Implementation of XML Database and Enhancement of Resource and Sensor agents of AgentTeamWork

Final Report

Cuong Ngo
August 18, 2006
Summer 2006

Introduction 4

DESCRIPTION OF CLASSES 4

RESOURCE 4
Description 4
Important Functions: 4
COLLECTION 5
Description 5
Important Functions: 5
SERVICE 5
Description 5
Important Functions: 5
DATABASEMANAGEMENTSERVICE 6
Description 6
Important Functions: 6
STORESERVICE 6
Description 6
Important Functions: 6
RETRIEVALSERVICE 6
Description 6
Important Functions: 6
DELETIONSERVICE 7
Description 7
Important Functions: 7
QUERYSERVICE 7
Description 7
Important Functions: 7
XPATHUTIL 7
Description 7
Important Functions: 8
XDBASE 8
Description 8
Important Functions: 10
XDBASEGUI 11
Description 11
Important Functions: 11
XCOLLECTION 13
Description 13
Important Functions: 13

FUTURE SERVICES 13

Service.java 14
RetrievalService.java 14
XDBase.java 14
XCollection 15

COMPILING AND RUNNING 15

RESOURCEAGENT MODIFICATIONS 15

INTRODUCTION 15
DESCRIPTION OF CHANGED/ADDED FUNCTIONS 17
initDB 17
updateDB 17
downloadXmlsFromFtp(String ftpDir, FTPFile[ ] ftpList, String dbColName, Vector dbList) 18
addXmlFromFtpToDB(String ftpDir, String filename, String dbColName) 18
addRuntimeToDb(String fileContent) 18

addAllChildNodes(Element root, Document document, Node currentGroupNode) 18
getRuntimePairs(Node node) 18
addSingleChildNode(Element root, String nodeName, String nodeValue, Document document) 18
queryForIpNames 19
startProbing 19

SENSORAGENT MODIFICATIONS 19

INTRODUCTION 19
DESCRIPTION OF CHANGED/ADDED FUNCTIONS 23
Init() 23

FUTURE WORK 23

Introduction

The new database comprises of the following classes: XDBase, Collection, Resource, XCollection, Service, DatabaseManagementService, StoreService, RetrievalService, QueryService, DeletionService, XPathUtil and XDBaseGUI. Each class will be described to what their function is and how to use each one. The basic idea of how the database works is that the database (XDBase) contains Collections that contains Resources. There are Service subclasses that allow communication with the database for specific services. XPathUtil is a utility class that performs all Xpath queries and the XDBaseGUI class is an applet-based GUI that interacts with the database. The XCollection is the interface in which the ResourceAgent communicates with the database.

XDBase acts as a native XML database, which means that the data is not contained in tables, but rather contained in DOM objects. The DOM objects hold the structure the XML file in tact.

To accommodate the new database, improvements were made to the ResourceAgent and how it interacts with it. The SensorAgents were also changed to be able to dynamically monitor resource nodes within a cluster.

Description of Classes

Resource

Description

This is a wrapper class for the DOM object that represents the XML file and the resource computing node. It contains the DOM object (Document), the name of the resource, the last date of modification and whether or not this resource computing node is available to use. Basically, each computing node and cluster information is represented by the Resource class.

Important Functions:

- *getLastModificationDate* – returns the last modification date
- *setModified(Date)* – sets the new last modification date
- *getName* – returns the name of the resource
- *isAvailable* – returns the availability of the resource node
- *setAvailability(Boolean)* – sets the availability of the resource node
- *getDocument* – returns the DOM object
- *getDocumentAsString* – returns the String representation of the DOM object. In other words, it returns the XML file as a String.

Collection

Description

This class represents a collection of resource computing nodes that share something in common. For example, the database contains a collection for the cluster information, cluster node runtime information and public node information. The Resources are held in a hashtable and are manipulated through functions of the Collection class. This class allows Resources to be added, deleted, modified and queried. In addition there are functions to display the number of Resources in the collection, retrieve the collection name and retrieve a list of all resources.

Important Functions:

- *changeAvailabilityTo(String, boolean)* – changes the availability of the specified Resource to the specified boolean value
- *changePublicNodeAvailabilityTo(String, boolean)* – changes the availability of the Resource found by the given Xpath expression to the specified boolean value
- *retrieveResource(String)* – returns the Resource specified by the resource name
- *storeResource(Resource, String)* – stores the Resource with the specified name(key in hashtable)
- *query(String, boolean)* – queries all the Resources in the Collection using the Xpath expression. If boolean parameter is true, only query available Resources
- *update(String, String[], String[])* – updates the specified Resource by using provided elements to change and their associated values to change them to. Returns the number of the elements changed.
- *update(String[], String[])* – updates all Resources in the collection by using provided elements to change and their associated values to change them to. Returns the number elements changed.
- *deleteResource(String)* – deletes the Resource specified, if it exists

Service

Description

This is base class in which service type classes subclass. It also contains basic database communication functions such as connecting to the database, sending a request and closing all connections.

Important Functions:

- *connectToDatabase* – connects to the database and creates a socket and output/input stream

- *sendRequest(int, String[])* – sends a request to the database represented by the integer value and required information by the DB
- *closeAllConnections* – closes output/input stream and the socket
- *connectAndSendInformationToDB(int, String)* – connect to the database, and sendRequest

DatabaseManagementService

Description

This class provides services that manage the database. Those services include: creating a collection, synchronizing the database and shutdown of the database.

