Towards Semantically Enhanced File-Sharing

Alan Davoust and Babak Esfandiari
Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada
Email: {adavoust, babak}@sce.carleton.ca

Abstract— **We characterize publication and retrieval of structured documents in peer-to-peer (P2P) file-sharing systems, based on the abstract notion of community, encompassing a shared document schema, a network protocol and data presentation tools. We present an extension of this model to manage multiple communities, and to describe relations between documents or communities. Our approach is based on the idea of reifying complex concepts to structured documents, then sharing these documents in the P2P network. The design of our prototype P2P client involves components interacting asynchronously using the blackboard model. This decoupled architecture allows the system to dynamically extend its query processing functionality by creating new components that implement the processing described in downloaded documents.**

I. Introduction

Distributed applications using a peer-to-peer (P2P) architecture have become a very active field of research, mostly because of the scalability offered by the decentralized control model typical of this architecture.

A hugely popular application of this paradigm is *Peer-to-peer File-Sharing* which allows users to locate files of interest in a P2P network and retrieve them to their local system. The infrastructure required for such an application is minimal : a protocol to propagate and answer queries across the network, and a file transfer protocol such as `ftp` or `http` to retrieve files. Generally, no assumptions are made about whether all query answers are really relevant to the users information needs, or that all the relevant documents are listed in the search results.

A first natural extension to this approach is to share structured data in a P2P network: by managing structured data the infrastructure can support more expressive queries and take up the role of a middleware layer between applications managing structured data.

In this work we describe a P2P application which can manage arbitrary schemas, with what we consider to be the defining purpose of file-sharing systems, which is data *retrieval*, i.e. the two-step process of locating then downloading a remote data item. Another defining feature of P2P file-sharing systems is that each peer has full control over its local repository, and populates it by publishing or downloading data from its peers.

In previous work [1] we have contrasted this approach with P2P database systems (Peer Data Management Systems, PDMS) which are rather meant to unify a (P2P-) distributed set of databases into a single large knowledge repository, over which expressive queries can be answered using the data in place.

Our prototype application U-P2P is essentially a schema-based P2P File-sharing system, where the shared data items are structured XML documents. We have extended the basic model of schema-based file-sharing in two ways. First, U-P2P offers the possibility of managing multiple schemas within the same application.

In the spirit of P2P file-sharing, our approach allows end-users to create new schemas, or discover and download the schemas published by others, then publish or download data structured according to the new schemas, and thus be able to manage a rich variety of data. This extension was first presented in [2].

The second extension of our system, presented here, addresses the problem of enhancing U-P2P to manage semantic data. A semantic enhancement to Peer-to-peer file-sharing would make this paradigm suitable for the distributed and emergent creation and sharing of any kind of knowledge. Such an enhancement would require the ability to annotate documents, relate documents to one another, qualify such relations with formal properties, navigate the relations to surf from one document to another, and infer implicit relations. The main challenge is to do so in a purely distributed manner.

Our current solution allows users to describe relations between documents using different data representations, and to perform basic navigation of these data relations. The flexible architecture of our system, based on a tuplespace, allows easy extensions, such as integrating semantic reasoning. We describe here the design of our system, and our method to dynamically extend its functionality.

The rest of this article is organized as follows. We first briefly present related work, then clearly define the P2P File-Sharing concepts that we use as basis for our data model and general approach. In section IV we define the type of relations that can be expressed in our data model, and the corresponding navigation query. Finally in section V we describe the tuple-space based design of our system, and the process of defining a new relation.

II. Related work

AmbientDB [3] is a schema-based P2P infrastructure meant to act as a communication middleware between various applications, such as household appliances managing music (an IPod, a computer, a cell phone, etc.). The schema-based approach, which could be described as a "database with distributed tables", is fairly close to our basic model for U-P2P. The purpose of AmbientDB, in

doi:10.4304/jsw.4.8.787-797

terms of query processing, is to manage relational queries over the database. While peers may come and go, the global data schema is static.

Distributed Hash Tables (DHT), such as Chord [4] or CAN [5], are distributed storage systems, and support optimized data retrieval based on unique identifiers. DHTs require a structured network, and do not give peers full control of their local storage space, since a system-wide optimization algorithm determines the storage location of the documents published to the shared repository.

Peer Data Management Systems (PDMS), such as Piazza [6], are database systems related by P2P schema mappings. They support expressive query answering over the schema, with some constraints on the network topology. As we have mentioned in the introduction, such systems are designed for expressive query answering over static data, as opposed to the File-Sharing purpose of replicating the data. Furthermore, in contrast to our approach of managing multiple schemas, in PDMS each peer manages a single relational schema which it relates to its neighbors schemas by semantic mappings, in order to propagate queries across the mappings. We have, however, explored the idea of mapping schemas in U-P2P in [7].

As we will discuss in section V-D, schema mappings can be considered to be a particular type of *relation* between documents, for which we hope to provide navigation through the new approach presented in this paper. This specific scenario is, however, future work.

Edutella [8] is a slight variation between the principles of file-sharing and of PDMS. Edutella is a PDMS where each peer maintains a local database of RDF metadata about educational resources, such as books or online courses. Edutella is built on a super-peer topology, where the super-peers collaborate to distribute and rewrite complex queries to the "leaf" peers.

The answers to the queries are the URLs of the educational resources of interest to the user, which can then be retrieved by a separate protocol such as HTTP, based on their URLs. Despite the ultimate file-sharing goal, the data of interest to Edutella – the meta-data – is static, which conforms to the principles of PDMS.

Bibster [9] is a schema-based P2P file-sharing system to share bibliographic data, based on the widely used Bibtex format. Bibster uses a rich ontology to support expressive queries and topic-related browsing, which could be from a user perspective the ultimate goal of our project U-P2P. However, Bibster is specifically tailored to the domain of bibliographic data, and all the semantic knowledge in Bibster which supports the expressiveness of the querieswas input by the designers of the system. End-users can simply contribute their own bibliographic data, but cannot contribute to building the *ontology* of bibliographic data.

