would like to avoid depending on the port number in order to locate a peer. The approach we took was using a well known port to map a registry where any peer at a host can register. Requests go to the registry which uses the peer's name to redirect the request to the appropriate peer in the same host. The details of this architecture are described in theArchitecture Overview Section.

5.4 Architecture Overview

This section gives a brief overview of the software architecture and highlights the main components of the software. This report is not exhaustive in describing the software structure since this was a large software development project where more than *70 java files* were added to the AutoMed API.

5.4.1 Overview of Peer Processes

A peer must run as a process at a host in the network. For the peer to be allowed to run properly a Registry process must be running at the same host. The registry is a simple process that runs at port number 8282 (or any well known port; we are using 8282 for this project). A peer can only join the network if it is registered with the registry. The configuration is illustrated in Figure 5.1 In the general case, a host will have more than one peers, other peers are not aware of port numbers of the different peers and should therefore request the port number of the particular peer from the registry before they can contact it. This interaction is illustrated in Figure 5.2. We now illustrate the architecture of peers and registries.

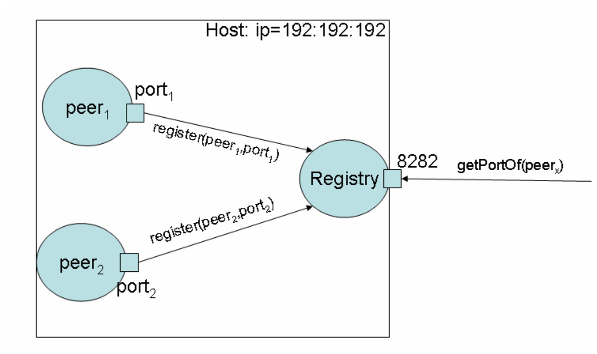


Figure 5-1 A host with two peers

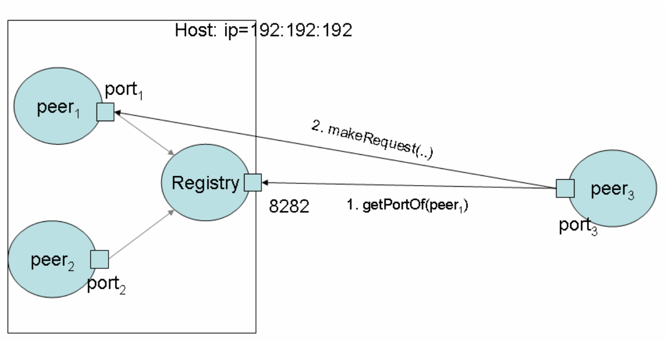


Figure 5-2 Communication between two peers

5.4.2 Communication Infrastructure

We have developed an infrastructure for communication which is reusable and modular, separating out the communication details and the logic of the different components. The basic components are illustrated in Figure 5.3. The AbstractDaemon is an abstract class which can run as a multithreaded process that can binds to a port and listens for requests. Any request is handled by one of its threads (the number of threads can be configured) while the response to the request is handled by the subclasses. The class was defined in such a way that it can be reusable and configurable. A level above the AbstractDaemon is the AbstractServer, which is a subclass of the AbstractDaemon. This class delegates any requests made to the AbstractDaemon to an object that implements the RequestListener interface.

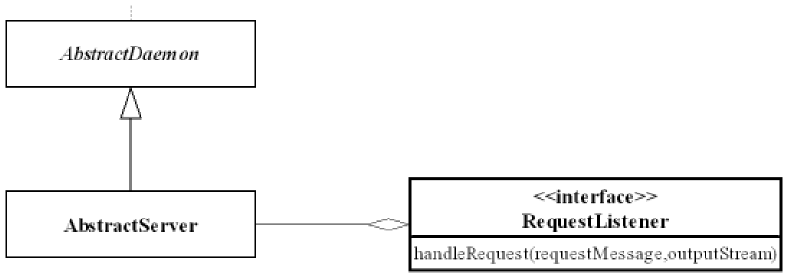


Figure 5-3 Core classes for communication

This architecture implies that any class that wants to behave as a server should just implement the RequestListener interface and register itself to an abstract server. i.e.

```
                    //concrete server//
         AbstractServer server = new AbstractServer(this);
                         server.run();

      public void handleRequest(..){//message handling code}
```

Figure 5-4 Implementing the AbstractServer

Alternatively it could chose to just subclass the AbstractServer and implement the interface.

The point here is that we can now implement servers by only considering the high level logic of the server while everything else will be handled by the lower levels. One example is the implementation of the Registry which is shown in Figure 5.5

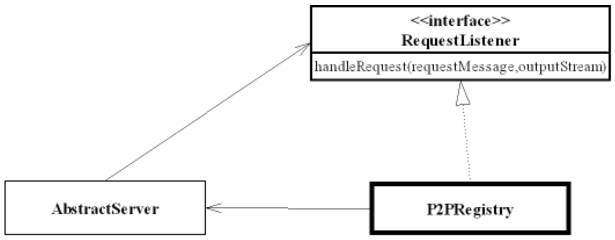


Figure 5-5 Implementing the Registry

The handleRequest method implementation is rather simple. It handles requests to register a local peer in the system and requests to get the port number of a local peer. Note that the message that is received is encapsulated within an object and it does not need any parsing by the server. The format of the message and how the encapsulation is implemented is covered in Chapter 7.

5.4.3 Implementing a Peer

An AbstractPeer is thought of being comprised of a client and a server. The client initiates requests on behalf of the peer while the server listens to request from other peers. When a client initiates a request, a connection is established between the client peer and the server peer, in which communication between the two can take place. The communication does not have to be a simple request-reply but there could be arbitrary message exchanges between the two.

An AutoMed peer is an abstract peer that wraps an AutoMed repository and can communicate with other peers according to some protocol. The definition of the protocol that was implemented for this system was described in the abstract in Chapter 4 while its implementation on the real system will be covered in Chapter 8. The overall architecture is demonstrated in Figure 5.6.

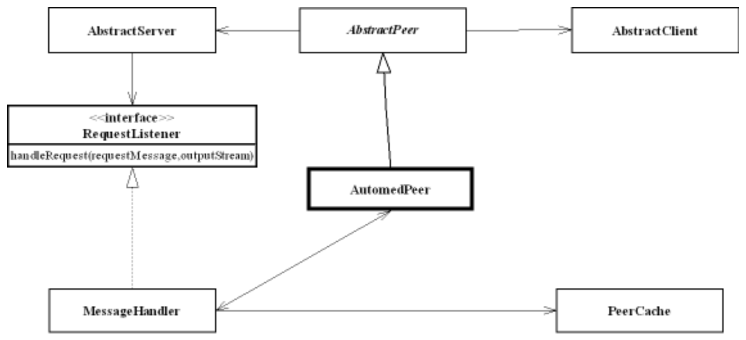


Figure 5-6 Design of a peer

The components that are specific to a particular protocol are the MessageHandler and PeerCache. Extending the functionality of the protocol will require extending those two components. PeerCache is the analogous to the global knowledge (cache) of a peer. For reasons discussed in Chapter 4 the current structure of the cache was kept quite simple but it can be subclassed to implement more complex caching. The current implementation of MessageHandler implements the protocol described in Chapter 4. A description of its implementation on the real system is given in Chapter 8.

Note that the message handler has also access to an AutoMed repository which comprises the local knowledge of the peer as described in the abstract model.

5.4.4 Implementing the directory service

The directory service was implemented as a process that runs on a well known address and can be accessed by other schemas through pre-specified messages. It maintains information about peers and public schemas stored in an SQL database and it can be queried by peers about public schemas and about which public peers implement which public schemas. The schema of the data that the directory service maintains is shown in Figure 5.7 while the software architecture is shown in Figure 5.8

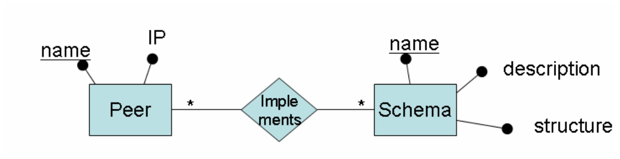


Figure 5-7 The information held by the directory service

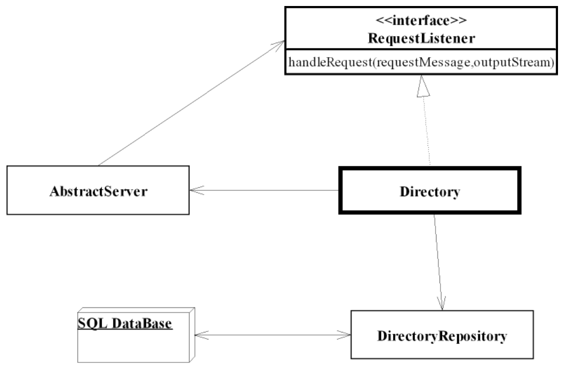


Figure 5-8 The architecture of the directory service

As it can be seen from the ER diagram in Figure 5.7 the directory service maintains the structure of public schemas in text format (see Chapter 6 for how to serialise AutoMed objects) so peers can get the structure of public schema through the directory service. Alternatively we could have implemented the system such that the directory service only contains the name and the structure of the schema can be obtained from the peer that implements it. We chose not to implement it this way for the cases where a peer publishes a schema and then leaves the network. With the current approach the schema definition stays within the network while using the alternative implementation there would be no way of locating the schema.

The directory can accept the following queries:

- get all the public schemas (name and definition)
- get the structure of a particular schema
- get the peers implementing a particular schema (name and IP)

A peer that changed host can notify the directory of the new address. The directory will alter the IP field from the peer's table.

The functionality that the directory service provides is similar to the one described in the model. In the future work section we include a discussion about using a more sophisticated directory service.

5.4.5 XML

The communication infrastructure of the system uses only XML messages (see Chapter 7) of how XML documents are used to build the messages of the protocol). Any message sent from one peer to another (and even the directory service) should be in XML format. Our decision to use XML was based on many reasons and the main ones are covered in the background section of this report.

The system was designed such that all the XML serialisation and de-serialisation is hidden from the user. XML documents are encapsulated within java Document objects (see java API). A layer of abstraction has been added to the system to give the illusion that the sender sends a java Document object to the receiver. A simplified version of the code that does that is shown in figure 5.9

```
public static void sendDocument(Document document, OutputStream out){
  OutputFormat of = new OutputFormat(document);
  of.setIndent(4);
  of.setLineSeparator("\n");
  XMLSerializer xs = new XMLSerializer(out, of);
  xs.serialize(document);
}
```

```
public static Document receiveDocument(InputStream is){
  DocumentBuilderFactory factory=
                       DocumentBuilderFactory.newInstance();
  DocumentBuilder builder = factory.newDocumentBuilder();
  return builder.parse(is);
}
```

Figure 5-9 Encapuslation of XML messages in java objects

5.5 The Final Product

The final product of this project is a working peer to peer system including a prototype graphical interface and an extensible API that provides the infrastructure for creating similar systems or extending the current one. We take a brief look at both of these aspects of the implementation.

5.5.1 A working peer to peer system

We have created a peer to peer system that can be run using a prototype graphical interface (at each peer). We also implemented a directory service that can interact with the peers. Each peer can wrap an AutoMed repository and communicate with other peers as defined in section 5.1. Each peer has the following capabilities:

- Publish one of the schemas in its AutoMed repository
- Obtain the definition of a pubic schema and store it on its repository
- Inform the directory service that it implements a particular public schema.
- Discover peers that implement a particular public schema by querying the directory service.
- Discover the public schemas of the network by querying the directory service.
- Cache information about other peers and public schemas
- Send queries to other peers expressed on a public schema and get results from the receiving peers data sources.
- Request the pathway from one public schema to another and store it in its repository, thus being able to derive pathways from its repository to public schemas using existing pathways in the network
- Request pathways and access methods from a public schema to the data sources of another peer and store this information in its repository. This way it can connect to data sources of other repositories dynamically.

- Pose “broker” queries as described in Chapter 4. The receiver of the query uses its data sources to answer the query but also forward the query to peers that it is connected to so that the query can travel around the network. There is a mechanism to avoid a query going around in cycles (as described in Section 4.3)
- Provide logging information regarding the messages it has sent for testing and monitoring purposes.
- Furthermore the system provides the ability to run multiple peers on a single host. A peer can change host and still be considered as the same peer (as long as it keeps the same name). Peers can automatically sign in to the network by notifying the directory service the IP of the host on which they reside.

5.5.2 An Extensible API

This project has built an infrastructure for peer to peer communication as described in section 5.4. In Chapter 6 we will describe an infrastructure developed for serializing and de-serialising AutoMed objects, and thus storing them from one repository to another. In Chapter 7 we will describe the framework developed for using the message format described in Chapter 4 and how to extend the current set of messages. In Chapter 8 we will describe the ways the current protocol can be extended or changed.

But even without extending the existing API, it can be used to develop more complicated systems than the prototype that was created in this project. Appendix C gives a developer’s guide to the existing API and demonstrates ways of developing more complex systems.