Important Functions:

- *createCollection(String)* – create a Collection using the specified collection name
- *synchronize* – write back database contents to local disk
- *shutdown* – shutdown the database

StoreService

Description

This class provides services that store items in the database. Those services include: storing a whole Collection and storing a Resource.

Important Functions:

- *storeCollection(String, Collection)* –store a Collection using the specified collection name
- *storeResource(String, String, Document*) – store a Resource, created from the document, using the specified resource name. The Resource is stored in the specified collection name.

RetrievalService

Description

This class provides services that retrieve items from the database. Those services include: retrieving a collection, retrieving a document, retrieving a Resource list and retrieving the last modification date of the Resource.

Important Functions:

- *retrieveCollection(String*) – get the Collection specified by the collection name

- *retrieveResource(String, String)* –get the document specified by the resource name in the specified Collection.
- *retrieveResourceList(String)* – get a list of Resources in the specified Collection
- *retrieveLastModificationDate(String, String)* – retrieve the last modification date of the specifiedResource in the specified Collection

DeletionService

Description

This class provides services that delete items in the database. Those services include: deleting a Collection and deleting a Resource

Important Functions:

- *deleteCollection(String)* – delete a Collection
- *deleteResource(String, String)* – delete a Resource in the specified Collection

QueryService

Description

This class provides services that query and update the database. Those services include: update a specific Resource, update an entire Collection, query a Collection, query for IP names and query for domain IP names.

Important Functions:

- *query(String,String*) – query the Collection using the Xpath string expression. Returns the results in a vector
- *update(String, String[], String[])* – update the entire Collection using the specified elements and their new values
- *update(String, String, String[], String[])* – update a single Resource in the specified collection with the specified elements and their new values

XPathUtil

Description

This class provides all the querying functions on the database by the Xpath query expressions. This class acts as a wrapper around the XPathAPI functions provided by Java. The query expressions are created specifically in each function. This is a static class that contains only static members.

Important Functions:

- *getClusterName(Node*) – retrieves the cluster name
- *getClusterAlias(Node)* – retrieves the cluster alias name
- *getClusterGateway(Node*) – retrieves the cluster gateway name
- *performXpathQuery(Node, String)* – perform an Xpath query on the node using given Xpath expression
- *getClusterIps(Node*) – get a list of cluster IPs from the group within the cluster
- *updateSingleNode(Node, String, String)* – update a single node(or Document) using the new values

XDBase

Description

This class represents the database itself. It contains a hashtable of Collections, which in turn contains a hashtable of Resources. The database waits for requests and then parses and processes them. The database also writes its contents back to local disk and can read those contents back the next time it starts up again. The database also keeps track of connected GUI clients (applet) and informs them of any changes in the database so that the GUI clients are up-to-date. To implement this, an additional thread is created besides the main thread. The second thread, guiThread, handles GUI clients by informing them of database changes. The guiThread waits until the database contents are changed. The main thread processes the request and if it is a successful transaction and the database has changed its contents, it changes a flag to indicate the database is “dirty” and wakes the guiThread. The guiThread wakes up and informs all the GUI clients of the database changes. The database also contains a private class called Connection. This class represents a connection made by a client.

