Monitoring the Dynamism of the Linked Data Space through Environment Abstraction

Riza Cenk Erdur
Department of Computer Engineering, Ege University
35100 Bornova, Izmir, Turkey
cenk.erdur@ege.edu.tr

Oylum Alatli
Department of Computer Engineering, Ege University
35100 Bornova, Izmir, Turkey
oylum.alatli@ege.edu.tr

Tayfun Gökmen Halaç
Department of Computer Engineering, Ege University
35100 Bornova, Izmir, Turkey
tayfunhalac@gmail.com

Oguz Dikenelli
Department of Computer Engineering, Ege University
35100 Bornova, Izmir, Turkey
oguz.dikenelli@ege.edu.tr

ABSTRACT

Management of data dynamics is a major issue for linked data applications. There are some approaches and applications addressing this problem which are either pull or push based. However, these one sided solutions do not solve the problem since pull based approaches bring an overhead to data consumers, and push based approaches require the collaboration of the data publishers which is contrary to the free nature of the Web of Data. In this paper, to monitor the dynamism of the linked data space, we present an infrastructure that combines pull and push approaches to monitor the changes of SPARQL queries over the linked data space. In the design of this infrastructure, environment abstraction of multi-agent systems domain is followed. The proposed infrastructure has been built upon the CArtAgO environment infrastructure, and VoID based federated query engine, WoDQA, is incorporated into CArtAgO to handle dynamic discovery of datasets and execution of monitored SPARQL queries.

1. INTRODUCTION

The linked data space has expanded dramatically following the introduction of the linked data concept by Tim Berners-Lee in his technical note [3] where he manifested the basic principles for making the data published on the web to become part of a global data space which is usually referred to as "Web of Data". Today, different organizations from different domains such as government agencies, media, entertainment, etc. have created a huge linked data space by publishing and linking their data, towards the realization of the semantic web vision.

In such a huge and continuously expanding linked data space, requirements can be much more complex than just publishing linked data and querying it using the power of SPARQL. For example, it becomes a critical task for asynchronous intelligent software[1] to monitor the changes on an application related part of the linked data space and react to these changes in an appropriate way. From the perspective of the developers of such kind of software, an infrastructure that simplifies the usage and monitoring of the linked data space would be an essential requirement.

In building an infrastructure for monitoring the linked data space, there are three critical issues that needs to be addressed. The first issue is to decide on the basic entity that will be monitored so that the changes in the linked data space can be detected. The second issue is to decide on the representation formalism that will be used for representing the changes once they are detected. Finally, the third issue in general addresses the monitoring architecture used. Communication mechanisms of the publishers and consumers, and hence the change notification mechanism are included in the monitoring architecture. These issues are briefly discussed below.

Regarding the first issue, our approach is to monitor SPARQL queries, since a SPARQL query defines a sub-graph of the linked data space and a sub-graph may include elements covered by several different datasets residing on the linked data space. Using SPARQL queries, we can also monitor resources and statements on the linked data space.

To deal with the second issue, we first need to consider the types of changes. Since we monitor SPARQL queries, the changes fall into three categories which are resultset, graph, and boolean result. These categories correspond to "select", "construct" (and also "describe"), and "ask" query types in SPARQL. To represent changes in graphs, we use the change ontology proposed by Palma et. al. [6] which is one of the most recent studies that deals with the OWL2 specification.

There are two basic approaches for designing the monitoring architectures regarding the communication mechanisms

[1]By the term asynchronous intelligent software, we mean any application that runs on a server, a desktop, or a mobile device which reacts autonomously to asynchronous events occurred outside of the application. When we use the term agent in this paper, we refer to such software as the well-known semantic web scenario does.

ISEM '13, September 04 - 06 2013, Graz, Austria.

http://dx.doi.org/10.1145/2506182.2506193.

of publishers and consumers [10]. These basic approaches are called pull and push. In the pull mechanism, the consumer has to monitor the data publisher by means of periodic queries. In the push mechanism, the consumer subscribes to the data publisher and then the data publisher acknowledges the changes to subscribers. In the literature, work conducted within the context of linked data dynamics focuses on either the pull or the push approach. However; because of the highly dynamic nature of the linked data space, we think that pull and push approaches are needed to be combined in order to monitor the linked data space. In this paper, we introduce a monitoring infrastructure that combines the pull and the push approaches.

As a result of the dynamism of the linked data space, datasets that are covered by a monitored SPARQL query may change over time. In such a case, the push approach cannot be used, since it is not exactly known by the application which resources to monitor. What is required is a pull mechanism which has to discover all new datasets each time before querying. This pull mechanism constitutes the linked data space interface of the proposed infrastructure. To realize this pull mechanism, we have used a federated query engine called WoDQA [1] which discovers related datasets using VoID (Vocabulary of Interlinked Datasets) [2] descriptions. By incorporating WoDQA, the proposed infrastructure becomes capable of monitoring a SPARQL query which spreads over multiple datasets.