5.6 Things We Didn’t Solve

- There were several issues left open by this project. Some of them have already been mentioned. Here we give a summary of the problems we did not solve:
- The current system assumes that model of the schemas that are transferred from one peer to another are present in both peers. This assumption will not always be true on a real system however. The future work section discusses the problems we faced regarding this issue and suggests some ways of tackling the problems.
- We did not work on implementing any complicated caching algorithms including materialisation of queries on the peers. Our caching apart from being simple is also volatile. That is, caching information is not persisted and is lost when the peer process terminates.
- Our approach for object serialisation assumes that the two peers are using the same version of the AutoMed API. If this is not the case, storing the objects on the receiver’s repository might not work (for example when the definition of a schema changes significantly). Similarly, when we are transferring an AccessMethod from one peer to another we are assuming that the receiving peer has the definition of the wrapper required by the access method. Again, this might not be the case.
- When restarting a peer it has no way of knowing which of the schemas in its repository are public unless it contacts the directory service. We have not provided any way of flagging schemas in the repository as “public”. Similarly for derived pathways (i.e. pathways obtained from other peers) there no way of distinguishing them from local pathways. Although implementing those adjustments is not a particularly challenging thing, they are quite vital. For example a derived pathway is likely to be less reliable than a local one.
- We did not focus on things we considered to be of “no academic interest”. However for a real life application these issues need to be addressed since they can affect the performance significantly. The main examples are:

- For peer communication, we are using TCP connection and have not investigated whether other protocols could be more suitable. Generally we did not focus on optimising message transfers.
- We are using DOM parsers for parsing XML documents while a SAX parser would probably be more suitable for traversing large messages.[Xerces]
- We implemented a custom, relatively simple, centralised directory service and have not investigated how a better service could be achieved using some existing hierarchical services like LDAP or DNS.

6 Serialisation of AutoMed Objects

The requirements of this project include transferring objects stored in one AutoMed repository across the network and realising them at a different repository. This means that we need to develop an infrastructure that will allow us to serialise the particular objects in order to transfer them and then de-serialise them so that they can be persisted on the receiving end. This chapter describes an infrastructure developed that allows any AutoMed object to be serialised and de-serialised appropriately. First we explained why a new infrastructure had to be developed instead of using existing frameworks and then we describe its architecture and implementation.

6.1 Why Not Use Existing Serialisation Frameworks

Before designing the system we considered using existing frameworks for serialisation and de-serialisation the two alternatives considered were:

- Using java Object Serialisation.
- Using the existing write and read methods present in the AutoMed API.
- We give an explanation of why both these alternatives were rejected

6.1.1 Using Java Serialisation

Any java object object can be serialised if it implements the Serializable interface. Serialising an object will result in the serialisation of all its member objects (apart from the ones that are defined as transient). Since the 1.4 java platform, there a framework that enables objects to be serialised in an XML form. The class `java.beans.XMLEncoder` enables someone to encode an object in XML format:

```
XMLEncoder e = new XMLEncoder(myOutputStream);
e.writeObject(myObject);
e.close();
```

The dual class `java.beans.XMLDecoder` does the reverse. It realises the Objects from the XML files.

This at first looked like a good choice for representing AutoMed objects and transferring them over the P2P network, however we could find several problems that made this approach inappropriate.

- The framework assumes that the objects follow some `bean` conventions and AutoMed does not seem to follow the bean philosophy.

- We do not have too much control on which parts are encoded and therefore things that we might not be interested in will be encoded anyway. (like information about graphical representation of objects)
- The serialisation and de-serialisation is done blindly looking at the structure of the object. For example if an object with object id 15 is transferred across the network a new object with id 15 will be created. This would be a wrong thing to do since another object with the same id might be present on the new receiving repository.
- The encoder in our case will encode the entire object graph and will probably try to encode branches that we are not interested in creating streams that are too large and possibly hard to manage. For instance, when serialising a Schema Object it will try to serialise the model objects attached to it (if it has a field that points to the model) and then everything attached to the model and so on. For our case we want to be selective about the parts of an object we want to serialise

6.1.2 Using Existing Write and Read Methods

AutoMed provides write and read methods for serialising and de-serialising certain kinds of AutoMed objects. Unlike the Java serialisation approach, this approach takes into account the semantics of the object and the serialisation is specific to the type of object that is being serialised. You use this to transfer schemas, pathways, transformations and models. Using this approach was rejected for its luck of flexibility and its limitation to express some complex structures. Some examples are:

- Serialising a pathway assumes that extensional schemas are already present in the repository
- A pathway with *ident* transformation cannot be handled properly
- Access Methods cannot be serialised
- There is no way of extending the read/write API to include new types of objects. For a new object to be considered the write and read methods need to be written from scratch.
- The serialisation is not expressed in XML and doing and extending the current read and write methods seemed to be a difficult thing as the design does not seem to be modular.

6.2 Overall Architecture

We chose to serialise AutoMed objects in an XML format for several reasons most of which have been discussed in the background section. At the heart of the architecture is an intermediate representation we call AutoMedTree. This has a tree structure and can represent any number of objects stored in the repository. The fact that it has a tree structure makes it easier to translate into XML (which has similar structure) but also translate from XML into an AutoMedTree. The overall architecture is illustrated in figure 6.1 In order to transfer a set of java AutoMed objects from one repository to another, they first need to be represented as an AutoMedTree, which is then translated into an `org.w3c.dom.Document` object (a Java representation of an XML document). The Document object can be then serialised into an XML document and transferred across the network. Realising the XML document into AutoMed objects requires the reverse procedure. We will first give a description of the AutoMedTree design and then describe the implementation of steps 1 to 6 of Figure 6.1.

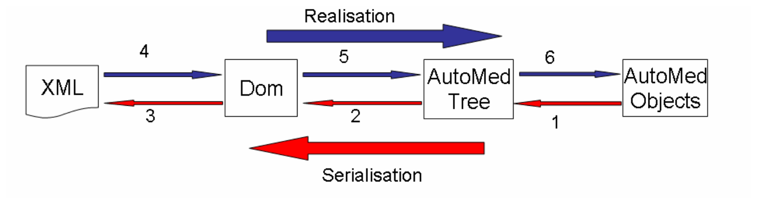


Figure 6-1 Overview of serialisation and deserialisaiton

6.3 *The AutoMedTree Object*

The *AutoMedTree* object can represent an arbitrary number of AutoMed objects. When representing an AutoMed object, in the general case, you will have to represent a number of objects that it relates to. For example, when representing a pathway, you will have to represent the schemas of the pathway (at least one extensional schema for every pathway). So we can have the notion of main objects which are the objects that need to be represented and relevant objects which are the objects that have to be present so that the information contained in the *AutoMedTree* is complete.

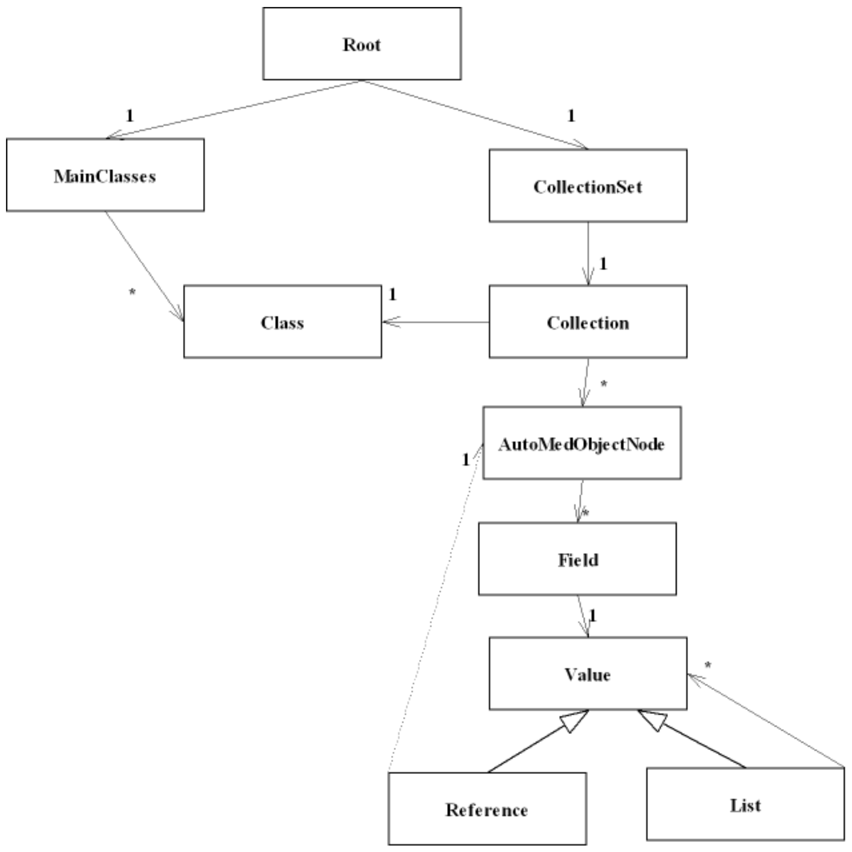


Figure 6-2 The AutoMedTree for representing AutoMed objects

The structure of the tree is summarized by Figure 6.2. The root of the tree contains two nodes. The MainClasses node describes the classes of the main objects. The CollectionSet node contains a set of collections of AutoMedObject Nodes. AutoMed objects in the tree are organized in collections according to their classes. All Schema objects for example are in the schemas collection. An AutoMed object is represented by an AutoMedObjectNode (AON). An AON holds a unique id that unambiguously identifies it within collection and a set of fields which define the structure of the object. The set of fields should be such that are sufficient to build the actual object. A field has a name and a value. The value can be a simple string value, a reference to an AON or a list of values. A reference holds the id of the AON it is referring so it is loosely references the object. A tree is said to be complete if all the referenced AONs are present in the tree.

Figure 6.3 shows an example of how an AON representing a schema is represented (using an XML notation). It has an id=11. No other schema in the tree should have the same id. The structure of a schema is defined by three fields. The first one is the list of schema objects it contains, which is represented a list of references to the objects. The other fields are the type and name of schema which are simple string values. The choice of how to represent a particular object class cannot be inferred just by looking at the code of the class. There needs to be a way of knowing which the necessary fields that need to be represented are and how the fields should be represented. The following section describes the architecture developed for achieving a flexible representation of AutoMed objects

```
<object id="11">
  <field id="schemaObjects">
    <objectList>
      <ref class="uk.ac.ic.doc.automed.reps.SchemaObject"id="55"/>
      <ref class="uk.ac.ic.doc.automed.reps.SchemaObject"id="68"/>
      <ref class="uk.ac.ic.doc.automed.reps.SchemaObject"id="94"/>
      <ref class="uk.ac.ic.doc.automed.reps.SchemaObject"id="124"/>
    </objectList>
  </field>

  <field id="type">
    <value id="0"/>
  </field>

  <field id="name">
    <value id="er_s1"/>
  </field>
</object>
```

Figure 6-3 XML representation of a schema

6.4 Representing AutoMed Objects as a Tree

We now describe the infrastructure for representing AutoMed objects as a tree. That is, transition number 1 of figure 6.1 For an object to be represented in an AutoMed tree, it has to implement the *PersistentObject* interface. This interface contains only two methods which the object should implement

```
public interface PersistentObject{
    public PersistentKey getKey();
    public Map getPersistentAttributes();
}
```

The getKey() method should return a PersistentKey (we omit details about this class) which should uniquely identify the object. For example, for a schema it should return a key containing the object id of the schema while for a pathway object it should return a key containing a list with the id's of the schemas in the pathway. In other words, the way of finding the key can only be known by the class of the object.

For example, in the class *Schema* the implementation of the method is:

```
public PersistentKey getKey(){
    return new PersistentKey(sid, Schema.class);
}
```

The *getPersistentAttributes* method returns a Map that maps field to their values. As a value can be a PersistenObject, a List or any other Object. If it is a persistent object the field value is a Reference, if it is a List the filed value is a List of values, while if it is any other object the field value is the string representation of the object. For instance in order to represent a schema object as in shown in figure 6.3 the implementation of the method should be:

```
public Map getPersistentAttributes(){
    Map map = new HashMap();
    map.put("name", this.getName());
    map.put("type", new Integer(this.getType()));
    map.put("schemaObjects", Arrays.asList(this.getSchemaObjects()));
    return map;
}
```

After implementing these two methods, a PersistentObject can be represented as a tree by calling:

```
Schema mySchema = Schema.getSchema("er_s1");
Root myTree = AutoMedTree.createTree(schema);
```

This creates a tree representation containing a description of the schema as shown and also any related objects. It does that by converting all the references to an AON until the tree is complete. The algorithm for doing that is rather complicated and will not be covered here, but what enables us to implement it is the fact that the PersistenKey of the PersistentObject will be used to build both the reference and the AON's id. Therefore you can locate an AON using the persistent key of the reference. For the example we are using, completing the tree will require to build an AON for each of the 4 references in the schemaObjects field and then make sure that all of their references are complete.

With this architecture you can represent more than one object. One example where this is vital, is when we want to represent a pathway to a data source, including the access method. The following code shows how this can be done:

```
myPathway...; myAccessMethod ...;
Root myTree = AutoMedTree.createTree({myPathway, myAccessMethod});
```

6.5 Representing an AutoMedTree as org.w3c.dom.Document object

Transforming the tree into a Document object (transition number 2 of figure 6.1) is a straight forward job. The Document object is also a tree structure and there is an one-to-one mapping between the nodes of the tree and the nodes of the Document. A simple recursive algorithm where each node in the tree defines the respective node in the Document was enough to implement the translation.