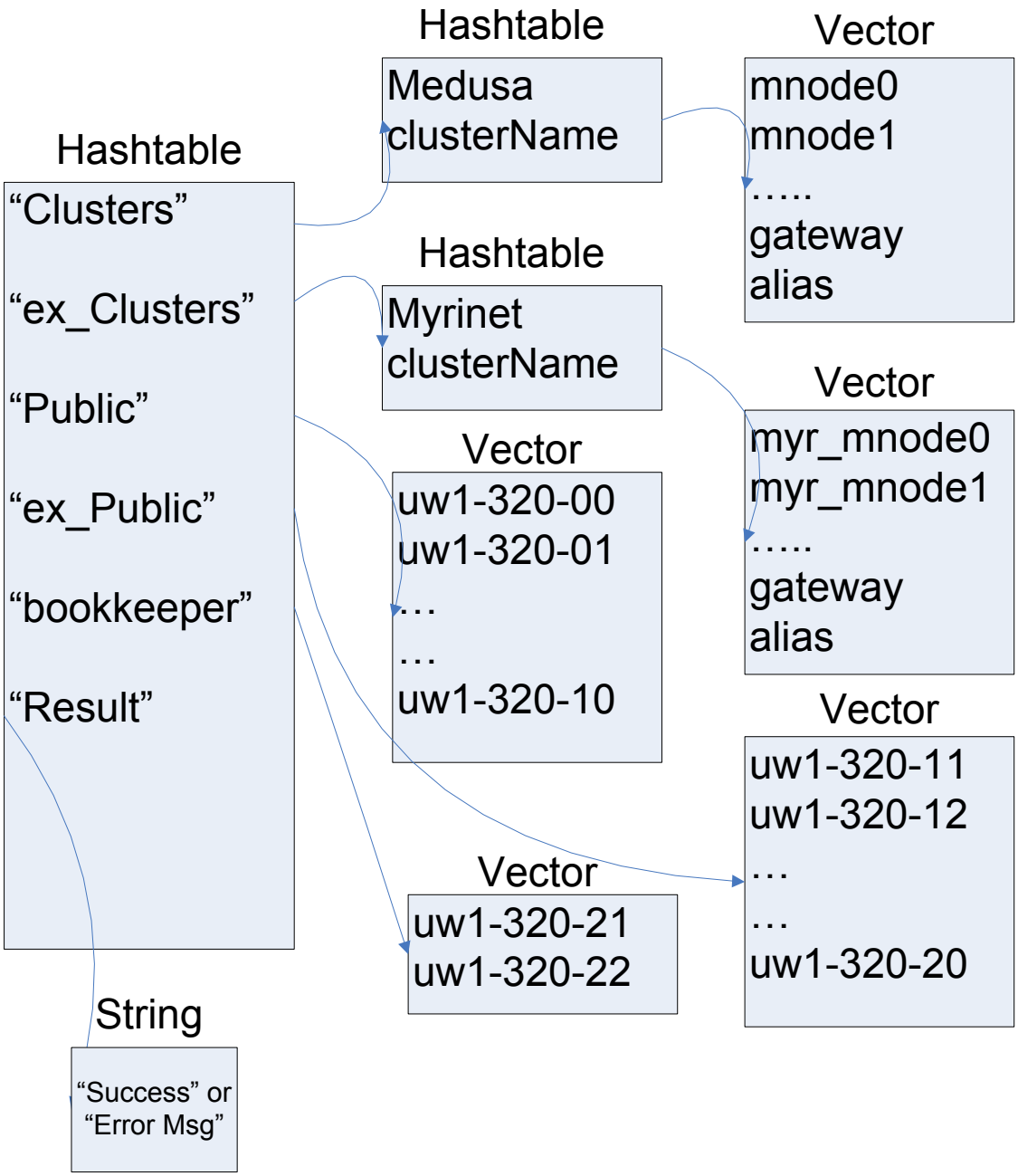

Fig 1. Structure constructed by database in a "QUERY_FOR_IP_NAMES" request

Above is the structure constructed by the database that is return to the CommanderAgent via the ResourceAgent when a "resource_needed" message is received.

Important Functions:

- *acceptConnections* – listens for clients attemping to connect with a timeout of 500ms
- *parseRequest(Connection)* – receive the client request, parses and processes it. If it was a successful transaction, wake up the guiThread. Send a result transaction message back to the client, if necessary.
- *createCollection(String)* – creates a new collection and store in the hashtable
- *synchronize* – write database contents back to the local disk using the pre-defined file name ("data.xml")
- *shutdown* – shutdown the database by first synchronizing and then closing the server socket. Also set the alive flag to false to indicate that the database desires to be shutdown.
- *storeCollection(String, Collection)* – stores a Collection using if the collection with the same key/name does not exist already exist.
- *storeResource(String[], Document)* - creates a Resource from the Document and a new Date object. It then stores the Resource in the specified collection, if it exists, and the collection does not already contain a Resource with the same key/name
- *retrieveCollection(String)* – retrieve the collection specified by the collection name/key. If the Collection does not exist, null is returned.
- *retrieveResource(String[])* – retrieve the Resource in the specified Collection. If the Resource exist, the DOM document is returned, otherwise null is returned.
- *retrieveLastModification(String[])* – retrieve the last modification date of the Resource in the specified Collection. If the Resource exist, the last modification date is returned, otherwise null is returned.
- *retrieveResourceList(String)* – retrieves the Resource list of the specified Collection. If the Collection does not exist, null is returned.
- *deleteResource(String[])* – deletes the Resource from the specified Collection. The Resource is deleted if the Collection exist and it contains the Resource.
- *deleteCollection(String)* – deletes the specified Collection if it exists.
- *query(String[])* – queries the specified Collection using the provided Xpath query expression. If the Collection does not exist, null is returned.
- *queryForIpNames(String[], Hashtable)* – query the database for available Resources (IP names). Construct a hashtable that contains clusters, extra clusters, public nodes, extra public nodes and bookkeeper nodes. If there are not enough nodes that satisfy

the user requirements, an error is returned to the user via the CommanderAgent.
- *markNodesAvailabilityTo(Hashtable, boolean)* – marks the availability state of nodes in the hashtable to the specified boolean value.
- *queryForDomainIps(String)* – queries for all IPs in the domain specified by the Xpath expression.
- *update(String[], String[], String[])* – updates, either one Resource or an entire Collection, using the provided values and elements to change. Returns the total number of elements changed.
- *addGUIClient(Connection)* – add this connection to the list of GUI clients in order to send the updated database.
- *updateGUI* – updates all the GUI clients of changes to the database. This thread will wait until notified by the main thread that there changes to the database. Once the thread is awake the database has changed, it sends the database contents to all the GUI clients. If a GUI client does not receive the snapshot, the database assumes that this GUI client has disconnected and removes the Connection from the list.

XDBaseGUI

Description

This is an applet-based GUI that interacts with the database. It can view the database contents, add and delete files. It can also connect to remote database, though issues have occurred while doing so using school computers because of the limited access being granted to applets.

Important Functions:

- *connectToDatabase* – connects to the database using the default address of localhost or user provided address in address text box
- *getData* – gets the data from the database
- *dataToCollectionList* – gets the list of collections and display them in the collection list Jlist
- *createGUI* – creates all the GUI elements. This takes a lot of time as Java and applets are really slow. It may seem like it is hanging but give it a few seconds depending on computer speed.

