Communication Protocol and Handling for a Decentralized Data Management Framework for Data Grids.

by

Brenden Conte

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the degree of

MASTER OF COMPUTER SCIENCE

Rensselaer Polytechnic Institute
Troy, New York

February, 2006
(For Graduation May, 2006)

CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to thank my parents, Gary and Karin, for their constant support; Boleslaw Szymanski for working with me on this, and Houda Lamehamedi for allowing me to work with her on this research.

ABSTRACT

Sharing large amounts of data between cooperative parties is a difficult process that has been approached with several incomplete solutions. The solution put forth by Houda Lamehamedi is a decentralized data management framework that uses replication to localize in-demand data. Such an architecture requires a Replica Management Layer to support replica generation and environment monitoring; a Resource Access layer to provide access to resources, including catalogs; and a Communication Layer which supports all aspects of communication between nodes. This paper describes the contributions to this approach, which were concentrated within the Communication and Resource Access Layers. It examines the developed protocol for such an environment, and details each aspect of the protocol while examining its functionality as it relates to the research project.

1. Introduction

In the scientific and educational communities, it is often necessary to share large amounts of data with colleagues. Means of doing this task, however, are not always apparent or practical. Accessing the required data remotely may be a solution; however poor connectivity or otherwise slow connections can render such a method useless. Likewise, it is not always practical to obtain a copy of that data, especially if the data set exceeds the recipients' storage capacity.

Grids are networks of disparate computers or nodes that are a part of the same global framework. Data Grids were born out of the highly data intensive modeling and data mining fields where large amounts of raw data required the high-performance computing resources, which were shared and distributed throughout the Grid, to process and compile it into results. Along those same lines, the collaborative environment of the grid allowed for the equally large amount of compiled results to be shared with other collaborators for future work. The Grids provided access to resources needed to conduct large-scale research and collaboration activities, however the management and execution of the Grid was manually driven, and required human intervention for everything. This environment created a demand for intelligent middleware technologies that could provide transparent access to the shared resources, with automated management of the Grid, its nodes, and its resources.

Today, there are several Data Grid projects that are deployed and in use, such as GriPhyN [2] and the EU DataGrid [3,4]. The Open Grid Service Architecture (OGSA) [5] attempts combine existing Grid technologies using the Open Grid Service Infrastructure (OGSI) [6], in which "a Grid service is a (potentially transient) service that conforms to a set of conventions, expressed as Web Service Definition Language (WSDL) interfaces, extensions, and behaviors" [1]. All of these projects, however, require a substantial amount of setup, configuration, testing, and maintenance to properly deploy and run a node. Adding and removing nodes, especially for intermittent connectivity, requires substantial manual efforts, since the support software and toolkits do not provide the proper framework needed for such a dynamic global configuration. In addition, the centralized management and configuration act as a hindrance to large-scale growth of these Data Grids. Replication between nodes was also a wasteful manual process, since it is impossible for a human to determine where data must be placed for optimal efficiency and productivity.

Working with Houda Lamehamedi, whose research in Decentralized Data Management Framework for Data Grids was based on two previous research works [6,7], we sought to develop a solution to network users of the various communities into a data grid, with decentralized data managed automatically by the network itself, or more specifically, the nodes within the network. This includes such tactics as replication of the data to other nodes, selection of appropriate neighbors based on request fulfillment, and intelligent search techniques for finding data over the distributed network so as to not cause resource consumption problems. Such a grid requires careful management of both network and storage resources as to not overload either. Manual administration and intervention should not be required.

Our approach utilized a 3-layer structure within each node:

- Replica Management Layer
- Resource Access Layer
- Communication Layer

For my part in the research, I contributed a majority of my efforts toward building a robust Communication layer, while also contributing to building the Resource Access Layer and providing support for the Replica Management Layer. The nodes were written in java and built using the Java Development Environment version 1.5.0.

