Scientific Middleware for Abstracted Parallelisation

Daniel Goodman

Oxford University Computing Laboratory, Oxford, England, OX1 3QD
Daniel.Goodman@comlab.ox.ac.uk

Abstract. In this paper we introduce a class of problems that arise when the analysis of data split into an unknown number of pieces is attempted. Such analysis falls under the definition of Grid computing, but fails to be addressed by the current Grid computing projects, as they do not provide the appropriate abstractions. We then describe a distributed web service based middleware platform, which solves these problems by supporting construction of parallel data analysis functions for datasets with an unknown level of distribution. This analysis is achieved through the combination of Martlet, a work-flow language that uses constructs from functional programming to abstract the parallelisation in computations away from the user, and the construction of supporting middleware. To construct such a supporting middleware it is necessary to provide the capability to reason about the data structures held without restricting their nature. Issues covered in the development of this supporting middleware include the ability to handle distributed data transfer and management, function deployment and execution.

1 Introduction

In this paper we introduce a class of problems that arise when the analysis of data split into an unknown number of pieces is attempted. Such analysis falls under the definition of Grid computing, but fails to be addressed by the current Grid computing projects [1,2,3,4,5,6], as they do not provide the appropriate abstractions. We then describe in detail a prototype web service based middleware which was created to provide a distributed implementation of the work-flow language Martlet [7]. Martlet allows users to write parallel programs to analyse distributed data without having to be aware of the details of the parallelisation, so addressing the described problems.

While applicable to a wide range of projects, this was originally created in response to some of the problems faced in the distributed analysis of data generated by the Climate*Prediction*.net[1] (CPDN) [8,9] project. Climate*Prediction*.net is a distributed computing project inspired by the success of the SETI@home[2] [10]

[1] http://www.climateprediction.net
[2] http://setiathome.ssl.berkeley.edu

project. Users download a model of the earth's climate and run it for approximately fifty model years with a range of perturbed control parameters before returning results read from their model to one of the many upload servers.

The output of these models creates a dataset that is distributed across many servers in a well-ordered fashion. This dataset is too big to transport to a single location so analysis must be performed by distributed algorithms if a user wants to analyse more than a small subset of the data. Users are unable to ascertain how many servers a given subset of this data spans, nor should they care. Their interest is in the information they can derive from the data, not how it is stored. In order to derive results it is intended that users submit analysis functions to the servers. However as this dataset provides a resource for many people, it would be unwise to allow users to submit arbitrary source code to be executed. This means a trusted work-flow language and a loosely coupled supporting middleware, capable of running in a highly distributed environment is required as an intermediate step. So facilitating the construction of analysis functions from an extendable set of approved functions. The contribution of this paper is to describe the prototype of such a framework, and how this it can interact with existing grid projects to extend their functionality.

2 Related Work and Motivation

There is a wide range of middleware already in existence to support grid services and distributed web services, starting with SOAP [11] containers like Apache Axis [12] and Jakarta Tomcat [13]. These are then used as the basis of other middlewares such as those produced by the Open Middleware Infrastructure Institute (OMII) [14] and Globus (GT4) [15] which are intended to provide a generic basis on which to build other services. Projects such as MONET [16] look at how to combine different existing services to provide required mathematical functions for users on demand. Projects such as those built on Styx [17] explore alternative ways of transferring data to computations through the use of streaming. In addition to these, there is the range of middlewares [1,2,3] produced to support particle physics etc. Philosophies put forward in work such as WS-RF [18] and WS-GAF [19] are used to under pin these differing middlewares. These middlewares are then built on providing extended functionality by a range of work-flow languages [4,5,6].

Existing combinations of middleware and work-flow languages are not sufficient for analysing the CPDN dataset. While there is a wide range of different languages, compatible with different middleware, supporting different tools, databases, scientific equipment etc, all of them address the same programming model. In this model there are a known number of data inputs that need to be mapped to computational resources, and as such none of them are able to handle data that is split into an unknown number of pieces when the work-flow is submitted. This problem occurs because these work-flow engines, although aware of the data type, don't *interpret* the data, but instead they simply pass it between functions. This restriction means that existing work-flow engines are

not able to take a generic work-flow and use it to generate work-flows that match the partitioning of data they are passed.

An example of a situation where the current solutions are ineffective is the calculation of an average across a list. Across a local list $[x_0, x_1, \ldots, x_{n-1}]$ a solution can be written as;

$$\overline{x} = \frac{\sum_{i=0}^{n-1} x_i}{n}$$

This is easily described with existing workflow languages. However if this list of numbers is now partitioned into a pieces $[x_0, x_{n_1-1}], \ldots, [x_{n_{(a-1)}}, x_{n_a-1}]$, and the solution rewritten as:

$$\begin{aligned}
y_0 &= \textstyle\sum_{i=0}^{n_1-1} x_i \\
z_0 &= n_1 \\
& \\
y_1 &= \textstyle\sum_{i=n_1}^{n_2-1} x_i \\
z_1 &= n_2 - n_1 \\
&\vdots \\
y_{a-1} &= \textstyle\sum_{i=n_{a-1}}^{n_a-1} x_i \\
z_{a-1} &= n_a - n_{a-1} \\
& \\
\overline{x} &= \frac{\sum_{i=0}^{a-1} y_i}{\sum_{i=0}^{a-1} z_i}
\end{aligned}$$

This can now not be written in existing work-flow languages unless the value of a is known at the point the work-flow is written or the programmer includes the specifics of how to handle the variable number of arguments. This adds complexity to the system that the user does not want and may not be able to deal with, and simultaneously introduces a much greater potential for the insertion of errors into the process.

