MEDIator: A Data Sharing Synchronization Platform for Heterogeneous Medical Image Archives

Pradeeban Kathiravelu[1] and Ashish Sharma[2]

***Abstract*— With the growing adaptation of pervasive computing into medical domain and increasingly open access to data, metadata stored in medical image archives and legacy data stores is shared and synchronized across multiple devices of data consumers. While many medical image sources provide APIs for public access, an architecture that orchestrates an effective sharing and synchronization of metadata across multiple users, from different storage media and data sources, is still lacking. This paper presents *MEDIator*, a data sharing and synchronization middleware platform for heterogeneous medical image archives. *MEDIator* allows sharing pointers to medical data efficiently, while letting the consumers manipulate the pointers without modifying the raw medical data. *MEDIator* has been implemented for multiple data sources, including Amazon S3, The Cancer Imaging Archive (TCIA), caMicroscope[1], and metadata from CSV files for cancer images.**

I. Introduction

Scientific data is getting larger and larger, with an increasingly varying degree of heterogeneity in its content and storage media. Medical image archives are populated with images and metadata by a few source providers and are read by many consumers through different interfaces, following the norm of big data science. As medical image archives store data in a well-structured hierarchy, adhering to a pre-defined schema with a commonly accepted taxonomy, medical image data can be retrieved at different granularity by the consumers.

Data should be easily accessed from multiple medical data sources and legacy warehouses, supporting the Open Science initiatives in biomedical engineering. Analyzing and mining multiple medical data sources require considerable computational resources, as they consist of different structures, complex APIs, and gigantic amount of hierarchical data. As the complexity of the image hierarchy of medical image sources keeps growing, consumers bookmark and share pointers to selected sub sets of images and metadata from different sources, that match a certain criteria or interest.

The shared pointers should be stored as bookmarks in a fault-tolerant manner, and shared among the users as replica sets. Since data sources have different and often incompatible interfaces, a generic data sharing and synchronization tool will be convenient, for sharing of such replica sets from the data sources. While sharing data is encouraged in science [1], algorithms and architectures should be designed for mashing up and sharing the medical data efficiently.

In-memory data grids distribute storage and execution over multiple physical nodes, while giving a logically centralized view of a larger resource pool. Hence, they provide a cost-effective and feasible alternative to supercomputers. An in-memory data grid can be an alternative for a traditional storage for the replica sets, as it provides faster storage, access, and execution. A deployment architecture consists of in-memory key-value stores such as Infinispan for sharing medical images will be scalable and pervasive.

This paper presents the research and prototype implementations of $MEDIator$, a data sharing and synchronization platform, exploiting the in-memory data grid platforms, while consuming data sources such as TCIA [2] via their public APIs. While $MEDIator$ is designed for medical image archives, the overall design and architecture of the framework can be generalized for any data sources. In the remaining of the paper, we will discuss the preliminary background information on $MEDIator$ in section II. Section III discusses solution architecture of $MEDIator$, where Section IV further presents the prototype implementation details of $MEDIator$. Section V closes the paper with conclusions and future work.

II. Background and Related Work

Medical images are stored in specific data sources such as TCIA that defines a schema for the metadata, or can be stored in a general-purpose data source such as Amazon S3, with user-enforced schema. TCIA public API provides methods to retrieve the images and metadata of different granularity [2], as shown by Figure 1. These methods are invoked by the public APIs such as the REST API. 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.

Data Grids and Access Integration Platforms: Infinispan [3] and Hazelcast [4] are two commonly used open source in-memory data grids, which are leveraged as scalable memory and execution platforms. Infinispan is a Java in-memory NoSQL store with implementations of distributed executor framework and MapReduce, which also has an Android version. OGSA-DAI is a middleware platform that provides seamless integration and access to heterogeneous data sources [5]. WS-DAI family of specifications provide a data access integration for web services [6]. While there

[1]Pradeeban Kathiravelu is with Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal (e-mail: pradeeban.kathiravelu@tecnico.ulisboa.pt).

[2]Ashish Sharma is with the Department of Biomedical Informatics, Emory University, Atlanta, Georgia, USA (e-mail: ashish.sharma@emory.edu).