The work done in this thesis has been used to produce data, and was included in “Decentralized Data Management Framework for Data Grids” by Houda Lamehamedi and Boleslaw K. Szymanski [8].

2. The Communication Layer

The communications layer is largely made up of the MessageHandler object, herein known as the “message handler”. The message handler handles all aspects of requests in the node, including generating new requests, determining the appropriate action for received requests, maintaining the state of requests when necessary, and often immediately processing the requests internally, generating the data for the node to send. The message handler is also in control of the data catalog for the node.

2.1 Protocol

I decided early in development that the protocol for inter-node communication would be based in XML. Using XML has the advantages of being a pre-existing standard in formatting data, and as such, already had pre-existing support for processing and data manipulation. It also allowed for infinite expandability as the project progressed – many new request types were programmed into the protocol over the course of the research, never interfering or requiring adjustments of previously written message types. It also eliminated any data ordering requirements (with the exception of nested data structures), and allowed for easier debugging, since the data was human readable.

2.2 Message Types

2.2.1 The Basic Message Format

The format of the inter-node requests varied slightly between various request types; however the basics of a request all had the same core, as seen in Figure 1 below.

```
<Message Version="version" Type="type">
   <Source>sourceIP</Source>
</Message>
```

Figure 1 – Basic message example

The basic request format includes a “Message” tag, with “Version” and “Type” attributes, and a source tag within the message content. The version attribute was installed as a method for future backwards-compatibility – if a particular request type changed in an upgrade, the format version could be incremented, thus acting as a safety variable for sending and receiving nodes alike. In future work, format negotiations could take place so that each node communicates in a lowest common denominator, if possible. In this research, however, no such versioning support was needed since the experiments took place under a controlled environment with all nodes running the same code; therefore, all the version strings were always “1.0”.

The “Type” attribute determines what type of request the message is, and is what gives the protocol its extensibility. Types will be discussed in the next section.

Finally, all requests contained a “Source” tag. The source tag contains where the original request came from, who sent this particular message. This allows for nodes to process a request, and communicate directly back to the requesting node if the situation calls for it.

2.2.2 Hello Message Type

The Hello request type is used to introduce the originating, or connecting, node into a grid network, usually immediately after the originating node connects to another node in the grid. This acknowledges to the established node that the connection is indeed from a connecting node, and that the connecting node should be recognized and included in the grid. The Hello message type is detailed in Figure 2 below.

```
<Message Version="version" Type=”hello">
   <Source>sourceIP</Source>
</Message>
```

Figure 2 - Hello message type example

This initial message also had the secondary effect of confirming that the connection was established and not immediately dropped.

2.2.3 Bye Message Type

The Bye request is used to cleanly disconnect from the grid network. It is issued to all connected nodes, including parents, children, and siblings to notify them that the disconnecting node is no longer valid. The Bye message type is detailed in Figure 3 below.

```
<Message Version="version" Type="Bye">
   <Source>sourceIP</Source>
</Message>
```

Figure 3 - Bye message type example

2.2.4 ResourceUpdate Message Type

The ResourceUpdate message type is the type of message that nodes will send to distribute their catalog entries to parents, children, and siblings. The ResourceUpdate message type is detailed in Figure 4 below.

```
<Message Version="version" Type="ResourceUpdate">
   <Source>sourceIP</Source>
   <Resource>
     <Add>newfile1</Add>
     <Add>newfile2</Add>
     <Del>deletedfile1</Del>
   </Resource>
</Message>
```

Figure 4 - ResourceUpdate message type example

The Resource tag within the ResourceUpdate message contains a list of Add and Del tags, indicating what files have been Added and/or Deleted since the last update. There is no order to the Add or Del tags – they can appear in any order.

