Project Number 001907

DELIS
Dynamically Evolving, Large-scale Information Systems

Integrated Project
Member of the FET Proactive Initiative Complex Systems

DELIS-TR-622

A P2P Data Management Platform for XML and RDF Data

Giovanni Cortese, Fabrizio Davide, Dominic Battré,
Felix Heine, André Höing, and Odej Kao

2007
A P2P Data Management Platform for XML and RDF Data

Giovanni CORTESE\textsuperscript{a}, Fabrizio DAVIDE\textsuperscript{a}, Dominic BATTRÉ\textsuperscript{b}, Felix HEINE\textsuperscript{b}, André HÖING\textsuperscript{b} and Odej KAO\textsuperscript{b}

\textsuperscript{a} Radiolabs \\
\textsuperscript{b} TU Berlin, Complex and Distributed IT Systems

Abstract. This document describes a peer-to-peer data management platform for XML and RDF data developed in the Delis project. The data management platform features two main components (BabelPeers for RDF data management and inference, XGR for XML data management, replication and query) running on the same underlying DHT technology. The platform is available for public use and experimentation. This report intends to provide useful information to those interested in its exploitation and usage, as such, it documents the platform functionalities, the main design choices behind it, as well as aims at providing guidance to the application programmer.

Introduction

This document describes a peer-to-peer data management platform for P2P and RDF data developed in the Delis project. The data management platform features two main components (BabelPeers for RDF data management and inference, XGR for XML data management, replication and query) running on the same underlying DHT technology. The platform is available for public domain use and experimentation. This chapter is organized as follows: after the overview in section 1, where we provide a description of main concepts and a list of the main features of the platform, we describe in more detail the services for managing XML data in section 2. Then, in section 3 and in section 4 we describe the technical details how BabelPeers works. This includes the data dissemination, the query processing and the load balancing features. This section is an excerpt from the Delis technical report [7]. In section 5 we conclude with programming level, detailed information describing how to use the platform.

1. Overview

1.1. Concepts and Features

The platform is a middleware providing a wide range of data management services to applications running in a distributed setting without a central control. Data management services include storage, replication, indexing, query processing, and are performed in a cooperative style by a network of peers. While some features will be more relevant to
one or the other component, the following section describes the system model, which is roughly applicable to both components.

XGR is focused on storing, indexing/querying, replicating large sets of ‘plain’ XML data in a P2P network. Babelpeers is focused on providing a similar set of functionalities on RDF data, with an emphasis on supporting semantic-rich indexing and querying of RDF triples. On the other hand, XGR component is optimized to support indexing of data types such as ‘vectors’ and streams, such as those generated by network monitoring agents or physical sensors.

The two services are then complementary; depending on the application needs, the application designer can choose the more appropriate service. Or, they can be used together: as shown in [1], an application can use BabelPeers e.g. to encode metadata information such as categories and classifications, while XGR is used to support more massive storage and retrieval of instance data.

1.2. System Model

The system we consider consists of a large number of agents, which can be providers or consumers of data (or both). A comparably high number of peers cooperate to realize a distributed data management service, where each peer serves firstly as a place to store data, secondly as a helper to query processing, and finally as a forwarder of data (see Figure 1).

The common storage is used to store both application-level data, and metadata such as schema and index information, which will guide queries to find the data stored in the network of peers. Depending on requirements, application data can be better stored in a local storage at the provider node, and only the system’s distributed indexing feature is exploited by applications in this case. Differently, applications can use the cooperative storage for both data and indexes. The former usage is associated either to data which change frequently (e.g. performance counters in a network monitoring agent), or that are generated on demand (e.g. via a Web Service). Peers arrange in a self-organising structured overlay, for the purpose of communication and basic data indexing. The system
implementation is built on top of a DHT-based self-organising communication overlay (see [12]). Producers interact with the system through one generalized insert primitive, which injects an object into the system. The object can be both dispatched to consumers which have declared an interest (through a subscription query), and stored in the common storage for the benefit of consumers which successively issue queries to which it may be relevant. The insert primitive allows the client to select between several alternative behaviors, including a desired persistency policy. Consumers can access data using two logical primitives:

- **retrieve**: allows a client to pose an ad-hoc query, which is executed immediately and returns to the user a list of matching objects, among those persistently stored in the system.
- **subscribe**: allows a client to post a long-running, continuous query. The client will be notified whenever an object has been inserted, or updated, which satisfies the subscription query.

2. XML Data Management

In this section, we describe XML data management services provided by the platform.

2.1. Concepts

The XML data management component of the platform, as provided by XGR, provides three main services:

- **Distributed Storage and Query**: a middleware for collaborative storage, indexing and query of XML data. Features a replication service for both data and indexes.
- **A collaborative pub/sub system**: it provides a topic-based publish-subscribe system; while based on it and conceptually similar to Pastry/ Scribe (see [?],6), it provides a layer of convenience functions on top of it, for example adding support for XML Xpath syntax for subscriptions, as well as ‘client APIs’ for allowing access to the network of brokers from a node which is not part of the overlay (currently through Java RMI).
- **Running queries**: half way between a query service and the pub/sub service, running queries support more complex query patterns than the pub-sub system, as well as being able to incrementally deliver new tuples matching query information to the application which issued the query.

The design of XGR is the result of our experience in integrating self-organizing P2P information storage and retrieval algorithms, with mechanisms to help them work in more practical way in realistic application scenarios.

2.1.1. Storage

A storage is a logical partitioning of the data space. It is responsibility of the application designer to choose the number of storages to be used (one or many), and the allocation of data to storages. Decision should be based so as to achieve a clear design of the application; there are no strong performance or technical criteria behind the choice of using one or many storages. A predefined storage for system metadata always exists in an XGR installation (the Storage Catalog).
2.1.2. Data Model

Applications store XML data in the network of peers. In order to implement a manageable indexing system, we decided to require the system designer to provide metadata information, which provides guidance with respect to how an XML document is broken into smaller parts, which are independently stored as a unit. The unit of distribution is then an XML fragment, as described by a 'storage type' descriptor (we will use the term 'type' for brevity in the following text). Each DHT fragment will be allocated to a DHT node based on configurable mapping strategies which are described later in this section. A type descriptor is a collection of metadata, which provide directives to the distributed storage manager about the storage strategy for objects. A (storage) type should not be thought of as a DTD or an XSchema, listing all the attributes and corresponding datatypes in the XML document. It is more of a set of metadata, to specify information which controls storage of the matching XML fragments. An application programmer can actually store any data she wants as an instance of a type, as long as the primary key and other indexed attributes are included.

Directives included in the type descriptor include criteria for

- forming the primary key ('id') of the fragment, based on values of selected attributes within the fragment itself;
- indexing the fragment on alternate, secondary key attributes;
- providing hints on the desired number of replicas, for fragments of this type
- and for selecting among different available strategies for mapping data to peers (see below).

We briefly cover how XML fragments are allocated to DHT nodes (mapping strategies). XGR is designed to support storage of data of different nature, and provides a degree of control to application designer with respect to placement of data. Also, behavior of insert and retrieve operations is affected by the strategy adopted.

XML fragments are stored in the DHT based on the primary key. Primary key is the attribute which is used to create the hashcode (DHT key), used to store and lookup the XML fragment in the DHT. An Xpath expression can be used to create a complex key (e.g. a concatenation of strings) as a key. A default 'id' attribute is used as pkey unless specified differently. However, granularity of storage varies depending on whether the datatype is single, log or a vector type.

- 'single': each object, as identified by its id, is individually stored in the peer having the closest node id. A pkey identifies a single instance of an XML fragment. Subsequent 'insert' operation using the same pkey are implemented as 'update' of the unique fragment associated to the pkey.
- 'log': several instances of the object, having same id, are stored at the peer having the closest node id, and appended to a queue, as appropriate for example for a time series of observations reported by a sensor or network management agent. A log is similar to an array (queue) of objects. Subsequent insert operation using the same pkey append new instances of the fragment to the log. All fragments with same pkey are stored altogether at the same XGR/DHT node, and distinct by a progressive id/ timestamp. For example, observations (e.g. positions, or other measurements) from a sensor are modeled as logs in XGR. An expiration mechanism is
often used in association with logs, to keep the size of stored data from growing indefinitely

- ‘vector’: multiple objects share the same primary id, and are stored in a vector at the responsible peer for id, while being individually addressable for selections or updates based on an instance key. ‘Sub-objects’ can selected using a second identifier, the ‘entrykey’. The attribute acting as entrykey is again specified in the type descriptor.