The fact that the end user should be able to contribute the data at all levels providing new schemas, defining new relations is an important aspect of our approach. From this perspective our work can be related to the so-called Web 2.0, which is characterized by a shift towards user-produced content. This shift is embodied by wikis, social networking, "tagging", and online repositories of content uploaded by end-users, such as Flickr[1] or YouTube[2].

An important difference between the Web 2.0 approach and the P2P approach is that in a Wiki web site, or on Flickr, users do not have the ultimate control over the data they publish. Registration to online repositories comes with a contract, where the host typically asserts its right to edit or modify data at its discretion, and *use* it with few restrictions.

In a P2P file-sharing system a peer may lose control of what happens to *copies* of its data – just as Flickr loses control of photographs downloaded by end-users – but the peer retains full control of the data in its local repository. The data stored by a peer cannot be modified by another user; if other peers download and modify a document, then multiple versions will co-exist in the network.

Sweet Wiki [10] is a project of a community-built repository of semantic data, based on the "wiki approach. While we do not claim that U-P2P supports a full-fledged semantic knowledge base, our approach is still comparable: users of Sweet Wiki may define new concepts and populate them with instances of the concept.

The general principles of Linked Data, as outlined in [11], also parallel our work, as the fundamental principle of Linked Data is the definition of URIs, and of links based on these URIs, to describe relations between data items of different repositories.

III. Model of P2P File-Sharing

A. Basic Model

We first present a general model for schema-based P2P file-sharing systems, which we will use as basis for our discussion. A full formal definition of these concepts can be found in [12].

1) Documents: In traditional file-sharing systems, the data items which are managed by the system are files, with their name, extension, date, and size, as meta-data attributes. In addition, a file has a binary "payload". In these systems, search queries are usually understood to be keyword-matching searches on the file name.

In some specialized file-sharing applications dedicated to sharing music such as Napster[3], some music-specific meta-data attributes are extracted from the shared files and can be used as search filters.

In our model, peers can share data structured according to any schema, and we define the notion of *document* to model a uniquely identifiable data item, representable by a set of attributes. These attributes can be either meta-data (with string or numerical values) or else binary.

This representation allows us to represent a complete file by a single abstract data item, rather than separating the file from its metadata, as is done in many knowledge representation approaches. For example, a music file can

[1] http://flickr.com
[2] http://www.youtube.com
[3] http://www.napster.com, no longer existing in its original form

be represented by a document containing both the meta-data attributes (artist, title, etc.) and the binary "payload", which is in our case simply an attachment.

We define two classes of attributes : *meta-data* attributes, and *attachments*, which are binary. We also distinguish one meta-data attribute with a special significance: this attribute called *documentId* is a unique identifier of a document, in the sense that two documents with different content (i.e. any different attributes) will not have the same identifier. In our prototype application, the identifier is system-generated using the one-way hash function MD5 [13].

2) Community: We define a *community* as an abstract grouping tool to access a collection of documents sharing the same schema.

A *Community* has:

- A unique name;
- A data schema, which is the schema of any document shared in this community;
- A protocol, which connects the peers who store documents shared in the community: a protocol can be defined as a function which maps a peer to a set of reachable peers. Notation: we note $prot(p)$ the set of peers accessible to p through the protocol $prot$.
- A set of data presentation templates which can be used by the application to present the documents shared in the community to a human user, or to generate a query interface based on the community schema, for example.

A community is the foundational structure that supports the sharing of documents in a P2P File-sharing system. In practice, a traditional file-sharing application such as Kazaa[4] or Limewire[5] defines a *community*, with its own schema and protocol. We note that several communities may have the same protocol.

In the following, we will say that documents are *shared-in* communities, meaning that they are described by the community schema and stored by a peer which is a *member of*[6] the community; and we will use the notation D_C to represent the set of all documents shared in the community C. We will refer to the unique name of a community as its *communityId*.

3) Peer: A peer is an abstract entity representing a human user, which interacts with a community and stores part of D_C. A peer has a unique identifier, and some storage space (a database or file system for example). A peer may *join* a community, which is typically done by downloading an application (e.g. Limewire) implementing the community protocol and capable of answering queries over the community schema. The peer may then contribute documents to the community: such documents must conform to the community schema and be placed in the peer's local repository.

Notation: we will note D_C^p the documents of community C stored on peer p.

4) Operations: A peer p which is *member of* a community C offers an interface to C consisting of the following operations:

- `publish(document` d`)`: d is added to D_C^p.
- `delete(document` d`)`: if d exists in D_C^p, then d is removed from D_C^p[7]
- `search(expression` $expr$`)`, where $expr$ is a boolean expression over attributes of C:
  The result of the search is a list of pairs $(p_k,\ d_i)$ defined by the following conditions:
  - $expr$ evaluates to *true* over d_i;
  - the peer p_k is in $prot(p)$ and d_i is in $D_C^{p_k}$

  The function returns a virtual view over the collection of search results $\{(p_k, d_i)\}$, containing only the metadata attributes of each document.
  Intuitively, the search results leave out the attachments, which form the bulk of each document, while showing the attributes which are directly human readable. The user will decide which documents to download according to the partial view offered by the metadata.
- `download (document` d`, peer` p_2`)`: if d exists in $D_C^{p_2}$ and p_2 is part of $prot(p)$ then d is copied to D_C^p.

5) Example: We consider the community *Cinema* which is a community to share documents representing movies. The basic schema of the community is $(title, director, year, data)$, where *title*, *director*, and *year* are meta-data attributes, and *data* is an attachment storing the actual video file[8].