From the application development perspective, an application knows what data to monitor and can accordingly define the SPARQL queries to be monitored. Thus, here a push mechanism is needed. Using the push mechanism, the application registers the SPARQL queries that will be monitored to the infrastructure and then waits for being notified in case of a change. This push mechanism constitutes the application side interface of the proposed infrastructure. The hybrid approach of combining pull and push simplifies the interaction of developers with the linked data space, since developers only need to submit their SPARQL queries to the monitoring infrastructure without a need for specifying the datasets on which the query parts would be executed.

During the design and the realization of the proposed monitoring infrastructure, we used the existing environment abstraction [11] of the multi-agent system research area. A&A (Agents and Artifacts) [5] is a well-known meta-model used in the modeling of agent environments and CArtAgO [9], which is an open source implementation of the A&A meta-model, provides the necessary constructs to program the agent environment. We specialized the basic A&A meta-model and the CArtAgO's basic constructs to design and implement the proposed infrastructure which provides SPARQL query monitoring facilities to linked data application developers through the components (i.e. artifacts) that it includes. Environment, which includes all the resources and services that an agent needs, shields the agent developer from the complexity of the outside world. In a similar manner, we aimed at shielding the linked data application developers from the complexity of developing on the linked data space, by following the environment abstraction of multi-agent systems during the design and implementation of the proposed linked data monitoring infrastructure.

In the rest of the paper, we first give a motivating scenario in Section 2. The conceptual architecture of the proposed infrastructure is given Section 3. Section 4 describes the implementation level details of our linked data environment infrastructure. Section 5 exemplifies the use of the developed infrastructure by giving a sample scenario. Related work is discussed in Section 6. Finally, the conclusion is given.

2. MOTIVATING SCENARIO

In order to demonstrate the need for the proposed monitoring infrastructure, we constructed a sample use case from the social semantic web domain which is inherently a highly dynamic domain. This sample use case demonstrates the need for the proposed monitoring architecture both from the push and the pull perspectives. The scenario in this sample use case is mainly based on an hypothetical enterprise which develops different software applications using several datasets in the social semantic web domain. We assume that one such application is an advertisement recommender application which uses data mining techniques to find out the most appropriate advertisements for a target mass by matching the advertisements with the demographic and personal interest data residing in the social network datasets. The application then sends by e-mail to the target mass the advertisements that are matched. We further assume that the other application is an activity recommender software which recommends its users various social activities also according to their personal interest data exposed through the social networks. Users are constantly exposing their interests through the social networks, a lot of new activities are published by organizations such as ticket selling websites, and many advertisements are being featured on the web. Thus, either the advertisement or the activity recommender software needs to monitor the linked data space to inform its user about new advertisements or activities that the user may be interested in.

Although these two recommender applications may use some different datasets such as the ones related with advertisement data and activity data to fulfill their requirements, there are also some common specific datasets in the social semantic web domain, using which both of the applications achieve their goals. For example, each application uses personal interest data residing in the social network datasets, and relates users' interest data with other datasets from the linked open data cloud so that detailed information can be gathered about the exposed interests. Since there are common parts of the linked data space that each application needs to monitor, it will be the case that each application monitors similar or the same queries. However, monitoring the same query in every application in a way that is independent of each other would bring an extra overhead for each application and cause a network traffic. Such a problem can be eliminated if the responsibility of monitoring the linked data space through queries can be given to a middleware specifically designed for that purpose. Then, through the proposed infrastructure, each application will be able to use in common the monitoring entity which is responsible for monitoring a specific query. This discussion demonstrates the need from the push approach perspective for the proposed infrastructure.

The other perspective that needs consideration is about the specification of the datasets for each query that will be monitored. To demonstrate this perspective, let us take the activity recommender software, which recommends the users new movies in which a favorite director, actor, etc. of a user participates, as an example. We assume that the user's in-

```
PREFIX theatre: <http://155.223.25.235:8180/sosep/ontology/moviesession.owl#>
PREFIX lmdb: <http://data.linkedmdb.org/resource/movie>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX users: <http://155.223.25.235:8180/dataset/socialsemantic/facebook/resource/>
PREFIX social: <http://155.223.25.235:8180/sosep/ontology/semanticFB.owl#>

SELECT ?newMovie ?visionDate {
      SERVICE <movietheatre/sparql> {
            ?session theatre:showtime ?visionDate .
            ?session theatre:visionFilm ?newMovie . }
      SERVICE <linkedmdb/sparql> {
            ?newMovie lmdb:actor ?actor . }
      SERVICE <dbpedia/sparql> {
            ?person owl:sameAs ?actor . }
      SERVICE <socialnetwork/sparql> {
            users:71373792 social:likes ?person . }
}
```