2.1.3. Queries and Query Processing

Queries are expressed in the W3C standard language XPath, which selects XML fragments in the peer-to-peer datastore. To date, no kind of “join” queries are possible, which retrieve related documents or parts of different documents which are related to each other. Depending on how the application formulates the query, you can get the whole document which matches the path query, or select the part of the document which is selected by the query.

Query processing is based on indexes. Query processing heavily relies on the index layer, which is layered atop the DHT. Indexes are data structures implemented as DHT data as well. They are distributed to the DHT peers, and are replicated to ensure their survival to node departure.

A query parser analyzes the query, and selects attributes for which an index exists; it then selects one or more indexes which will help processing the query. While access based on primary key is mapped to DHT lookup operations, indexes are required for accessing data based on values other than primary key attributes/elements, or for supporting complex queries.

XGR supports three index types: equality-predicate indexes, range-predicate indexes, and type-predicate indexes. Type-predicate makes it possible to process queries which do not include any condition on indexed attributes without broadcasting the query to all nodes (e.g. retrieve position information from all node providing such data).

To support type-predicate indexes, XGR builds and maintains an ‘implicit’ pkey index, used to retrieve all the XML fragments belonging to a type. Since, depending on the application domain, using type-predicate indexes can lead to severe storage-load unbalance (the index can grow very large, and nodes which are responsible for the index will be overloaded), enabling them for a type is optional, and we also introduced in the latest release of the platform a mechanism for splitting the index on several nodes.

2.1.4. Local vs. DHT-based storage

The system allows a choice on whether storing in the P2P network only the index information needed to lookup an XML document, or both the index and the XML data.

Both documents and indexes are stored in the network by default. It is also possible for the application designer to choose to store the XML data locally. In this case only the index is stored in the DHT; no replication is in effect for the document, while replication occurs normally for index entries. Typically, for sensor data or fast-changing data this is the right option; while documents which must be available even when the originating peer is not connected should be stored in the DHT. Another way to implement local-only storage would be to store/ publish its http URL and provide access to it with a local web server.
Virtual data refers to a feature of the platform, to provide added flexibility in dealing with data values managed in Local Storage mode. Virtual data need not to be encoded and stored as XML tuples. Data could be maintained in the format which is used by the application, for example as rows in a RDBMS table, files on the file system, lines in a local log file, etc. and can be 'virtually' inserted in the XGR storage: only a reference to the data item is stored in the XGR storage and the data value is accessed by the platform using a custom class implemented and registered by the data provider in the local XGR node (see section 5).

2.1.5. Data Replication

Storage type descriptors also provide hints on the desired replication level, which is applied to data objects. Several approaches for data replication exist in the literature. We selected and use a leafset-based replication manager provides for replicating the data, and for maintaining the desired level of replication, in spite of events such as departure or failure of nodes. A leafset based replication manager creates replicas of data on neighbours in the DHT, i.e. on nodes which have close ids. When a node leaves the DHT, neighbors take responsibility for the id space formerly managed by the departing node, resulting in an easier transition. As a final note, the implemented system replaces the replication system provided by Pastry through its PAST component. The reason to implement our own system include the need to deal properly with changing data, specifically logs, as well to implement a more controllable behavior in the system restart phase.

2.1.6. Storage Catalogue

All metadata (type definitions, index definitions, long running queries etc.) is described in XML and stored in the DHT. The system Catalog is a dedicated storage to keep information describing the current configuration of the system.

2.1.7. Multicast

The system includes an application-level multicast facility, which leverages prefix routing of Plaxton trees (implemented as an extension of FreePastry and Scribe [6]. It is used both as an internal mechanism (e.g. for disseminating management and metadata information), and it can be used by applications if they need.

2.1.8. Running Queries

Running queries (see the opendatastream primitive in the programmer guide section later in this document) allow a requestor to get updated results to the query as they become available. Running queries are stored in the DHT and consumers listen to it through a multicast channel; for each insert operation executed by a data producer, a check is done whether it matches a running query; in this case the appropriate channel is notified.

Running queries are similar to multicast communication, and the two differ since they are expected to be efficient and scale properly in different scenarios. In the multicast scenario, channels are defined by the data producer; they are typically coarse-grained (in a DHT system the most practical algorithms for publish-subscribe, that we adopted, are topic-based). If more selective filtering is required, it needs to be done at each consumer side. Potentially, if filtering criteria from most consumers are more selective than the criteria used by the publisher, wasted communication results. With a running query, a
channel for each consumer exists - filtering is done at the producer side. Only data that match the subscription travels in the network.

3. Semantic Resource Discovery

In this section, we describe how we model resource description and discovery with RDF. While the example will be in the domain of resource modelling and discovery in Grids, the system is open to match anything which can be described using RDF.

3.1. Concepts and Example

RDF descriptions consist of a set of triples, whose elements are called subject, predicate, and object. The triples can be read like sentences. A set of triples can also be regarded as a graph, where each predicate is a directed edge connecting the subject vertex with the object vertex.

We start with an example how we use RDF [11,4] in combination with RDFS [5] to describe resources and background knowledge about resources. For simplicity, we use a reduced information model, and we omit namespaces. The techniques shown here apply to other models as well.

Consider a cluster named SFB running the Debian distribution with kernel version 2.4 of the Linux operating system. The cluster nodes are equipped with Itanium processors. A part of the RDF graph describing this system is shown in figure 2(a). Both, the operating system and the processor are described by blank nodes as we are not interested in these entities themselves but rather in their type and their properties.

Queries can be modeled as graphs, too. A simple query graph is shown in figure 2(b). It looks similar to the first graph, but there are some differences. First, the entity which is queried is denoted with a question mark. Second, the type of the operating system is specified as Linux compared to Debian in the resource description. Third, the type of the processor is specified as a combination of two types: Intel and 64Bit. Thus, the
query is more generic than the description. The entire query graph should be read as a
template which is to be matched with a subgraph of the resource description graph.

In order to be able to match this query with the resource description, we need addi-
tional background knowledge, which we encode in RDF Schema. It encompasses infor-
mation like Debian is a subclass of Linux. Figure 3(a) shows how this knowledge is
encoded in RDF/S.

Debian
  └── subTypeOf
      └── Linux

Itanium
  └── subTypeOf
      └── Intel

64Bit
  └── subTypeOf
      └── Intel

(a) Graph for Schema Knowledge.

Debian
  └── hasCPU
      └── SFB

Linux
  └── hasOS
      └── Debian

Itanium
  └── type

64Bit
  └── type

2.4
  └── kernelVers

Intel
  └── type

(b) Model Graph after Applying RDFS Rules.

Figure 3. Resource Matching with RDF Schema.

The background knowledge is integrated in the query process by applying the RDF
Schema entailment rules [8]. These rules are executed in a forward-chaining manner,
thereby generating new triples. For instance, entailment rule “rdfs9” of [8] generates new
dfs:type edges. In the resulting graph, shown in figure 3(b), a subgraph can now be
found which matches the query graph. The answer to the query is the SFB entity.

3.2. Integrating Background Knowledge

The introductory example already shows one type of background knowledge which is
very important. Information about the class hierarchy can be used during the matchmak-
ing process. This allows locally extending the class hierarchy, including very specific
classes for entity types that are elsewhere unknown. Through the RDF Schema mech-
nism, these entities are also published to be objects of more generic types, thus being
discovered by queries searching for the generic entities.

RDF does not distinguish between classes and instances and therefore allows de-
scribing both identically with triples. Thus, generic information about types can be en-
coded as well. In figure 4(a), the class Itanium is used as subject and several triples
describe further information that hold for every Itanium processor. This approach is an
alternative to the way the same information has been encoded in the above example,
allowing a clearer distinction between several aspects.

This way of modeling provides for example the possibility to use filter expressions
in queries that restrict suitable word lengths of processors. If the word length is encoded
as a class called 64Bit as shown in figure 3(a), queries for CPUs with a word length
greater or equal that 32 bit are not possible without listing every word length explicitly
or modeling this by \texttt{subClassOf} relations. In figure 4(a), the word length is an integer literal, which can be used to filter the results.

These two options can also be combined together. Consider e.g. a processor class hierarchy where Itanium2 is defined to be a subclass of Itanium, see figure 4(b). A provider describes its resource to have an Itanium2 processor, see figure 5. Now look at the query in figure 6. It asks for a cluster with a processor that is manufactured by Intel.

How can this query be answered? First, the RDF Schema rules use the \texttt{subClassOf} relationship defined in figure 4(b) to generate a new triple stating that the CPU of ClusterX is also of type Itanium. This new triple provides a link between ClusterX and the background knowledge for Itanium shown in figure 4(a), which includes the manufacturer Intel. Summarizing, the query is answered using three steps: First, three different RDF fragments are combined. Second, RDF Schema reasoning is applied to the union of these fragments. Third, the query pattern is matched to the resulting RDF graph.