Peers P_1 and P_2 are members of this community. P_1 stores the document *nosferatu* with the attributes *(nosferatu, 'Nosferatu', 'F. W. Murnau', 1922, [nosferatu.mov])*, P_2 stores the documents *metropolis* and *twotowers* with the attributes *(metropolis, 'Metropolis', 'F. Lang', 1927, [metropolis.mov])* and *(twotowers, 'The two Towers', 'Peter Jackson', 2002, [towers.mov])*, respectively. The square brackets indicating that the value of the attribute *data* cannot really be represented here. In the following examples we will leave out the *documentId* from the tuple-representation of documents, using instead the notation: document *documentId* with attributes $(a_1, a_2 \ldots a_n)$.

B. Fundamental Properties of P2P File-Sharing Systems

P2P File-sharing systems — a set of peers interacting sharing documents within one or several communities, using the operations defined above — share a number of properties:

[4]http://www.kazaa.com, no longer existing in its original form

[5]http://www.limewire.com/

[6]this notion will be defined in the next subsection

[7]we note that the document is only removed from the local repository of peer p.

[8]according to our definition, the binary content of an *attachment* attribute is part of the *document*; in this example it may seem that the document simply *references* a video file which would be separate, but conceptually the attribute *value* is the full binary file and not a reference, so the video file cannot be dissociated from the *document*. In fact this specification leaves the implementation open, in the sense that the video file could be directly included into an XML document as a #CDATA section, for example.

- the system is *unstructured* : we consider that peers are free to connect with any peer they choose, and hence we can make no assumption regarding the topology of the resulting network;
- the system is highly dynamic: peers may join or leave at any time, creating a potentially high level of churn;
- each peer has full control over its local storage, i.e. peers store documents on their local storage space because they choose to, as opposed to DHTs for instance, where the storage location of each document is decided according to an optimization algorithm.
- 'best effort' query answering : At any given time, a peer accessing a community through searches and downloads, can only access documents stored by peers which are *reachable* via the community protocol at that time. This limited view of the community implies a 'best-effort' query answering, typical of P2P file-sharing.

We consider that these properties best reflect the decentralized control which is the basis of the "peer-to-peer" paradigm.

C. Communities as "Active Documents" with application-specific semantics

1) Reifying the Concept of Community: The concepts and discussion in this paper are implemented in a prototype file-sharing application, U-P2P (for Universal Peer-To-Peer, first presented in [2]), which allows peers to simultaneously interact with several communities, and even to create new communities by defining new document schemas.

This feature relies on the idea of "reifying" the concept of a community, i.e. describing the community in a *community definition document*, in order to share this document with other peers.

Such *community definition documents* are structured according to a schema — communityId, schema, protocol — and can be shared in a special bootstrap community based on this particular schema. Peers can join and search this community — the *Community* community — in order to discover or create (and *publish*) new communities. We note that the *communityId* is used as *documentId* in the community definition document.

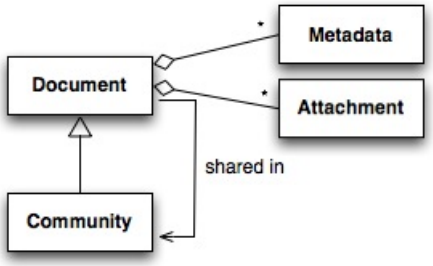

Figure 1. Data Model of U-P2P

Figure 1 illustrates how communities become special documents by this "reification" process. Based on this principle, a peer can join a community simply by downloading the community definition document; thisdocument effectively provides an interface to the community, in the form of the community protocol and schema.

2) Example: We extend the example in section III-A.5. Peers P_1 and P_2 must now be members of the *Community* community, and store locally a copy of the document defining the community *Cinema*. In addition, P_1 may now create a new community $Actors$ in which documents describing Actors are shared. The community schema is $(name, bio, photo)$, where $name$ and bio are meta-data attributes (bio is a short text about the actor), $photo$ is an attachment containing a photograph of the actor. P_1 also includes a rendering template which allows users to view a nice multimedia document with a description of the actor and her picture. In this community P_1 stores the documents $depp$ with attributes *('Johnny Depp', 'Johnny Depp was born on 09th of June 1963, and...', [jd.jpg])*, and $livtyler$ with attributes *('Liv Tyler', 'Liv Tyler is the daughter of... ', [lt.jpg])*. The data stored by the peers is illustrated in Figure 2.

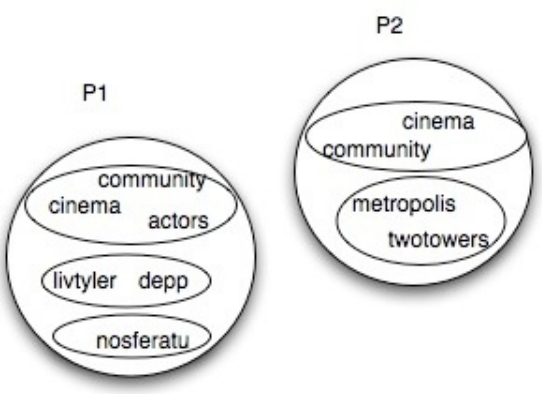

Figure 2. Example peers and documents stored by these peers in a scenario with multiple communities. Within each peers, the documents are grouped by community.

3) Semantics and Active Documents: In our model, *Community definition documents* have the special property that peers automatically join a community simply by downloading and *interpreting* such documents.

This interpretation – and its reverse operation of "reifying" the definition of a community to a document, imply a special semantics associated with the *Community* community.

Concepts of the material world (e.g. films or actors) can be described in structured documents, but the file-sharing application does not in any way manage the semantics of such data.

Our approach is to enable the application, our prototype U-P2P, to selectively manage the semantics of U-P2P specific concepts, i.e. to be capable of interpreting a specific class of documents with these semantics.

This class of documents, encoding *behavior*, can thus

be considered to be "active documents", as defined by [14].

