A Data Sharing and Synchronization Middleware Platform for Heterogeneous Medical Image Archives

Pradeeban Kathiravelu
INESC-ID Lisboa
Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
Email: pradeeban.kathiravelu@tecnico.ulisboa.pt

Ashish Sharma
Department of Biomedical Informatics
Emory University
Atlanta, Georgia, USA
Email: ashish.sharma@emory.edu

***Abstract*—Big data science has more consumers and fewer producers of data. Scientific data is getting larger and larger and consists of a varying degree of heterogeneity in its content and storage media. Medical image archives are populated by the images and meta data by a few source providers and are read by different interfaces. With the growing adaptation of pervasive computing into medical domain and increasingly open access to data, the meta data stored in legacy data stores is shared and synchronized across multiple devices and users. With the increasing complexity of the medical image hierarchies, consumers of medical images should be able to bookmark pointers to the sets of images and meta data from different sources, and share the pointers with others.**

While many medical image sources provide APIs for public access, an architecture that permits an effective sharing and synchronization of meta data across multiple users, from different storage media, is still lacking. This paper presents *MEDIator*, a data sharing and synchronization middleware platform for heterogeneous medical image archives. *MEDIator* allows sharing pointers to data effectively, while letting the consumers manipulate the pointers without modifying th raw data. *MEDIator* has been implemented for multiple data sources, including the Amazon S3[1], The Cancer Imaging Archive (TCIA)[2], caMicroscope[3], and meta data from CSV files for cancer images and metadata.

I. Introduction

Data sources contain data of different granularity. Data is organized in a hierarchical structure, where different levels are used to present data in specific formats. Folders and documents make a good example of this. A unique identifier is assigned to each of the data units. A search across the data source would present the user with the list of matching criteria. Interesting sub set of the matching criteria may be bookmarked by the user and shared with others. Medical image archives have a hierarchy of information with a well-structured schema. Hence, medical image information can be retrieved at different granularity by the users, exploiting this hierarchy.

Sharing data among the consumers can be achieved by sharing pointers to data. Replica set is a shared pointer to a stored data, which is created, shared, and modified by the data consumers. Storing a selected sub set of data items that matches a specific criteria as a replica set is an optimal way to bookmark large data sources. Users can create replica sets as a sub set of their search queries, and share their replica sets with other users, and update their replica sets periodically. Replica sets can be used as a way of tracking and sharing information.

Users should be able to create and update their replica sets, and share it with others using a unique identifier. While different data sources have different interfaces, a generic data replication and synchronization tool will be convenient, such that sharing of replica sets across heterogeneous data sources will be possible.

In-memory data grids distribute storage and execution over multiple physical nodes, while giving a logically centralized view of a larger resource pool. An in-memory data grid can be an alternative for a traditional storage to store the replica sets, as it will provide faster storage and access. Infinispan is a Java in-memory NoSQL store with implementations of distributed executor framework and MapReduce [1], which also has an actively maintained Android version. Sharing pointers instead of actual data will reduce the redundancy and increase the efficiency of the sharing and accessing of medical data. A deployment architecture consists of in-memory key-value stores such as Infinispan for sharing medical images will be scalable and pervasive.

While sharing data is encouraged in science [2], algorithms and architectures should be designed for mashing up and making the public data sharing easy and effective. This paper presents the research and prototype implementations of $MEDIator$, a replication and data synchronization platform, exploiting the in-memory data grid projects, while consuming data sources such as TCIA via their public APIs.

II. Related Work

A. Medical Image Archives

Medical image archives often follow a well-defined hierarchy of images with different granularity. TCIA public API provides methods to retrieve the images and meta data of different granularity [3]. Figure 1 depicts the methods that retrieve image and metadata at different granularity from TCIA. These methods are invoked by the public APIs such as the REST API. As shown by Figure 1 an initial search on TCIA may contain parameters such as modality, in addition to collection name, patient ID, study instance ID, and series instance UID. Each of the searches in TCIA returns the output in a finer granularity.