This example shows an important feature needed for resource discovery. The system must be able to combine information originating from various sources, as the pieces of information which have been combined might reside on different nodes in the network. This is a main motivation for the system design of BabelPeers as described in the next section.

4. Babelpeers - System Design

In this section, we describe the methods and algorithms used in the BabelPeers system. This includes the way we disseminate the information, the query processing algorithms, and load balancing issues.

The BabelPeers component aims to provide a system that resolves these two main requirements. For scalability, it uses a peer-2-peer (p2p) network based on distributed hash tables (DHT) to store and query the resource information. For flexible resource de-
scriptions, it uses the resource description framework (RDF) from W3C. BabelPeers also supports RDF Schema inheritance to integrate multiple possibly incompatible resource description schemas.

Main parts of the system are a data dissemination algorithm that places the resource information on specific nodes and ensures fault-tolerance through replication, an efficient query algorithm that allows querying the aggregated information of all participating peers, an RDF Schema processor that evaluates the RDF Schema rules, and a load-balancing mechanism that ensures good performance of the system. Additionally, we have started to develop a wrapper for BabelPeers that allows us to use BabelPeers as a replacement for the Globus WS-MDS component. We have evaluated the performance of BabelPeers both through simulations and real benchmarks with up to 128 nodes.

4.1. Data Dissemination

As explained in section 3, we aim to integrate knowledge (in the shape of triples) originating from different sources, i.e. different nodes in the p2p network. Even broadcasting a query to all nodes in the network and collecting the results could not deliver the same quality of results one gets if the knowledge is integrated. Thus we need to disseminate the knowledge to well-defined nodes in order to be able to find and access it efficiently during query processing.

For this purpose, we use a p2p network based on distributed hash tables (DHT). In a DHT, each data item is associated with an identifier from an identifier space, e.g. $0, \ldots, 2^{128} - 1$. Each node in the network is responsible for a certain range of this identifier space. Every item to be stored is then pre-distributed to the node responsible for the identifier of the item. For fault tolerance, items are additionally replicated over multiple nodes. In our case where data items are RDF triples, we disseminate each triple to three different nodes based on its components subject, predicate, and object. Thus we can later access the triples even if only one of the components is known.

For the RDF Schema reasoning, we follow a forward chaining approach. This means that we generate and store instances of every new triple which follows from the RDF Schema rules, like the additional type triples in the previous section. Our dissemination scheme has the advantage, that all triples which are needed to do this forward chaining will be located on the same node. Thus, after dissemination, we can run the reasoning process on each node locally, generating the new triples. However, these newly generated triples are then disseminated to the network to be accessible via the standard indices over subject, predicate, and object.

The whole process is visualized in figure 7. Each node has some triples in its “local triples” store. These triples are disseminated via the p2p network to the responsible nodes, which store them in their “received triples” store (step 1). Using only the triples in the received store, RDF Schema reasoning is performed to generate new triples locally (step 2). The new triples are stored in the “generated triples” store, and then disseminated again over the network (step 3). At the target nodes, they are again stored in the received store, where they might fire new RDF Schema rules. The whole process terminates as soon as no more rules can be fired.

Updates and deletions are handled via a soft-state process. Thus, every triple carries an expiration time and is automatically removed from the network when it does not get refreshed. This means that the dissemination process runs periodically to keep the knowledge in the network up-to-date.
We provide two different kinds of query processing strategies. The goal in the first case is an exhaustive query evaluation, i.e. an evaluation that returns all matches to a query in the RDF graph. Depending on the type of query, you might get enormously large answers from a resource discovery system. If you ask a very generic query, e.g. by asking for resources with a Unix-type operating system, it is likely that you receive a really large answer list. Just like using Google to search for information in the WWW, you will not look at every answer in detail, but only at a few of them. To support this type of query, we developed a second strategy named top-$k$ that retrieves efficiently only $k$ answers instead of generating an exhaustive list of all answers. The answers can be ranked according to an attribute of the resources. This is useful when you are e.g. interested in all Linux servers in the Grid, but look for the server that has the largest main memory.

### 4.2.1. Exhaustive Query Evaluation

In the previous section we have given an intuitive definition of RDF graphs. The graph that contains the resource description is called a model graph. Each labeled, directed edge can be represented as a triple $t = (s, p, o)$. Its components are called subject, predicate, and object respectively. A subject can be an RDF URI reference or a blank node; a predicate has to be an RDF URI reference; an object can either be an RDF URI reference, a blank node, or a literal. For simplicity we assume that each triple $t$ is element of $X \times X \times X$. The set of all triples in the network is denoted with $T_M$.

A query as described in figure 2(b) can be modeled as a set of answer-variables and triple patterns. Following [10], we define a conjunctive query $Q$ as a formula

$$ H(Q) : \exists x_1, \ldots, x_n \exists s, p_1, o_1 \land \ldots \land \exists s_m, p_m, o_m \left( \left( s, p_1, o_1 \right) \land \left( s_2, p_2, o_2 \right) \land \cdots \land \left( s_m, p_m, o_m \right) \right) $$

where $x_1, \ldots, x_n$ are variables and each $(s_i, p_i, o_i)$ is a triple pattern, consisting of URIs, blank nodes, literals, and variables. $H(Q)$ denotes the head of the query, the set of variables that will appear in the answer set. Each variable $x_k$ appears in at least one triple pattern. The triple patterns $(s_i, p_i, o_i)$ of the query make up the set $T(Q) \subseteq (X \cup V) \times (X \cup V) \times (X \cup V)$, where $V$ denotes the set of variables. $T(Q)$ is also called the query graph. Note that it can contain variables that do not appear in the query head.

The goal of the query processing is to find all assignments of variables to URIs, blank nodes, and literals such that all triples of the query can be found in the model.
graph, or formally: Find all valuations $v : V \mapsto X$ such that $T(Q)[?x_1/v(?x_1), \ldots, ?x_n/v(?x_n)] \subseteq T_M$.

**Example:** Consider the model graph of figure 2(a) and assume that background knowledge (given in RDF Schema) augmented this graph by information such as “Itanium processors are 64 Bit processors by Intel” and “Debian is a Linux distribution.” The query depicted in figure 2(b) could be modeled as

$$\text{?cluster} :- (?\text{cluster}, \text{hasCPU}, ?v_1) \land (?v_1, \text{type}, \text{Intel}) \land (?v_1, \text{type}, \text{64Bit}) \land$$

$$\text{(?cluster, hasOS, ?v_2) \land (?v_2, \text{type}, \text{Linux})}$$

A valid valuation given the extended model graph would be $v = \{(?\text{cluster}, \text{SFB}), (?v_1, _:1), (?v_2, _:2)\}$ where _:1 and _:2 represent the blank nodes in the model graph.

Once we have found valid valuations, we can project them to only those variables which appear in the head of the query. These variables are denoted with $H(Q) \subseteq V$.

Focus of this section is how to find valid valuations. The query processing described in this section is similar to work presented by Liarou et al. in [10] but extended by sophisticated means to determine the order in which triples of the query are processed (see [9]). These means are crucial to the overall performance of the system. The general idea of the query processing strategy is to calculate the result of a query iteratively. We start with a single triple pattern of the query graph $T(Q)$ and do a lookup for possible valuations of the variables occurring in this triple pattern. Then we extend this intermediate result by doing a lookup for a second triple pattern and joining the results. We execute this operation until all triple patterns have been regarded.

Assume that the triple patterns in $T(Q)$ can be ordered by a heuristic such that triples patterns which have few matches are processed first. This heuristic will be described later on. We denote with $\text{next}(T(Q))$ the triple pattern of $T(Q)$ that shall be processed next.

Triple patterns can be processed if they contain at least one fixed value (URI, blank node, or literal, but not a variable) or a variable for which we have determined possible valuations already. We denote with $\sigma_{\text{next}(T(Q))}(T_M)$ the selection of triples from the model graph that match the next triple pattern to be processed.

During query processing, intermediate results are stored in a relation $R$, whose columns store the valuations of variables. Initially, $R$ is empty.