Community definition documents are the first example of this class of documents: a U-P2P client will for example extract schema information in order to provide a query interface tailored to the community schema, and use data presentation templates to render documents in a community-specific way.

In this work, we further extend our model by defining a second class of "active documents", that will encode *query processing directives*. Users will be able to publish such documents, U-P2P clients will be able to interpret such documents and automatically extend their functionality.

IV. Related Documents

Our multiple-community model allows us to share documents with different schemas, which in a more knowledge-representation perspective, allows us to model instances of different concepts. Our users in the running example of this paper can share *movies* as well as documents about their favorite *actors*. The next step in knowledge representation will be to be able to model relationships between entities.

Our solution to this problem is to relate the documents modeling these entities. In this section, we show how our model can be further extended to manage relations between documents; this includes both the issues of representing and querying these relationships.

A. A graph of related documents

We consider a model of documents with attributes, related by binary relations. We use the abstract mathematical definition of a relation : a relation R is a set of couples (A, B), where A and B are documents. To each relation R we associate a *label* L_R.

All the documents and their relations then form a directed graph, where the nodes are documents, and there is an edge with a label L_R from a node A to a node B iff (A, B) is in R.

a) Example: In the running example of this paper, featuring *Movies* and *Actors*, we could define for example a relation with the label *Sequel*, which would contains couples of movies, and a relation with the label *stars-in*, with couples from $Actors \times Movies$ (cartesian product). An example graph of related documents is illustrated in Figure 3.

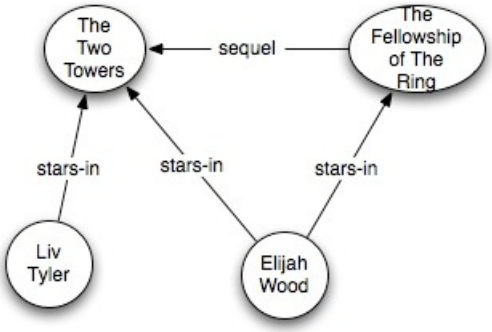

Figure 3. Example graph of Documents and Relations

B. Representing Relations in the U-P2P Data Model

1) A U-P2P-specific URI Scheme: The first point to be noted is the fundamental requirement to describe relations between any entities is a *naming* scheme such as a URI [15] to identify these entities. In our setting, this means that *documents* must be named, or identified in an unambiguous way.

As noted in section III-A.1, each document of a community has a special attribute *documentId* that uniquely identifies a document within this community. With each community being uniquely identified within the *Community* community, by the *comunityId*, we thus have a two-level hierarchical naming scheme, which allows us to unambiguously refer to any document.

In our multiple-community framework we thus obtain the following URI scheme, which was introduced in [12]: `up2p:`*communityId*`/`*documentId*. A specially interesting characteristic of this scheme is that it identifies *any copy* of a replicated document, and we will use a search-based dereferencing mechanism (described below) that will allow us to retrieve any available copy of the document. This way we can retrieve a document even if the peer that first published it has disconnected from the network, as long as it has been downloaded (and is made available) by other peers.

Based on this addressing scheme, we introduce an additional type of attribute, which we will call an *endpoint* attribute. The values that endpoint attributes can take are well-formed `up2p:` URIs.

The simplest way of representing a Relation is to use endpoint attributes. If a document A in community C, has an endpoint attribute named "a-name" with the value `up2p:communityId2/docIdB`, then this attribute can represent the fact that the document A and the document B with the documentId "docIdB", in the community identified by the communityId "communityId2", are related by a relation with the label "a-name". To *uniquely* identify this relation, we can use the identifier of C in combination with the endpoint attribute name "a-name".

2) Example: In our running example about movies and actors, the relation 'stars-in' could for example be represented by a multi-valued endpoint attribute in the schema of the *Actors* community. The community schema would then be : *(name, bio, photo, acts-in*$^+$*)*, the $+$ indicating that the attribute is multi-valued. The document representing Liv Tyler could then be : *('Liv Tyler', 'Liv Tyler is the daughter of... ', [lt.jpg],* `up2p:Cinema/twotowers`*)*

3) Alternative Representations: We note that the use of endpoint attributes is a requirement to represent relations, but this particular representation is not the only possibility. In some cases, documents will exist without any linkage, and creating the link a posteriori should

still be possible, for example by representing the link in an external document, in the spirit of the Resource Description Framework (RDF, [16].

The link between some documents A and B (with label *lbl*) would be represented by a third document L in a special community "RDFLinks" with the following schema:$(documentId, doc1, label, doc2)$. The document L would then be : *(L,* `up2p:C/A`*, lbl,* `up2p:communityId2/docIdB` *)*.

We consider it important not to commit to particular representation choices, as the purpose of this system is to accomodate new documents and schemas input by end users, whose ideas and choices we cannot predict and do not wish to constrain.

4) Dereferencing a URI: The different representation that may be chosen for a relation will be important if we want to support automated queries of the graph of documents, but the basis of any algorithm will the URI dereferencing mechanism, which will always be the same:

1) if the peer is a member of the community *comId*, then the peer may jump directly to step 4.
2) *Community*.search(communityId=*comId*); [results include $(p_j, comId)$]
3) *Community*.download$(pj, comId)$;
4) *comId*.search(documentId=docId)

This search-based dereferencing mechanism has a high cost, but we note that (a) the communityId significantly reduces the search space and (b) not identifying a document by its location provides the opportunity of getting any one of multiple copies, which makes the dereferencing more robust to churn (i.e. if a peer disconnects the documents it was storing are not necessarily unavailable).

C. Navigation

In this section we discuss how users may query the graph of documents.

We consider that the most basic information needs of the users do not require the full expressiveness of a graph query language such as SPARQL [17], and we consider for now the most simple graph query, which we will define as the *navigation* query. This query can be expressed as follows:

Definition 1: Navigation query: *Navigate(document A, Relation R)*: given a document A, a relation R, find the documents B such that (A, B) is in R.