Figure 2.1: The federated query monitored by activity recommender software

terests can be queried through social network's SPARQL endpoint, the social network uses DBpedia resources to express interests, and a movie theater dataset publishes showtimes by linking them to linked open data cloud. To realize this goal, the activity recommender system uses the query shown in Figure 2.1. This is a federated query, since different parts of the monitored information resides in several RDF datasets. The query result changes if a new movie is added to the movie theater's dataset, or the user likes a new person in the social network, etc. Also, the number of movie theaters which are considered by the application may be more than one and change over time. It is obvious from the use case that the activity recommender software needs to specify the datasets explicitly for each query that will be monitored. This may not be an acceptable situation when the dynamic nature of the linked data is considered. The datasets that are specified for a monitored query may change dynamically. What is needed here is a mechanism that will act on the behalf of the application (i.e. activity recommender software) in choosing the right datasets dynamically for each monitored query. This mechanism, which in our case is a federated query engine that can discover related datasets using VoID descriptions, constitutes the pull-side of the proposed infrastructure.

3. CONCEPTUAL ARCHITECTURE

The proposed monitoring infrastructure has been designed and implemented based on the environment abstraction [5, 11] used in multi-agent systems. In the multi-agent systems literature, the well-known A&A meta-model is used to model the environment in a multi-agent system. During the design of the proposed infrastructure, we also used the A&A meta-model and specialized it for our purposes. Thus, to form a basis for our discussion on the conceptual architecture of the proposed infrastructure, we will first give an overview of the A&A meta-model.

The artifact concept lies at the core of the A&A meta-model. Artifacts are the building blocks of the environment and provide specific functionalities for agents. An artifact has a usage interface which defines the operations that an agent can execute on that artifact. There are two types of actions provided for an agent that can be performed on an artifact. The first one is the use action through which the agent can execute the operations in the usage interface of the artifact. The other one is the focus action through which an agent can start to observe specific properties of the

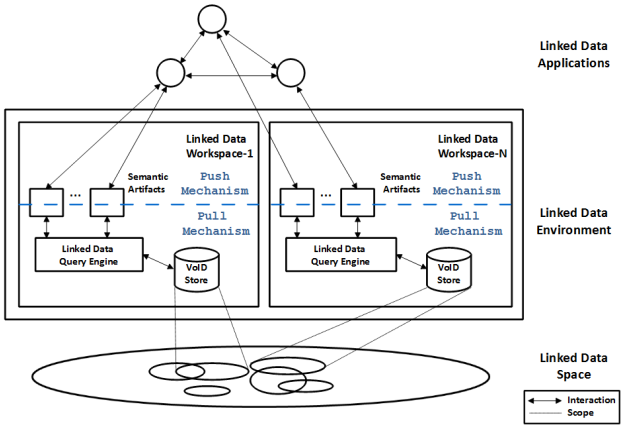

Figure 3.1: Conceptual architecture which combines push and pull mechanisms to monitor the linked data space

artifact. In our case, the focus action needs special consideration, since an application focuses on artifacts which are monitoring specific SPARQL queries. Additionally, events generated as a result of the operations triggered by other agents can also be observed by a specific agent. Finally, the artifacts can interact with each other through their link interfaces. A workspace is defined as a logical container of agents and artifacts. It organizes agents and artifacts from a topological perspective and defines the scope for interacting with the environment. An environment can be considered as a collection of workspaces.

With the workspaces and artifacts, the A&A meta-model provides us with the necessary basic constructs to design the combined pull-push monitoring infrastructure for the linked data space. Figure 3.1 shows the conceptual architecture of the proposed infrastructure which is built upon the environment abstraction and hence upon A&A meta-model. This conceptual architecture consists of three layers as shown in Figure 3.1.

The top layer includes linked data applications that need to interact with the datasets on the linked data space, as well as with each other. Using the push mechanism, the applications at this layer registers SPARQL queries to the semantic artifacts residing in the linked data workspaces at the middle layer. It is assumed that the application knows the

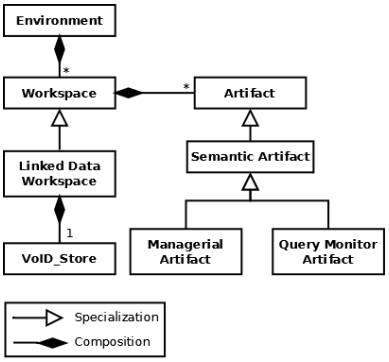

Figure 3.2: Specialized A&A Meta-model

vocabularies related with the datasets which the workspace includes.

The middle layer called "linked data environment" is at the core of the architecture. Both the push and the pull mechanisms are included in this layer. Using, the pull mechanism in this layer, the linked data environment interacts with the linked data space through the linked data query engine and VoID stores. A federated query engine should be used to realize the pull mechanism. Based on the VoID stores, the linked data query engine decides which datasets will be covered by a specific query at each time it pulls the data. On the other hand, the linked data applications interact with the linked data environment using the push mechanism which is also in this layer. Using the push mechanism, the linked data applications register the SPARQL queries that they need to monitor to the linked data environment via the semantic artifacts and then wait for the change notification events coming from the semantic artifacts by focusing on these artifacts.