2.2.5 Request Message Type

The Request message type is the message that is sent when a node needs data that is not locally available. The details of a request message are shown in Figure 5 below.

```
<Message Version="version" Type="Request">
   <MsgID>MsgID</MsgID>
   <TTL>TTL</TTL>
   <Source>sourceIP</Source>
   <LastNode>IP of Sending Node</LastNode>
   <Request>
     <Type>file</Type>
     <Name>filename1</Name>
   </Request>
   <History Track="true/false">
     <Node>IP1</Node>
     <Node>IP2</Node>
   </History>
</Message>
```

Figure 5 - Request message type example

The MsgID field in the request message type is a locally unique id that the node can use to easily distinguish various requests, and ties into the resource management layer. In our implementation, the MsgID was an incrementing integer value, one per request, which ensured there would be no duplicate message IDs during our tests. There is no such integer limitation, however – the only requirement is that no other such MsgId be in use at the time.

The TTL field, or “Time To Live” field, works much like its TCP/IP counterpart, in that it creates a maximum retransmission limit to prevent endless transmission loops and limits exceedingly long links. As long as the TTL after node processing was greater than 0, the request would be retransmitted. If a node, after processing the request and decrementing the TTL, tried to forward a request with a TTL of zero, the node would simply discard the request instead of forwarding it.

The LastNode field contains the IP of the last node to forward the request. By definition, the value is always overwritten by a node with its own IP before being forwarded to other nodes. If the message is received from the node that originated the request, both the Source and LastNode values will be the same; otherwise a request answer is always sent to the Source value with a DataOffer message (detailed below).

The Request value contains both a Type tag and Name tag. The Type tag was to be used to identify the type of resource the request was for, under the implication that different types of data may have different storage, index, and retrieval mechanisms. For the current research, the only valid type was type “file”. The “name” tag was used as the identifier for the resource in question. It is assumed that all names/identifiers would be globally unique, although the current system has no method of enforcement.

Finally, the History tag controlled the optional history tracking functionality of request messages. If the Track attribute was set to false, the History tag was ignored, and forwarded on unchanged. If the Track attribute was true, however, the node would append its IP within a Node tag to any other Node tags within the History tag. In this case, the ordering of the Node tags indicates the order that the nodes processed the request, and thus its integrity is maintained. The end result was a complete history of every node that the request had been forwarded through, in chronological order. This function was mainly used for debugging, and could also be used for network performance testing and maintenance.

2.2.6 DataOffer Message Type

The DataOffer message is generated whenever a node receives a Request message, and has the resource sought after by the original requesting node. It takes the following form, as shown in Figure 6 below.

```
<Message Version="version" Type="DataOffer">
   <MsgID>MsgID</MsgID>
   <Source>sourceIP</Source>
   <LastNode>IP of Sending Node</LastNode>
   <DestNode>IP of original node</DestNode>
</Message>
```

Figure 6 - DataOffer message type example

The MsgID tag contains the original MsgID field from the request message, so that the original requesting node will know which request the DataOffer is in response to. The Source and LastNode are set to the IP of the responding node, while DestNode contains the Source IP from the original request, indicating that this offer is to go to the original requesting node.

2.2.7 DataAccept Message Type

A DataAccept message is only generated by a node that had sent out a request message and subsequently received a response via a DataOffer message from another node in the grid. The format of a DataAccept message is detailed below in Figure 7.

```
<Message Version="version" Type="DataAccept">
   <Source>sourceIP</Source>
   <DestNode>IP of offering node</LastNode>
   <Request>
     <Type>file</Type>
     <Name>filename1</Name>
   </Request>
</Message>
```

Figure 7 - DataAccept message type example

The DestNode is the Source of the DataOffer message, indicating the acceptance of the offer. Which resource to accept is dictated within the Request tag. Like the Request message type, the request tag within the DataAccept message contains a Type tag and a Name tag, indicating which resource was requested. The Type and Name values are retrieved from the Resource Management layer, using the MsgID returned from the offering node.