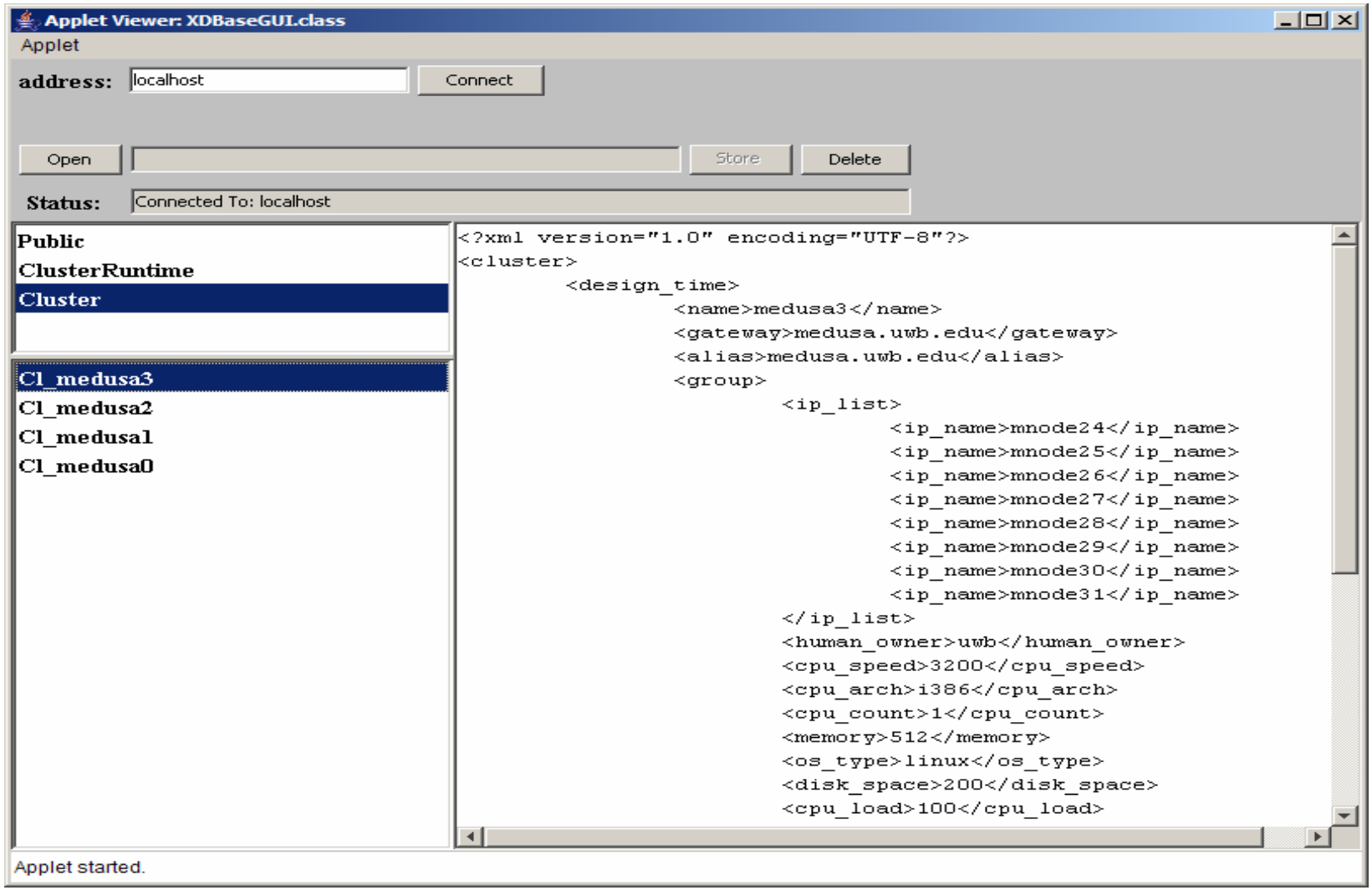

Fig 2. Applet GUI

XCollection

Description

This class is an interface between the ResourceAgent and the database. It provides functions that the ResourceAgent will need to start the correct Service to communicate and send requests to the database.

Important Functions:

- *initCollection(String)* – creates a collection with provided name
- *shutdown* – shutdows the database
- *synchronize* – writes database contents back to local disk
- *insert(String, String, String)* – inserts a Resource into a Collection
- *delete(String, String)* – delete a Resource from a Collection
- *retrieve(String, String)* – retrieve a Resource from a Collection
- *query(String, String)* – query a Collection using the Xpath expression
- *retrieveResourceList(String)* – retrieve a list of Resources from a Collection
- *retrieveLastModification(String, String)* – retrieve the last modification date of the Resource
- *update(String, String[], String[], String)* – update a single Resource or an entire Collection using Xpath expressions and new values
- *retrieveLastFTPConnection(String)* – retrieve the timestamp of the last FTP connection
- *queryForIpNames* – query the database for IP names that fulfill the requirements.
- *queryForDomainNames(String[])* – query the database for all IPs within a domain using the Xpath expression

Future Services

To add future services or request that the database should handle, decide if the service fits into one of the existing service classes. If it does, then implement the function or if it does fit into one of the existing classes, then create another subclass and then implement the function. For example if I wanted to create another request that retrieve the list of Collections, the function would be in RetrievalService class. Create a function called retrieveCollectionList. Connect to the database and send the request. The database should add a function to process that request as well. Below would be the changes that need to be added.