[1]http://imaging.cci.emory.edu/camicroscope/

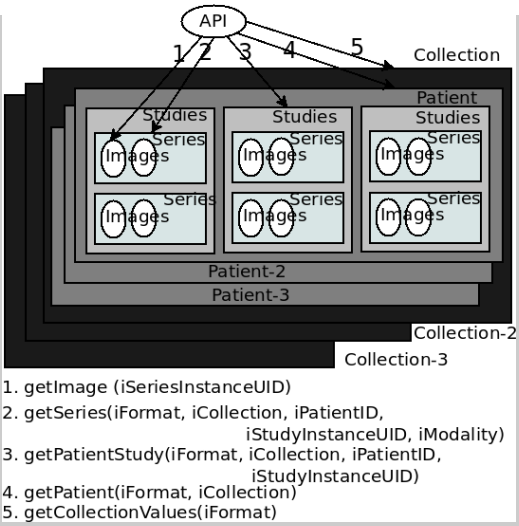

Fig. 1. Retrieving images and metadata in TCIA

are generic data sources integration platforms, a compact platform for medical image archives to share content across multiple users with light-weight pointers without actually duplicating the original content is still lacking.

III. Solution Architecture

Sharing data among the consumers can be achieved by sharing pointers to the data, which reduces the redundancy and increases the efficiency of sharing and accessing the data as data duplication is minimized. Replica set is a pointer to stored data, which is created by the users as a bookmark, by choosing a sub set of data items that matches a search query or a specific user-defined criteria. Users can create replica sets with unique identifiers, and share their replica sets with other users and duplicate the replica sets, and update or delete their replica sets later. Hence, replica sets can be used as a way of tracking and sharing information.

Multiple nodes running Infinispan are leveraged and configured to create a cluster of $MEDIator$. Having multiple instances running over different nodes provide fault-tolerance, as when one node crashes or fails, the other nodes have the backup replica of the partitions stored in the terminated node. Figure 2 depicts a higher level use case view, with $MEDIator$ configured to execute over multiple physical nodes including servers, desktops, laptops, and smart Android devices forming an Infinispan cluster. Heterogeneous data sources are accessed by the $MEDIator$ cluster. $MEDIator$ cluster is accessed by multiple users through any of the physical nodes.

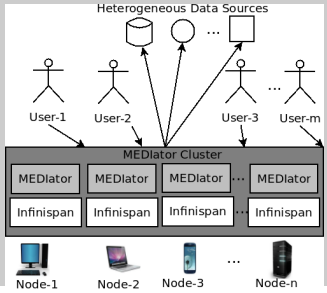

Fig. 2. Higher Level Use Case View of $MEDIator$

When the user performs searches for the images, series, collections, and the other metadata, the results will be returned to the user, and the user can chose a subset of the returned results to create and store a replica set in the Infinispan maps. 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.

A. $MEDIator$ APIs

$MEDIator$ consists of 3 public APIs, to be able to connect to specific data source, to integrate multiple data sources and metadata into the system, and to create replica sets for the data sources.

- $InterfaceAPI$: Integrates with the data sources for data retrieval, which is often provided by the remote data sources, as a mean to query and retrieve the raw data. 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.
- $PubConsAPI$: Manages replica sets for each specific data source.
- $Integrator$: 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 in a coarse granularity. $PubConsAPI$ and $Integrator$ provide replica set management.

$DataProSpecs$ extends $PubConsAPI$ to design and implement the logic of replica sets management. The methods of $DataProSpecs$ can be invoked by knowing the respective ID of the replica set and the user ID of the user who owns the replica set. UserID is an identifier such as a user name or a 'secret key' from the user. ReplicaSetID is a large number that can be randomly generated. UserID serves as the key of the userReplicasMap, where replicaSetID serve as the key for the replica sets maps.

$createReplicaSet()$ 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, making it necessary to input the userID.