[1]http://aws.amazon.com/s3/
[2]http://www.cancerimagingarchive.net/
[3]http://imaging.cci.emory.edu/camicroscope/

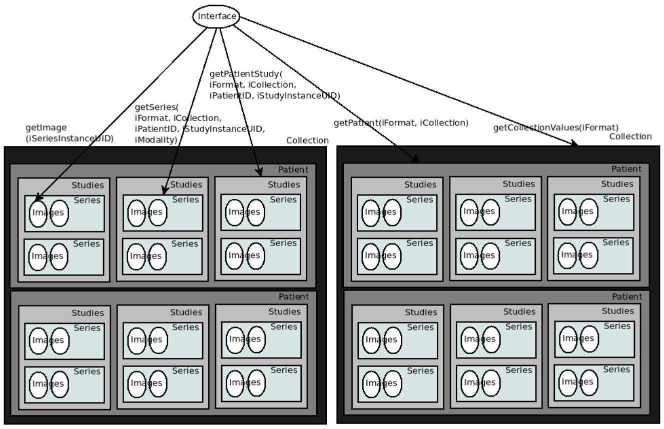

Fig. 1. Retrieving images and meta data

B. In-Memory Data Grids

In-memory data grids are extensively used in industry and academia, to provide a scalable memory and execution platform. Many commercial and open source in-memory data grid platforms exist. Infinispan and Hazelcast [4] are two commonly cited open source in-memory data grids. Infinispan is often cited in researches [5], [6], [7].

III. Solution Architecture

Multiple nodes running Infinipsan are leveraged and configured to create a cluster of $MEDIator$. Having multiple instances running over different nodes provide fault-tolerance, as when one node terminates, the other nodes have the backup replica of the partitions stored in the terminated node. Figure 2 depicts a higher level deployment view.

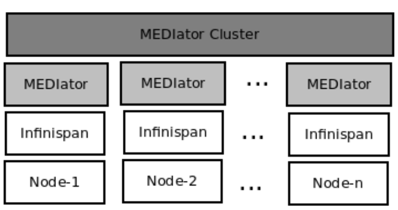

Fig. 2. Deployment

A. Data Access and Integration

A search query may contain different parameters that can define the scope of the search and the outcomes, and often return the outputs in a finer granularity. $MEDIator$ lets the users to create, update, retrieve, and delete replica sets, and share the replica sets with others.

When a sub set of such information is chosen and stored as a replica set, it is sufficient to store the unique identifiers of the matching data units of finer granularity than the original search query, as it would be sufficient to reproduce the data that is represented by the replica set. Hence, when using TCIA as the data source, storing an array of patient ID would be sufficient to identify the selected sub set of the collection including the array of patients. Similarly an array of study instance UID is sufficient to represent the selected sub set of any patient, and an array of series instance UID is sufficient to represent the selected sub set of any study, as it will contain an array of series. Hence $MEDIator$ stores the relevant IDs for the respective replica sets.

$MEDIator$ consists of 3 APIs, to be able to connect to specific data source, to integrate multiple data sources and meta data into the system, and to create replica sets for the data sources. Replica sets are stored and processed in the Infinispan maps, where integrating with the data sources is done by the $InterfaceAPI$ interface. $InterfaceAPI$ is often provided by the data sources, as a mean to query and retrieve the raw data from the remote data sources.

Methods for RESTful invocations are designed for each of the interfaces defining the APIs. Methods that are defined in the interfaces should be implemented by the classes that implement these interfaces. Table I depicts the methods defined for each of the interfaces

TABLE I. Methods defined for the Interfaces

Data Source Management	**ReplicaSet Management**	
$InterfaceAPI$	$PubConsAPI$	$Integrator$
retrieve	createReplicaSet	updateExistenceInDataSource
	duplicateReplicaSet	doesExistInDataSource
	getRawData	getRawData
	getReplicaSet	getMetaData
	putReplicaSet	putMetaData
	updateReplicaSet	updateMetaData
	deleteReplicaSet	deleteMetaData

$PubConsAPI$ and $Integrator$ provide replica set management. $Integrator$ integrates multiple heterogeneous data sources into $MEDIator$ and provides a meta map and maps for each of the data sources to access the relevant information for each of the entry, while $PubConsAPI$ provides and manages replica sets for each specific data source. $Integrator$ lets the users to store meta data from multiple heterogeneous sources in a coarse grain access.