2.2.8 Benchmark Message Type

The Benchmark message type is used to gather crude information about the overall speed of a connection or node. The format of the message is shown below in Figure 8.

```
<Message Version="version" Type="Benchmark">
   <MsgID>MsgID</MsgID>
   <Source>sourceIP</Source>
   <Payload>garbage data</Payload>
</Message>
```

Figure 8 - Benchmark message type example

The MsgID used by the Benchmark message type is independent of the MsgID values used in the Request, DataOffer, and DataAccept message types. Like those message types, the MsgId is stored in the Resource Management layer along with a millisecond time measurement, and the sending node forgets of its existence until a BenchmarkResponse message type is received.

The Payload tag value is a chunk of arbitrary data to enlarge the message size to something non-trivial, but not so large as to consume significant network resources. In the course of our research, we used a sixty-five kilobyte sequence of ‘x’ characters.

2.2.9 BenchmarkResponse Message Type

The BenchmarkResponse message type is issued in direct response to a Benchmark message type. Its format is nearly identical to the Benchmark message type, except there is no payload data, as shown in Figure 9.

```
<Message Version="version" Type="BenchmarkResponse">
   <MsgID>MsgID</MsgID>
   <Source>sourceIP</Source>
</Message>
```

Figure 9 - BenchmarkResponse message type example

The MsgID in the BenchmarkResponse is identical to the MsgID supplied in the original Benchmark message.

2.3 Message Processing

2.3.1 Connecting/Disconnecting

When a node wishes to connect to a grid network, it must first connect to another node already on the network. To do this, it connects and issues a Hello message to the destination node. If the connection appears to be valid, the connecting node considers the connection valid, and the destination node to be its parent.

When a node’s message handler receives a hello message type, it stores the source IP and sets its object as containing Hello message data before returning. In this manner, the other parts of the node can see that there was a connection request, and subsequently handle the addition of the node to the grid appropriately within other services, like the Routing and Connectivity service.

After issuing the Hello message, the node can then send its resource catalog to its neighbors via a massive Catalog Update. This is covered in section 2.3.2.

When a node is shut down or otherwise wishes to disconnect from the grid network, the node will issue a Bye message to its parent, siblings and children. Those nodes are expected to remove the disconnecting node from all references as a peer, parent, or child node, and remove any catalog entries that are sourced from the disconnecting node.

2.3.2 Catalog Updates

There were going to be two situations in which this type of message is used. The first is when a new node connects to an existing node, after it has issued a hello message. In this case, the ResourceUpdate message would include a complete listing of the node's local catalog. It would also be used during normal operation when the node's local catalog changes – it would issue this request to update parent, sibling, and children nodes, using the Add or Del tags within the Resource tag to indicate which files were added or removed since the last catalog update.

In the actual implementation, the functionality for sending the complete catalog was built into the nodes, however it was not used in the later builds. The finer-grained implementation for catalog updating was thus not implemented.

2.3.3 Requests

A Request message is created by a node when it is looking for data that does not exist locally to that node. The message handler will generate the request message, record the generated MsgID value and the requested filename together within the Resource Access Layer, and hand it off to the rest of the node, where the Routing and Connectivity service will then flood that request to all its parents, children, and siblings.

When a node receives a Request message, it first examines the request and checks its local catalog for the requested resource. If the requested resource is not found, it then decrements the TTL, checks the History Track attribute, adding tracking information if necessary, and then broadcasts the requests to all parents, siblings, and children – except for the node that sent the request.

If a node receives a Request message and is able to find the resource locally, the node will generate a DataOffer message, and using the Source tag value from the Request message, send a reply with the MsgID to the requestor. There is no state associated with this action, and no further processing is done for this sequence unless a DataAccept is subsequently received.