Service.java

```
protected static final int RETRIEVE_COLLECTION_LIST = 17;
```

RetrievalService.java

```
public vector retrieveCollectionList() {
        printMsg( “Retrieving collection list: “ );

        Vector result = null;

        if ( connectAndSendInformation( RETRIEVE_COLLECTION_LIST, null ) ) {
                try {
                        result = (Vector)objectIn.readObject( );


                        if ( result != null ) {
                                printErr(“database not initialized” );
                        }
                        else if (result.isEmpty( ) ) {
                                printErr( “database has no collections” );
                        }
                } catch ( Exception e ) {
                        printErr( “ trying to receive collection list “ );
                } finally {
                        closeAllConnections( );
                }
        else {
                closeAllConnections( );
        }

        return result;
}
```

XDBase.java

```
private static final int RETRIEVE_COLLECTION_LIST = 17;

public synchronize void parseRequest(Connection client ) {
        …
        Switch(request) {
                …
                case RETRIEVE_COLLECTION_LIST:
                        result = retrieveCollectionList( client );
                        sendResult = false;  // don’t need to send boolean result to client
                        break;
                …
}

private boolean retrieveCollectionList( Connection client ) {
        boolean result = true;

        Vector colList = retrieveCollectionList( );
```

```
        try {
                client.Out.writeObject( colList );
                client.Out.flush( );
        } catch ( Exception e ) {
                result = false;
        }

        return result;
}

private Vector retrieveCollectionList( ) {
        if ( collections == null )
                return null;

        Vector colList = new Vector( );

        Enumeration keys = collections.keys( );
        while ( keys.hasMoreElements( ) ) {
                colList.add( ( String )keys.nextElement );
        }
        return colList;
}
```

XCollection

```
pubic Vector retrieveCollectionList( ) {
        RetrievalService service = new RetrievalService( );
        return service.retrieveCollectionList( );
}
```

Compiling and Running

To compile all the agents and all the necessary files of the database, run the shellscript *compile.sh.* To run the database execute the shellscript *startDB.sh* and to shutdown the database, execute *shutdown.sh*. All shellscript files are currently located in *home/uwagent/MA/agentsNew.* As of this moment, the only argument the database needs is the port number, which it is passed in as the command argument. At the moment, the port of the services that connect to the database is 8000. So at the moment, please don't change the shell script startDB.sh because it has the port number of 8000.

ResourceAgent Modifications

Introduction

Changes to the database made it possible to retrieve a resource itinerary with cluster information, but the ResourceAgent needed to be modified to handle such changes. Collections were added and deleted to accommodate the new database. We no longer need the "Resource"

collection as it does not give information about whether or not that resource node was a private or public node. Instead, we now have a "Cluster" collection that contains an XML file with cluster information. Such information includes the cluster name, gateway, gateway alias, CPU speed, memory and cluster nodes. A "ClusterRuntime" collection contains all the cluster nodes of every cluster and their updated status. The runtime information of each cluster node is created and stored in the database when the cluster XML file is downloaded from the FTP server, ftp.tripod.com. A "Public" collection contains all the public nodes. The "ftpinfo" collection contains the timestamp of the last FTP connection. The ResourceAgent only contacts the FTP server if the last timestamp was over 24 hours ago. The "prbinfo" collection remains the same as it contains the probe frequency of the SensorAgents to determine which ResourceAgent should be the primary and launch SensorAgents to monitor nodes.

The ResourceAgent will now contact the database to retrieve all nodes, either in a private cluster or public, within a domain that the user as provided. The default domain at the moment is "UWB" and that will cause the database to return all nodes in the UWB domain to the ResourceAgent. The information received from the database will also be quite similar to that of when the database returns a list of available resource node to the CommanderAgent via the ResourceAgent.