As shown in Figure 3.1, the semantic artifacts are the basic building blocks of the combined pull-push architecture. To define the semantic artifacts, we have specialized the original basic artifact concept in the A&A meta-model. Semantic artifacts are also further specialized to include the sub-categories shown in Figure 3.2. These sub-categories are called managerial artifacts and query monitor artifacts. While managerial artifacts deal with the managerial aspects such as registrations of queries, query monitor artifacts provide the means for focusing on a specific part of the linked data space and are responsible for keeping track of the changes in that focused part. Managerial aspects of the environment are adding/removing VoIDs, and creating proper query monitor artifacts for the registered query. There is a different type of query monitor artifact for each type ("select", "construct", "describe", "ask") of SPARQL query. A query monitor artifact executes the query which it is responsible for, and then detects the changes in the result of the query, and notifies the change to the interested agents.

The other concept we have introduced to specialize the A&A meta-model for our purposes is the linked data workspace. We define linked data workspace as a logical entity that specifies the boundaries of a subset of the linked data space which all applications need to interact with. The main element which a linked data workspace includes is the VoID store. VoID is an RDF schema vocabulary that provides the terms and patterns for defining meta-data of the linked datasets. The VoID store contains the VoID documents which specify the logical boundaries of a specific linked data workspace. VoID documents belonging to datasets that are newly assigned to a linked data workspace can be dynamically added to that workspace's VoID store through the managerial artifact. Thus, the scope of a linked data workspace can be widened at run-time as new datasets related with that linked data workspace begin to appear in the application domain. Similarly, the scope can be narrowed at run-time by removing the VoID document belonging to a dataset which is undesired any more. This way, the dynamism of the linked data space can be handled, fulfilling the requirement given in the motivating scenario.

4. LINKED DATA ENVIRONMENT IMPLEMENTATION

We have used CArtAgO framework [9] to implement the conceptual architecture presented in the previous section. CArtAgO is an open source Java based implementation of the A&A meta-model. Two main entities of CArtAgO are artifact and workspace. Using CArtAgO, one can design an environment in terms of a set of first-class computational entities called artifacts which are collected in workspaces. A CArtAgO environment consists of one or more workspaces which are used to divide the environment into meaningful parts. A workspace can contain some predefined artifacts such as the factory artifact which in charge of instantiating other artifacts required within the context of the application. Artifacts can hold some values representing the state of a part of the environment and can provide some operations which can change the state of the environment.

Artifacts can be created and destructed dynamically by either agents or other artifacts. Artifacts provide means for interacting with the environment either by being changed or by being sensed. An agent must join a workspace to interact with an artifact in it. Agents can join or leave workspaces at run-time. Joining a workspace means that the agent needs to gather information within the context of the workspace. To be informed about the changes on an artifact, an agent must focus on that artifact. Because, when a property of an artifact changes, the artifact sends a percept to the agents which have already focused on it. The percept which includes the new value of the changed property of the artifact is interpreted in the agent's business logic when the percept is sensed by the agent.

Figure 4.1 shows the overview of the architecture implemented on top of CArtAgO[2]. The top layer in this figure includes the linked data applications which use the linked data environment. Our approach abstracts the application from the datasets and provides a transparent access to the datasets through the environment. These applications are the agents in the A&A meta-model. They join the linked data workspace and then execute and monitor SPARQL queries on the datasets within the context of the workspace. Different applications, which probably interacting with each other, can utilize the same workspace[3].

[2]Framework implemented on top of CArtAgO can be downloaded from http://seagent.ege.edu.tr/etmen/lde

[3]Current implementation allows multiple applications to access to the same workspace through the RMI support of the CArtAgO workspace. More suitable option for the web is

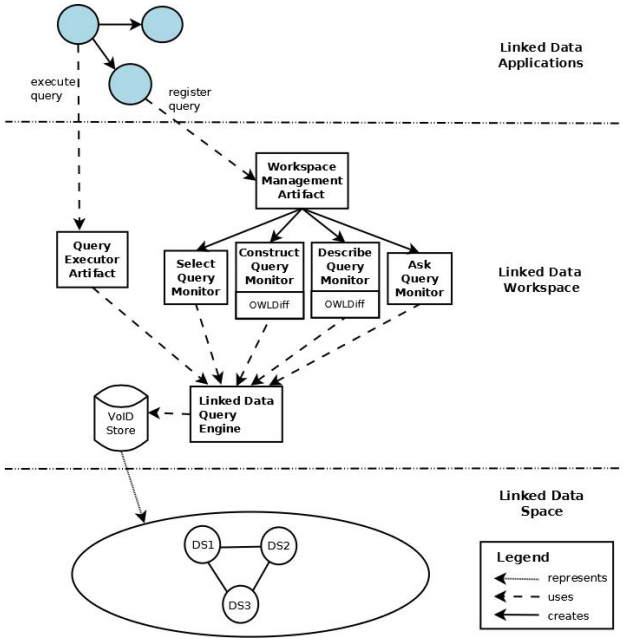

Figure 4.1: Linked Data Environment Infrastructure

To create a linked data environment infrastructure using CArtAgO, we first implemented linked data workspace concept of the conceptual architecture by extending the default workspace component of CArtAgO. Then, we defined new artifact types which are responsible for monitoring linked data space. The middle layer shown in Figure 4.1 depicts this specialized workspace with new types of artifacts in it. Implemented artifacts are workspace management artifact (WMA), the query monitor artifacts for different SPARQL query types, and the query executor artifact.