B. Integration with Medical Data Sources

Clinical data is deployed in multiple data sources such as TCIA, caMicroscope, and Amazon S3. Figure 3 depicts the deployment of the system with multiple medical data sources. A set of clinical data was uploaded to S3, where the metadata 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 metadata maps in Infinispan. The CSV files containing metadata or $resourceIDfileName$ mapping are stored locally in the file system or in a remote repository.

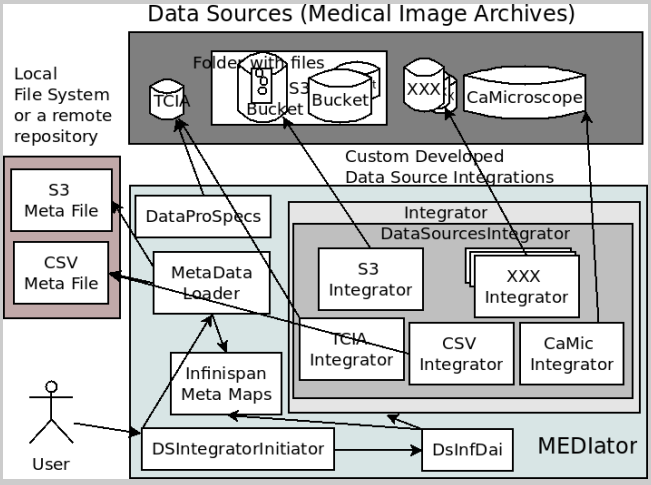

Fig. 3. Deployment Diagram of the System and the Data Sources

Each data source is connected to $MEDIator$ by implementing a class that extends Integrator interface. $DatasourcesIntegrator$, an abstract class provides the common features for the data sources integration. It is extended by $CsvIntegrator$, $CaMicIntegrator$, $S3Integrator$, and $TciaIntegrator$ which respectively function as the integrators for CSV, $caMicroscope$, $AmazonS3$, and $TCIA$. $MetaDataLoader$ loads the CSV files and stores them into Infinispan maps, against the ID, such as the patient ID.

A sub class of $InfDataAccessIntegration$, $DSInfDai$ holds the instances of map to store all the metadata. $DSIntegratorInitiator$ invokes the instances of $DSInfDai$ and $MetaDataLoader$ to parse the metadata, and store the instances into the respective maps.

C. MEDIator Representation of Medical Image Sources

Infinispan distributed cache instances are created in $InfDataAccessIntegration$ as the data structures to hold the replica sets for each of the users. $userReplicasMap$ provides a mapping of userIdentifier → array of replicaSetIDs. Hence, $userReplicasMap$ provide a listing of all the replica sets for the given user. replicaSetsMap is a mapping of replicaSetID → replicaSet. A meta map provides a mapping of replica set ID against a boolean array. The boolean value is set to true if the data source represented by the index of the boolean element in the array is available in the considered replica set.

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

The $metaMap$ stores a binary array against each of the key (such as, patientID) to point the existence of metadata in the data sources defined. Further, meta maps such as $tciaMetaMap$, $csvMetaMap$, $s3MetaMap$, and $caMetaMap$ are defined for each data source, against the replica set ID. $s3MetaMap$ 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 metadata loaded from the CSV files against the respective ID. $tciaMetaMap$ stores a boolean array against the replica set ID, where a $true$ for the array element in the map indicates that a metadata with specific granularity is stored in the replica set.

Methods for updating, creating, and deleting metadata from these maps are available where creating or deleting metadata 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 data sources such as CSV, CA, TCIA, and S3. Updating the existence will flop the respective boolean value for the 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. Data sources with a hierarchy such as $TCIA$ follow the same model, with a boolean array stored against each replica set ID at the top level.

$MEDIator$ is multi-tenanted where multiple users coexist without the knowledge of existence of the other users, sharing the same cache space. Involving a time stamp for the class extending $PubConsAPI$, downloaded items can be tracked, and the diffs can be produced for the user download. Thus a download can be paused and resumed later, downloading the images that have not been downloaded yet.

$MEDIator$ can be secured with user authentication. SAML tokens can be created for user names and passwords with the membership information, which can be validated later. This is designed as an Integrator for the Single Sign-on (SSO), as it involves multiple data sources. A create and validate API with user stores such as LDAP or OpenID should be designed and implemented, extending $MEDIator$.