Simplified, with this, we can process a query by applying the following step iteratively:

$$R' = \pi_V \left( R \bowtie \pi_{\text{var}(\text{next}(T(Q)))} \left( \sigma_{\text{next}(T(Q))}(T_M) \right) \right)$$

where $V$ is the set of variables occurring in the head $H(Q)$ of the conjunctive triple pattern $q$ or in triple patterns of $T(Q)$ that have not been processed yet. The operations are (beginning from the inner most operation):

1. query those triples from the model graph that match the triple pattern $\text{next}(T(Q))$
2. project the result to only the variables of the triple pattern $\text{next}(T(Q))$
3. calculate the natural join with the previous results
4. project the result to only those variables (columns) that are relevant for the result or the further processing of the query
Once all triple patterns have been processed, relation $R'$ contains the result of the query.

The query evaluation can be accelerated significantly by reducing the network traffic caused by triple lookups. As explained, each triple pattern of the query consists of RDF URI references, literals, and variables. A lookup of a triple pattern consisting of three URI references can return at most one triple. A lookup of a triple pattern consisting of two URI references and a variable can, however, return a large number of triples. If we restrict the triple patterns to be returned to only those that may be relevant in the future, this can save a lot of bandwidth. When a variable occurs the first time in a lookup we receive a set of possible valuations for this variable. As triple patterns are conjunctive, this set of possible valuations can only be pruned but never be extended by future lookups. Therefore, we encode the set of possible valuations with Bloom filters while doing lookups and request only those triples that fall into our existing candidate set.

Furthermore, we improve the query processing throughput by first determining for each triple pattern that we can process in the next step, how many matching triples are expected. Then we continue with the triple pattern that delivers the least results. Using this order creates small candidate sets for variables and helps retrieving smaller triple sets with future lookups. These strategies are described in detail in [9].

4.2.2. Top $k$ Query Processing

As motivated before, the top $k$ query processing strategy serves the purpose of retrieving only the best $k$ matches (e.g. the fastest clusters that run Linux). Query processing runtime is reduced because not all matches need to be generated.

The general idea of the evaluation function is to iterate over all possible assignments to triples, assume one, and proceed to a recursive evaluation until we encounter contradictions, find a complete match, realize that we have found a sufficient number of matches, or until we cannot assign any more triples. So far, this is straight forward. The crucial part is to find the next possible assignment (called candidates) for a given query triple. If this is done on a triple by triple basis, the algorithm results in sending a huge number of very small messages over the network, each one collecting a single triple.

In order to avoid this situation, we developed a combined caching and look-ahead mechanism. The look-ahead tries to guess which candidates might be used next in the backtracking and to fetch a larger set of these candidates. At the same time, old candidates might be reused during the backtracking and will thus be cached to avoid re-fetching over the network.

Each query triple consists of three components, which might either be fixed values or variables. Each variable in a query triple might either be bound or unbound. A variable is bound if it was seen before in a higher recursion level, and unbound if it occurs for the first time. We split the triple into one component which defines the key of the DHT and two remaining components. If possible, we choose a fixed URI as DHT key; otherwise we have to use a variable which is already bound to some value. For the other two components of the query triple, we can encounter six different cases:

1. two unbound variables,
2. an unbound variable plus a bound variable,
3. an unbound variable plus an fixed URI or literal,
For each of these cases we define a specially optimized type of cache. These caches are depicted in figure 8.

The caches have to query the next chunk of up to \( c \) candidates for a query. They deliver these chunks triple by triple. For scalability reasons, the peer which will process the query does not store any state information, so the requesting peer is in charge of submitting the state along with the actual request. The state can consist of the set of triples we want to gather information about (first three cases below) or of a set of markers, which define the last triples for which we know information already (last three cases below).

The simplest cache for fixed/fixed components (see figure 8(a)), which occur if a RDF query contains a triple with three URIs, does a simple lookup without look-ahead. The state of the cache can be “triple exists in RDF graph” (represented by a check mark in the figure), “triple does not exist in RDF graph” (cross), or “unknown whether triple exists in RDF graph” (circle). For fixed/bound component pairs (see figure 8(b)) the caching is simple as well. A peer requests a chunk of triples by specifying the fixed component and a set of candidates for the bound component for which it wants to retrieve the state. For bound/bound components (see figure 8(c)) we build up a request containing a set of unknown combinations of already known values for the bound variables. The fixed/unbound cache (see figure 8(d)) is similar to the fixed/bound cache, except that it is sufficient to request the next \( c \) elements starting after a given position. Therefore, we submit the fixed element and the last inspected value for the unbound element as the request. The bound/unbound cache (see figure 8(e)) extends this by storing and sub-
mitting markers for the last known elements in several rows. The peer who processes a request starts sending triples at the first marker until $c$ triples have been sent or continues at the next marker if the row (candidates) do not provide $c$ triples. For the unbound/unbound cache (see figure 8(f)) it is again sufficient to submit a single marker which determines the next triples to be delivered.

4.3. Load Balancing

Load balancing is an important issue in every distributed network. In our system, optimal performance can only be achieved if all nodes take the same share of the overall load.

Load balancing is described in detail in the related technical report [7] and in [2] and [3].

4.4. Example application: WS-MDS Interface

As mentioned already, our goal is to create a flexible and scalable resource description service. To show that BabelPeers fulfills these requirements, we implemented a prototype discovery service based on a BabelPeers backend for the Globus Toolkit, which can replace the original WS-MDS service.

The Globus WS-MDS implementation is based on a hierarchical model, where data sources are located at the leaves of a tree (or DAG). Each data source publishes its data to index services and the index services propagate and aggregate the information upwards in the hierarchy. This means either that an index service at the root of the hierarchy has to cope with high data and query load or that data needs to be aggregated and therefore information is lost.

A DHT based p2p model without a distinguished root-node provides a global view without aggregation and without a single point of entry/failure. Thus, we think it is beneficial to use BabelPeers within Globus-based Grids.

However, there is a gap between the WS-MDS XML and the RDF data model. To provide compatibility with existing services, which publish their properties as XML documents, we employ a generic XML to RDF translation. Unfortunately, querying the resulting documents is complicated, as they have an artificial structure. Thus we favor a more natural way of using RDF as described in section 3. Although it requires more work to write appropriate translators, it can provide powerful and flexible resource queries. The same holds for the query language. There is no one-to-one mapping of XPath to the RDF query language SPARQL used in BabelPeers. We tested an automated partial translation, but again we favor using native SPARQL to unleash the full power of the system. The architecture is shown in figure 9. The dashed boxed indicate translators that have to be implemented individually.
5. Programming Guidelines and APIs

5.1. XGR Programmer Guide

This chapter describes how to work with XGR, specifically:

- how to access XGR services through the RApplication command line utility,
- how to access XGR services through the RMI API (remote interface to an XGR node running on a separate JVM), and
- the Java API (interface to an XGR node running in the same JVM hosting the client application)

Note: RMI API and Java API are almost identical except for main class names; in the following chapters the RMI API examples references the Java API examples when identical

The following basic concepts are assumed:

- all actions (insert, delete, query, etc.) are related to a storage instance that must be created before being used
- a predefined storage instance (XGRStorageCatalog) stores the meta-information (types, indexes, storage instances) for all data in the distributed repository
- managed data values are XML documents; the root element qualified name is the data type that defines a number of properties influencing how the document will be stored (e.g. indexing, data primary key, etc.)

5.1.1. Node Binding

Bind to the node instance providing the XGR services.

RAplication command line option:

-host <host>:<port>: XGR node RMI host:port

Examples:

#> java interplay.xgr.test.RApplication -host localhost:15001

Java API:

import interplay.xgr.XGR;
import interplay.xgr.XGRNode;
import interplay.xgr.network.XGRNodeImpl;

// init XGR properties
Properties props;
XGR.init(props);

String mode = "instance_name"; // internal instance name
int mode = XGRNode.CLIENT; // node role in the network

// create a new node instance
XGRNode node = XGRNodeImpl.newNode(name,mode);

// activate the node instance (network join)
5.1.2. Storage Creation

Create a new Storage definition: a storage instance is defined by a name and a description.

RApplication command line option:

```
-create #<storageid>#<descr>: create/update a storage definition
```

Examples:

```
#> java interplay.xgr.test.RApplication -host localhost:15001
   -create "#TestStorage#Test Storage Instance"
```

Java API:

```
import interplay.xgr.XGRNode;
import interplay.xgr.XGRStorage;

XGRNode m_node = < node binding >;

// create/retrieve storage instance
String storageid = "TestStorage";
XGRStorage storage = m_node.storage(storageid);

if ( storage != null ) {  
   // storage created/retrieved  
   // set [optional] "description" property
   Properties props = new Properties();
   String descr = "Test Storage Instance";
   props.setProperty("description", descr);
   storage.register(props);
}
```

Java RMI API:

```
import interplay.xgr.XGRNodeR;
import interplay.xgr.XGRStorageR;

XGRNodeR m_node = < node binding >;
```
// create/retrieve storage instance
String storageId = "TestStorage";
XGRStorage storage = m_node.storage(storageId);
if ( storage != null ) {
  // storage created/retrieved
  // set [optional] "description" property

  Properties props = new Properties();
  String descr = "Test Storage Instance";
  props.setProperty("description", descr);
  storage.register(props);
}

5.1.3. DataType/IndexType Registration

Register a new DataType or IndexTypeTypes: registration is performed by inserting the
data or index type description (XML document) on the XGRStorageCatalog storage in-
stance.