6.6 From the Document object to XML and Back

These two transitions (transitions number 3 and 4 respectively in Figure 6.1) are straight forward and were implemented using existing packages. Serialising the document object was done using the Xerces API [Xerces] while parsing of the XML document into a document object was done using the `javax.xml.parsers package` of the standard Java API distribution.

6.7 Representing an org.w3c.dom.Document Object as an AutoMedTree

This transition (transition number 5 of figure 6.1) is quite more complicated than the reverse transition (transition 2). Generally, the *realisation* path shown in the figure is much more complicated than the *serialisation* path. This is because along the realisation path we are moving from semantic-poor structures to semantic-rich structures and amongst other things we have to validate that the poorer structures are correct representations of the richer structures. Moving along the serialisation path this is always the case. So for example, although one AutoMedTree can maps to exactly one Document when moving along the serialisation path, it is not the case that a Document always represents a valid AutoMedTree.

To implement this transition we defined a *TreeBuilder* object which traverses the Document checking for its validity while trying to build the Object. An exception might be thrown if:

- The structure of the document does not reflect the structure of the AutoMedTree defined in Figure 6.2.
- The class names used in the document do not exist.
- The resulting Tree is not complete

When using the TreeBuilder, all that needs to be done to convert a Document into a Tree is call the following

```
try{
   new TreeBuilder(myDocument).createTree();
}catch(AutomedTreeException ate){
   Logger.error("The document is not a valid tree");
}
```

6.8 Creating the AutoMed Java Objects from the AutoMedTree

This is the hardest aspect of the process. We could not find a simple generic way of doing this and therefore for every class that needs to be realised we should add a specialised piece of code that will handle the object creation. Note that we found several techniques for having a generic framework of realising objects (i.e. the Persistent Object Framework) but these require designing the whole project following certain conventions that are not being followed in AutoMed.

Object creation is handled by the ObjectBuilder class which traverses the trees and tries to create the appropriate objects. The builder only works if the model of the SchemaObjects that are being built is aready part of the repository. It works by going through Collections of the MainClasses (see Figure 6.2) and trying to build the objects that correspond to each AON of the collection. When it finds a Reference node it tries to check whether the referred object has already been created. If this is not the case it goes tries to build that object before moving on and so on.

For example when it tries to build the SchemaObject of Figure 6.3 the specialized code for building schemas knows that it has to first create a schema and then add the objects of the schema one at a time. A tricky point is that some of the objects need to be build before others, for instance an entity needs to be created before any of its attributes. If the builder finds the attribute first, it will find a reference to the entity and see that the entity is not present and therefore build the entity before the attribute.

The main problem faced when implementing the ObjectBuilder class was that we had to deal with many of the details of the AutoMed implementation which affect the way objects are represented. For example, a rename transformation results in deleting the object to be renamed and adding a new object with the new name. When renaming nodal objects [BKL+04] not only the nodal object are deleted and re-added but also all the objects that are attached to it. A rename transformation is therefore applied to SchemaObjects of special type Array which encapsulates all the objects that are being removed and added again.

Other problems we faced regarded the representation of the pathway. It is represented as a list of transformations (as expected) but you cannot re-create the pathway by applying the transformations in the same order. For instance, in an ident transformation, both schemas of the ident need to be present at the time of the transformation. Therefore we need to create both subnets of the transformation first and then apply the ident transoformation. To implement this we represented the pathway as an *Abstract Network* (AbstractNetwork class) which tries to build the transformations in the correct order.

Building the objects of a Tree using the object builder requires the following code:

```
ObjectBuilder myBuilder = new ObjectBuilder(myTree);

List mainObjects = myBuilder.createObjectsFromTree();
```

This code will return a list of the realised object builder of the tree. The builder will try to name the schemas with the same name as in the tree. If there is a name conflict it will be trying alternative naming until the schema is resolved. Consequently, the object builder can give you a Map that maps names in the tree with the actual schemas created.

6.9 Capabilities of the Framework

We have created a framework that can be used to serialise and realise AutoMed objects including schemas, schema objects, transformations, pathways and access methods. The infrastructure allows the serialisation of any number of objects at the same time allowing a more flexible transfer of objects. For example, a pathway to a data source schema and its access method can be included in a single message with this approach and thus they can be realised with a single call to the ObjectBuilder as shown above. Using the framework for serialisation and de-serialisation is an easy task (as demonstrated in the previous sections).

Furthermore, the framework is extensible. It is easy to define new types of objects to be serialised. For example, to be able to serialise a Model object you just need to make the Model class extend the PersistentObject interface and implement its two methods as described in Section 6.4. Implementation of these methods is straight forward and requires defining the fields that have to be serialised in order to be able to rebuild the Model object. Then you need to extend the ObjectBuilder class to define how these fields will be used to re-build the object.

The objects are serialised in an XML format but the fact that we are using an intermediate representation (AutoMedTree) means that there is flexibility to chose different formats. What needs to be done is to change the transitions 2 and 5 of Figure 6.1 such that they serialise the AutoMedTree in the required format while transitions 1 and 6 and the structure of the AutoMedTree model remain unaffected.

6.10 Areas of improvement

We recognised three main areas imrovement

We could not find a generic way for creating the objects from the AutoMedTree class as explained in Section 6.8. This reduces the extensibility of the framework because there needs to be an extension to the definition of the ObjectBuilder class every time a new class needs to become serialisable. We believe that under the current AutoMed architecture a completely generic implementation cannot exist. However there is possibly a better way of implementing the object creation than the one currently used.

There is some redundancy in the way the objects are represented. In particular, when serialising a pathway all the schemas of the pathway are completely serialised. This is not always necessary because typically the structure of some schemas is implicit by the transformations that were applied before reaching the schema. This redundancy allows any of the schemas to be defined as extensional when they are being created, allowing more flexibility on the receiver’s side. The price is that a longer message is created which delays message transfer.

There should be a way of identifying which version of the AutoMed repository the object was serialized with and identifying whether the serialisation is compatible with the repository on the receiving end.

7 The Messaging Framework

This chapter describes the infrastructure developed to handle message passing between peers and the API developed to allow application programmers to use this infrastructure. We start by defining what the framework tries to achieve and then give a description of the design of the framework including the design of the message format. Finally we describe how the API can be used for taking advantage of the current set of message types and how to add new message types to the system. We are assuming that the reader is familiar with the concepts introduced in Section 4.1

7.1 What the Framework achieves

The framework implements the message format described in Chapter 4 and builds an infrastructure based on this format where the communication can happen in a high level where all the communication details are hidden from the application programmer. The application programmer should not work with low level messages but with application specific messages. An application specific message, or just "*application message*", is aware of the intended purpose of the message and the semantics of the information that it encapsulates providing the appropriate access methods to the information.

Our aim is be able to send an application message from one peer and have a new application message arrive to the other without being aware of any intermediate steps. For example, a peer wants to request a pathway from another peer. There should be an application message that is responsible for requesting pathways. Using this message should be as simple as possible and not include any details about the message format. So the requesting peer should do something like the one at Figure 7.1 Failure or error replies should be encapsulated into exceptions and therefore a failure reply message should throw an exception to the request making it easier for the application programmer to handle.

On the receiving end the code for handling the request should be similar to the one described at Figure 7.2. The message arrives as an application message and the concrete type of the message identifies the purpose of the message and provides the proper access methods for handling the message.

This framework enables the programmer to specify her software at a higher level of abstraction and program more on the domain level and less on the implementation level. The framework also provides a way to add new message types into the system as we will see on Section 0. Next we take a look at the design that enables the development of this infrastructure.

```
ApplicationMessage request =
            new PathwayRequest("startSchema", "targetSchema");
try{
  PathwayInformation pathInfo = request.getReply();
  Pathway path = pathInfo.getPathway();
  usePath(path);
}catch(FailedReplyException fre){
  hanldFailure();
}
```

Figure 7-1 Requesting a schema from another peer in our target framework

```
public void handleRequest(ApplicationMessage message){
  ... check the type...
  PathwayRequest request = (PathwayRequest) message;
  fromSchema = request.getFromSchema();
  toSchema = request.getFromSchema();
  ... getPathway ...
  if(success){
    new PathwayInformation(pathway).send();
  }
  else{
    new FailedOperation().send();
  }
```

Figure 7-2 Handling a request for schema in our target framework

7.2 Design of Message Format

The message format used by the system is an implementation of the language described in Chapter 4. In this section is to define how to translate the syntax described in this chapter into an XML notation. The translation should be obvious as there is an one-to-one association between the message structure of the syntax described in Chapter 4 and the structure of XML. A semi-formal description of the XML based syntax is shown in Figure 7.3. The fact that the syntax is not formal causes some ambiguities which should be resolved with common sense. A formal syntax definition which is based on the structure of Document Object Model is defined in Appendix B. Here we give the informal definition because we believe it is easier to understand and there is a direct association with the abstract syntax defined in section 4.1.

A message is of valid format if is a valid XML document and follows the constraints of the syntax defined here (the latter should imply the former). The syntax allows white-spaces between the different components and the child elements of performative can be in any order as long as the compulsory elements are present. The example used in Section 4.1.1 translated into XML is shown in Figure. Appart from the syntax, everything else is the same including the semantics of the different components of the message. We will call a message that respects this format a *"system message"*;

```
$Message::=
    <$performative>
      <context>$String<\context>
      <sender>$String<\sender>
      <receiver>$String<\receiver>
      <messageId>$String<\message_id>
      [<inReplyTo>$String<\inReplyTo>]?
      <content>$MapPairs<\content>
    <\$performative>

$Performative::= $TagString
$Map::=<$TagString>$MapPairs<\$TagString>
$MapPair::=$Map | <$TagString>$Value<\$TagString>
$MapPairs::=[MapPair]*
$Value::=$String|$Map|$List
$List::=<list>[$ListValue]*<\list>
$ListValue::=<item>$Value<\item>
$String::=list of characters
$TagString::=list of character legal for an xml tag name
```

Figure 7-3 A semi-formal description of the syntax of the message format

```
<ask-all>
      <context>pathway<\context>
      <sender>peer1<\sender>
      <receiver>peer2<\receiver>
      <messageId>labelX<\messageId>
      <content>
            <fromSchema>er_s1<\fromSchema>
            <toSchema>
                  <list>
                        <item>er_s2><\item>
                        <item>er_s3><\item>
                        <item>er_s4><\item>
                  <\list>
            <\toSchema>
            <constraints>
                  <minimum_quality>0.52<\minimum_quality>
                  <maximum_length>15<\maximum_length>
            <\constaints>
      <\content>
<\ask_all>
```

Figure 7-4 An example of an XML messages based on the syntax

7.3 Architecture of the Framework

7.3.1 Overview

The framework is designed on three layers of abstraction. The application program is only exposed to the top layer called Application Message Layer while other layers are hidden. A message arriving at a peer must go through all three layers before it arrives at the application program. As explained above any message has to be in XML format so the XML layer makes sure that the message is in XML format and it parses it as an org.w3c.dom.Document object. The abstract layer ensures that the XML obeys the syntax described in the previous section and creates an Abstract Message which encapsulates a message of the format we are using. The application extracts the meaning from an Abstract Message to create objects that are specific to the application and encapsulate the semantics. These are the objects that the application programmer handles.

We will next take a look at these layers in detail and explain how they work and how they were designed. We then explain the techniques used to hide the implementation from the lower layers and the infrastructure for installing new application message into the system.