This query allows users to *browse* the graph of documents: view a document A, follow a relation R to a related document B.

In the next section we will discuss how such navigation can be supported in a P2P file-sharing system.

D. Supporting Navigation in U-P2P

Answering the *navigation* query *Navigate(A,R)* in U-P2P may involve different processing, depending on the representation of R. It may be for example:

- dereferencing an 'endpoint' attribute of A
- doing reverse navigation, i.e. searching all endpoint attributes of potential B documents, for references to A.
- if the relation is described in a document L separate from A and B, then searching the community (or communities) storing candidate documents for L, then then dereferencing URIs listed in L to obtain B.

Any of these query processing algorithms can be easily implemented using the elementary *search* operation defined in our original model (see section III), and the dereferencing operation[9] described in section IV-B.4.

In order to avoid committing to a particular representation for Relations, our solution is to define navigation algorithms as scripts, i.e. sequences of *search*, *download*, and *dereference* operations, in documents which can be dynamically loaded and interpreted by U-P2P (active documents). We detail this solution the following section.

V. Extensible Query Processing based on a Tuple-Space Design

In this section we describe the design of our prototype application U-P2P, which implements a multiple-community File-Sharing application, extended to support navigation across links, as presented in section IV.

We first describe the implementation of the fundamental file-sharing operations presented in section III-A.4. These fundamental operations constitute simple building blocks that can be combined to perform more complex processing.

Our design in based on a tuple-space architecture, where the three major components of our application — user interface, local database, and network adapter — interact to perform these basic operations.

This extensible design allowed us to seamlessly integrate the additional processing necessary to support navigation queries, by adding software agents interacting with the tuple-space, which in turn use the existing building-block operations.

Finally, we will show how this architecture can support dynamic addition of agents to further extend the query processing functionality of the application.

A. Basic P2P functions

The high-level design of U-P2P comprises three major components, interacting via a tuple-space:

- A user interface collects input from the user and submits queries to the tuple-space. When the user performs searches, the User Interface generates a unique identifier for this query, which is used to track the query and the responses to it; when responses arrive, the user interface stores a temporary collection of Responses.

[9] we note that the dereferencing procedure itself only involves searches and downloads, which shows that the entire navigation process will eventually be implemented by the basic operations search, download, publish, remove.

- The local repository stores documents of the communities that the peer is member of, and responds to searches, and retrieval queries (documents or document attribute values).
- A Network Adapter propagates the queries to other peers reachable through community protocols, and also inputs queries originating from other peers (to be evaluated against the local repository). The Network Adapter also performs Downloads (retrieves documents stored by other peers), and outputs the retrieved documents into the tuple-space in Download Response tuples. The Adapter maintains a "Routing table" of Search and Download Requests in order to return responses to the peer that originated the corresponding request.

Each of these three components inputs and outputs tuples to the tuple-space via a *proxy agent*, which manages a set of templates specific to the component. Each time that a tuple in the tuple-space matches a template of the agent, the tuple is retrieved and processed by the agent. Responses, if any, are output into the tuple-space.

The architecture of U-P2P is illustrated in Figure 4.

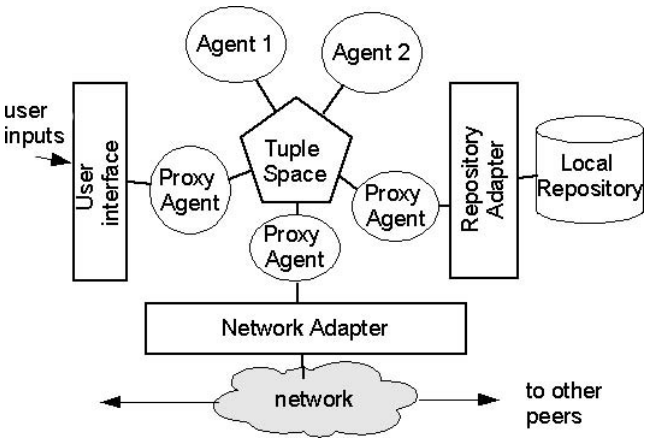

Figure 4. Architecture of U-P2P

Queries and responses are encoded as tuples with a verb in the first field, and appropriate parameters in the following fields. Searches have a special query identifier field that is generated by the user interface of the peer that outputs the original query, and used to track and route responses appropriately.

The fundamental P2P File-sharing operations are implemented as tuples exchanged by the different Proxy Agents. The Tuple representation of the requests and responses, as well as the agents which may output each type of tuple are listed in table I.

Each of these Request and Response tuples will trigger certain processing from the different Proxy Agents. The processing triggered by each tuple for the different Proxy Agents is as described in the two-entrance table II. N/A indicates that the Proxy Agent does not use a template matching the tuple.

Proxy Agents that process tuples do not remove these tuples from the Tuple-space, but they store a copy of each processed query until it disappears from the tuple-space. An additional agent, the Cleaning Agent, monitors all the tuples in the tuple-space and removes them after a short lifetime in the tuple-space. The lifetime is an adjustable parameter of the application; it must be long enough for all agents to read the tuple at least once, and short enough for tuples to be removed before another identical Tuple is likely to appear. This situation is unlikely to occur with only the basic Agents present, but in dynamic extensions where new agents may be created, this type of "collision" must be avoided.

B. URI Dereferencing and Navigation

The extended model presented in section IV, which adds the concept of Relations to connect documents in a graph, introduces the additional elementary operation *Dereference (URI)*, and the complex *Navigation* query.

In order to support URI dereferencing, a new type of tuple has been created, the Dereference Request Tuple. This tuple has the following fields: *("DereferenceURI", uri, queryId)*. An additional Agent has been created, which listens for "DereferenceURI" request tuples, and translates them to searches as described in section IV-B.4, and outputs these Search Request tuples into the tuple-space. The Search Request tuples have the same query identifier as the input DereferenceURI query.