Classes XGRDataType and XGRIndexType can be used to handle the type defini-
tion.

RApplication command line option:

- register #<xmlData>: insert/update a datatype specification

Examples:

#> DATATYPE='<?xml version="1.0" ?>
  <xgr:datatype xmlns:xgr="XGRSchema" >
    <xgr:id>xgr:test</xgr:id>
    <xgr:desc>Test Data Type</xgr:desc>
    <xgr:replicas>2</xgr:replicas>
    <xgr:expire>86400</xgr:expire>
    <xgr:index>
      <xgr:name>TIDX</xgr:name>
      <xgr:path>xgr:value1</xgr:path>
    </xgr:index>
  </xgr:datatype>'

#> INDEXTYPE=' <?xml version="1.0" ?>
  <xgr:indextype xmlns:xgr="XGRSchema" >
    <xgr:id>TIDX</xgr:id>
    <xgr:desc>Test Index Type</xgr:desc>
    <xgr:type>rangenum</xgr:type>
    <xgr:range>10</xgr:range>
  </xgr:indextype>'

#> java interplay.xgr.test.RApplication -host localhost:15001
   -register "#$DATATYPE" -register "#$INDEXTYPE"

Java API:

import interplay.xgr.XGRNode;
import interplay.xgr.XGRData;
import interplay.xgr.storage.XGRDataType;
import interplay.xgr.storage.XGRIndexType;

XGRNode m_node = < node binding >;

// create a DataType instance
XGRDataType datatype = new XGRDataType();
datatype.type = "xgr:test";
datatype.desc = "Test Data Type";
datatype.replicas = 2;
datatype.expire = 86400;

XGRDataType.IdxEntry idx1 = new XGRDataType.IdxEntry("TIDX","xgr:value1");
datatype.indexes.add(idx1);

// get type as XGRData
XGRData data = datatype.asXGRData();

// insert/update datatype definition in catalog
m_node.catalog().insert(data);

// create an IndexType instance
XGRIndexType indextype = new XGRIndexType();
indextype.id = "TIDX1";
indextype.desc = "Test Index Type";
indextype.type = XGRIndexType.RANGENUM;
indextype.range = 10;

// get type as XGRData
XGRData index = indextype.asXGRData();

// insert/update index definition in catalog
m_node.catalog().insert(index);

Java RMI API:
import interplay.xgr.XGRNodeR;
import interplay.xgr.XGRData;
import interplay.xgr.storage.XGRDataType;
import interplay.xgr.storage.XGRIndexType;

XGRNodeR m_node = < node binding >;

// Same as for Java API

5.1.4. Data Insertion

Insert/update a data item (XML document): if an item with the same primary key (and
entry key if datatype is a vector) is already stored, its value is replaced with the new
value. The item key/id is obtained from the data using the data type specification.

Data insertion/update is performed on a storage instance.

Virtual Data Items (see 6.12) are inserted by specifying a dataptr that is interpreted
by the a specific data handler, e.g. a local file name (the handler for the datatype must be
registered and loaded on the node)

RApplication command line option:
Examples:

```java
卜 DATAVALUE='<?xml version="1.0" ?>
  <xgr:test xmlns:xgr="XGRSchema" >
    <xgr:id>Nome1</xgr:id>
    <xgr:desc>Test Data Value</xgr:desc>
    <xgr:value1>145</xgr:value1>
    <xgr:value2>Tue Feb  7 23:50:21 2006</xgr:value2>
  </xgr:test>
卜 java interplay.xgr.test.RApplication -host localhost:15001
     -insert "#TestStorage#$DATAVALUE"

卜 echo "$DATAVALUE" > data.txt
卜 java interplay.xgr.test.RApplication -host localhost:15001
     -insert "#TestStorage#file:data.txt"

卜 cat >> extdata.txt < EOF
  <xgr:ext_test xmlns:xgr="XGRSchema" >
    <xgr:id>Nome1</xgr:id>
    <xgr:desc>External Test Data Value</xgr:desc>
    <xgr:value1>145</xgr:value1>
    <xgr:value2>Tue Feb  7 23:50:21 2006</xgr:value2>
  </xgr:ext_test>
EOF
卜 java interplay.xgr.test.RApplication -host localhost:15001
     -insert "#TestStorage#xgr:ext_test#file:extdata.txt"

NOTE: "xgr:ext_test" must be registered as a virtual data type
(a specific handler must be loaded on the node)
```

Java API:

```java
import interplay.xgr.XGRNode;
import interplay.xgr.XGRData;
import interplay.xgr.XGRStorage;

XGRNode m_node = < node binding >;
String storageid = "TestStorage";
String value = "<?xml version="1.0" ?>"+
  " <xgr:test xmlns:xgr="XGRSchema" >"+
    " <xgr:id>Nome1</xgr:id>"+
    " <xgr:desc>Test Data Value</xgr:desc>"+
    " <xgr:value1>145</xgr:value1>"+
    " <xgr:value2>Tue Feb  7 23:50:21 2006</xgr:value2>"+
  " </xgr:test>";

// get type as XGRData
XGRData data = new XGRData(value);
```
// get storage handler
XGRStorage storage = m_node.storage(storageid);
if ( storage == null) {
    // failed to get storage instance, abort...
}

// insert/update datatype definition
storage.insert(data);

// insert a virtual data item
// NOTE: "xgr:ext_test" must be registered as a virtual
data type (a specific handler must be loaded on the node)
String datatype = "xgr:ext_test";
String dataptr = "file:extdata.txt";
storage.insert(datatype,dataptr);

Java RMI API:
import interplay.xgr.XGRNodeR;
import interplay.xgr.XGRData;
import interplay.xgr.XGRStorageR;
XGRNodeR m_node = < node binding >;
String storageid = "TestStorage";
String value = "<?xml version="1.0" ?>"+
"<xgr:test xmlns:xgr="XGRSchema" >"+
" <xgr:id>Nome1</xgr:id>"+
" <xgr:desc>Test Data Value</xgr:desc>"+
" <xgr:value1>145</xgr:value1>"+
" <xgr:value2>Tue Feb  7 23:50:21 2006</xgr:value2>"+
" </xgr:test>";

// get type as XGRData
XGRData data = new XGRData(value);

// get storage handler
XGRStorageR storage = m_node.storage(storageid);

// Same as for Java API

5.1.5. Data Removal

Remove a single data item: the data item to be removed is identified by its primary key and, for vector/log data type, the entry identifier (for pure log datatypes the entry identifier is the record number (20 char string padded with 0), e.g.: "00000000000000000123").
Data removal is performed on a storage instance.
RApplication command line option:

- remove #<storageid>#<datatype>#<datakey>[#<entryid>]: remove a data item

Examples:

#> java interplay.xgr.test.RApplication -host localhost:15001
   -remove "#TestStorage#xgr:test#Nome1"
5.1.6 Data Item Lookup

Lookup a single storage entry value: the data item to be retrieved is identified by its primary key and, for vector/log data type, the entry identifier (for pure log datatypes the entry identifier is the record number (20 char string padded with 0), e.g.: "00000000000000000123").

Data lookup is performed on a storage instance.

Note: this is a low level API that allows retrieving values for index entries and log/vector container entries; client applications should use the retrieve API.