When a linked data workspace is instantiated, a WMA, which is a managerial artifact, is created in the workspace. The main responsibility of the WMA is the creation of query monitor artifacts. The WMA includes a hash-table which holds monitoring records. Each monitoring record is composed of the query being monitored, the agents focused on a specific part of data space by means of that query, and the artifact that is used for monitoring that query. Agents communicate to the WMA for a query monitor request. the WMA manages these requests, and creates a new monitor artifact for each new query.

An important component of the linked data workspace is an RDF model which corresponds to the VoID store. Each workspace covers several RDF datasets and the queries in that workspace are spread over these datasets. VoID store is the projection of the linked data space shown at the bottom layer of the architecture. This way, the data within the context of the workspace is specified. At run-time, this space may be expanded or narrowed by manipulating the VoID store. As mentioned previously, a workspace is used to divide the linked data cloud to smaller spaces. This eliminates the necessity of manual dataset and query management for the developer.

providing a REST interface for the workspace, which will be implemented as a future work.

Each linked data workspace includes a linked data query engine. It is responsible for distributing queries over related datasets in the workspace. We used WoDQA federated query engine [1] in the implementation. All artifacts share the WoDQA instance created in the workspace. To execute a query, an artifact passes the query to WoDQA, and then WoDQA discovers the relevant datasets which the sub-parts of the query will be sent. WoDQA employs VoID descriptions of datasets which are encapsulated by the linked data space. VoID descriptions allow to specify the topology of the linked data space. Vocabularies used in a dataset, urispaces which the dataset's resources are defined in, linksets between dataset, and the relations between triple patterns of the query are used to decide the datasets and decompose the query. Based on these descriptive properties of datasets, WoDQA selects the relevant datasets effectively, and queries only the selected datasets for a query. Thus, executing a query does not place a burden on all datasets in the linked data space.

There are two main requirements for an agent interacting with the linked data workspace. The first one is executing a SPARQL query on that linked data space whenever it is needed. For this case, an agent requests the execution of the query from the query executor artifact. Once requested, this artifact executes the query by means of WoDQA and returns the results to the agent. The second requirement is monitoring changes in the results of the queries. To fulfill this requirement, we implemented four types of query monitor artifacts each of which corresponds to a specific SPARQL query type. These artifacts are SelectQueryMonitor, DescribeQueryMonitor, ConstructQueryMonitor and AskQueryMonitor. All these artifacts use the WoDQA of the linked data workspace. When an agent needs to monitor a query in the workspace, it registers this query to the WMA. To do this, the agent calls the registerQuery operation provided by WMA (i.e. the agent runs WMA's use method with the registerQuery operation as a parameter) which takes the query that will be monitored, and the time interval that the agent specifies as the monitoring frequency. When the WMA receives the request of the agent for its registerQuery operation, it first searches the hash-table for an artifact that monitors the query given to it. If the WMA finds such an artifact, it returns the artifact's id to the requester agent. If there is not an artifact monitoring the given query, the WMA creates a new one and returns its id to the requester agent. When the requester agent receives the artifact id of the query monitor artifact, it focuses on that artifact in order to receive the percepts (i.e. change notifications) that will come.

As mentioned previously, WMA is responsible for creating the proper query monitor artifact for a query to facilitate the programming of the environment. The agent only registers a query and focuses on the created query monitor artifact. Then, it handles the percepts which are sent by the query monitor artifact for the changes in data. For each query monitor artifact, a monitoring thread is created to handle the time interval requested by the agent. This thread, periodically, performs the executeQuery operation of the query monitor artifact according to the requested time interval. executeQuery is the operation which executes the query by means of WoDQA and investigates whether there is any change in the query result.

An artifact's monitoring cycle causes a load on the datasets

and on the monitoring system. An important factor on this load is the frequency of query executions. We allow the developer to specify different monitoring frequencies for different queries or even for different registrations of the same query. Therefore, to minimize the query load on the datasets, the implemented environment makes sure that only one artifact monitors a specific query. The artifact which is responsible for monitoring a query checks whether there is any change for the smallest frequency, and uses the most recent result when sending percepts. To make this pull mechanism more efficient, the artifact should be able to determine the query frequency proactively. Also, caching the results retrieved from each dataset and following the change frequency of each dataset could be considerable. For this reason, as future work, we plan to enhance the query monitor artifacts by incorporating such intelligent caching mechanisms.