The simplest navigation, involving only a URI dereferencing, is now supported by U-P2P: the user interface generates the Dereference Request Tuple, with the appropriate URI parameter and a new queryId, outputs it to the tuple-space, then listens for Search Request responses with the same queryId. The open architecture of the tuple-space allows such extensions to be very easily integrated into the application.

In order to support navigation of Relations that require more complex processing, we have chosen to provide indirect support by dynamically creating new agents which implement the appropriate sequence of searches, downloads, and dereference requests. For this, repeating our approach of reifying communities into active documents, we define a special community $Agents$, for which we associate a special semantics: this semantics is materialized by a schema by which query processing directives can be specified to be implemented in the agent.

U-P2P then *interprets* the agent definition (its query processing directives) at runtime, creating new agents in interaction with the internal tuple-space.

C. Agent Definition Language

In order to define a Navigation algorithm, we have defined a simple tree schema (implemented using XML) to define the behavior of an Agent as a set of forward-chaining rules based on the Linda coordination language [18]. Each rule is formed by a *Head*, which is a template matching tuples that will activate the rule, and a *Tail*, which is a sequence of tuple-instructions (tuple inputs and outputs) to be executed when the rule fires. Each

Operation	Tuple Representation	Output by
Publish	('Publish', $communityId$, $Document$)	user interface
Remove	('Remove', $communityId$, $documentId$)	user interface
Search Request	('Search', $communityId$, $expr$, $queryId$)	user interface, network adapter
Search Response	('SearchResponse', $communityId$, $documentId$, $peerId$, $Document*$, $queryId$)	repository, network adapter
Download	('Retrieve', $peerId$, $communityId$, $documentId$)	user interface, network adapter
Download Response	('RetrieveResponse', $communityId$, $documentId$, $Document$)	repository, network adapter

TABLE I.
FUNDAMENTAL FILE-SHARING OPERATIONS IN A TUPLE SPACE ARCHITECTURE.

Tuple / Agents	**User Interface**	**Repository**	**Network Adapter**
Publish	N/A	Document stored in the local repository of the community identified by *communityId*	N/A
Remove	N/A	document identified by *documentId* removed from local repository of community identified by *communityId*. If the document is not present, nothing happens.	N/A
Search Request	N/A	The query *expr* is evaluated against local repository of community *communityId*; for each matching document *d*, a SearchResponse tuple is output.	If the agent is currently listening for Responses to the same Request (identified by *queryId*), then the Request is ignored. If not, the Request is sent to other peers through the network, according to the network protocol of the community *communityId*.
SearchResponse	If the agent is currently listening for Responses with this *queryId*, then the temporary collection of Search Responses identified by the *queryId* is updated with the new Response	N/A	If the agent is currently listening for Responses with this *queryId*, then the Response is extracted and sent to the peer that originated the Request *queryId*.
Download Request	N/A	If the *peerId* is that of the local peer, then the Agent creates a Download Response tuple with the document identified by *documentId* from the local repository of community *communityId*. The DownloadResponse tuple is output to the tuple-space.	If the *peerId* is *not* that of the local peer, then the network adapter connects to the peer identified by this *peerId*, and retrieves the document. It then places the document in a Download Response tuple and outputs the tuple to the tuple-space.
Download Response	If the Agent is listening for this Download Response, then the Document is extracted from the tuple and rendered to the user in the User interface.	If the Download Response was not output by the Repository Agent itself, then the Document is extracted from the Response and stored in the local repository of the community *communityId*.	If the Download Response was not output by the Network Adapter Agent itself, then the Document is extracted from the Response and sent to the peer that originated the query.

TABLE II.
PROCESSING TRIGGERED BY THE DIFFERENT TUPLES IN THE DIFFERENT PROXY AGENTS

tuple-instruction is a *verb* (the Linda primitives in / out / read) followed by a tuple definition, which may include variables.

A Tuple is formed of any number of fields, which may be *formal* or *literal*. A formal field is a typed field which has no value but which matches any field of that type, and a literal field is a typed field with a precise value. "Template" tuples are used as parameters for "read" operations, in the sense that the read operation will return tuples matching the template field-by-field. Formal fields are mostly used as templates, as literal fields only match if their values are identical (a property which can be used as a filter).

In order to limit the complexity of our language we limit ourselves to String fields, which avoids the need to specify the type of fields. Our variable binding process is based on the concept of formal fields, to which we add a variable name instead of a value. This allows us to read values from matched tuples, and store these values in a named variable. In order to reuse a variable, we

use what we call "variable" fields, which will appear at runtime to be litteral fields, but whose values are filled in dynamically from stored variables. For example, a formal field bound to variable x will allow us to read a value and store it in the variable x, whereas a field declared as a *variable* field will be filled at runtime with the current value for x.

A single agent may have multiple rules sharing the same set of variables, and this avoids the need of loop constructs or condition statements, since the former can be replaced by a short rule which can be triggered repeatedly, and a conditional fork can usually be replaced by two rules triggered by the two alternative inputs of the condition.

The instructions should be based on the existing UP2P functions, i.e. at least one rule should be triggered by the Navigation Request query $(\text{“}Navigate'', URI, label, queryId)$, and the tuples that the agent will output to the tuple-space should be requests for the basic operations of U-P2P, i.e. search, download, dereference URI, etc.

In order to make the language expressive enough, we have enriched the basic U-P2P processing capabilities with the following additional tuple requests (and appropriate processing in the existing agents):

- some very simple data processing functions (string concatenation),
- an additional operation to query temporary collections of Search Responses,
- a Lookup Request to retrieve individual attribute values from documents.

The schema, which we will call Agent Definition Language, is illustrated in Figure 5.