B. Generic Design

Classes extending PubConsAPI design and implement the logic of replica sets management, where integration with the data sources is done by two interfaces - Interface API and Integrator. Interface API provides data retrieval from the data source, where Integrator integrates multiple data sources into the system and store the meta data of the data sources into Infinispan.

A generic design was created as an example implementing the Interface API and PubConsAPI.

Two distributed cache instances exist in InfDataAccessIntegration.

```
protected static Cache userReplicasMap;
protected static Cache replicaSetsMap;
```

userReplicasMap is a mapping of userId → Array of replicaSetIDs. UserID could be the logged in user name. (for now, testing with random strings). replicaSetsMap is a mapping of replicaSetID → replicaSet.

Though this could be replaced with a single cache instance with the mapping of userID → replicaSets, having two cache instances will be more efficient during searches, duplicates, and push changes. Hence, two cache instances design was chosen.

$InfDataAccessIntegration$ implements the PubConsAPI for publisher/consumer, where $InterfaceManager$ implements the $InterfaceAPI$ for the interface between the data source and the data replication and synchronization tool. Invoker classes extending the abstract class InterfaceManager, implement the respective data source integration to invoke these methods. The execution flow is depicted by Figure 6.

DataProSpecs - PubConsAPI: The methods of $DataProSpecs$ can be invoked by knowing the respective replicaSetID of the replica set and the user ID of the user who owns the replica set. User ID is a random string that is input by the user. Probably this can be some 'secret' or pass from the user. userIDs serve as the keys of the userReplicasMap. Apart from that, there is no data structure to hold the list of users.

$createReplicaSet()$ requires the userID along with the elements to be stored in the replica set. It returns the $replicaSetID$, which is further used to uniquely identify the replica set. $getReplicaSet()$ requires only the relevant $replicaSetID$ to return the respective replica set. Similarly, $updateReplicaSet()$ requires $replicaSetID$ as well as the elements to be stored in the $replicaSet$, replacing the previous elements. $deleteReplicaSet()$ requires both the $userID$ and $replicaSetID$. Similarly, $duplicateReplicaSet()$ requires both $replicaSetID$ and the userID of the user to whom the replicaSet is shared to. Creating, deleting, and duplicating a replication set requires modification to the user replicas map, which holds the list of the replica sets for the particular user. Hence the necessity to input the userID. ReplicaSetID is a large negative number (UUID) randomly generated. It can be safely assumed to be harder to guess. Hence, this model is assumed to provide adequate security for this prototype.

C. Integration with Multiple Data Sources

Clinical data is deployed in multiple data sources such as TCIA, CA Microscope, and Amazon S3. Figure 3 depicts the deployment of the system with multiple data sources. A set of clinical data was uploaded to S3, where the meta data mapping of patientID → fileName was available as a CSV file. Similarly CSV file depicting clinical information is available, as multiple properties against the UUID, such as patient ID. These files are parsed and stored into meta data maps in Infinispan. The CSV files containing meta data or $resourceIDfileName$ mapping are stored locally in the file system or in a remote repository.

Each data source is connected to the replication tool by implementing a class that extends Integrator interface. DataSourcesIntegrator is an abstract class implementing the common features of the currently implemented 4 data sources - CSV, CA Microscope, S3, and TCIA. CsvIntegrator, CaMicIntegrator, S3Integrator, and TciaIntegrator respectively implement the integrators for each of these data sources. $MetaDataLoader$ loads the CSV files and stores them into Infinispan maps, against the ID, such as the patient ID. TciaIntegrator invokes TciaInvoker to retrieve the images and meta data from TCIA.