D. Software Architecture

Figure 4 depicts the architecture of $MEDIator$. 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.

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

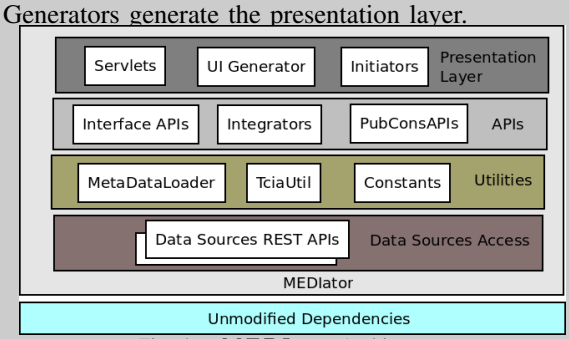

Fig. 4. $MEDIator$ Architecture

Replica sets management is handled by both $Integrator$ and $PubConsAPI$ interfaces. If multiple data sources should be managed consecutively, having maps pointing to metadata from heterogeneous sources, $Integrator$ interface should be implemented in a class with the respective distributed maps to hold the metadata. If a single data source is involved with a fine grain control over a data source, it is sufficient only to extend and implement $PubConsAPI$. $InterfaceAPI$ integrates the data sources with $MEDIator$, such that the relevant data can be queried, manipulated, and downloaded. $InterfaceAPI$ is implemented by the respective class for each data source.

IV. Implementation: $MEDIator$ for TCIA

Different complex data sources require custom development extending the generic framework and core $MEDIator$ architecture, as creating and customizing replica sets require a more specific data structure. A prototype $MEDIator$ web application was built for TCIA, with Apache Tomcat (Embedded) to get the user inputs from the HTML pages and present the output in pages formatted by Apache Velocity Templates. Respective servlets were created to receive the inputs from HTML pages to the backend Java code.

$InterfaceManager$ implements the $InterfaceAPI$ for the interface between the data source and $MEDIator$. Invoker classes such as $TciaInvoker$ extending the abstract class $InterfaceManager$, integrate the data sources. Metadata 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. 4 maps store the metadata for each replica set against its ID, in different granularity - collections, patients, studies, and series, along with a meta map that points to the 4 maps.

```
protected static Cache<String, Boolean[]> tciaMetaMap;  /*
    collections , patients , studies , series */
```

Each of the 4 maps stores the respective IDs of the metadata as the value against the replicaSetID. $tciaMetaMap$ contains a boolean array of length 4, which reflects which of the granularity of metadata is selected as a whole. A $true$ value indicates the existence of replica set entry in the respective map. For TCIA, if a few collections are selected, the first element of the array is set to true, and similarly patients, studies, and series are defined by the other 3 elements.

The name of the collections, patientID, studyInstanceUID, and seriesInstanceUID are stored against the respective replicaSetID in collectionsMap, patientsMap, studiesMap, and seriesMap. 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. Conclusion

$MEDIator$ is a platform providing an ubiquitous access to medical metadata 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 complex data types with an index to query and structure them as replica sets.

References

[1] A. Szalay and J. Gray, 2020 Computing: Science in an exponential world, *Nature, vol. 440*, pp. 413-414, 2006.
[2] 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.
[3] Marchioni, F. (2012). Infinispan data grid platform. Packt Publishing.
[4] Johns, M. (2013). *Getting Started with Hazelcast*. Packt Publishing.
[5] Antonioletti, M., Atkinson, M., Baxter, R., Borley, A., Chue Hong, N. P., Collins, B., ... & Westhead, M. (2005). The design and implementation of Grid database services in OGSA-DAI. *Concurrency and Computation: Practice and Experience, 17* (2-4), 357-376.
[6] Antonioletti, M., Krause, A., Paton, N. W., Eisenberg, A., Laws, S., Malaika, S., ... & Pearson, D. (2006). The WS-DAI family of specifications for web service data access and integration. *ACM SIGMOD Record*, 35(1), 48-55.