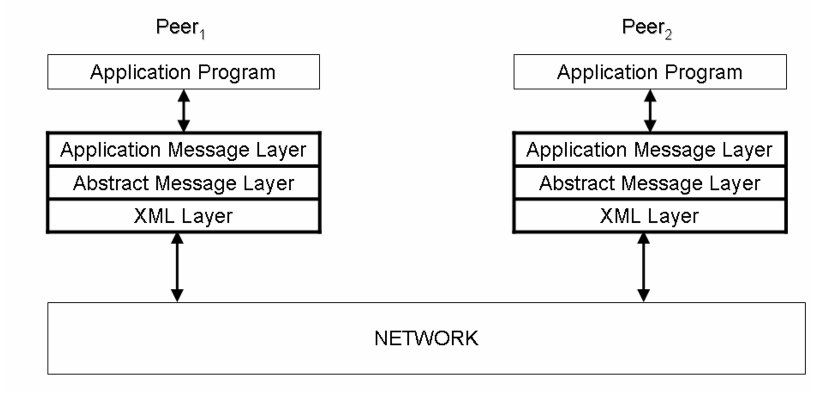


Figure 7-5 The layered architecture of the messaging framework

7.3.2 Framework Layers

7.3.2.1 XML Layer

Any valid message in our system is in XML format. On the receiving end, the XML layer gets messages as a stream of bytes and parses it as a java Document object that encapsulates the document which is passed up to the Abstract Message Layer. If the message does not respect the XML syntax it passes up a null object. When sending a message the opposite procedure occurs. The implementation details of the procedure was explained in 5.4.5. This layer does not check whether the message respects the system message format. This is done by the *Abstract Message Layer.*

7.3.2.2 Abstract Message Layer

The Abstract Message Layer ensures that the XML message respects the system message format and it creates an *AbstractMessage* that encapsulates a system message providing proper access methods to the different components of the message. An outline of an AbstractMessage code is given in Figure 7.6. Notice how it abstracts from the format of the message and only describes the different pieces of information of the message.

```
AbstractMessage{
   ...
   public String getPerformative(){...}
   public String getContext(){...}
   public String getSender(){...}
   public String getReceiver(){...}
   public String getMessageId(){...}
   public String getReplyTo(){...}
   public Map getContent(){...}

   public org.w3c.dom.Document getXMLRepresentation(){...}
}
```

Figure 7-6 Outline of the AbstractMessage

Since the AbstractMessage is not aware of the meaning of the particular message it is unaware of what the different objects in the content represent or their types. The map returned by getContent() can contain objects of the following types:

$$TypeOfMap := \begin{cases} String \\ org.w3c.dom.Element \\ List\ of\ TypeOfMap\ elements \\ Map\ of\ Type\,(String => TypeOfMap) \end{cases}$$

The type *TypeOfMap* is just used to make the definition easier. Note how the definition of the type allows for arbitrary nesting of Maps and Lists. Element is a java representation of an XML node. It is used to represent the XML structure of an *AutoMedTree* when object definitions are being sent from one peer to the other as explained in Chapter 6. This can be used to create the actual java objects as described in the previous section. Note that the syntax defined in this chapter does not allow for arbitrary Element nodes to be part of the content. However, the formal syntax of Appendix B allows for this with the rule ObjectDefinitionElement. This rule was omitted in this chapter to avoid over-complicating the syntax notation.

As an illustration of how the content is represented consider the example of Figure 7.4 and the code of Figure 7.7 and assume that the message is encapsulated by the *AbstractMessage testMessage*.

Note that the get method of a Map takes as an argument an object and uses it as a key to get an element in the map of type Object.

```
AbstractMessage testMessage=...;
Map content = testMessage.getContent();
String fromSchema = (String)content.get(“fromSchema”);
List toSchemas = (List)content.get(“toSchema”);
Map constraints = (Map)content.get(“constraints”);
String minQual = (String)constraints.get(“minimum_quality”);
String maxLength = (String)constraints.get(“maximum_length”);
String mapName = (String)constraints.get(MAP_NAME_KEY);

System.out.println(“fromSchema is : ”+fromSchema);
System.out.println(“toSchema is   : “+toSchemas);
System.out.println(“minQual is    : ”+minQual);
System.out.println(“maxLength is  : “+maxLength);
/*
Will print:
fromSchema is : er_s1
toSchema is   : [er_s2,er_s3,er_s4]
minQual is    : 0.52
maxLength is  : 15
*/
```

Figure 7-7 Illustration how content is represented

The design and implementation of the Abstract Message Layer had to deal three main issues:

- How to encapsulate messages in AbstractMessage Objects
- How to create a Document object from the AbstractMessage.
- How to create an AbstractMessage from a Document object
- We briefly describe how each of these issues is addressed.

7.3.2.3 Encaplulating Messages in Objects

An AbstractMessage includes a constructor that takes all the components of the message as a parameter. This constructor is used when you try to create an AbstractMessage to encapsulate some sort of data and its declaration looks like Figure 7.8. The constructor code just sets the appropriate fields using the parameters. The constructor is used in the expected way. For example, for encapsulating the message of Figure 7.4 the code of Figure 7.9 could be used.

```
public AbstractMessage(String performative, String context,
                       String sender, String receiver,
                       String messageId, String inReplyTo,
                       Map content)
{... construct the message ...}
```

Figure 7-8 AbstractMessage Constructor

```
Map constraints=new HashMap();
constraints.put(“minimum_quality”, new Double(0.52));
constraints.put(“minimum_lenght”, new Integer(15));
List toSchema = Arrays.asList(new String[]{“er_s2”,”er_s3”,”er_s4”);

Map content = new HashMap();
content.put(“constraints”,constraints);
content.put(“toSchema”,toSchema);
content.put(“fromSchema”,”er_s1”);

AbstractMessage msg =
   new AbstractMessage(“ask-all”,”pathway”,”peer1”,”peer2”,”labelX”,
                       null, content);
```

Figure 7-9 Constructing an AbstractMessage

7.3.2.4 Creating a Document Object from an AbstractMessage

Once the AbstractMessage is created, it is easy to create a Document object out of it. Calling the method *getXMLDocument*() creates an XML node for each of its fields which it appends on the performative element which is the root node of the document.

7.3.2.5 Creating a Document from an AbstractMessage

This is the trickiest part of the three. Remember that *not* every valid XML document is a valid AbstractMessage so there are several validity checks that need to be done. A *MessageBuilder* object tries to create the message using from the Document by going through the Document's structure and checking every compulsory component of the message is present and the values of each is according to the syntax. It should account for white space nodes which are generally inserted while serialisation in order to make the resulting text more readable. The document builder has to deal with three cases.

- The first case is when the Document object is null which means that the message was not in XML format (see Section7.3.2.1). In this case, a special abstract message with performative *invalid_message* and context *invalid_xml* is created.
- The second case is when the Document not null but it does not follow the syntax of the message format. Similar to the previous case a special it creates an

AbstractMessage with performative *invalid_message* and context *invalid_xml*. As a content it contains one field with key “documentDescription” and value the input document. Therefore proper error handling can take place.

- The last case is when the message is valid in which case the appropriate AbstractMessage is created.

Note how an AbstractMessage is created in all cases including the ones where the message is of wrong format. This behaviour is very beneficial for the Application Message Level, where it is important that we have a good framework for *error handling*.(see Section 0).

Using this framework to create an *AbstractMessage* this way is quite simple. The code of Figure 7.10 could be used. The definition of the constructor is similar to Figure 7.11.

```
Document document = getFromStream(); //from the XML layer
AbstractMessage message = new AbstractMessage(document);
```

Figure 7-10 Encapsulating a received message in an abstract message

```
public AbstractMessage(Document document){
   messageBuilder.build(document,this);
}
```

Figure 7-11 The constructor of the AbstractMessage when used to encapsulate received messages

We have seen how the Abstract Message Layer is used to create messages that encapsulate the contents of messages and abstract from the format and hiding any implementation details. We now look how these messages are used to build the Application Message Level.

7.3.3 The Application Message Level

The Application Message Level represents each type of message as an Object that has knowledge of the intended purpose of the message and provides services specialised to handle domain specific issues. The soundness of the design of this layer is dependent on two basic axioms regarding the semantics of the language which where defined in the protocol Chapter:

- The performative and context uniquely identify the intended purpose of the message
- The performative and context uniquely identify what the structure of the content should be.

Based on these axioms we achieve the following:

- By reading the performative and context we know which particular Application Message should be used
- Each Application Message knows how the structure of its message should be

These observations makes it easy to understand the design of this layer and why the approach described next is appropriate.

The aim of this layer is to allow the user define a set of application specific objects one for each type of message that the implemented protocol can handle and having each AbstractMessage being encapsulated in the appropriate object. After defining those messages then the framework should enable the user to define such objects and send them across the network to some other peer, where another object of the same type is created without being aware of AbstractMessages or any of the lower layers. In other words the peers communicate by sending each other ApplicationMessages. There should also be an infrastructure for error handling.

7.3.3.1 The Design

An Application Message should implement the *ApplicationMessage* interface which is shown in figure 7.12 These four methods are the only methods in the concrete class that betray the underlying implementation and are the methods used by the messaging framework and not by the application programmer (notice that they are protected). The rest of the methods of are supposed to be application specific.

The getPerformative and getContext functions help identify which ApplicationMessage will be used for each AbstractMessage by a process that will be described later. The getAbstractMessage returns an AbstractMessage that corresponds to the ApplicationMessage. The concrete object should therefore have knowledge of how to create the appropriate AbstractMessage (which occurs when the message is being sent). Similarly it should have knowledge of how to create the ApplicationMessage from the AbstractMessage (which occurs when the message is being received) in order to implement the *createFrom* function. There is a key point which needs to be understood regarding the *createFrom* function and it is explained next.

```
public interface ApplicationMessage{
  protected String getPerformative();
  protected String getContext();
  protected AbstractMessage getAbstractMessage();
  protected ApplicationMessage createFrom(AbstractMessage message);
}
```

Figure 7-12 The Application Message Interface

There should be *two* ways to create to create an abstract *ApplicationMessage*. The first one is the application specific way and it should be the only one that is exposed to the application programmer. For instance, creating a message of type "*ask-all pathway*" as described in Figure 7.5 might be done through a constructor that looks like Figure 7.13. The programmer should create the object in this manner in order to send an object across the network.

```
public AskAllPathway(String fromSchema, List toSchemas,
                         double minQuality, int maxPath)
{... create the object ...}
```

Figure 7-13 creating an AskAllPathway object

The second way of creating the object is through the *createFrom* function which is called by the framework when a message of the particular type arrives. To get an understanding of the use of this method, we should explain the way the framework handles incoming messages.

The framework maintains a list that contains one object from each concrete *ApplicationMessage* class in the system. (These objects act as *factory* objects of the class as we will see next) When an AbstractMessage arrives, the framework uses the performative and *context* of the message to find which is the appropriate ApplicationMessage. If no appropriate appropriate object is found for the particular message, it uses a *special* ApplicationMessage that handles such cases. The message selected is not the one actually returned but it is used as a factory object to create other objects through the createFrom method. A simplification of a code that does this whole process is given in Figure 7.14

From what was said in the previous paragraph we should conclude that there should be a way to create a *factory object*, which is the third way of creating the object. When the object is created this way, it should not hold any information other than *the knowledge of how to implement the createFrom method*. Also, it is important to appreciate that a factory object might not create an AbstractMessage of the same type! In particular, when the message content of the received message is not of the correct structure it should create an AbstractMessage that encapsulates this error instead. This brings us to the subject of error handling covered in the next section.

```
public AppicationMessage getMessage(AbstractMessage message){
   Collection factoryObjects ... ;
   ApplicationMessage factory =
         findFactory(message.getPerformative(),message.getContext();
   return factory.createFrom(message);
}
```

Figure 7-14 Finding the ApplicationMessage for an incoming AbstractMessage

```
public ConcreteApplicationMessage(){
  ... create a factory object of the class...
}
```

Figure 7-15 There should be a contructor that creates a factory object for each concrete class

7.3.3.2 Error Handling

A subtype of an ApplicationMessage is the *ErrorMessage*. An ErrorMessage encapsulates an error in the communication. The error could be of many types. Here are some examples, some of which have already been discussed in previous sections:

- The message received was not of the correct format
- The message received cannot be handled by the system
- There was a problem when implementing a request
- A request was denied
- There communication timed out
- A peer could not be reached
- ...

An error message is an ApplicationMessage that implements the ErrorMessage interface defined as shown in Figure 7.16. MessagingException is an Exception that encapsulates the type of error. The *throwException()* method can be used by the application programmer to have more natural error handling in her application. For example, requests can be sent through a special RequestSender class which throws an exception when an error message is received, as illustrated in Figure 7.17. Then making a request will look like Figure 7.1.

```
public interface ErrorMessage extends ApplicationMessage{
   public void throwException() throws MessagingException;
}
```

Figure 7-16 The ErrorMessage interface

```
RequestSender{
  ...
  ApplicationMessage sendRequest(ApplicationRequest request)
                      throws MessagingExeption
  {
      ApplicationMessage reply = myPeer.sendRequest(request);
      if(reply instanceof ErrorMessage){
           ((ErrorMessage)reply).throwException();
      }
      return reply;
  }
}
```

Figure 7-17 Using the ErrorHandling infrastructure

7.3.4 How to Install new MessageTypes in the System

It is very easy to add a new message type in the system or remove old ones. This is very important since when designing a protocol you will want a flexible way of defining the protocol messages. For this framework, all that needs to be done is to create the appropriate *ApplicationMessage* class and register it with the *message registry*. Registering it only requires to change the *MessageRegistry* class to add a new entry as shown in Figure 7.18. Note that it is easy to implement this using a configuration file, which would make the process even more flexible.

```
public class MessageRegistry{
   ...
   static{
       ... //other entries
       addEntry(new MyApplicationMessageClass()); //add new
   }
```

Figure 7-18 The Message Registry

7.3.5 How to use the API

To demonstrate how to use the API we give a simple example of the code used to send and receive the message of Figure 7.4. We are assuming that the ApplicationMessage that corresponds to ask-all queries with pathway as context is called AskAllPathway. An illustration of the code used for sending the message is shown in Figure 7.19 while the code for receiving is shown in Figure 7.20. Notice how simpler this is compared to similar operations in the Abstract Message Layer. The simplicity becomes more appreciated as the complexity of the message increases. For example when you receive a request which you want to forward you can just change the fields that you want to modify and send the same object over. You will want to do this for broker queries as described in Chapter 4.

```
//SEND REQUEST//
List targetSchemas = getRequestedTargetSchemas();
AskAllPathway askPath =
  new AskAllPathway("er_s1",targetSchemas,0.52,15);
try{
  ApplicationMessage reply = requestSender.sendRequest(askPath);
  handleReply(reply);
}catch(MessagingException me){
  ... handleException ...
}
```

Figure 7-19 Using API to send a request

```
//RECEIVE REQUEST//
AbstractMessage message = getRequest(...);
...
if(message instanceof AskAllPathway){
  askPath=(AskAllPathway)request;
  int qual = askPath.getMinimumQuality();
  ... //use access methods to get values
}
```

Figure 7-20 Using the API to handle a request

8 Protocol Implementation

This chapter describes the issues regarding the implementation of the protocol defined in Chapter 4. In particular, it explains the framework developed to allow Application programmers easily define and extend one such protocol using all the infrastructure defined in the previous chapters. We are not focusing on the algorithms used for the communication since these were covered abstractly in Chapter 4 but we are focusing on the infrastructure that was constructed in order to implement them. This chapter assumes that the reader is familiar with chapters 3, 4 and 7.