The first query monitor artifact is SelectQueryMonitor which monitors select type SPARQL queries. After SelectQueryMonitor executes a query, it compares the result with the previous one. This artifact has two observable properties: result property which is a Jena resultset containing the last retrieved result, and difference property which is an object containing the added and deleted rows of the resultset. When those properties are updated, CArtAgO infrastructure generates a percept that the agents focused on the SelectQueryMonitor artifact can sense. An application can use which property it requires when it receives the percept.

ConstructQueryMonitor and DescribeQueryMonitor artifacts monitor the construct type and describe type queries respectively. They both have an observable changeDesc property, and use OWLDiff [4] instance. OWLDiff is an ontology comparison tool that detects and lists the syntactic and semantic differences (by using the Pellet reasoner) between two versions of an ontology. These artifacts compare two RDF graphs using OWLDiff and create the difference graph conforming the Change Ontology for OWL. When the comparison of the previous and the current result graphs creates a new difference graph which includes two versions of the result graph where the found differences are annotated, this graph is set to changeDesc property. Whenever the changeDesc is updated, the CArtAgO infrastructure generates a percept for the agents focused on the related artifact. Since construct and describe queries return a graph, agents monitoring these queries receive a description of the change in the result.

The final query monitor artifact, AskQueryMonitor, monitors ask type SPARQL queries. The observable property of the AskQueryMonitor artifact is a result which is defined as boolean. After the artifact executes the ASK query, the result property is updated if the query result is different. Therefore, this artifact generates a percept which includes boolean query result value.

The infrastructure explained in this section separates the linked data application development process into three steps. The first step is describing the linked data spaces which the application will use by designing the linked data workspaces. Thus, different kinds of queries can be constructed on different linked data spaces. The second step is defining the queries which the application needs to execute or monitor, and then associating these queries to appropriate workspaces. Finally, the last step is using the results of the queries and interpreting the change notifications in the application's business logic. Thanks to the implemented linked data environment infrastructure, monitoring dynamic linked data space becomes simpler for the linked data developer.

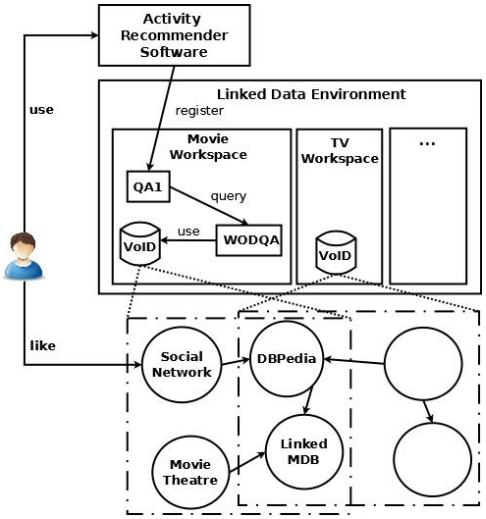

Figure 5.1: Architectural diagram for example linked data environment

5. USE CASE IMPLEMENTATION

In order to demonstrate the usage of the proposed linked data environment, we implemented the activity recommender application which is one of the applications that are introduced in the motivating scenario given in Section 2. For the activity recommender application, we used four interconnected datasets, two of which are well known DBpedia and LinkedMDB datasets. In order to provide the datasets required for implementing this application, we had to construct the other two datasets, which are Social Network and Movie Theater Showtime datasets, ourselves. Social Network dataset is composed of the Facebook data of authorized users. The data about the likes of those users are retrieved from the Facebook social network and stored into an RDF dataset. On the other hand, Movie Theater Showtime dataset contains artificial data about the showtimes of an imaginary movie theater.

For our implementation, we constructed a movie workspace as an instance of the linked data workspace concept introduced in Section 3. Other possible workspaces could be TV workspace for recommending programs, music workspace for recommending concert events, etc. To define the region of linked data space that movie workspace would monitor, we recorded VoID description of the aforementioned four datasets to the VoID store in the workspace. The architectural diagram of this solution can be seen in Figure 5.1.

We developed the activity recommender system using the the movie workspace. It monitors the workspace for the new movies of the favorite actors of its users. In order to do this, activity recommender software creates a SelectQueryMonitor artifact named as QA1 by the help of the WMA of the movie workspace and configures the artifact to execute the query seen in Figure 5.2 at intervals specified by the user. The query searches for the movies which have a session in a movie theater and their casts include an actor liked by the specific user. This represents an RDF graph which resides on more than one dataset and is needed to

```
SemanticChangeInterpreter interpreter =
      new SemanticChangeInterpreter() {
            public void interpret(ArtifactId sourceArtifact,
                  boolean change) {
            // ...
            }
            public void interpret(ArtifactId sourceArtifact,
                  Model change) {
            // ...
            }
            public void interpret(ArtifactId sourceArtifact,
                  ResultSet change) {
            // handle movie query result...
            }};
EnvironmentAdapter adapter = new EnvironmentAdapter(
            "recommender71373792", "movieWorkspace",
            interpreter);
adapter.focusQuery(QueryFactory.create(
                  "SELECT ?newMovie ?time {"
                  + "?session theatre:showtime ?time . "
                  + "?session theatre:visionFilm ?newMovie . "
                  + "?newMovie lmdbmovie:actor ?actor . "
                  + "?person owl:sameAs ?actor . "
                  + "?users:71373792 social:likes ?person . "),
                  864000000);
```