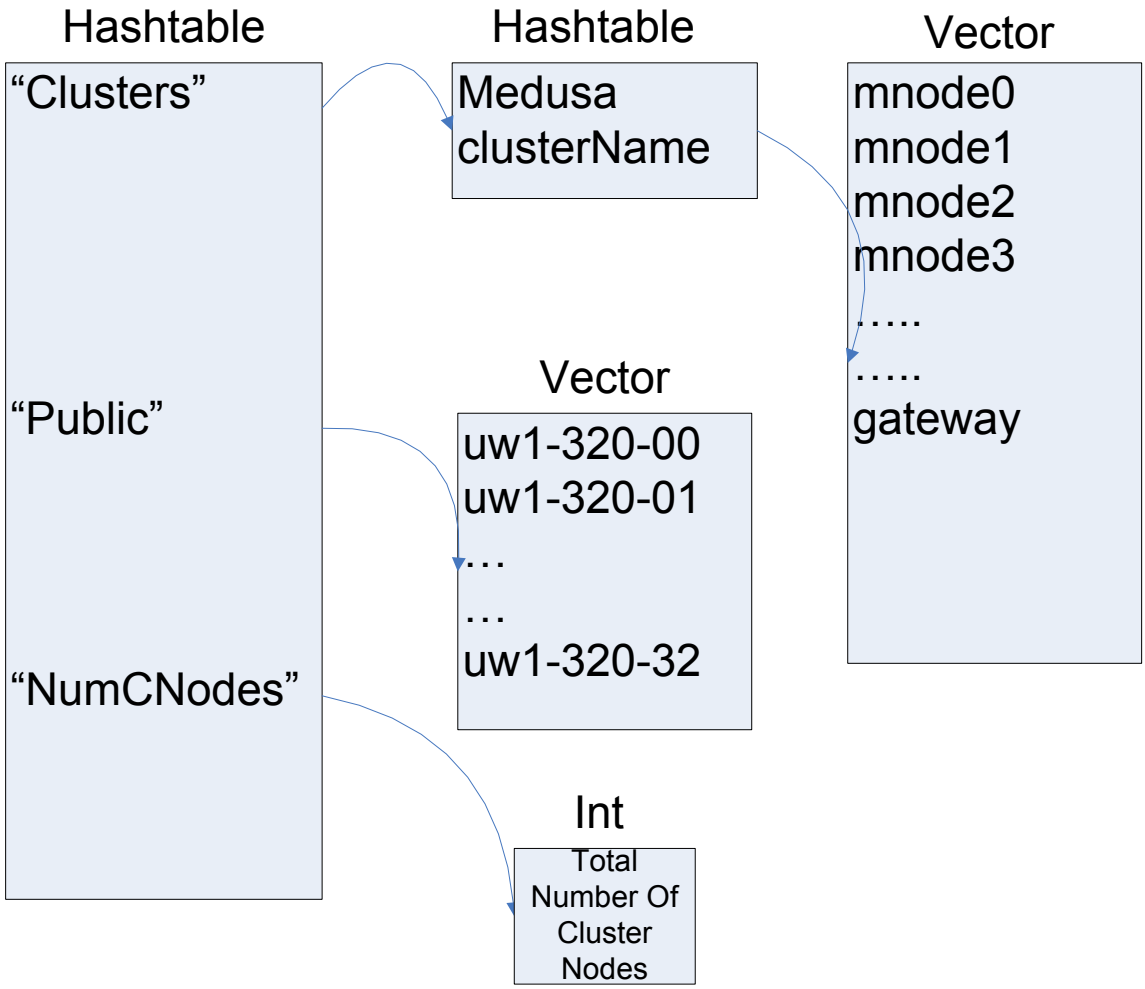

Fig 3. Structure constructed by database from a "QUERY_FOR_DOMAIN_IPS" request

When this information is retrieve from the database, the ResourceAgent constructs a string array argument to pass to its child SensorAgent.

Description of Changed/Added Functions

initDB

Create the "Cluster", "ClusterRuntime", "Public", "ftpinfo" and "prbinfo" colletction.

updateDB

Instead of checking the last modification of each XML file when downloading from the FTP server to the database, there is now only one check in the beginning. The "ftpinfo" collection contains the last time any files were downloaded from the FTP server. If the "ftpinfo.xml" file does not exist or the timestamp was over 24 hours ago, then proceed to

connect, download files from the FTP server and update the “ftpinfo.xml” timestamp. Download files in the “Cluster” and “Public” folder/directory on the FTP server.

downloadXmlsFromFtp(String ftpDir, FTPFile[ ] ftpList, String dbColName, Vector dbList)

This function was modified to not check the timestamp of the file in the database. This is because if we are at this point, then it has been over 24 hours since last FTP connection so we download regardless. Now it checks to see if the file also exists in the database and deletes it first. It is very important to delete the file first because the database does not allow you to just replace the file, it must be deleted first.

addXmlFromFtpToDB(String ftpDir, String filename, String dbColName)

This function downloads the file from the FTP server and stores it in the database. The modifications include ignoring carriage return characters that are downloaded with the FTP file. Secondly, if the file downloaded is a cluster file (dbColName == “Cluster”) a cluster runtime file must be created.

addRuntimeToDb(String fileContent)

This function is new and creates a cluster runtime file for each node in the cluster. First, create a DOM document for the cluster file (fileContent) in order to easily get the nodes from the cluster by creating an XPath expression to each “group” of the cluster. Iterate through all the nodes and query the cluster file for node information. Add all the runtime information and insert the new DOM document into the “ClusterRuntime” collection.

addAllChildNodes(Element root, Document document, Node currentGroupNode)

This function adds all the runtime information that the runtime file should have.

getRuntimePairs(Node node)

This function gets all the runtime pairs. It creates a string array with the first index containing an XML element. The next index contains the value of that element. For example, index 0 contains cur_cpu_speed and index 1 contains the CPU speed stated in the cluster file.

addSingleChildNode(Element root, String nodeName, String nodeValue, Document document)

Creates an element child by inserting it into the root element by using the nodeName as the name of the element and the nodeValue as its text value.

queryForIpNames

The structure that is returned is shown in Fig 1. The information needed from the CommanderAgent is the number of sentinels, multiplier, number of bookkeepers and the resource requirements. With the provided requirement, 3 Xpath expressions are created: cluster, cluster runtime and public Xpath expressions. Send the request to the database and wait for response.

startProbing

This function is changed so that it is able to monitor inside private clusters. First, a query is made to the database for all nodes within a domain. Look at Fig 2 for the structure that is returned. A string argument is created to spawn the root SensorAgent. The root SensorAgent is spawned on the same node as the ResourceAgent.

SensorAgent Modifications

Introduction

The SensorAgents were modified to handle private clusters as well as public nodes and to monitor the nodes dynamically. Previously, the nodes to be monitored were static and a predefined finite number of nodes. Now, nodes are monitored dynamically, defined at runtime and can be as many as possible. To accommodate the changes, the tree structure of the SensorAgent was modified. Look at Fig 4. for the tree structure. The root spawns has 2 children, the first 2 gateway children and the last 2 are the public client/server root. The rest of the tree is pretty similar except that when you are in a gateway part of the tree, you spawn cluster nodes and public nodes. The picture shows 5 clusters with 5 pu The 5^{th} cluster would reside on the left most children and would have id of 65. The G1 client and server root would be id 66 and 67, respectively. If only 3 clusters existed, then G4 would not exist and G3 client and server would be 68 and 69, respectively.