We start by discussing the general architecture describing the different key components and then explain how these relate to and integrate with the infrastructure described in the previous chapters. Then we explain the exact steps required for using the API developed in order to define new communication types and new protocols. We conclude with a case study of how one of the communication type described in Appendix A.1 can be implemented in the real system which should make apparent how to implement any of the algorithms described in the abstract protocol.

Implementing a protocol requires implementing the messageHandler defined in chapter Chapter 3. Chapter 4 describes an abstract framework for implementing this function and an actual implementation. Here, we illustrate how the architecture of the actual implementation follows closely this abstract framework and give an example of how one type of communication can be implemented.

8.1 General Architecture

In Chapter 4 we described formally an abstract architecture for implementing the messageHanlder function. The core concept of the architecture was that each message type received was delegated to an appropriate handler which could be defined separately from the main function enabling to formalise each type of communication in isolation. This approach is followed by the design of the actual system.

The assumption that enables the architecture described above is that the message type should uniquely identify how the message should be handled and the rationale for why this assumption is valid is described in Chapter 7. Similar to the model, our system maintains a collection of handlers which can handle different types of communications. For each message received the appropriate handler is selected deal with it as illustrated in Figure 8.1.

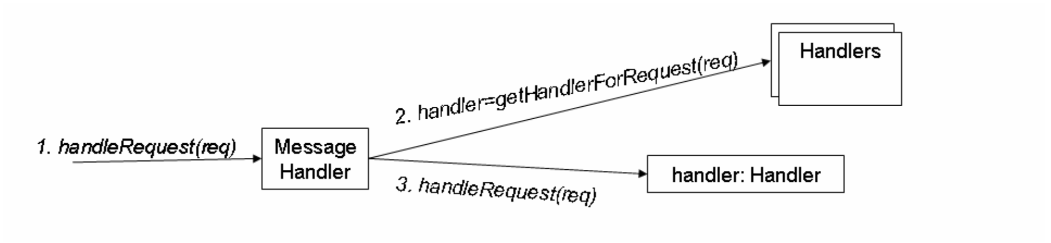


Figure 8-1 The overall structure of the message handler

Differences between the real design and the formal model do exist. In general a reply message to a request is delivered directly to the handler and it does not go through the handler selection process. Furthermore *timeouts* can be handled by the underlying system and are not part of the MessageHandler job as in the formal model. Moreover, the communication does not proceed in *rounds* like the abstract model although this does not affect the definition of the functionality significantly.

8.1.1 Handlers

Handlers come in two types: the handlers that initiate a communication (initiators) and the handlers that accept a request (receivers). Sending a request should be done through an *InitiatorHandler* while a request is handled by *ReceiverHandler*. Typically, for each type of communication we should define one of each. Handlers should have access to:

- the *Repository* of the Peer
- the Peer's cache
- the directory service
- the underlying communication infrastructure.

The *MessageHandler* is built on top of the *Application Message Layer* described in section 7.3.3. The general structure of this component in relation with the other communication components of the system is discussed in Section 5.4.3. It operates by having a collection of *ReceiverHandlers* and upon receiving an *ApplicationMessage* it uses its type (class) to find the appropriate handler to which it delegates the handling of the message.

A ReceiverHandler should implement the ReceiverHandler interface which is outlined in figure [**figure**]. The getMessageTypes method identify the types (classes) of messages which it can handled and are used by the MessageHandler to identify the appropriate ReceiverHandler for each request. The handle message does the actual handling of the message. The second parameter of the method is used to send the reply back if a reply is needed.

To initiate a communication one should use the appropriate InitiatorHandler. An InitiatorHanlder implements the InitiatorHandler interface. It typically contains information specific to the particular type of the communication it is handling. It initiates the communication and handles any subsequent messages sent or received as defined by the protocol.

```
public Interface ReceiverHandler{
  public Class[] getMessageTypes();
  public void handleMessage(ApplicationMessage message,
                            OutputStream out);
}
```

Figure 8-2 The ReceiverHandler interface

```
public Interface InitiatorHandler{
  public void initiateCommunication();
}
```

Figure 8-3 The InitiatorHandler interface

8.1.2 Using the Messaging Framework

The general architecture integrated with the Messaging Framework described in Chapter 7 is illustrated in Figure 8.2 using as example a simple request-reply communication type that comprises of the sender sending a request and receiving a reply. Notice that once the MessageHandler on the receiving end finds the appropriate handler and delegates the message handling to it, the communication proceeds between the two handlers without any more involvement of the MessageHandler.

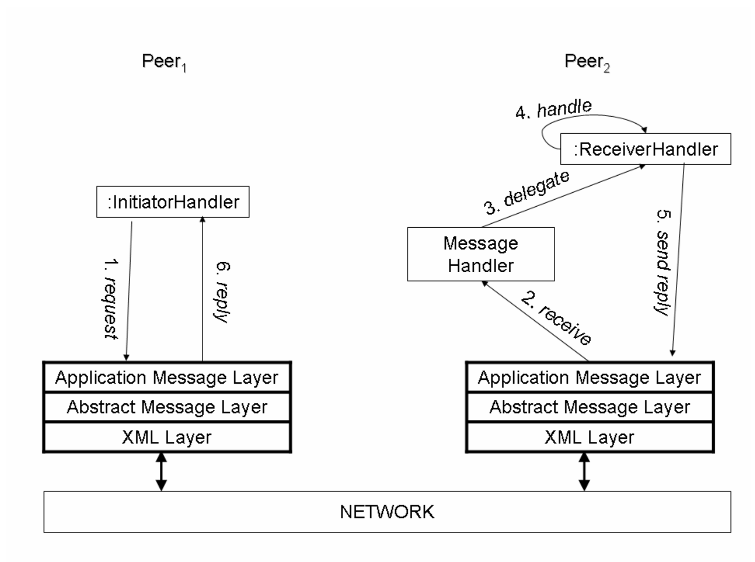


Figure 8-4 Illustration of the architecture

8.2 Flexibility of the framework

8.2.1 Adding a Communication Type

In Chapter 4 we defined the protocol as a set of communication types between peers. Each communication type defines both the type and content of the messages as well as the semantics of the conversation between the peers, including the number of messages and the reaction of the peer to each message. Therefore the ease of adding a new communication type in the network is a good measure of the flexibility of the framework that we developed.

Adding a new communication type can be done in 4 steps:

- define the new set of ApplicationMessages a
- install them into the system as described in chapter [**chapter**].
- define one InitiatorHandler and one ReceiverHandler to handle the messages according to the semantics of the new communication.
- install the ReceiverHanlder in the system

Installing a receiver handler in the system means letting the message handler know that it exists and that it can handle particular types of requests. Doing that is very easy and it is similar to the registration of ApplicationMessages in the system as in 7.3.4 All that needs to be done is add one line of code in the HandlerRegistry class.

```
public class HandlerRegistry{
...
static{
... //other entries
addEntry(new MyReceiverHandlerClass()); //add new
}
...
}
```

Figure 8-5 The Hanlder Registry

8.2.2 Extending the Cache Structure

Defining a protocol will probably require to extend the structure of the existing peer cache. As explained in Chapter 4 for the protocol we are using there no complicated caching algorithms are implemented. If a protocol requires more advanced caching policies the current structure of the cache will need to be extended. For this you will need to extend the *PeerCache* class (see Chapter 5) to add the new features. The new cache should have backward compatibility with the current one if you want the current message handlers to work properly.

8.3 Case Study

We illustrate how to use this API using a simple case study. We outline the changes that need to take place in order to add a new communication type and in particular the communication that takes place in order to for one peer to be able to request the schema definition of a public schema and store the result in its repository. The handling of this procedure is described abstractly in Appendix A.1

For this the following steps need to take place.

8.3.1 Add the new message types

There are two messages for this communication type:

	Request Message	**Reply Message**
performative	ask	tell
context	schema	schema
content	name (Name of schema)	definition (schema definition)

For this we need to define two ApplicationMessage classes. We will call them AskSchema and RequestSchema. Their definition is outlined in Figure 8.6.

```
public class AskSchema extends ApplicationMessage{
  String schemaName

  //define the constructor available to the handlers
  public AskSchema(String name){
     this.schemaName = name;
  }
  public String getSchemaName(){return schemaName};
  ... implement the other methods as described
  ... in the previous chapten
}
```

```
publc class TellSchema extends ApplicationMessage{
  AutoMedTree schemaTree; //as in chapter [**chapter**]

  //define the constructor available to the handlers
  public TellSchema(Schema schema){
    //get the Tree as in chapter [**chapter**]
    this.schemaTree()=getTreeFromSchema(schema);
  }
  public AuotoMedTree getSchemaTree(){
    return this.schemaTree;
  }
  ... Implement the ApplicationMessage methods ...
}
```

Figure 8-6 Implementation of the concrete ApplicationMessages

8.3.2 Register Messages

This is done as described in Chapter 7. For this example the code in Figure 8.7 should be used.

```
public class MessageRegistry{
   ...
   static{
       ... //other entries
       addEntry(new AddSchema());
       addEntry(new TellSchema());
   }
```

Figure 8-7 Register Messages

8.3.3 Define Handlers

The definition of the InitiatorHandler is and the ReceiverHandler is in Figure 8.8. Notice how ErrorMessages sent are encapsulated in exceptions which helps make coding the communication more natural.

```
public   class   RequestSchemaInitiator   extends   InitiatorHandler{
//contructor
  public RequestSchemaInitiator(String schemaName, Peer other){
    //build message
    AskSchema ask = new AskSchema(schemaName);
    //request schema
    try{
      TellSchema tell = (TellSchema)requstSender.send(ask);
      AutoMedTree schemaTree = tell.getAutoMedSchema();
      buildObject(schemaTree); //as explained in chapter[**chapter**]

    }catch(MessagingException me){
        ... check the type of the exception and handle it ...
    }


  }
}
```

```
pubic class ReqSchemaReceiver extends InitiatorHanlder{
   public Class[]getMessageTypes(){
     return new Class[]{AskSchema.class}
   }


   public void handleRequest(ApplicationRequest req,
                              OutputStream out){
     AskSchema ask = (AskSchema)req;
     String name=ask.getSchemaName();
     Schema schema = getSchema(name);
     if(schema==null){ //schema not found
         //this will send an ErrorMessage
         replySender.sendFailureReplyTo(ask,out);
     }
     else{
        TellSchema tell = new TellSchema(schema);
        replySender.send(tell,out);
     }
  }
}
```

Figure 8-8 Implement Hanlders

8.3.4 Register Request Handler

This is illustrated in Figure 8.9

```
public class HanlderRegistry{
   ...
   static{
       ... //other entries
       addEntry(new ReqSchemaReceiver());
   }
```

Figure 8-9 Implement ReceiverHandler

9 Evaluation and Future Work

9.1 Evaluation

The objectives set for this project defined in the Introduction were:

- To provide a formal framework for designing and analysing such *[Peer-to-Peer Integration using Both as View Rules]* networks
- To extend the AutoMed API to support P2PDI and use this API in order to develop a working system.

We now analyse the extent that this project was successful in achieving these objectives.

9.1.1 Formal Approach

"To provide a formal framework for designing and analysing such networks"

We have defined a mathematical framework that models the structure of the network and formalised semantics of the different component of the network.

However, the correspondence between the model and the real world was argued very informally and the suitability of the framework to model any arbitrary real network was left to the intuition of the reader. Furthermore the model takes a black box approach in analysing query evaluation and it does not try to analyse the query semantics in any way. Therefore it cannot be used (at this stage) to analyse correctness of query evaluation in such a system.

We formalised the *operational semantics* of the network using the model and defined their effects on the *network state* thus modelling the *evolution* of the network and the *network history*. Based on that we provided a framework which can be used to describe different protocols of peer interaction and analyse their effect on the model. We have also showed how to model different situations like link failures, network partition, message transmission delay etc.

When defining network evolution we used the simplifying assumption that the computation proceeds in synchronised rounds. This assumption appears to be oversimplified and potentially unsafe. However we provided a discussion where we demonstrated that the assumption is safe for the type of communication the system will typically use. We also showed that we can overcome this assumption by using rounds of very high frequency.