RApplication command line option:
-lookup #<storageid>#<datatype>#<datakey>[#<entryid>]: lookup data
-lookup #<url>: lookup data by URL (DHT://<storageid>/<datatype>@<datakey>)

Examples:

#> java interplay.xgr.test.RApplication -host localhost:15001
   -lookup "#TestStorage#xgr:test#Name1"

#> java interplay.xgr.test.RApplication -host localhost:15001
   -lookup "#TestStorage#xgr:log#Name2#0000000000000000123"

#> java interplay.xgr.test.RApplication -host localhost:15001
   -lookup "#TestStorage#xgr:vector#Name3#Item2"

Java API:

```java
import interplay.xgr.XGRNode;
import interplay.xgr.XGRStorage;
import interplay.xgr.storage.data.XGREntryValue;

XGRNode m_node = < node binding >;
// create/retrieve storage instance
String storageid = "TestStorage";
XGRStorage storage = m_node.storage(storageid);
if ( storage == null ) {
   // failed to get storage instance, abort...
}

// lookup entry
String type = "xgr:test";
String datakey = "Name1";
XGREntryValue value = storage.lookup(type,datakey);
if ( value == null ) {
   // no data found
}

// lookup vector entry
String type = "xgr:test";
datakey = "Name3";
entryid = "Item2";
value = storage.lookup(type,datakey,entryid);
```

Java RMI API:

```java
import interplay.xgr.XGRNodeR;
import interplay.xgr.XGRStorageR;
import interplay.xgr.storage.data.XGREntryValue;

XGRNodeR m_node = < node binding >;
// create/retrieve storage instance
String storageid = "TestStorage";
XGRStorageR storage = m_node.storage(storageid);
```
5.1.7. Data Retrieve

Retrieve a data item value collection (vector of XML document): the data item to be retrieved is identified by its primary key and an XPath filter; all entries related to the primary key and matching the filter are returned (for non log/vector data type only one entry or none is returned).

Data items can be retrieved in bulks specifying a bulk max length and a starting entry id (void for the first request; the last value returned by the previous request for further requests).

Data retrieve is performed on a storage instance.
RApplication command line option:

```
-retrieve #<url>[#<filter>#<max>#<start>]:
```

lookup data by URL (DHT://<storageid>/<datatype>@<datakey>)

Examples:

```
#> java interplay.xgr.test.RApplication -host localhost:15001
-retrieve "#TestStorage#xgr:vector#Name1#/xgr:vector[id="Item2"]"
```

Bulk retrieve (blocks of 3 entries, filter with no condition):

```
#> java interplay.xgr.test.RApplication -host localhost:15001
-retrieve "#TestStorage#xgr:vector#Name1#/xgr:vector#3"
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xgr:vector xmlns:xgr="XGRSchema">
  <xgr:id>Name1</xgr:id>
  ...
</xgr:vector>
```

```
4 ---------------------
NextEntryPtr: Item4
```

```
#> java interplay.xgr.test.RApplication -host localhost:15001
-retrieve "#TestStorage#xgr:vector#Name1#/xgr:vector#3#Item4"
```

Java API:

```
import interplay.xgr.XGRNode;
import interplay.xgr.XGRStorage;
import interplay.xgr.XGRData;
import interplay.xgr.storage.data.XGREntryPtr;

XGRNode m_node = < node binding >;
// create/retrieve storage instance
String storageid = "TestStorage";
XGRStorage storage = m_node.storage(storageid);
```
if (storage == NULL) {
    // failed to get storage instance, abort...
}

// lookup entry
String type = "xgr:vector";
String datakey = "Name1";
String filter = "/xgr:vector[id="Item2"]";
Vector values = storage.lookup(type, datakey);
for (Iterator i = values.iterator(); i.hasMore(); ) {
    XGRData data = (XGRData) i.next();
}

// lookup entry in bulk mode
String type = "xgr:vector";
String datakey = "Name1";
String filter = null;
// bulk spec
int max = 3;
String start = ";
// bulk iteration
while (start != null) {
    // get next bulk of data (starting with ")
    values = storage.lookup(type, datakey, max, start, filter);

    start = null;
    for (Iterator i = values.iterator(); i.hasMore(); ) {
        Object next = i.next();

        // check for end of bulk (pointer to next bulk start entry)
        if (next instanceof XGREntryPtr) {
            start = ((XGREntryPtr)next).entryPtr;
            break;
        }

        // data value
        XGRData data = (XGRData) next;
    }
}

Java RMI API:

import interplay.xgr.XGRNodeR;
import interplay.xgr.XGRStorageR;
import interplay.xgr.XGRData;
import interplay.xgr.storage.data.XGREntryPtr;

XGRNodeR m_node = < node binding >;
// create/retrieve storage instance
String storageid = "TestStorage";
XGRStorageR storage = m_node.storage(storageid);
// Same as for Java API
5.1.8. **Storage Query**

Retrieve a data item value collection (vector of XML document): the data item to be retrieved are identified by an XPath expression.

Data items can be fetched in bulks of defined length using a query cursor. [Note: if the result set is very large, issuing a query without using the cursor may be very memory expensive for the XGR node that performs it as the full set is returned as single java Vector instance]

Data query is performed on a storage instance.

RApplication command line option:

```
-query #<storageid>#<xpathquery>[#<fetchnr>]:
```

query data (full query or fetching)

Note: the query mode (full or fetching) defines how the application implements the query, not the result

Examples:

```
#> java interplay.xgr.test.RApplication -host localhost:15001
-query "#TestStorage#/xgr:test[xgr:value1 > 100]#10"
```

Java API:

```java
import interplay.xgr.XGRNode;
import interplay.xgr.XGRStorage;
import interplay.xgr.XGRData;
import interplay.xgr.XGRQueryCursor;

XGRNode m_node = < node binding >;
// create/retrieve storage instance
String storageid = "TestStorage";
XGRStorage storage = m_node.storage(storageid);
if ( storage == NULL ) {
  // failed to get storage instance, abort...
}

// full query
String query = "/xgr:test[xgr:value1 > 100]";

Vector values = storage.query(query);
for (Iterator i = values.iterator(); i.hasMore(); ) {
  XGRData data = (XGRData) i.next();
}

// cursor query (bulk of 10 entries)
int bulklen = 10;

XGRQueryCursor cursor = storage.openQueryCursor(query);
while ( cursor.status() == XGRQueryCursor.OPEN ) {
  // retrieve next bulk
  Vector values = cursor.fetch(bulklen);
  for (Iterator i = values.iterator(); i.hasMore(); ) {
    XGRData data = (XGRData) i.next();
  }
}
Java RMI API:

```java
import interplay.xgr.XGRNodeR;
import interplay.xgr.XGRStorageR;
import interplay.xgr.XGRData;
import interplay.xgr.XGRQueryCursorR;

XGRNodeR m_node = < node binding >;

// create/retrieve storage instance
String storageid = "TestStorage";
XGRStorage storage = m_node.storage(storageid);
if ( storage == NULL ) {
    // failed to get storage instance, abort...
}

// full query
// same as Java API

// cursor query (bulk of 10 entries)
int bulklen = 10;
XGRQueryCursorR cursor = storage.openQueryCursor(query);
// same as Java API
```

### 5.1.9. DataStream Registration

Register an handler for receiving notification on data insertion matching an XPath expression.

DataStream registration is performed on a storage instance and is related to data inserted in the storage instance.

RApplication command line option:

```
-datastream #<storageid>#<query>#<times>: open a datastream and receive data
times = number of data items received before closing the stream [-1 never]
```

Examples:

```
#> java interplay.xgr.test.RApplication -host localhost:15001
   -datastream #TestStorage#/xgr:test[xgr:value1 > 100]#20
```

To test datastreams, an option to simulate periodic inserts on the storage is provided too:

```
-producer #<storageid>#<xmldata
or '/file:'filename>#<times>#<interval>: start a data producer
times = number of data published [default 1]
interval = delay (secs) between msgs
```

Data variables:
- `<NR>`: data sequence
- `<DATE>`: current date
- `<TIMESTAMP>`: current timestamp
Data variables are replaced in each data sent with the current values

Examples:

```java
#> DATAVALUE=' <?xml version="1.0" ?>
<xgr:test xmlns:xgr="XGRSchema" >
  <xgr:id>Nome1</xgr:id>
  <xgr:desc>Test Data Value</xgr:desc>
  <xgr:value1><NR/></xgr:value1>
  <xgr:value2><DATE> (<TIMESTAMP>)</xgr:value2>
</xgr:test>'
#> java interplay.xgr.test.RApplication -host localhost:15001
   -producer "#TestStorage#$DATAVALUE#100#10"
```

Java API:

```java
import interplay.xgr.XGRNode;
import interplay.xgr.XGRStorage;
import interplay.xgr.XGRMessageReceiver;
import interplay.xgr.XGRData;

XGRNode m_node = < node binding >;
// create/retrieve storage instance
String storageid = "TestStorage";
XGRStorage storage = m_node.storage(storageid);
if ( storage == NULL ) {
    // failed to get storage instance, abort...
}

// data insert notification handler
XGRMessageReceiver handler = new XGRMessageReceive() {
    public void handleMessage(XGRMessage msg) {
        String streamid = msg.subtopic; // the stream identifier
        XGRData data = (XGRData) msg.data; // the data inserted
    }
}