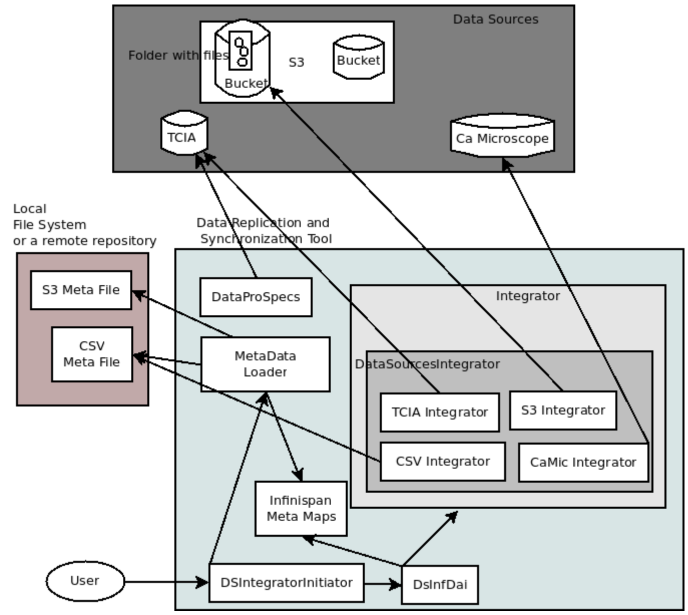

Fig. 3. Deployment Diagram of the System and the Data Sources

$DSInfDai$ a class extending $InfDataAccessIntegration$ holds the instances of map to store all the meta data. $DSIntegratorInitiator$ invokes the instances of $DSInfDai$, $MetaDataLoader$ and the other respective classes to start parsing the meta data, and store the instances into the respective maps. Figure 4 represents the class diagram of data source integrators, along with the 3 interfaces of the replication tool.

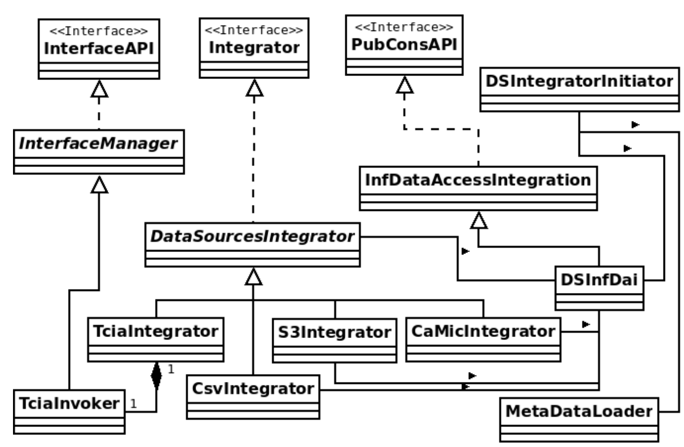

Fig. 4. Class Diagram of data sources integrators along with the higher level API

```
protected static Cache<String, Boolean[]> metaMap; /*csv
    , ca, tcia, s3*/
protected static Cache<String, String[]> csvMetaMap;
protected static Cache<String, String> s3MetaMap;
protected static Cache<String, String> caMetaMap;
```

The $metaMap$ stores a binary array against each of the key (such as, patientID) to point the existence of meta data in the data sources defined, in this case, CSV, CA Microscope, TCIA, and S3. $s3MetaMap$ provides the file name for the respective

ID, which can be used to find the location of the file as it follows the pattern of of $S3\_BASE\_URL + folderName +$ "$/" + fileName$. Similarly, $caMetaMap$ stores the URL that the respective object is stored. $csvMetaMap$ contains the meta data loaded from the CSV files against the respective ID.