If a node is able to find the resource in its catalog, but rather than a path value it retrieves an IP, the message handler will forward the request to the returned IP only, instead of using a broadcast forward. This targeted forward would still follow all the system rules as long as all other parameters allowed it (such as the TTL being greater than zero). This functionality was built into the node, however when the catalog exchange was disabled, nodes ceased receiving foreign catalog entries, and thus IPs were not stored in a local catalog.

Support for recording multiple resource locations was built into the node, however the node upon scanning and retrieval, would only accept the first entry. Future modifications could use an algorithm to decide the best resource to use, or if they're all IPs, a multiple targeted forwarding system could be used.

When the generated DataOffer message is received by the original requesting node, it first checks the MsgId and the Request Name tag against the data stored in the Resource Access layer. If it finds a match, it indicates that the DataOffer message is a response to an open request, and it generates a DataAccept response to send back to the offering node – but not before it deletes the request reference from the Resource Allocation Layer. By deleting the reference, it will cause any future DataOffer messages to be silently ignored. This mechanism of deleting the reference thus indicates that the request is either fulfilled or being fulfilled. In future iterations, it could also mean that the request was canceled, however there was no cancellation support in the code used in this iteration. When the node finally returns, it will have a DataAccept message ready to be sent by the Routing and Connectivity service.

If the offering node subsequently receives a DataAccept message, the node will then examine the Request contents of the message to see exactly which resource the original node is accepting. With that information, the node will then connect to the original node, sending the requested data.

The request mechanism was built with this three-way acceptance mechanism – Request, DataOffer, DataAccept – because the requesting node needed to have the authority to dictate which node it wanted to accept the data from. While not implemented in this version, it would be possible for the requesting node to accept several offers within a window of opportunity, and then decide which offer suited the requesting node best – perhaps after benchmarking the possibilities, for example. If the offering nodes simply sent their data to the requesting node upon the receipt of a request, there would be no control in receiving data by the requesting node. If multiple hosts had the requested data in their local catalogs – which is a very possible and plausible situation, given the replication system in the grid network – they would all send their data to the requesting node, at best creating unnecessary duplicated network traffic, and at worst corrupting the data being received by the requesting node. Using the tree-way acceptance mechanism, this problem is avoided by allowing the receiving node to dictate the transfer of data.

2.3.4 Benchmarks

When a node needs to benchmark a connection, it generates a Benchmark message, which contains a pre-determined amount of garbage data to pad the message to a certain size. The benchmarking node uses this node to calculate the time it takes to send the message and receive a response from the destination node.

The destination node of a Benchmark Request simply responds with the same MsgID as the original request, without the payload.

While a very crude method of measuring bandwidth and latency, we chose to use this method over simple latency tests, like ping, and large-scale bandwidth data collection and monitoring. Using a single Benchmark request message with a non-trivial yet still small amount of data, the benchmarking node gets a feel for both latency and throughput, as well as the overall load on the destination node. A heavily loaded node would accept the Benchmark message, but take more time to respond and send data back to the source node than a lighter loaded node on a slightly slower connection.

There were no tests to determine an optimal payload size, which would be needed for a wider, large scale deployment.

3. The Resource Access and Replica Management Layers

The Resource Access Layer had the task of keeping track of all available resources and tabs on all open requests and benchmarks.

The storage medium for the Resource Access Layer was a Berkeley Database Environment. The message handler used this layer to store persistent data between its various object instances, using the following databases for its various data storage.

- Files
- openRequests
- openBenchmark
- requests

The Berkeley Database Environment is a key-value associative database, which suited the needs of our nodes since all the stored data was simple.

3.1 Files Database

The files database is a type of resource catalog. The name of the 'files' for the database originates from the resource type being requested – in our research, the type of request in a request message was always of type "file". The design assumed that other resource types for which a catalog would exist would contain appropriately named databases specific for them. This separation was designed since different resource types may have different methods of identifying how to access such resources.

This task of initially populating the database was moved from within the message handler to within the node's startup code, so that by the time the first request arrives, the message handler can simply query the database immediately.