We defined a language (message format and semantics) for inter-peer communication. The benefits of the language are discussed in Chapter 4. We used this language to describe formally a protocol for P2P interactions and demonstrated some "*basic capabilities*" of the protocol which we defined as the minimum requirements that any such protocol should provide so that the potentials of the network can be harnessed. We demonstrated how to use the model to provide a formal complexity analysis which we applied at the network and got some worst case results.

The results obtained by the analysis for certain scenarios were encouraging and demonstrated how as the network evolves the peers get more knowledge of the network topology making the evolved network faster. However the results regarding cases where a peer needs to obtain a global view of the network (for example in order to discover a path to a public schema) the results of the evaluation were disappointing. In some of the cases the complexity increased exponentially with the size of the network while in other cases we provided a formal argument that the least bound in complexity is $O(n^2)$ for example for the scenario where a peer needs to find the best quality pathway to some public schema. However we illustrated how some heuristics can greatly increase efficiency and provided argument to support that in practice the worst case analysis is very rarely met.

We should admit that our analysis failed to make a convincing argument that the model will be *scalable* (for certain types of operations) for arbitrary topologies and network semantics. We argued that the problem of scalability can only be addressed at the application level where the requirements and specifications are known. For example did not cover any complex caching algorithms or data materialisation which greatly affect the efficiency of the network since this an application specific issue.

9.1.2 Implementation

"To extend the AutoMed API to support P2PDI and use this API in order to develop a working system."

The largest part of this project in terms of time consumption was the implementation of a relatively large software (over 70 java files were created) which resides in the AutoMed API providing P2PDI functionality based on the formal model we defined. The correspondence between the model and the design of the implementation was demonstrated throughout chapters 5 to 8.

We tried to build a flexible and extensible *API* which would make it easy developing different protocols. The flexibility of the different components is illustrated in the chapters that describe the implementation. Several deficiencies of the framework are discussed at the end of chapter 5 and 6. Unfortunately due to the short timeframe of the project and the focus on other things we did not find time to document the API properly which will make it hard to use by others.

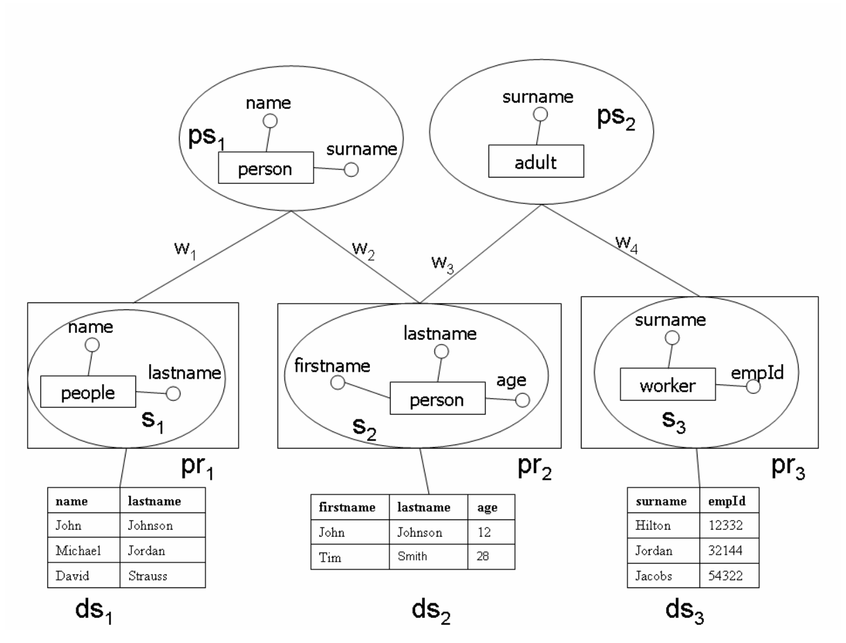
The working system works well and demonstrates the capabilities of the API and the features of the network. However it is very simple as it was design to illustrate the general features making it unsuitable to be used for any real applications. Testing was mainly focused on ensuring that the different operations described by the protocol work as expected and were not able (due to limited resources and time) to build any large enough network in order to test scalability.

9.1.2.1 Testing

The testing was based on a simple network of two simple public schemas and three peers as illustrated in figure 9.1 For building the network we followed the scenario described below:

- The three data sources (ds1,ds2,ds3) where wrapped by three AutoMed repositories one for each of the three peers (pr1,pr2,pr3) as shown in the diagram. The three export schemas (S1,S2,S3) were created (This was done using standard AutoMed tools)
- Peer pr1 created a pathway (w1) transforming s1 to ps1. Similarly pr3 transformed s3 to ps2.
- The three peers were logged into the system using the graphical tool (gui) as explained in C (was used to implement the following steps)
- Peer pr1 published ps1 and pr3 published ps2

- The two schemas were visible to pr2 and it downloaded their definitions installing both of them in its repository.
- Peer pr2 defined a set of pathways from s2 to ps1 (w2) and from s2 to ps2 (w3) using standard AutoMed tools. It then notified the directory that it implements ps1 and ps2. Peers could see who implements which schemas.
- Different peers posed queries to each other expressed on the common public schemas with the expected results.
- Peer pr1 posed a broker query to requesting the surname of a person and the result obtained contained data in ds3 which showed that the query propagated correctly.
- Peer pr1 requested a pathway from ps1 to ps2 from pr2. A pathway from s1 to ps2 could be seen in the AutoMed editor that there was a new pathway that linked s1 to ps2 (passing through ps1). It then notified the directory that it implements the schema (notice that now there is a cycle in the network).
- The same broker query was sent and the results were the same which indicated that the cycle did not affect the network.
- Peer ps1 requested a pathway pr2's data sources. The AutoMed editor shoed that there was a pathway from s1 to ds1 in pr1's repository which was used to query the datasource.
- Peer pr2 was shut down and pr3 posed a query to pr1 expressed on ps2. The query returned the expected results which demonstrated that the peers are now connected even without pr2
- Messages sent to pr2 gave an error notice saying that the request timed out.
- Peer pr2 was launched from a different host. At first, messages sent from pr1 to pr2 still gave the same error. When pr1 refreshed its cache the connection was achieved properly.



name	lastname
John	Johnson
Michael	Jordan
David	Strauss

firstname	lastname	age
John	Johnson	12
Tim	Smith	28

surname	empId
Hilton	12332
Jordan	32144
Jacobs	54322

Figure 9-1 the test network

9.2 Future Work

This project was the first implementation of a system of this nature. We identify that the work done both in terms of analysing and implementing the system is by no means complete. In this section we list the things that were either left open by this project or were opened by the analysis performed and the lessons learned during our interaction with the subject.

9.2.1 Build a Simulation

We will give special focus to an idea generated by this project and we believe it to be a very interesting research proposal. We believe that the mathematical model of the system we developed can be a basis for building a simulation of the network. This idea was described in section 3.6.3

The model might need to be refined to abandon the synchrony assumption. I believe that this can be achieved relatively easily. We need to change the definition of *NetworkHistory* to:

$$NetworkHistory : \mathbb{R} \rightarrow NetworkState$$

which means that the network evolution occurs in dense time. Sending a message should occur after some delay which will allow us to ensure than no two actions happen at the same time. Simulating such a system can be achieved by discrete event simulation techniques, where sending a message or applying an update will generate an event.

We now give a very brief overview of the rest of the future issues, many of which have been described during the report. We group them in two sets the first one consist of issues of research interest while the second is more engineering and implementation related.

9.2.2 Research Issues

- Analyse and implement different caching algorithms to test their effectiveness under different situations
- Determine what is the correct definition of pathway quality (introduced in Chapter 3) and how to measure it.
- Using a single directory service might be restrictive. Analyse how the model should change to allow for more than one directory and the consequences that this would bring.
- Explore how the searching algorithms followed by Piazza where peers can upload information about their data in order to optimise the locating of peers and distribution of queries.
- Investigate the possibility of removing the directory from the network and distributing its information amongst the peers in line the P2P philosophy.
- Analyse how this system can be used to allow updating the data of other peers instead of just reading it.
- The network assumes that all peers are co-operating and it does not deal with malicious peers. Investigate how such a system can deal with misbehaving participants

9.2.3 Implementation Issues

- Extend the system to allow peers locate which peers are live in the network. Several techniques used for traditional P2P system could be employed.

- Implement the directory service on a more robust and hierarchical framework like LDAP.
- The current system cannot cope in the cases where the model of any schema received is not in its repository. Try to find a solution to this issue considering that transferring the model along with the schema might not be scalable and that the same model is often represented differently in different repositories.
- Explore the possibility of integrating the inter-peer communication with WebServices. The fact that the communication is XML based will probably make the integration easier.

9.3 Conclusion

This project was the first attempt to develop a system of this nature. We identified from the beginning that producing a complete result would be impossible within the given timeframe. I viewed the project as a journey where I learned many lessons and gained some insights to the subject which I tried to share in this report and summarised in this chapter. I tried to look at the problem from many points of view which generated some new ideas that could be explored in the future. I suggested formal frameworks for modelling and analysing the system which are by no means complete but could become a starting for researching around this area. I also created a working system and an API that I claim to be useful and extensible. However as we have seen in these chapter all of these achievements are only partly successful.

Personally, I would consider this project a success if the ideas described in the report can become a useful reference for future work on the subject.

Appendix

A Communication Types of Protocol

We will describe our protocol as a set of communication types and the behaviour of the peers for each of them.

A.1 Communication Type: Request Schema

A.1.1 Communication description

The initiator of the communication (client) requires the definition of a public schema from the receiver of the message (server).

The client sends a message containing the public schema name ($schema_{\&}$) and the server should reply with the structure of a message ($schema$ where $SchemaName(schema_{\&}) = schema$). The receiver should add this information to its local knowledge.

A.1.2 The Messages

	Request Message	**Reply Message**
performative	ask	tell
context	schema	schema
content	name (Name of schema)	definition (schema definition)

A.1.3 Definition of Handlers

Extra state information: On the client's side, we map messageId's to schema names. That is, the id of the message that made the request to the name of the schema that it requested:

$reqSchNm : ID \rightarrow Schema_{\&}$

```
makeRequestHandler::
  peer;
  state = (round,pendingRequests,reqSchName)
  mail = NULL; //not hanlding a mail. starting a fresh request
  sname; //the name of the schema

  //send one message
  mailBox={ ask[schema]{name:sName}>>server }

  //change the state
  reqSchNm(msgId)=sname; //msgId is the id of the message above
  pendingRequests(msgId)=round+3; //after 3 rounds times out

  //no change in local or global knowledge
  updates={};

  return(updates,mailBox,state);
```

```
receiveRequestHandler::
  peer;
  state;
  mail;

  sName=mail.content.name;

  if(sName==null){
    mailBox={ error[schema]{description:"wrong format"}>>sender }
  }
  else{
        //can only return schema within its local knowledge
        $sch = \begin{cases} SchemaName(sName) \text{ if } peer.pp(sch) \neq undefined \\ undefined \;\; otherwise \end{cases}$
        if(sch=undefined){
          mailBox={ fail[schema]{description:”cant find”}>>sender  }
        }
        else{
          mailbox={ tell[schema]{definition:sch}>>sender  }
        }
  }
  updates={}

  return(updates,mailBox,state}
```

```
receiveReplyHandler::
  peer;
  state;
  mail;

  if(mail.performative=fail/error){
    //error handling
  }
  if(mail=NULL){
    //remove relevant pending messages that timed out
  }
  if(mail.performative=tell){
    reqId=mail.inReplyTo;
    sName=reqSchNm(reqId);
    schema=mail.content.definition;
    updates={ public({schema}) }; //add new public schema
    mailbox={};
  }
  return(updates,mailbox,state)
```

For the first communication type we have defined the handlers quite thoroughly considering error cases and timeouts. These are standard procedure and will therefore be excluded from now on.

A.2 Communication Type: Make a query

A.2.1 Communication description:

The initiator of the communication (client) requests by the receiver (server) the evaluation of a query expressed on a public schema.

The client sends a message containing the query and the server should reply with the result of the evaluation of the query.

A.2.2 The Messages

	Request Message	**Reply Message**
performative	ask	tell
context	query	query

content	schema (query schema) question (the query)	result (query result)

A.2.3 Definition of Hanlders

Extra State information: on the client's side, we map messageId's to queries.

$queries : ID \rightarrow Question$

```
makeRequestHandler::
  peer;
  state;
  mail = NULL; //not hanlding a mail. starting a fresh request
  sname; //the name of the schema
  qn; //the query

  //send one message
  mailBox={ ask[query]{schema:sname,question:qn}>>server }

  //change the state
  questions(msgId)=sname; //msgId is the id of the message above
  return({},mailBox,state);
```

```
receiveRequestHandler::
  peer;
  state;
  mail;

  sName=mail.content.name;
  schema=SchemaName(sName); //assuming in local knowledge
  question=mail.content.question;
  query=(question,schema);

  publicPath = peer.pp(schema);
  query2 = transQ(query,pathway);//expressed on export schema
  pathways = {peer.dp(ds)|ds∈DSchema}
  result2 = eval*(query2,pathways); //expressed on export schema
  result = transR(result2,publicPath);//expressed on public schema
  mailbox={ tell[query]{result:result}>>sender  }

  return({},mailBox,state}
```

```
receiveReplyHandler::
  peer;
  state;
  mail;
  perform_necessary_checks_and_maintainance();
  updates={ ?? }; //possibly cache this request
  return(updates,{},state)
```

A.3 Communication Type: Request Pathway to DataSources

A.3.1 Communication Description

The initiator requests a set of pathways from a particular public schema to a set of data source schemas.