// register the handler for notifications
String query = "/xgr:test[xgr:value1 > 100]";
String streamid = storage.openDataStream(query, handler);
// main loop
...
// close the data stream
storage.closeDataStream(streamid);
```

Java RMI API:

```java
import interplay.xgr.XGRNodeR;
import interplay.xgr.XGRStorageR;
import interplay.xgr.XGRMessageReceiverRImpl;
import interplay.xgr.XGRMessageReceiver;
```
import interplay.xgr.XGRData;

XGRNodeR m_node = < node binding >;

// create/retrieve storage instance
String storageid = "TestStorage";
XGRStorageR storage = m_node.storage(storageid);
if ( storage == NULL ) {
  // failed to get storage instance, abort...
}

// data insert notification handler
// save as Java API

// RMI wrapper to notification handler
// [implements the RMI UnicastRemoteObject
// related to the XGRMessageReceiverR interface]
XGRMessageReceiverRImpl rmi_handler =
  new XGRMessageReceiverRImpl(handler);

// register the handler for notifications
String query = "/*xgr:test[xgr:value1 > 100]";
String streamid = storage.openDataStream(query,rmi_handler);

// main loop
...

// same as Java API

5.1.10. Message Publish/Subscribe

Topic based message publish/subscribe: messages are published on a channel instance associated to a topic (channelid) and are received by application subscribed to the channel.

Messages have the following attributes:
- topic: the channel topic (channelid)
- subtopic: a string classifying the message
- data: message data value (Object)

The channel subscription allow to specify filters on received messages:
- subtopic filter: regular expression on subtopic value [default: ‘.*’ (match all)]
- data filter (only for XGRData values): XPath condition [default: null]

Two special channel groups are created and managed by XGR:
- DataStream channels (channelid: DataStream#<storageid>#<streamid>): to be used by mean of the DataStream registration API
- Storage channels (channelid: XGRStorageNotification-<storageid>): all data insert/remove on the storage instance related toDATatypes with the flag notify=true are notified on the channel with: subtopic: the action performed (‘insert’, ‘remove’) data: an XGRDataEntry instance (data info/key and value)

RApplication command line option:
Message publishing:
-publish #<channelid>#<subtopic>#<string
or 'data:'xmlstring>#<times>#<interval>: start msg publisher
  times = number of message sent (default 1)
  interval = delay (secs) between msgs

Data variables:
  <NR>: data sequence
  <DATE>: current date
  <TIMESTAMP>: current timestamp

Data variables are replaced in each data sent with the current values

Examples:

#> java interplay.xgr.test.RApplication -host localhost:15001
   -publish "#TestChannel#stringmsg#Message NR <NR> (<DATE>)#100#10"

#> DATAVALUE=' <?xml version="1.0" ?>
   <xgr:message xmlns:xgr="XGRSchema" >
     <xgr:text>Message Text</xgr:text>
     <xgr:value1><NR></xgr:value1>
     <xgr:value2><DATE> (<TIMESTAMP>)</xgr:value2>
   </xgr:message>'

#> java interplay.xgr.test.RApplication -host localhost:15001
   -publish "#TestChannel#xmlmessage#data:$DATAVALUE#50#5"

Message subscription

-subscribe #<channelid>#<subtopic>#<filter>[#<times>]: subscribe to messages
  subtopic = value or regexp
  times = msg received before unsubscribe [-1 -> never]

Examples:

#> java interplay.xgr.test.RApplication -host localhost:15001
   -subscribe "#TestChannel#stringmsg##100"

#> java interplay.xgr.test.RApplication
   -host localhost:15001
   -subscribe "#TestChannel#xml.*#/xgr:message[xgr:value1 > 10]#100"

Java API:
import interplay.xgr.XGRNode;
import interplay.xgr.XGRChannel;
import interplay.xgr.XGRMessageReceiver;
import interplay.xgr.XGRMessage;
XGRNode m_node = < node binding >;
// get channel for topic
String channelid = "TestChannel";
XGRChannel channel = m_node.channel(channelid);
if ( channel == null ) {
    // failed to get channel, abort...
XGRMessage msg = new XGRMessage();
msg.subtopic = "stringmsg";
msg.data = "Message NR 1";
channel.publish(msg);

// message XML data publishing
String subtopic = "xmlmessage";
String xmlvalue = "
<?xml version="1.0"
<xgr:message xmlns:xgr="XGRSchema"
>  
<xgr:text>Message Text</xgr:itext>
  
<xgr:value1>100</xgr:value1>
  
<xgr:value2>Feb 10 2006 12:00:00</xgr:value2>
</xgr:message>
";
XGRData data = new XGRData(xmlvalue);
channel.publish(subtopic, data);

// message subscribing
// message notification handler
XGRMessageReceiver handler = new XGRMessageReceive() {
    public void handleMessage(XGRMessage msg) {
        String topic = msg.topic;
        String subtopic = msg.subtopic;
        Object data = msg.data;
        ....
    }
};

// register for message receive
String filter = "xml.*";
String datafilter = "/xgr:message[xgr:value1 > 10]";
channel.subscribe(handler, filter, datafilter);
....

// unsubscribe
channel.unsubscribe(handler);

Java RMI API:
import interplay.xgr.XGRNodeR;
import interplay.xgr.XGRChannelR;
import interplay.xgr.XGRMessageReceiverRImpl;
import interplay.xgr.XGRMessageReceiver;
import interplay.xgr.XGRMessage;
import java.rmi.server.UnicastRemoteObject;
XGRNodeR m_node = < node binding >;
// get channel for topic
String channelid = "TestChannel";
XGRChannelR channel = m_node.channel(channelid);
if ( channel == null ) {
    // failed to get channel, abort...
}

// message publishing
// same as Java API

// message subscribing
// message notification handler
// same as Java API

// RMI wrapper to notification handler
// [implements the RMI UnicastRemoteObject
// related to the XGRMessageReceiverR interface]
XGRMessageReceiverRImpl rmi_handler =
    new XGRMessageReceiverRImpl(handler);

// register for message receive
String filter = "xml.*";
String datafilter = "/xgr:message[xgr:value1 > 10]";
channel.subscribe(rmi_handler, filter, datafilter);

// unsubscribe
channel.unsubscribe(rmi_handler);

// shut client RMI interface
UnicastRemoteObject.unexportObject(m_rmiHandler, true);

5.1.11. Statistics Retrieval

XGR nodes keep statistics on storage function calls (number and average time to perform). An API is provided to retrieve a report on statistic values as an XML document with the following format:

<network>
  <runtime>
    <rt-indicator> value </rt-indicator>
    ....
  </runtime>
  <statistics instance="node-instance-name">
    <indicator> value </indicator>
    ....
  </statistics>
  ....
</network>
For each indicator the following infos are reported:

- **id**: indicator name (class instance and function)
- **type**: indicator type (avg: average value)
- **startts**: timestamp of start sampling time
- **currts**: timestamp of current sampling time
- **counter**: total number of function calls
- **value**: indicator value (average time for avg indicator types)

RApplication command line option:

```
-stats #reset: reset node/network statistics
-stats #[mode]: retrieve node/network statistics (mode: xml|tab)
```

Examples:

```bash
#> java interplay.xgr.test.RApplication -host localhost:15001
-stats "#reset"
```

```bash
#> java interplay.xgr.test.RApplication -host localhost:15001
-stats 
```

**Node Runtime Info**

```
<runtime>
<usedMem>942992</usedMem>
<freeMem>1088040</freeMem>
<maxMem>66560112</maxMem>
<totMem>2031616</totMem>
<nrThreads>2</nrThreads>
</runtime>
```

**Node XGRNode.BOOT-1 statistics**

```
1 -------------------
<statistics instance="XGRNode.BOOT-1">
<indicator>
  <id>XGRStorage.XGRStorageCatalog.Cursor</id>
  <type>avg</type>
  <startts>1139755303019</startts>
  <currts>1139762479979</currts>
  <counter>0</counter>
  <value>0.0</value>
</indicator>
<indicator>
  <id>XGRStorage.XGRStorageCatalog.Fetch</id>
  <type>avg</type>
  <startts>1139755303019</startts>
  <currts>1139762479979</currts>
  <counter>0</counter>
  <value>0.0</value>
</indicator>
```