Figure 5.2: Registration of a query to the movie workspace

be monitored by the application. But, using our infrastructure, it is enough to use the example code shown Figure 5.2 in the application to monitor the query. The behavior in the *interpret* method of *SemanticChangeInterpreter* is done when a percept is generated by the artifact which is responsible to monitor this query. The artifact passes the query to the local WoDQA instance with the period specified by the user, for example once in a day as shown in the example code. WoDQA spreads the query over the datasets (as it is shown in the query in Figure 2.1) and returns the resultset to the artifact. Upon receiving the resultset, the artifact compares the new resultset with the previous one. If the artifact detects a difference, which means that a new movie of the favorite actor of the user is being shown, it notifies the activity recommender software with a percept. When the activity recommender receives this percept, it picks up new movies from the percept and shows a notification about the showtime of each new movie to its user.

As the implementation of the activity recommender application shows, the query monitor artifact employs the combined pull-push approach for handling the dynamism of the linked data space. The query monitor artifact pulls the data from related datasets by querying the datasets repeatedly via WoDQA query engine, and then pushes the changes to the applications focused on it. This mechanism abstracts the developer from two challenges. The first one is the distribution of the queries on linked data space, and the other one is the management of change detection. From the application developer perspective, our observation is that the proposed infrastructure shields the developer of the activity recommender application from the complexity of the linked data cloud by taking over the responsibility of handling the dynamism of linked data.

6. RELATED WORK

The research area that investigates the means and ends of detecting, representing and notifying the changes on linked datasets is called linked data dynamics. The current linked data dynamics research [10] classifies change detection and description mechanisms into three levels of granularity: dataset, resource and statement. The granularity specifies the entity whose change will be detected, e.g. addition and deletion of statements for statement granularity. However, since linked data applications are influenced by the changes in the results of the queries they run, adapting a query level granularity can be beneficial in practice. Detecting changes at query level involves specifying changes in the results of queries. Among the linked data dynamics approaches such as sparqlPuSH [7], DSNotify [8], SDShare[4] and WebHooks[5] in the literature, only sparqlPuSH [7] and DSNotify [8] use the query level granularity. But, the novelty of the approach proposed in this paper is monitoring a dataspace which spreads over several datasets and is described by a federated query that is dynamically constructed.

Another perspective to compare linked data dynamics approaches is the method of change notification. Change notification is related with the means linked data applications are notified about the dataset changes. Approaches to change notification are divided into two categories: pull and push.

In pull based approaches client applications periodically check for change. This approach brings an overhead for the client. On the contrary, in push based approaches, instead of checking periodically, data sources (or change detection systems) notify the client about changes whenever a change occurs in the monitored data. If a pull based system notifies clients about the changes that they have registered for, this system is characterized as supporting both pull and push approaches.

An example system which supports both pull and push based approaches together is PubSubHubbub[6] which is a web based publish-subscribe protocol. This protocol relies on three components: client application, hub and data provider. The protocol can be applied with or without the support of the data provider. Data providers that cooperate with a hub notify it about the data changes. When the hub gets the notification, it notifies the clients which subscribed for those changes. On the other hand, if the data provider does not cooperate with the hub, the hub queries the data provider periodically and notifies the subscribed clients. PubSubHubbub protocol has a scalable nature since it allows to build distributed notification architectures on the web. In our infrastructure, it is currently possible to build such architectures by constructing artifacts which monitor remote artifacts, since CArtAgO supports remote artifacts.

SparqlPuSH is an implementation of PubSubHubbub protocol for RDF data stores. It implements the part of the PubSubHubbub protocol that utilises data provider support. It requires new data to be loaded to the RDF store through its interface to be able to detect the change and notify clients. However, forcing data publishers for cooperation decreases the flexibility that is provided by linked data principles. More important than that, sparqlPuSH can monitor changes only in a single RDF store. However, linked data applications may need to combine data from many RDF stores by means of federated queries [1] instead of isolating them from each other. Therefore, monitoring interconnected datasets together in the context of a query is more suitable for linked data applications.

Another combined pull-push architecture is DSNotify [8]. DSNotify does not require the cooperation of data pub-

[4]http://www.sdshare.org/spec/sdshare-current.html
[5]http://www.webhooks.org/
[6]http://code.google.com/p/pubsubhubbub/

lishers. It monitors data providers, takes differences, and communicates them to the subscribed applications itself. However, like sparqlPuSH, DSNotify can only monitor a query on only one endpoint, and does not support federated queries. From the perspective of notification, our approach is a combined pull-push architecture for monitoring federated SPARQL queries. It pulls the data by querying more than one RDF store, and pushes the changes to the clients.