Methods for updating, creating, and deleting meta data from these maps are available where creating or deleting meta data will update the availability to true/false for the respective index in the $metaMap$. $XXX\_META\_POSITION$ defines the position in the metaMap, where XXX stands for CSV, CA, TCIA, and S3. Updating the existence will flop the respective boolean value in the respective entry. The $metaMap$ ensures easy indexing, and helps to search which of the data sources contain the respective information for any given key, such as a given patient ID.

D. MEDIator Extensions

$MEDIator$ has further characteristics to functions as a resilient middleware platform.

multi-tenancy: The Replication Tool is multi-tenanted, and it is aware of the multiple tenants or users using the system. Each user co-exist in the replication tool at the same time, without the knowledge of existence of the other users, sharing the same cache space.

Access Logging: Involving a time stamp for the class extending $PubConsAPI$, downloaded items can be tracked, and the diffs can be produced for the user download. Hence, a download can be paused and resumed later, downloading the images that have not been downloaded yet.

Security: $MEDIator$ can be further secured with user authentication. SAML tokens can be created for user names and passwords with the membership information, which can be validated later. A create and validate API with user stores such as LDAP or OpenID should be designed and implemented, extending and leveraging $MEDIator$.

E. Software Architecture

Figure 5 depicts the architecture of the data replication and synchronization tool, showing its components and dependencies.

Dependencies are used unmodified. Apache HTTP Client and Mashape Unirest are configured and leveraged by the InterfaceAPI, to query and retrieve the raw data from the data sources. Infinispan, as the core dependency, is configured and exploited as the shared storage and distributed execution medium. Presentation layer dependencies such as Embedded Tomcat, Apache Velocity, and Log4j2 facilitate prototype application development with web pages, and configurable logging. Apache Maven is used as the project management dependency to build, deploy, and execute the project.

Data sources access layer consists of means of accessing the data sources, such as querying and downloading the data stored in Amazon S3 or TCIA, via the REST API. Utilities such as MetaDataLoader and TciaUtil provide utility methods and functionalities for the tool throughout its execution.

InterfaceAPIs, Integrators, and PubConsAPIs are implemented at the top level, extending their respective base classes.

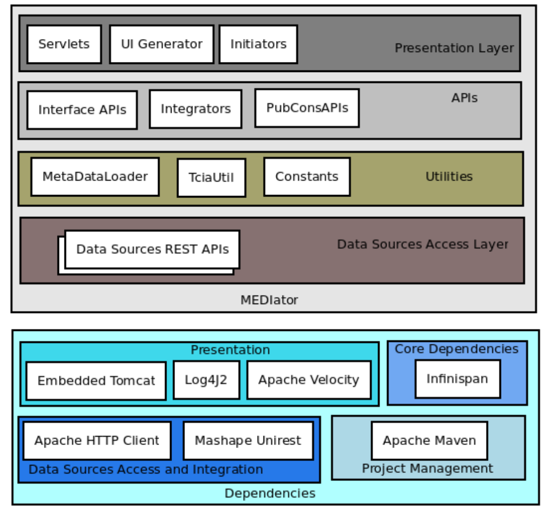

Fig. 5. Architecture

The presentation layer consists of the Initiators that function as the starting point of the prototype applications. Servlets and UI Generators generate the presentation layer of the tool.

Replica sets management is handled by both $Integrator$ and $PubConsAPI$ interfaces. If multiple data sources should be managed consecutively, having maps pointing to meta data from heterogeneous sources, $Integrator$ interface should be implemented in a class with the respective distributed maps to hold the meta data. If a single data source is involved with a fine grain control over a data source, $PubConsAPI$ should be implemented. Functionality such as security with SAML token should be implemented as an Integrator, as this involves SSO for multiple data sources.

$InterfaceAPI$ integrates the tool with the data source, such that the relevant data can be queried, searched, manipulated, and downloaded. Hence, respective classes should implement this interface for each of the data source. The TCIA integration by the $tcia\_rest\_api$ package stands as a decent sample on implementing this interface.