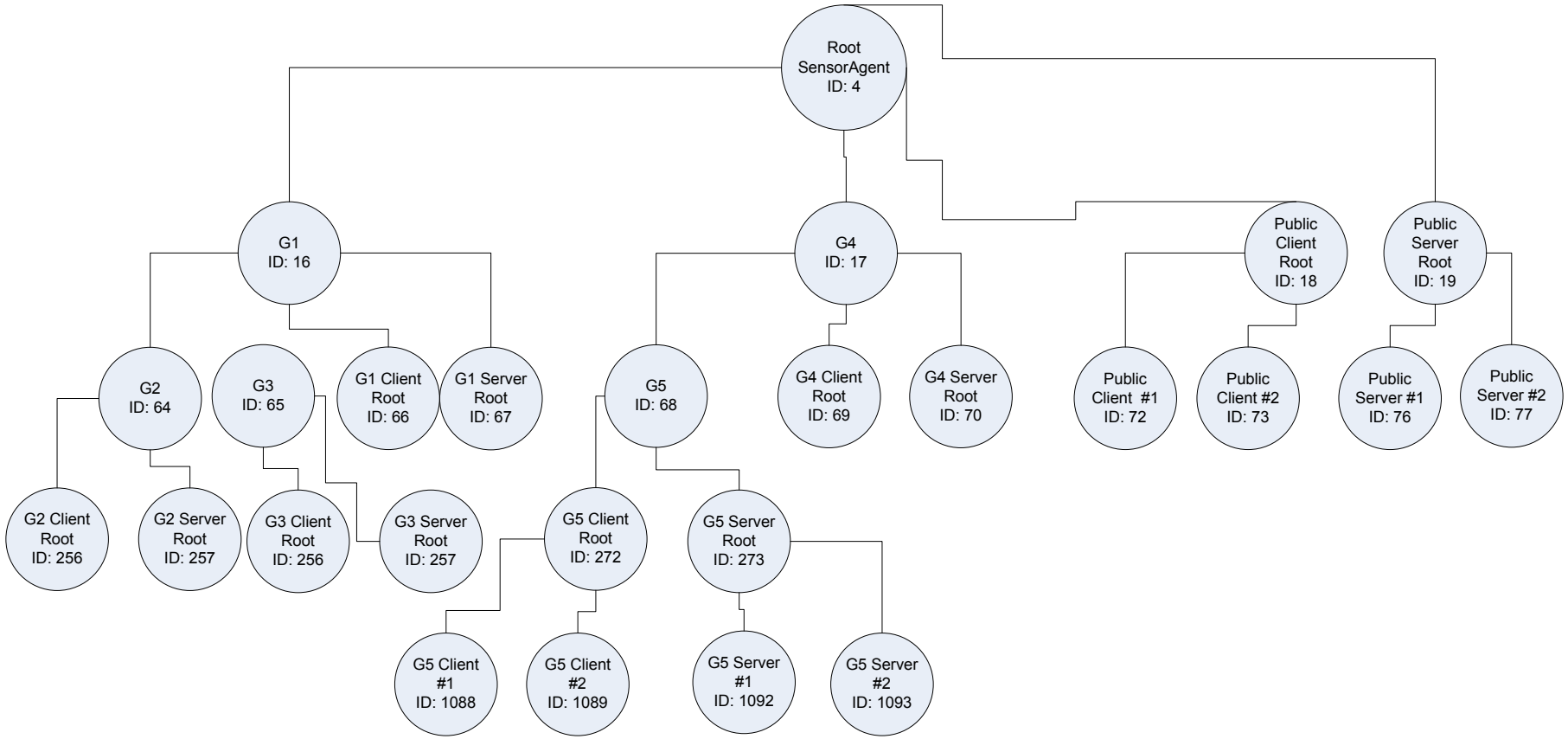

Fig 4. SensorAgent Tree Modification

	RootSensor
0	myfunc
1	port
2	probe freq
3	total nodes
4	total local nodes
5	total clusters
6	cluster name
7	c1 name
8	c2 name
9	c3 name
10	c4 name
11	c5 name
12	c1 #
13	c2 #
14	c3 #
15	c4 #
16	c5 #
17	public/local
18	
19	
20	
21	
22	
23	cluster 1
24	
25	
26	
27	
28	
29	
30	
31	cluster 2
32	
33	
34	
35	
36	
37	cluster 3
38	
39	
40	
41	
42	cluster 4
43	
44	
45	cluster 5
46	
47	
48	
49	
50	
51	

	root
	8000
	20000
	30
	5
	5
	localhost
	c1
	c2
	c3
	c4
	c5
	7
	5
	4
	3
	6
	p1
	p2
	p3
	p4
	p5
	p6
	c1 gateway
	c1 node1
	c1 node2
	c1 node3
	c1 node4
	c1 node5
	c1 node6
	c1 node7
	c2 gateway
	c2 node1
	c2 node2
	c2 node3
	c2 node4
	c2 node5
	c3 gateway
	c3 node1
	c3 node2
	c3 node3
	c3 node4
	c4 node1
	c4 node2
	c4 node3
	c5 gateway
	c5 node1
	c5 node2
	c5 node3
	c5 node4
	c5 node5
	c5 node6