Another important aspect of data dynamics research is the means to express changes, in other words, change description. Our approach notifies the client about changes according to SPARQL query type. To describe changes in the results of construct and describe queries, we employ Change Ontology for OWL [6] which is a fine grained vocabulary. This vocabulary can express changes in RDF graphs at three abstraction levels: atomic, entity and composite changes. On the other hand, for select and ask queries, we use an internal representation for the difference.

7. CONCLUSION

In this paper, we proposed to exploit environment abstraction to simplify the use of linked data sources in the applications and also to monitor the changes in the data sources in accordance with the requirements of the applications. Besides the advantage of the flexibility provided by linked data for integrating data from various sources on the web, linked data space has an intrinsic complexity for the linked data developer because of the large amount of dynamically changing data and links it contains.

Environment is an accepted abstraction of multi-agent systems research. Our approach uses this abstraction for shielding the application developer from the complexity of the linked data space. In our approach, we access the linked data space via an environment layer which can be divided into sub-spaces called workspace. Workspaces contain programmable artifacts for accessing the data in the data sources that are used within the context of the sub-space. These artifacts employ WoDQA query engine which automatically distributes the query execution on data sources by means of VoID descriptions. Thus, a programmable linked data environment decreases the complexity by taking the burden of knowing contents of all the data sources that the application depends on from the developer.

Besides providing the environment as a means to program on linked data space, we focused on monitoring the dynamism in the data sources. Since the linked data applications generally depends on queries which are distributed on more than one data sources, we focused on a query level granularity to monitor the dynamism. The proposed linked data environment makes use of push and pull strategies together for the purpose of handling the dynamism of the data and transmitting changes to the applications. Artifacts specialized for monitoring are responsible for executing SPARQL queries and providing the pull and push services.

We have exemplified the proposed approach with a use case in the social semantic web domain. This experience showed us that separating the responsibilities of the application and the environment makes it easier to develop from the linked data developer perspective.

As future work, we will firstly evaluate the performance of our approach under heavy load such as registering lots of queries in a workspace by lots of applications. A possible solution to handle the load is distributing the workspaces via environment infrastructure. Another possible future work is improving the environment implementation with a caching capability. Since generally similar queries are registered onto a workspace in its linked data space, it will be beneficial using a cache for familiar sub-parts of queries to obtain results without accessing to remote data sources.

Acknowledgment

This work is supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK), Project Number: 111E027.

8. REFERENCES

[1] Ziya Akar, Tayfun Gökmen Halaç, Erdem Eser Ekinci, and Oguz Dikenelli. Querying the web of interlinked datasets using void descriptions. In Christian Bizer, Tom Heath, Tim Berners-Lee, and Michael Hausenblas, editors, *5th Linked Data on the Web Workshop (LDOW 2012), 21th World Wide Web Conference (WWW 2012)*, Lyon, France, 2012.

[2] Keith Alexander, Richard Cyganiak, Michael Hausenblas, and Jun Zhao. Describing Linked Datasets - On the Design and Usage of voiD, the 'Vocabulary of Interlinked Datasets'. In *WWW 2009 Workshop: Linked Data on the Web (LDOW2009)*, Madrid, Spain, 2009.

[3] Tim Berners-Lee. Linked data - design issues. http://www.w3.org/DesignIssues/LinkedData.html, 2006. [Online Technical Note; accessed March-2013].

[4] Petr Kremen, Marek Smid, and Zdenek Kouba. Owldiff: A practical tool for comparison and merge of owl ontologies. In *Proceedings of the 2011 22nd International Workshop on Database and Expert Systems Applications*, DEXA '11, pages 229–233, Washington, DC, USA, 2011. IEEE Computer Society.

[5] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the a&a meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, December 2008.

[6] Raúl Palma, Peter Haase, Óscar Corcho, and Asunción Gómez-Pérez. Change representation for owl 2 ontologies. In *OWLED*, 2009.

[7] Alexandre Passant and Pablo N. Mendes. sparqlpush: Proactive notification of data updates in rdf stores using pubsubhubbub. In *SFSW*, 2010.

[8] Niko Popitsch and Bernhard Haslhofer. Dsnotify - a solution for event detection and link maintenance in dynamic datasets. *Web Semant.*, 9(3):266–283, September 2011.

[9] Alessandro Ricci, Michele Piunti, Mirko Viroli, and Andrea Omicini. Environment programming in cartago. In Amal El Fallah Seghrouchni, Jürgen Dix, Mehdi Dastani, and Rafael H. Bordini, editors, *Multi-Agent Programming:*, pages 259–288. Springer US, 2009.

[10] Jürgen Umbrich, Boris Villazón-Terrazas, and Michael Hausenblas. Dataset dynamics compendium: A comparative study. In *COLD*, 2010.

[11] Danny Weyns, Andrea Omicini, and James Odell. Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, February 2007.