$getRawData(Stringkey)$ method should be implemented in the classes implementing the $PubConsAPI$ and $Integrator$ interfaces to be able to download the raw data from the data sources such as TCIA, CA Microscope, or S3, for the given replica set, potentially using Java web start.

IV. Implementation

Based on the design, $MEDIator$ was implemented as a data sharing and synchronization middleware platform for heterogeneous medical image archives.

Different complex data sources require custom development extending the generic framework. As creating and customizing the replica set require a more specific data structure, further implementations are done, extending the core class hierarchy.

A. Generic Implementation

Base classes of the APIs were implemented to provide a generic synchronization tool, that stores the entire search queries of any data source as the replica set. When the user logs in, $logIn()$ checks whether the user has already stored replicaSets from the Infinispan distributed Cache. If so, execute them all again. This would be changed later as we do not have to execute all. Rather, we need to execute for the diffs. When the user performs new searches, for the images, series, collections, and the other meta data, the results will be returned to the user, and the user can chose ta subset of the returned results to create a replicaSet. The replicaSet for the image will

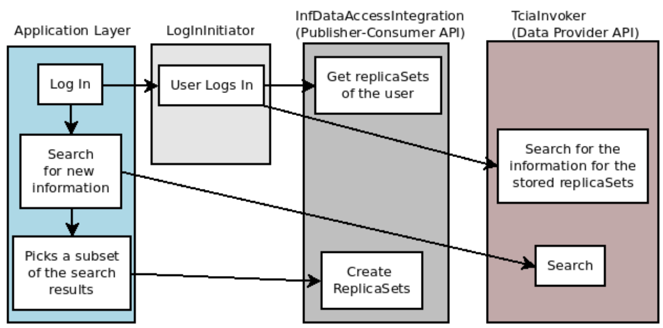

Fig. 6. Execution Flow

be as,

```
TCIAConstants.IMAGE_TAG + "getImage?SeriesInstanceUID=" +
     seriesInstanceUID
```

For other information (meta data), such as collections and seies,

```
TCIAConstants.META_TAG + query;
```

Here, query takes the below format.

```
"getSeries?format=" + format +
                 "&Collection=" + collection +
                 "&PatientID=" + patientID +
                 "&StudyInstanceUID=" + studyInstanceUID +
                 "&Modality=" + modality;
```

When a new instance starts now, and invokes the log in action for the same user, it will execute the queries for the stored replicaSets again, and reproduce the same results.

B. TCIA Implementation

An extension based on the base design was developed for TCIA, as shown by Figure 7, which provides a core class hierarchy of the system.

TciaInvoker - Interface API: $TciaInvoker$ extends $InterfaceManager$ to implement the interfacing layer between the TCIA data source and the data replication and synchronization tool. Meta data such as collections, patients, studies, and series are retrieved at different levels, though the default download manager of TCIA downloads the data in series level, composed of the images of the series in a single zip archive. While having the default userReplicasMap to contain the IDs of the replica sets for each user, the replica set itself is stored in multiple maps instead of a single replicaSets map, to provide an efficient storage and access.

$DataProSpecs$ extends the $InfDataAccessIntegration$ class. 5 maps are created as below to represent the replica sets.

```
    protected static Cache<Long, Boolean[]> tciaMetaMap;
    protected static Cache<Long, String[]> collectionsMap;
    protected static Cache<Long, String[]> patientsMap;
    protected static Cache<Long, String[]> studiesMap;
    protected static Cache<Long, String[]> seriesMap;
```

$tciaMetaMap$ contains a boolean array, which reflects which of the granularity of meta data is selected as a whole. For TCIA, if a few collections are selected, the first element of the array is set to true, and similarly, the other meta data are marked to true or false as shown by the below code segment.

```
        Boolean[] metaMap = new Boolean[4];
        metaMap[0] = collection != null;
        metaMap[1] = patientID != null;
        metaMap[2] = studyInstanceUID != null;
        metaMap[3] = seriesInstanceUID != null;

        putReplicaSet(replicaSetId, metaMap);
```

The name of the collections, patientID, studyInstanceUID, and seriesInstanceUID are stored against the respective replicaSetID in collectionsMap, patientsMap, studiesMap, and seriesMap respectively. Hence changes are done at the respective maps. Duplicating the replicaSets duplicate the contents of the entire row to a new replicaSetID. Similarly, deleting a replicaSet deletes the respective information from all the maps.