The database key is the identifier for the resource. In general, when a request message is parsed, the retrieved value of the Name tag is used to search the database keys and find the proper entry. In this case, the database key is the file name. The value for each database consists of either the path to the local file, or an IP address of where the node received a matching catalog entry with the identifier. When a request came in and the message handler did a local scan of its catalog to see if it could fulfill the request, it scanned the database keys, looking for the identifier. The identifier is assumed to be globally unique, although for our research there was no mechanism to enforce such a restriction. If a resource had multiple known locations, the value stored was a semicolon-separated list of the locations known for the resource.

3.2 openRequests and openBenchmark Databases

When a Request message is issued by a node, the details of the request are stored in the openRequests database before being sent. The key stored is the MsgID generated by the message handler, and the value is name of the resource being requested. When the message handler later receives a DataOffer message, it uses the MsgID supplied within the message to search the database and identify which request the DataOffer pertains to. If the MsgID is found, it retrieves the value of the database entry and uses it in the Name fields in the subsequent DataAccept message. If the MsgID is not found within the openRequests database, it means that the request is no longer valid – it may have already been fulfilled (or is in the process of being fulfilled), or the request may have been cancelled – either way, the node no longer needs the data.

The openBenchmark Database is identical to the openRequests database, except instead of storing MsgIDs for requests, the MsgIDs are specific to benchmark queries.

3.3 Requests Database

The requests database is the database used to keep tally of the number of requests the node saw. The key for the database is the resource name, and the value is simply the number of requests – whether the request was fulfilled locally or forwarded on to other nodes. This database allowed for data analysis of traffic and requests.

4. Discussion and Conclusion

The implemented protocol successfully manages all the required tasks for the research, while allowing ample room for expandability in future modifications and adjustments. The three-way exchange required gives the requesting node full authority as to what data it accepts, and recorded data along with benchmarking can give the node ample data for replica management.

The databases used within the Resource Access Layer, while simple, were effective, and retained data between sessions. Future projects may require a more robust data storage solution. The given code and protocols can assure that this can happen with minimal disruption of referencing code.

The entire model aims and succeeds at solving a small set of problems and goals, local to this research. There are certainly more hurdles to overcome with a larger, more robust grid network, but with the solid base provided here, it can be easily done.

5. Cited References

[1] K. Czajkowski, A. Dan, J. Rofrano, S. Tuecke, and M. Xu, "Agreement-based Grid Service Management (OGSI-Agreement)", *Global Grid Forum*, GRAAP-WG Author Contribution, June 2003.

[2] Grid Physics Network (GriPhyN). http://www.griphyn.org

[3] The European Data Grid Project, The DataGrid Architecture 2001, http://eu-datagrid.web.cern.ch/eu-datagrid/

[4] D. Bosio, J. Casey, A. Frohner, L. Guy et al, "Next Generation EU DataGrid Data Manage-ment Services", Computing in High Energy Physics (CHEP2003).

[5] K. Czajkowski, A. Dan, J. Rofrano, S. Tuecke, and M. Xu, "Agreement-based Grid Service Management (OGSI-Agreement)", Global Grid Forum, GRAAP-WG Author Contribution, June 2003.

[6] H. Lamehamedi, B. K. Szymanski, Z. Shentu, E. Deelman, ``Data Replication Strategies in Grid Environments," *Proceedings of ICAP'03*, Beijing, China October 2002, IEEE Computer Science Press, Los Alamitos, CA, 2002, pp. 378-383.

[7] H. Lamehamedi, B. K. Szymanski, Z. Shentu, E. Deelman, ``Simulation of Dynamic Data Replication Strategies in Data Grids" *Proceedings of IPDPS'03, Homogeneous Computing Workshop*, Nice, France April 2003.

[8] H. Lamehamedi, B. K. Szymanski, "Decentralized Data Management Framework for Data Grids" *Future Generation of Computing Systems,* in review.