Users (albeit fairly advanced users) can thus create and install new "canned queries" to support Navigation of particular Relations.

a) Example: We give in table III the script of an Agent implementing the Navigation query based on the "RDF Links" representation described in section IV-B.3. Notations[10] :

- the verb of each instruction is listed before each tuple, and the tuple is limited by the brackets, the fields separated by semi-colons.
- `[formal: <name>]` denotes a *formal* field where the value in the read tuple is stored in variable `<name>`.
- `[var: <name>]` denotes a *variable* field, i.e. a litteral field where the value is populated at runtime from the variable
- fields with simply a value are litteral fields containing the value.

D. Towards semantic reasoning agents

Our current design, described up to this point, supports navigation of Relations described in documents by *endpoint* attributes. The openness of our model makes the support extensible to variations around the basic principle of endpoint attribute links, such as the *RDF Links* example presented in section IV-B.3.

Navigating such links requires only data lookups, even though our distributed model makes these lookups a little bit tricky to handle and define.

The next developments envisioned in this project are to support a broader definition of a Relation, i.e. one that may be *inferred* or *computed* from other Relations or Documents.

We propose to define the following categories of Relations.

Suppose two documents A and B are related in a relation R (i.e. (A, B) is in R). R may be:

- a *data-defined* relation, i.e. there exists a data representation of all the couples in R; the examples of our previous sections are all data-defined relations;
- *inferrable*, i.e. we can determine if (A, B) are in R from other couples in R or in another relation R'. For example instances of the relation "sequel" between movies can be inferred from other instances, or from the opposite Relation "prequel", as these relations are partial order relations;
- *computable*, i.e. given A and R a new document B can be computed (created) such that (A, B) is in R. The typical example of computable Relations is a *mapping* Relation defined between the schemas of two communities: in some cases, given a mapping definition between two community schemas, and a document from one community, a mapping agent could generate the corresponding document in the second community;
- *co-computable*[11], i.e. given A, B and an intensional definition of R we can determine whether (A, B) is in R. For example, documents with a date attribute could be compared in an older / newer Relation based on the value of the date attribute;
- a combination of the above, e.g. a relation may be data-defined but also partially co-computable.

Supporting such relations, which are more about *reasoning* then about graph queries, would be an important step towards a real *knowledge* sharing system, where users could not only input base facts but incrementally build on facts to infer new facts and answer more expressive queries.

An example scenario would be the addition of a versioning relation, which we could base on the OWL property *priorVersion*. Suppose a user finds an error in a document downloaded from U-P2P and wishes to correct that error for the benefit of the other users. Making changes to that particular document and publishing it won't have much of an effect: one would need to make changes to all copies of the document.

A more efficient alternative would be to create a *priorVersion* link between the new document and the old

[10]these notations are used instead of the exact XML for a better readability

[11]The term "co-computable" was coined by analogy with the NP / CO-NP complexity classes.

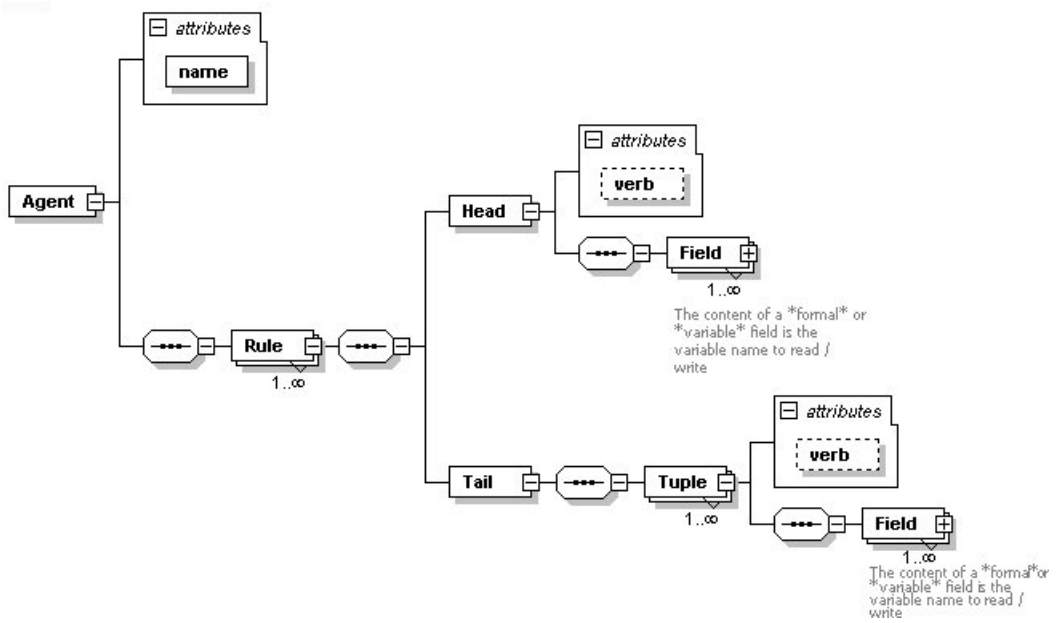

Figure 5. Schema structure of a valid Agent Definition.

```
Rule 1 :
Head : read (Navigate; [formal: uri]; [formal: label]; [formal:queryId])
Tail: out (Concatenate; label=; [var: label])
in  (Concatenate; label=; [var: label], [formal:labelcondition])
out (Concatenate; AND endpoint1=; [var: uri])
in  (Concatenate; AND endpoint1=; [var: uri]; [formal:uricondition])
out (Concatenate; AND endpoint1=; [var: uri])
in  (Concatenate; [var:labelcondition]; [var: uricondition]; [formal:searchexpr])
out (Search; RDFLinks; [var:searchexpr]; [var:queryId])

Rule 2 :
Head : in (SearchResponse; RDFLinks; [formal : documentId]; [formal: peerid] ;
[formal: ]; [formal:queryId])
Tail: out (Lookup; TemporarySearchResponses; [var: documentId]; endpoint2)
  in (Lookup; TemporarySearchResponses; [var: documentId]; endpoint2; [formal:uri2])
  out (DereferenceURI; [var:uri2]; [formal :queryId])
```

TABLE III.
DEFINITION OF AGENT "NAVIGATE RDFLINKS"

one, in a community of triples implementing a data representation of *priorVersion*. The simple agent described in our "RDFLinks" example (table III) would provide the basic navigation function necessary to follow a path of versioned documents, step by step. But a real *semantic enhancement* to this agent would be to add a rule in this agent to implement the transitivity of the relation. This rule would basically involve listening for responses to the navigate query, and recursively outputting new navigate queries, thus extending the search to generate the transitive closure of the *priorVersion* relation.