child #1

0	gateway
1	8000
2	20000
3	16
4	7
5	2
6	c1
7	c2
8	c3
9	5
10	4
11	c1 node1
12	c1 node2
13	c1 node3
14	c1 node4
15	c1 node5
16	c1 node6
17	c1 node7
18	c2 gateway
19	c2 node1
20	c2 node2
21	c2 node3
22	c2 node4
23	c2 node5
24	c3 gateway
25	c3 node1
26	c3 node2
27	c3 node3
28	c3 node4

child #1.1

0	gateway
1	8000
2	20000
3	5
4	5
5	0
6	c2
7	c2 node1
8	c2 node2
9	c2 node3
10	c2 node4
11	c2 node5

child #1.2

0	gateway
1	8000
2	20000
3	4
4	4
5	0
6	c3
7	c3 node1
8	c3 node2
9	c3 node3
10	c3 node4

child #1.3

myfunc	0 Client Root
port	1 8000
probe freq	2 20000
children	3 3
uneven?	4 uneven
child1	5 c1 node2
child2	6 c1 node3
child3	7 c1 node4

child #1.4

0	Server Root
1	8000
2	20000
3	2
4	
5	c1 node6
6	c1 node7

child #2

0	gateway
1	8000
2	20000
3	9
4	3
5	1
6	c4
7	c5
8	6
9	c4 node1
10	c4 node2
11	c4 node3
12	c5 gateway
13	c5 node1
14	c5 node2
15	c5 node3
16	c5 node4
17	c5 node5
18	c5 node6

child #2.1

0	Gateway
1	8000
2	20000
3	6
4	6
5	0
6	c5
7	c5 node1
8	c5 node2
9	c5 node3
10	c5 node4
11	c5 node5
12	c5 node6

child #2.2

myfunc	0 Client Root
port	1 8000
probe freq	2 20000
children	3 1
uneven?	4 uneven
child1	5 c4 node2

child #2.3

0	Server Root
1	8000
2	20000
3	0
4	

child #3

myfunc	0 Client Root
port	1 8000
probe freq	2 20000
children	3 2
uneven?	4
child1	5 p2
child2	6 p3

child #4

0	Server Root
1	8000
2	20000
3	1
4	
5	p5
6	p6

Fig 5. SensorAgent Arguments

The above spreadsheet shows an example set of arguments for the SensorAgent. The tree in Fig 4 is just a part of the total tree created by these arguments. The spreadsheet shows the argument passed to each of the SensorAgent children, from the SensorRoot to some of its grandchildren.

Nodes stats are still sent from server to corresponding client and then to its client root. The client root will then send to its parent, either the root SensorAgent or a gateway. The gateway will send to its parent until the root SensorAgent receives stats from all its children. It will then send the information to the ResourceAgent to update the database.

Index 0 – 6 in a root or gateway SensorAgent has significant meaning. 0 is the function of the agent. 1 is the port number. 2 is the probing frequency. 3 is the total number of nodes. 4 is the total number of local or public nodes. 5 is the total number of clusters. 6 is the name of this cluster this agent is at. The cluster name and amount of nodes in each cluster are next. After that, the public/local nodes are defined. After the public/local nodes, each cluster's gateway and nodes are defined.

Index 0 – 4 in a client/server root or client/server has significant meaning as well. 0 is the function of the agent. 1 is the port number. 2 is the probing frequency. 3 is the total number children to spawn. 4 is a flag to determine whether or not this agent is responsible for migrating while probing or one its children is responsible for. Index 5 is the start of any children to spawn or migrate to.

If there is an uneven amount of nodes within a cluster, lets say 7, then there won't be a perfect client/server match-up of pairs. Thus, we want to have 3 pairs and the 3rd pair's client will migrate and probe. Meaning that that client will monitor one node and measure, and migrate to another node and measure. This will allow the client to still be able to perform bandwidth test will the server even if it is migrating. Only the LAST client of the cluster will be migrating. We only want the client to migrate because the corresponding server has already setup the ttcp server and no synchronization is needed after the client migrates. If the server migrated, then the ttcp server will have to be started again on the new node and client/server pair will have to re-synchronize in order to measure bandwidth.

At the moment, gateway SensorAgent only needs to spawn the receivingMessageThread to receive messages from its children. It does not probe at all. All it is responsible for is forwarding messages upward and spawning children. Later on, that may be something to do so we can monitor bandwidth between clusters/gateways.

Description of Changed/Added Functions

Much of the original code is still around, such as the synchronization among client/server pairs. The only thing changed is setting up the arguments for the children and the migration code. The old code assumes that when a client migrates, then the corresponding server also migrates and synchronization between the client and server must be done again. Now, since only the client migrates, there is no need to re-synchronize as the server has already set up the ttcp server and can measure bandwidth.

Init()

This function was changed drastically to create the string arguments for spawning children shown in figure 4. Please refer to the code and comments for most of the explanations as it makes understanding the code easier to see the comments and code together.

Future Work

Some ideas that I have includes monitoring the gateways so inter-cluster bandwidth can be measured. There still needs to be communication between CommanderAgent and ResourceAgent so that when the user job is done, nodes allocated can be deallocated. We would need to add another service and process that in the database. Another idea would be have it so that we can use any port for the database. That would need coordination from the user supplying the port number for the CommanderAgent, changing the startDB.sh shell script, and the ResoureAgent to change the port number on the services. That means we will need to add a static function in Service.java to change the port number. The applet would also have to allow for a port number input area.