The client sends the name of a schema and the receiver sends a set of pathways from that schema to all the data source schemas that are within its local knowledge.

A.3.2 The Messages

	Request Message	Reply Message
performative	ask-all	tell-all
context	final_pathway	final_pathway
content	schema (the public schema)	paths (list of pathway definitions)

A.3.3 Message Handlers

```
makeRequestHandler::
  peer;
  state;
  mail = NULL; //not hanlding a mail. starting a fresh request
  sname; //the name of the schema
  //send one message
  mailBox={ ask-all[final_pathway]{schema:sname}>>server }
  return({},mailBox,state);
```

```
receiveRequestHandler::
  peer;
  state;
  mail;

  sName=mail.content.schema;
  schema=SchemaName(sName); //assuming in local knowledge
  publicPath = peer.pp(schema);
  pathways={append(publicPath,fpw)|ds∈DSchema ^ fpw=peer.dp(ds)}
  mailbox={ tell-all[final_pathway]{paths:publicPaths}>>sender  }
  return({},mailBox,state}
```

```
receiveReplyHandler::
  peer;
  state;
  mail;
  perform_necessary_checks_and_maintainance();
  pathways=mail.content.paths;
  //add all pathways in local knowledge
  updates={new_path(pw)|pw∈pathways};
  return(updates,{},state)
```

A.4 Communication Type: Request Linking Pathway

A.4.1 Communication Description

The initiator requests a pathway from one public schema to another.

The client sends the name of the two public schemas, ps1 and ps2. The server gets the pathway form its export schema to ps1, w1, and the pathway from its export schema to ps2, w2, and sends w=(w1|w2). The client's can now link to ps2 through (w1'|w) where w1' is the pathway from the client's export schema to ps1.

A.4.2 The Messages

	Request Message	**Reply Message**
performative	recommend	tell
context	pathway	pathway
content	fromSchema (public schema name) toSchema (target schema name)	path (the pathway)

A.4.3 Definition of Handlers

```
makeRequestHandler::
  peer;
  state;
  mail = NULL; //not hanlding a mail. starting a fresh request
  fromName; //public schema
  toName;   //target schema
  //send one message
  mailBox={
```

```
recommend[pathway]{fromSchema:fromName,toSchema:toName}>>server }
  return({},mailBox,state);
```

```
receiveRequestHandler::
  peer;
  state;
  mail;

  fromName=mail.content.fromSchema;
  toName=mail.content.toSchema;
  fromSchema=SchemaName(fromName);//assuming in local knowledge
  toSchema=SchemaName(toName); //assuming in local knowledge

  path1=peer.pp(fromSchema);
  path2=peer.pp(toSchema);
  linkPath=(path1|path2);
  mailbox={ tell[pathway]{path:linkPath}>>sender  }
  return({},mailBox,state}
```

```
receiveReplyHandler::
  peer;
  state;
  mail;
  perform_necessary_checks_and_maintainance();
  pathway=mail.content.path;
  //add pathway in your local knowledge
  updates={new_path(pathway)}

  return(updates,{},state)
```

A.5 Communication Type: Request Implementers of Schema

A.5.1 Communication Description

The initiating peer requests from directory service Dir the names of peers that implement a public schema.

The requesting peer sends Dir the name of the public schema. Dir sends a list with the names of the peers that implement those schemas based on its global knowledge (cache).

A.5.2 The Messages

	Request Message	Reply Message
performative	recommend-all	tell-all

context	peer	peer
content	schema (public schema)	peers (list of peers)

A.5.3 Definition of Message Handlers

Extra State information: On the client's side, we map messageId's to schema names. That is, map the request id to the schema name that the request relates to.

$schemas : ID \rightarrow Question$

```
makeRequestHandler::
  peer;
  state;
  mail = NULL; //not hanlding a mail. starting a fresh request
  sName; //name of schema

  //send one message
  mailBox={recommend-all[peer]{schema:sName}>>Dir }
  //update state
  schemas(msgId)=sName;
  return({},mailBox,state);
```

```
receiveRequestHandler::
  peer=Dir;
  state;
  mail;

  sName=mail.content.schema;
  peers=peer.cache(sName);
  mailbox={ tell-all[peer]{peers:peers}>>sender  }
  return({},mailBox,state}
```

```
receiveReplyHandler::
  peer;
  state;
  mail;
  perform_necessary_checks_and_maintainance();
  reqId=mail.inReplyTo;
  schName=schemas(reqId);

  peers=mail.content.peers;
  updates={impl(schName,peers)}
  return(updates,{},state)
```

A.6 Communication Type: Request All Public Schemas

A.6.1 Communication Description

The initiating peer requests from directory service Dir the names of all the registered public schemas

The requesting peer sends Dir the request. Dir sends a list with all the public schemas that it knows of using its global knowledge.

A.6.2 The Messages

	Request Message	Reply Message
performative	ask-all	tell-all
context	schema_list	schema-list
content	---	schemas (list of schemas)

A.6.3 Definition of Hanlders

```
makeRequestHandler::
  peer;
  state;
  mail = NULL; //not hanlding a mail. starting a fresh request

  //send one message
  mailBox={ask-all[schema_list]{}>>Dir }
  return({},mailBox,state);
```

```
receiveRequestHandler::
  peer=Dir;
  state;
  mail;

  schemas={ps∈PSchema|peer.cache(ps)≠undefined)
  mailbox={ tell-all[schema-list]{schemas:schemas}>>sender  }
  return({},mailBox,state}
```

```
receiveReplyHandler::
  peer;
  state;
  mail;
  perform_necessary_checks_and_maintainance();
  schemas=mail.content.schema;
  updates={public(schemas)}

  return(updates,{},state)
```

A.7 Communication Type: Advertise Pathway

A.7.1 Communication Description

The initiating peer informs the directory service Dir that it can provide a pathway to a particular public schema.

The requesting peer sends Dir the request containing the name of the schema. Dir sends a confirm/deny message depending if the update was done successfully.

A.7.2 Messages

	Request Message	Reply Message
performative	advertise	confirm/deny
context	pathway	pathway
content	schema (name of schema)	---

A.7.3 Definition of Hanlders

```
makeRequestHandler::
  peer;
  state;
  mail = NULL; //not hanlding a mail. starting a fresh request
  sName; //name of public schema
  //send one message
  mailBox={advertise[pathway]{schema:sName}>>Dir }
  return({},mailBox,state);
```

```
receiveRequestHandler::
  peer=Dir;
  state;
  mail;

  sName=mail.schema;
  peerName=mail.sender;
  updates={ impl(sName, {peerName})  }

  mailbox={ confirm[pathway]{}>>sender  }
  return(updates,mailBox,state}
```

```
receiveReplyHandler::
  peer;
  state;
  mail;
  perform_necessary_checks_and_maintainance();

  if(mail.getPerformative==deny){
    //should handle this
  }
  return({},{},state)
```

B Message Syntax

Message ::= *PerformativeElement*

PerformativeElement ::=

ElementNode(*Name*){*ContextElement*, *SenderElement* *ReceiverElement*, *MsgIdElement*, [*InReplyToElement*]$^{?}$, *ContentElement*}

MapValue ::= *MapValue*(*Name*)

MapValue(*mapName*) ::= **ElementNode**(*mapName*){[*MapElement*] *}

KeyedValue ::= **ElementNode**(*Name*){*Value*}

MapElement ::= *MapValue* | *KeyedValue*

Value ::= *TextValue* | *MapValue* | *ListValue*

TextValue ::= **TextNode**(*String*)

ListValue ::= **ElementNode**("**list**"){[*ListItem*]*}

ListItem ::= **ElementNode**("**item**"){*Value*}

ObjectDefinitionElement ::=

ElementNode("**object_definition**"){*XMLNode**}

String ::= `Non empty string of characters`

Name ::= `A` *String* `that is a valid tag name`

SimpleKeyedValue(*text*) ::= **ElementNode**(*Name*){*TextNode*(*text*)}

ContextElement ::= *SimpleKeyedValue*("**context**")

SenderElement ::= *SimpleKeyedValue*("**sender**")

ReceiverElement ::= *SimpleKeyedValue*("**receiver**")

MsgIdElement ::= *SimpleKeyedValue*("**msgId**")

InReplyToElement ::= *SimpleKeyedValue*("**inReplyTo**")

ContentElement ::= *MapValue*("**content**")

XMLElement ::= **ElementNode**(*Name*){*XMLNode**}

XMLText ::= **TextNode**(*String*)

XMLNode ::= *XMLElement* | *XMLText* | `...any other` *XML* `node...`

C Guide to the Graphical Tool

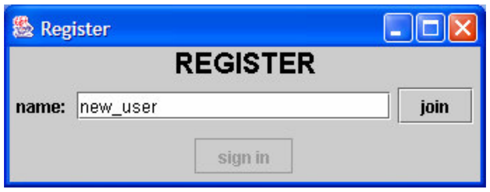
A simple graphical tool which is used as a prototype to test and demonstrate the functionality of the underlying framework was developed. We give a simple guide of how to use the tool to perform several tasks.

C.1 Starting the Application

To run the application properly a Registry must be running on the local host. You can run the registry from package `uk.ac.ic.doc.automed.p2p.P2PRegistry` from the AutoMed API. The application itself can be run from `uk.ac.ic.doc.automed.p2p.gui.PeerApplication`.

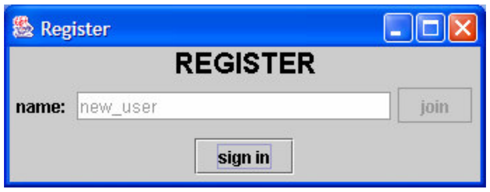
C.2 Join the local host

When the application starts you a dialog box like the one below will appear. You will have to chose a peer name and register with the local Registry by clicking the join button.



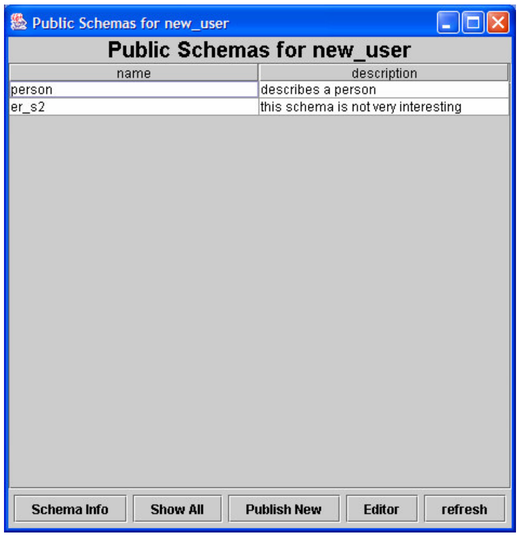
C.3 Sign in the Network

Once you join the registry the sign in button will become enabled. This will notify the directory service that you are logging in from the particular host using the chosen name. If you are not registered with the directory a register button will appear and clicking it will register and sign you in.



C.4 The Main Frame

Once you sign in the main frame of the application will appear. This looks like the one below and provides several buttons that will be described later. It displays the public schemas that the peer is implementing according to the directory service. In this frame and generally in all of the frames the information displayed comes from the peer's cache which might not be up-to-date. You can refresh the cache by clicking the refresh button which is present in all the frames of the application. We will call this frame "main frame".

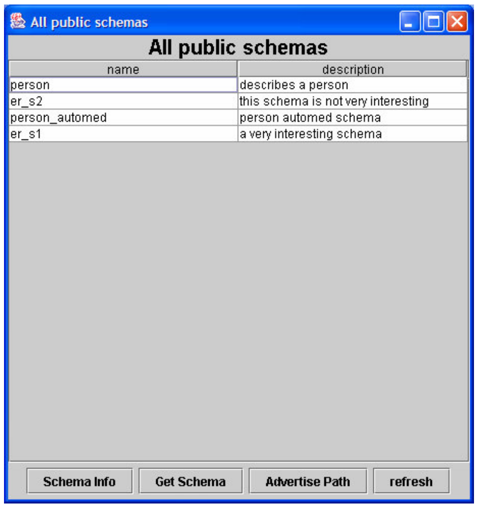


Public Schemas for new_user

name	description
person	describes a person
er_s2	this schema is not very interesting

C.5 View All Public Schemas

To view all the public schemas in the directory service you need to click on the "Show All" button in the main frame. The frame below will appear. For each public schema it provides the name and a description of the schema. We will call this frame "all-schemas frame".