From this broader picture it appears clearly that our elementary query based on relations, the *navigation* query, could in some cases easily be implemented by an agent using our Agent Definition Language, whereas in some cases our language expressiveness is clearly insufficient.

An important challenge for us is to incrementally enrich the Agent Definition Language and / or the basic data processing support of the native U-P2P agents, and for each step to establish for which class of Relations we can support the Navigate query.

At this point, we consider that our partial support for navigation is consistent with the "best-effort" approach to searches characteristic of P2P file-sharing. One experimental goal of this work is also to allow users to generate and share new content (including query processing agents) which may improve the completeness of navigation query answering.

VI. CONCLUSION

We have presented a framework to share structured data in a P2P file-sharing infrastructure. Our system extends the traditional model of P2P file-sharing, by supporting arbitrary schemas, and the navigation of *relations* between documents, while only sharing structured documents us-

ing the basic functions of a P2P file-sharing community, i.e. *publish*, *remove*, *search*, and *download*.

Our approach relies first on the "active XML" approach to "reify" the concept of a community or of a navigation query, by associating an application-specific semantics to the documents where these concepts are defined.

Secondly, our model uses a URI scheme specific to U-P2P communities, as the basis to annotate existing documents, or to relate documents to one another. Annotations or relations can be described in documents using *endpoint* attributes as links to the annotated documents, and we associate a search-based dereferencing mechanism to navigate such links.

Finally, through this semantic extension we have laid the foundations to manage relations with intrinsic semantic properties, such as being transitive, sub-properties of others, or computable from values of document attributes.

Our extensible architecture can accomodate new query processing agents that will be able to navigate such semantic relations, perform simple reasoning, and infer new implicit relations from those described in the data.

REFERENCES

[1] A. Davoust and B. Esfandiari, "Towards semantically enhanced p2p file-sharing," in *SEMELS conference*, 2008.

[2] A. Mukherjee, B. Esfandiari, and N. Arthorne, "U-p2p: A peer-to-peer system for description and discovery of resource-sharing communities," in *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 701–705.

[3] P. A. Boncz and C. Treijtel, "Ambientdb: Relational query processing in a p2p network," in *DBISP2P*, ser. Lecture Notes in Computer Science, K. Aberer, V. Kalogeraki, and M. Koubarakis, Eds., vol. 2944. Springer, 2003, pp. 153–168.

[4] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *Networking, IEEE/ACM Transactions on*, vol. 11, no. 1, pp. 17–32, 2003. [Online]. Available: http://dx.doi.org/10.1109/TNET.2002.808407

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, vol. 31, no. 4. ACM Press, October 2001, pp. 161–172. [Online]. Available: http://dx.doi.org/10.1145/383059.383072

[6] "The piazza peer data management system," *IEEE Trans. on Knowl. and Data Eng.*, vol. 16, no. 7, pp. 787–798, 2004.

[7] N. Arthorne and B. Esfandiari, "Peer-to-peer data integration with distributed bridges," in *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*. New York, NY, USA: ACM, 2006, p. 14.

[8] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmr, and T. Risch, "Edutella: A p2p networking infrastructure based on rdf," 2002, pp. 604–615.

[9] P. Haase, B. Schnizler, J. Broekstra, M. Ehrig, F. van Harmelen, M. Menken, P. Mika, M. Plechawski, P. Pyszlak, R. Siebes, S. Staab, and C. Tempich, "Bibster–a semantics-based bibliographic peer-to-peer system," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 2, no. 1, pp. 99 – 103, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/B758F-4DS962C-2/2/f3e50ac7d66b0f157852a6639b14ed82

[10] M. Buffa, F. Gandon, G. Ereteo, P. Sander, and C. Faron, "Sweetwiki: A semantic wiki," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 6, no. 1, pp. 84–97, 2008.

[11] T. Berners-Lee, "Linked data," World Wide Web design issues, July 2006. [Online]. Available: http://www.w3.org/DesignIssues/LinkedData.html

[12] A. Davoust and B. Esfandiari, "Peer-to-peer sharing and linking of social media based on a formal model of file-sharing," Department of Systems and Computer Engineering, Carleton University, Tech. Rep. SCE-09-04, 2009.

[13] R. Rivest, "The MD5 Message-Digest algorithm," Internet Engineering Task Force, RFC 1321, apr 1992. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1321.txt

[14] P. Ciancarini, R. Tolksdorf, and F. Zambonelli, "Coordination middleware for xml-centric applications," in *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2002, pp. 336–343.

[15] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform resource identifier (URI): generic syntax," Internet Engineering Task Force, RFC 3986, jan 2005. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3986.txt

[16] "The rdf suite of specifications," 2004. [Online]. Available: http://www.w3.org/RDF/

[17] E. Prud'hommeaux and A. Seaborne, "SPARQL Query Language for RDF," W3C, Tech. Rep., 2006. [Online]. Available: http://www.w3.org/TR/rdf-sparql-query/

[18] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Commun. ACM*, vol. 35, no. 2, pp. 97–107, 1992.