```
#> java interplay.xgr.test.RApplication -host localhost:15001
```
Node Runtime Info

UsedMem: 863704
FreeMem: 1167328
TotMem: 2031616
MaxMem: 66650112
NrThreads: 2

Node XGRNode.BOOT-1 statistics

<table>
<thead>
<tr>
<th>Indicator</th>
<th>Type</th>
<th>Cnt</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>XGRStorage.XGRStorageCatalog.Cursor</td>
<td>avg</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>XGRStorage.XGRStorageCatalog.Fetch</td>
<td>avg</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>XGRStorage.XGRStorageCatalog.Insert</td>
<td>avg</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>XGRStorage.XGRStorageCatalog.Lookup</td>
<td>avg</td>
<td>1</td>
<td>60.0</td>
</tr>
<tr>
<td>XGRStorage.XGRStorageCatalog.Query</td>
<td>avg</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>XGRStorage.XGRStorageCatalog.Remove</td>
<td>avg</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>XGRStorage.XGRStorageCatalog.Retrieve</td>
<td>avg</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>XGRStorageMgr.Copy</td>
<td>avg</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>XGRStorageMgr.Fetch</td>
<td>avg</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>XGRStorageMgr.Lookup</td>
<td>avg</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>XGRStorageMgr.Remove</td>
<td>avg</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>XGRStorageMgr.Retrieve</td>
<td>avg</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>XGRStorageMgr.Scan</td>
<td>avg</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>XGRStorageMgr.Store</td>
<td>avg</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>XGRStorageMgr.StoreReplica</td>
<td>avg</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>XGRStorageMgr.Unstore</td>
<td>avg</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>XGRStorageMgr.UnstoreReplica</td>
<td>avg</td>
<td>0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Java API:

```java
import interplay.common.StatsManager;

// get stats
// Note: the value returned is not an XML document, but an XML sequence (the childs of the network tag)
String stats = StatsManager.asXMLSequence();
```

Java RMI API:

```java
import interplay.xgr.XGRNodeR;

XGRNodeR m_node = < node binding >;

// get stats
// Note: the value returned is an XML document [root element: nework]
String stats = m_node.statistics();
```

5.1.12. Virtual Data Management

VirtualData refers to data values managed in Local Storage mode virtually inserted in the XGR storage: only a reference to the data item is stored in the XGR storage and the data value is accessed by the platform using a custom class implemented and registered by
the data provider in the local XGR node (e.g. data could be related to rows in a RDMBS table, files on the file system, lines in a local log file, etc.). The data reference is managed by XGR as an opaque value; the syntax and semantic is defined by the custom handler.

The value handler class is responsible for:

- returning the data value as an XGREntryValue instance
- managing store/unstore/update requests
- managing data access request (XPath matching, data key/entry id value retrieval, index values retrieval)

A default implementation of access methods related to the XML data value is provided, so that a typical implementation needs only to return the value of the data item in the XML format (XGRData instance).

Handler instances are instantiated by a custom Factory class that must be registered on the node for a list of data types. RApplication and RMI support is limited to the inserting data specifying the data type and data reference; the node the request is sent to must have the related handler loaded/registered; the main usage of the functionality in applications embedded in the node instance.

Java API: a simple handler example managing data on files

```java
// FileValueHandlerFactory class: factory for FileValueHandler virtual data handler

import interplay.xgr.storage.XGRValueHandlerFactory;
import interplay.xgr.storage.data.XGREntryValueHandler;
import interplay.xgr.storage.data.XGREntryValue;
import interplay.xgr.storage.data.XGREntryDataPtrValue;

public class FileValueHandlerFactory implements XGRValueHandlerFactory {
    protected String m_storedir;

    public FileValueHandlerFactory(String storedir) {
        m_storedir = storedir;
    }

    public XGREntryValueHandler getValueHandler(String datatype,
                                                 XGREntryValue value) {
        XGREntryValueHandler hd1 = null;
        if ( value instanceof XGREntryDataPtrValue ) {
            hd1 = new FileValueHandler(m_storedir,
                                        ((XGREntryDataPtrValue)value).dataptr);
        }
        return hd1;
    }
}
```

// FileValueHandler class: simple data value handler for data stored on files
public class FileValueHandler extends XGREntryValueHandler {
    protected String m_fileptr;
    protected String m_filename;
    public final String PTRPRFX = "file:";

    // Constructor
    // fileptr: data value reference "file:<filename>"
    // filepath: base file directory
    public FileValueHandler(String filepath, String fileptr) {
        m_fileptr = fileptr;
        m_filename = filepath + "/" +
                      fileptr.substring(PTRPRFX.length());
    }

    // DataValue access method:
    // return the file content as an XGRData instance
    public XGREntryValue get() {
        // open the file and returns the content (XML data)
        String strvalue = < m_filename content >;
        XGREntryValue value =
            new XGREntryDataValue(new XGRData(strvalue));
        return value;
    }

    // Store request callback: do nothing
    public String store() {
        // do nothing (returns the dataptr)
        return m_fileptr;
    }

    // Remove request callback: do nothing
    public void unstore() {
        // do nothing
    }

    // Update request callback: do nothing
    public String update(String dataptr) {
        // do nothing (returns the dataptr)
        return m_fileptr;
    }
}

Handler registration on main:

import interplay.xgr.XGRNode;
import interplay.xgr.netwokr.XGRNodeImpl;
import interplay.xgr.storage.XGRStorageApp;
import interplay.xgr.storage.XGRStorageManager;

XGRNode m_node = < node binding >;
// Create handler factory
String filebasedir = "/datastore";
FileValueHandlerFactory factory =
    new FileValueHandlerFactory(filebasedir);

// Register the HandlerFactory for datatypes on local node
// storage manager
XGRStorageManager mgr =
    ((XGRNodeImpl)m_node).storageApp().storageManager();
mgr.registerValueHandler("xgr:vtype1", factory);
mgr.registerValueHandler("xgr:vtype2", factory);
mgr.registerValueHandler("xgr:vtype3", factory);

5.2. Babelpeers Programmer Guide

This section describes the main APIs which allow to use the services of Babelpeers component. A first approach to use Babelpeers is through the command line interface. As with XGR, a command line interface is provided for testing purposes which allows to request all the middleware services, and a java class ("de.tuberlin.cit.babelpeers.CmdLine") which mimics the functionality of the interface is provided for programmatic access:

port <port> set the port number of the BabelPeers node
host <host> set the host name of the BabelPeers node
full set the query mode to full results
count set the query mode to count only
exhaustive select the exhaustive query processor
dllite select the dllite query processor
abox <file> send an abox file
tbox <file> send a tbox file
query <file> send a sparql query from file
shutdown shutdown the BabelPeers node
quit quit this client

More advanced applications may want to use lower level APIs, which are provided to operate with specific services. The following documents the APIs for working with categories

5.2.1. Working with Categories

The following methods (provided by class "de.tuberlin.cit.babelpeers.CategoryManagement") allow fine grained access to category functionality:

public void loadOntology()  
    Load the ontology used for category management to the local peer
public void addCategory(String categoryName)  
    Creates a new category
public void removeCategory(String categoryName)  
    Removes a category
public String getCategoryDescription(String categoryName)  
    Retrieves the description of a category identified by its name.
public void setCategoryDescription(String categoryName, String description)  
    Sets or alters the description of a category.
public Set<String> getChildCategories(String categoryName)
public Set<String> getAllChildCategories(String categoryName)  
Retrieves the set of all child categories for this category 
(including the transitive closure).

public void addChildCategory(String categoryName, String child)  
Inserts a new direct child category.

public void removeChildCategory(String categoryName, String child)  
Removes a category from the set of child categories. 
This does not remove the category "child" itself, 
just the link from name to child.

public Set<String> getParentCategories(String categoryName)  
Retrieves the set of direct parent categories for this category, 
without the transitive closure.

public Set<String> getAllParentCategories(String categoryName)  
Retrieves the set of parent categories for this category, 
including the transitive closure.

public void addParentCategory(String categoryName, String parent)  
Inserts a new parent category.

public void removeParentCategory(String categoryName, String parent)  
Removes a category from the set of parent categories. 
This does not remove the category "parent" itself, 
just the link from categoryName to parent.

public void relateCategories(String categoryName1, String categoryName2)  
Creates a "related-to" association between the two categories.

public void unRelateCategories(String categoryName1, String categoryName2)  
Removes a "related-to" association between the two categories.

public Set<String> getRelatedCategories(String categoryName)  
Retrieve a set of all categories that are related to this category.

public void classifyResource(String resource, String categoryName)  
Add a category to a resource (e.g. a resource is an URL to some document).

public void unClassifyResource(String resource, String categoryName)  
Remove a category from a resource 
(e.g a resource is URL to some document). 
Only a single category is removed, not all 
classifications for this resource.

public Set<String> getResources(String categoryName)  
Retrieve a set of all resources that are classified by categoryName.

public Set<String> getAllResources(String categoryName)  
Retrieve a set of all resources that are classified by categoryName, 
also those that are classified by subcategories.

References