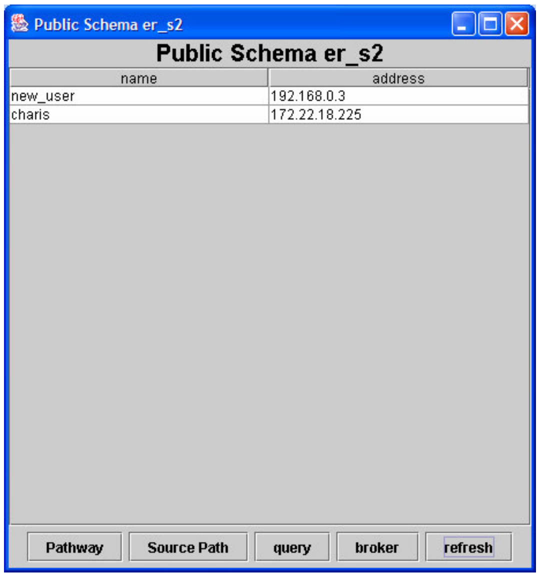


All public schemas

name	description
person	describes a person
er_s2	this schema is not very interesting
person_automed	person automed schema
er_s1	a very interesting schema

C.6 View Peers Implementing a Schema

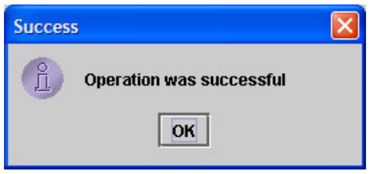
For each of the public schemas you can view the schema information by clicking the "Schema Info" button either in the main frame or the all-schema frame. The following frame will appear. The peer information is a list of peers that implement the public schema each entry containing the peer's name and the address of the machine that it is currently residing. There is no guarantee that any of the peers displayed is live at the time. We call this frame "schema info frame".


Public Schema er_s2

name	address
new_user	192.168.0.3
charis	172.22.18.225

C.7 Get Schema

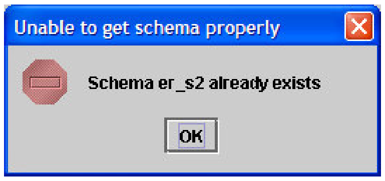
You can store the peer information from any public schemas in your AutoMed repository. To do that, select the schema you like from the all-schema frame and click on the "Get Schema" button. When schema is stored you will get the following notification.


Success

Operation was successful

C.8 Error Handling and Logging

If there was a problem with an operation that is supported by the tool you will get an error dialog notifying you of the fact. There is also a logging system that logs the exceptions thrown and other error messages and gives some useful monitoring and debugging information. At the time that this document was written the logging was done on the system output but it can be easily extended to provide a more user friendly logging.


Unable to get schema properly

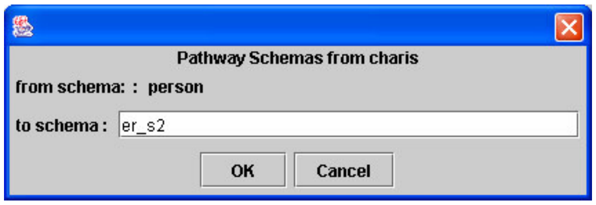
Schema er_s2 already exists

C.9 Advertise Pathway to a Public Schema

You can notify the directory service that you are providing a pathway to a public schema by selecting the schema on the all-schema frame and clicking the "Advertise Path" button. After doing that and go to the main frame and press refresh you will see the schema added to the list of schemas that you implement.

C.10 Get a Linking Pathway

You can get a linking pathway from one public schema to any other by finding a peer that implements both the source and target schema, go to the schema info frame, select the peer that has the linking pathway and click on the "Pathway" button. This will bring up the box shown below. You will need to fill in the target schema name and click OK. If the operation is successful the pathway will be installed in your repository and you will therefore can provide other peers with the linking path as well. If you had a pathway to one of the schemas of the pathway you can now advertise that you also have a pathway to the other schema.


Pathway Schemas from charis

from schema: : person

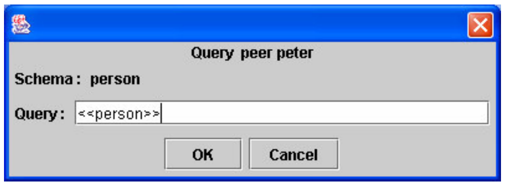
to schema : er_s2

C.11 Get Pathway to Data Sources

You can get a pathway from a public schema to a set of data sources of another peer. To do that you need to go to the schema info frame of the public schema that you are interested in. Select the peer which you want to get the pathway from, click on the “Source Path” button and wait for the result. When the result confirmation message arrives the pathway will be installed in your repository including all the necessary AccessMethods to the data sources.

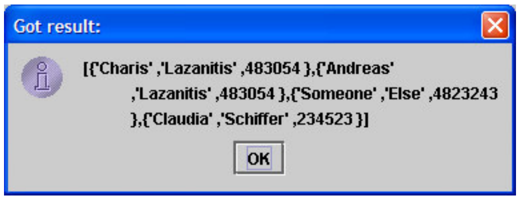
C.12 Send a Query

You can pose a query to a peer expressed on a pubic schema and get a reply from the receiver’s data sources. To do that, go to the schema info frame of the schema, select the peer you are interested in and click on the “query” button. The following dialog box will appear. Type the query click OK and wait for the result. The results are displayed in a dialog box as shown below. If there is an error with the query you will get a message notifying you of the nature of the error.


Query peer peter

Schema : person

Query : <<person>>


Got result:

[{'Charis' ,'Lazanitis' ,483054 },{'Andreas' ,'Lazanitis' ,483054 },{'Someone' ,'Else' ,4823243 },{'Claudia' ,'Schiffer' ,234523 }]

C.13 Broker Query

A Broker Query is one where the receiving peer answers it as a normal query but also forwards it to other peers and adds the results together before it sends the reply. The procedure to request a broker query is the same as the normal query. You just click on the “broker” button instead of the “query” button.

C.14 Launching the AutoMed Editor

Clicking the Editor button on any of the frames launches the editor tool which is included in the AutoMed API for viewing and modifying objects in the repository. The tool will immediately display any changes on the repository that come about during the peer’s participation in the network

D How to Set Up a Peer to Peer Network

To set up a peer to peer network you need a directory service and a set of peers that know the location of the directory service. How to run a peer is explained in Appendix [**appendix**]. Here we will explain how to run the directory service and how to let peers know where the directory service is.

D.1 Running the Directory Service

The directory service can be run from class `uk.ac.ic.doc.automed.p2p.directory.Directory`

Before you run the directory for the first time you will need to initialise the Directory repository from the class:

D.1.1 `uk.ac.ic.doc.automed.p2ps.directory.DirectoryRepository`

D.1.2 Configuring the Directory

There should be a file in the `.automed` folder in your home directory. There needs to be a file called `directory.cfg` which contains 5 lines of text as shown below.

```
cpu2.doc.ic.ac.uk
jdbc:postgresql://db.doc.ic.ac.uk:5432/myDB
myUserName
myPassWord
org.postgresql.Driver
```

The first line is the location of the Directory, which is the address of the machine that the directory service process is running on. The subsequent lines give information about accessing the SQL server that the directory is accessing and are the URL, user name, password and the driver name respectively. These lines are used for the directory to locate where its data is located. For the directory service to run properly the last four lines should be set to the appropriate values.

The first line is for peers to locate the directory service. The service is known to run at port `8383` and therefore there the internet address (which could be an IP address) is enough to locate the service.

D.2 Setting up the Network

Make sure that the directory is running properly at some address *dirAddress*.

All peers of the network should have *dirAddress* as the first line in the `directory.cfg` file described above.

The peers can run as described in Appendix C and they will automatically join the Network.

References

[APHS02] K. Aberer, M. Punceva, M. Hauwirth and R. Schmidt. *Improving data access in P2P systems*. IEEE Internet Computing, 2002

[Aust69] J.L. *Austin How to Do Things with Words*. 1969

[Automed] http://www.doc.ic.ac.uk/automed/ AutoMed website

[BB04] Leopoldo Bertossi and Loreto Bravo. *Query Answering in Peer-to-Peer Data Exchange Systems.*

[BBK+01] Frank Babek, Emma Brunkill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, Hari Balakrishan. *Building Peer-to-Peer Systems With Chord, a Distributed Lookup Service*. 2001

[BKL+04] Michael Boyd, Sasimol Kittivoravitkul, Charalambos Lazanitis, Peter McBrien and Nikos Rizopoulos. *AutoMed: A BAV Integration System for Heterogeneous Data Sources*. In Proc. of CaiSE2004.

[CGL+04] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzenini, Riccardo Rosati, and Guido Vetere. Hyper: *A Framework for Peer-to-Peer Data Integration on Grids.*

[CGLR04] D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. *Logical foundations of peer-topeer data integration*. In Proc. of PODS 2004,

[Cla99] I. Clarke. *A distributed decentralised information storage and retrieval System*. Master thesis, University of Edinburgh, 1999.

[CPRM02] Enna Z. Coronado Pacora and Manuel Rodrigues Martinez. *SRE: Search and Retrieval Engine of TerraScope Earth Science Information Systems*. 2002

[CRB+03] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, Scott Shenken. *Making Gnutella-like P2P systems Scalable. In Proc. of ACM SIGCOMM* 2003

[CSWH01] Ian Clarke, Oskar Sandberg, Brandon Wiley and Theodore Hong. Freenet: *A Distributed Anonymous Information Storage and Retrieval System*. Anonymity 200, LNCS 2009, 2001

[DBK+01] *Building Peer-to-Peer Systems With Chord, a Distributed Lookup Service.*

[DistNet] http://www.distributed.net. distributed.net website

[DOM]	http://www.w3.org/DOM/ The w3c Document Object Model.
[FFM+94]	Tim Finin, Richard Fritzson, Don McKay, Rodin McEntire. *KQML as an Agent Communication Language*. In the Proceedings of the Third International Conference of Information and Knowledge Management (CIKM94)
[FKL+04]	Enrico Franconi, Gabriel Kuper, Andrei Lopatenko, and Ilya Zaihrayeu. *The coDB Robust Peer-to-Peer Database System*. In the proceedings of The Second Workshop on Semantics in Peer-to-Peer and Grid Computing 2004
[GHI+01]	Steven Gribble, Alon Halevy, Zachary Ives, Maya Rodrig, Dan Suciu. *What Can Databases Do for Peer-to-Peer?*
[Gnutella]	http://gnutella.com Gnutella website.
[GSB+01]	Steve Graham, Simeon Simeonov, Toufic Boubez, Doug Davis, Glen Daniewls, Yuichi Nakamura, Nyo Neyama. Davis, DaneBuilding *Web Services with Java.* SAMS publishing. 2001
[HIST03]	A. Halevy, Z. Ives, D. Suciu and I. Taranov. *Schema Mediation in Peer Data Management System*. In ICDE, 2003.
[Len02]	Maurizio Lenzerini. *Principles of peer-to-peer data integration*.
[Lenz02]	Maurizio Lenzeniri. *Data Integration: A Theoretical Perspective*. In Proc. of ACM PODS 2002
[Losh02]	Pete Loshin. *Big Book of Lightweight Directory Access Protocol (LDAP) RFCs.* Morgan Kaufman, 2002.
[LS02]	Qin Lv, Sylvia Ratnasamy, Scott Shenken. *Can Heterogeneity Make Gnutella Scalable?* 2002
[McB03]	Peter McBrien. *Advanced Databases*. Ch 5. Lecture Notes for course 312 in Department Of Computing,Imperial College, November 2003
[McB03b]	Peter McBrien. *Networks and Communications*. Lecture notes for course “Networks and Communications” for the Department Of Computing, Imperial College. January 2003
[McB04]	Peter McBrien. *AutoMed in a Nutshell*. Technical Report.
[McB04b]	Peter McBrien. *P2P Data Integration Project*. Briefing Notes.
[MP03]	Peter McBrien and Alexandra Poulovassilis. *Defining Peer-to-Peer Integration using Both as View Rules.* Proc. DBISP2P, at VLDB
[MP03b]	P.J. McBrien and A. Poulovassilis. *Data integration by bi-directional schema transformation rules*. In Proc. ICDE2003. IEEE 2003.
[NN92]	Hanne Riis Nielson, Flemming Nielson: *Semantics with Applications: A Formal Introduction*, Wiley 1992

[PM98] Alex Poulovassilis and Peter McBrien. *A general formal framework for schema transformation*. Data and knowledge Engineering, 1998

[Poul04] Alex Poulovassilis. *A Tutorial on the IQL Query Language*. Technical Report. 2004

[Ripe01] Peer-to-Peer Architecture Case Study: *Gnutella Network*. In Proc of the First International Conference on Peer-to-Peer Computing (P2P'01)

[SAL+03] M Stonebraker, P.M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin and A. Yu. Mariposa: *A Wide-Area Distributed Database System*. VLDB, 2003

[Smith88] Reid G Smith: *The contract net protocol: high-level communication and control in a distributed problem solver*. 1988

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In Proc. of SIGCOMM'01

[TH04] Igor Tatarinov, Alon Halevy. *Efficient Query Reformulation in Peer Data Management Systems*. SIGMOD 2004

[TIM+03] Igor Tatarinov, Zachary Ives, Jayant Madhavan, Alon Halevy, Dan Suciu, Nilesh Dalvi, Xin Dung, Yana Dadiyska, Gerome Miklau, Peter Mork. *The Piazza Peer Data Management Project*. SIGMOD Record Vol. 32, No. 3, 2003

[UDDI] IBM. *Publishing your Webservices:UDDI.*

[Wool02] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley and Sons, 2002.

[Xerces] http://xml.apache.org/xerces-j/ The apache Xerces project.