V. Evaluation

A prototype web application was built with $MEDIator$ architecture. Apache Velocity was used to generate the web pages for the replication tool. Apache Tomcat (Embedded) was integrated into the program such that it will get the user inputs from the HTML pages and present the output in pages formatted by Apache Velocity Templates. Respective servlets were created inside the servlets package to receive the inputs from HTML pages to the backend Java code.

A cluster with 6 identical nodes (Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz and 12 GB memory) was used for evaluations. With synchronous backups enabled, $MEDIator$ showed a fault-tolerant behavior when a node failed and/or rejoined later. The time to initialize $MEDIator$ remained constant, regardless of the underlying data sources.

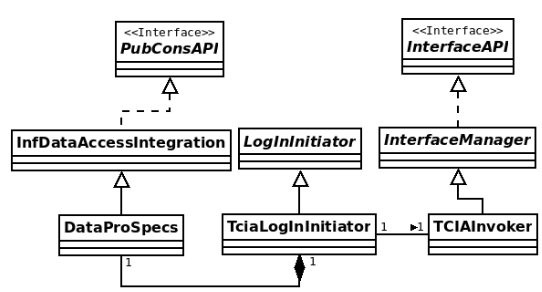

Fig. 7. Core Class Hierarchy

VI. Conclusion

$MEDIator$ is a platform providing an ubiquitous access to medical meta data from the image archives, while providing fault tolerance and load balancing, leveraging the distributed shared memory platforms. A prototype has been implemented with multiple data sources containing cancer images, with Infinispan as the in-memory data store.

Consumers download the data by searching the image repository using the browser. The information that the consumer is interested in, gets updated whenever the data producers update or add patient information. Further improvements to $MEDIator$ should enable automated downloads to the consumers. While $MEDIator$ focuses on medical image archives, the design can also be implemented for any other data types with an index available to query and structure them as replica sets.

Acknowledgments: This work was partially supported by the Google Summer of Code 2014 project, under the mentoring organization *Biomedical Informatics, Emory University*.

References

[1] Francesco Marchioni. *Infinispan Data Grid Platform.* Packt Publishing Ltd, 2012.

[2] A. Szalay and J. Gray, 2020 Computing: Science in an exponential world, *Nature, vol. 440*, pp. 413-414, 2006.

[3] Prior, F. W., Clark, K., Commean, P., Freymann, J., Jaffe, C., Kirby, J., ... & Marquez, G. (2013, July). TCIA: an information resource to enable open science. In *Engineering in Medicine and Biology Society (EMBC), 2013 35th Annual International Conference of the IEEE* (pp. 1282-1285). IEEE.

[4] Johns, M. (2013). *Getting Started with Hazelcast*. Packt Publishing Ltd.

[5] Palmieri, R., di Sanzo, P., Quaglia, F., Romano, P., Peluso, S., & Didona, D. (2012, January). Integrated monitoring of infrastructures and applications in cloud environments. In *Euro-Par 2011: Parallel Processing Workshops* (pp. 45-53). Springer Berlin Heidelberg.

[6] Rosa, L., Rodrigues, L., & Lopes, A. (2011, November). Goal-oriented self-management of in-memory distributed data grid platforms. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on* (pp. 587-591). IEEE.

[7] Ruivo, P., Couceiro, M., Romano, P., & Rodrigues, L. (2011, December). Exploiting total order multicast in weakly consistent transactional caches. In *Dependable Computing (PRDC), 2011 IEEE 17th Pacific Rim International Symposium on* (pp. 99-108). IEEE.