In this article we briefly introduce the language we have developed as a solution to this problem. We then look in detail at the implementation of the middleware that was designed to support this language and at how future work in this project might develop.

3 Overview of Martlet

Martlet [7] is the name of the work-flow language supported by this middleware design and implementation. The language supports most of the common features expected of a work-flow language, but in addition it also has constructs inspired by the inductive constructs of functional programming languages [20], which it uses to abstract the parallelisation of the data and work-flow. In addition it supports two classes of data structure, 'local' and 'distributed' to enable it to reason about the data it handles.

We chose to design a new language rather than extending an existing one because the widely used languages are already sufficiently complex that an extension for our purposes would quickly obfuscate the features we where aiming to explore. Moreover, at the time the decision was taken to follow this path, there were no suitable open-source work-flow language implementations to adapt. It is hoped that in due course the ideas that have been developed in this language will be added into other languages.

In functional programming languages it is possible to write extremely concise powerful functions based on recursion. For instance the reverse of a list of elements can be defined in Haskell as:

```
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

This simply states that if the list is empty, the function should return the list, otherwise it should take the first element from the list and turn it into a singleton list. Then it should recursively call reverse on the rest of the list and concatenate the two lists back together. The importance of this example is the separation between the base case and the inductive case, which allows the function to avoid ever mentioning the length of the list. Using these ideas it has been possible to construct a language that abstracts the level of parallelisation of the data away from the user, leaving the user to define algorithms in terms of a base case and an inductive case. For example, the distributed average problem looked at in section 2, taking the distributed matrix `a` and returning the average in a column vector `b`, can be written in Martlet as.

```
//Declare abbreviations for URIs, this is done to improve
//readabillity of the function
define
{
    uri1 = baseFunction:system:http://cpdn.net:8080/Martlet;
}

proc(a,b)
{
//Declare the required temporary variables, existing data
//structures are taken as an argument to parameterise where
//these temporary variables will be first needed and how
//many pieces the distributed ones must be in.
    y = dismatrix(a);
    z = disinteger(a);
    i = integer(b);

//The base case where we sum all the columns into a single
//vector y, and record the number of columns using the integer
//z. This happens in parallel on each piece of the matrix.
```

```
    map
    {
        matrixSum:uri1(a,y);
        matrixCardinality:uri1(a,z);
    }

//The inductive case, where we sum together the distributed
//parts giving the sum of all the columns in the matrix and
//number of columns.
    foldr
    {
        matrixSumToVector:uri1(y,b,b);
        integerSum:uri1(z,i,i);
    }

//Dived the column vector b by the number of columns i
//to get the average.
    matrixDivide:uri1(b,i,b);
}
```

This function starts off by stating any abbreviations that are going to be used for the URIs that appear in the script. This is done simply to improve readability. Next the input references are declared, followed by the construction of the temporary values required to implement this function. Each variable is constructed from an existing variable to allow the construction of the appropriate number of pieces for distributed variables, and to provide information on where these variables should be created to save unnecessary movement of data. Having constructed all the required variables first y_i and z_i are computed for each piece of `a` as was done in the example on page 3. This is done by the two statements in the `map` environment. The contents of the map environment will be run independently and in parallel on each piece of any distributed data structure referenced within the environment. When all the y_is and z_is have been computed, these results then need to be merged together. This reduction is performed by the *foldr* environment. The statements contained within this environment merge the results into local data structures which act as accumulator parameters. Having calculated the sum of all the y_is and z_is the final step is to divide through the sum of y with the sum of z. The important thing to note about this function is that, although this function works on partitioned data structures, at no point is a reference made to how the data structure is partitioned, and so the user therefore remains unaware of the partitioning and the function can be reused on differently partitioned datasets without having to make changes to the code.

4 Implementation

The middleware is constructed using a service oriented architecture. This is built on top of Apache Axis [12] running on Jakarta Tomcat [13]. This choice was made

because currently this is one of the best supported, widely used and functional SOAP containers and it is the corner stone of the middleware produced by the OMII[14]. This means that, while we have a very stable platform to work from, we are not segregating ourselves from other work that is going on in this field. In due course we expected that we will migrate to the OMII middleware, in order to gain use of all the additional functionality supported by the OMII stack such as WS-Security [21], but at the start of this project there were no OMII releases available. The middleware itself is designed using an architecture that is similar to MONET [16], taking on board ideas put forward by WS-GAF [19] and WS-RF [18] on the use of handles and URIs to manage resources. It is not however constructed with a strict adherence to any one set of ideas or principles.

The middleware is abstracted so the different conceptual parts of the system are separated into different concrete parts. The structure of the communication between these parts and the use of XML documents posted by each node on the web means that nodes can be added and removed at will, without having to inform the whole system of the change.

Data is passed by reference, which has two main advantages. First, the data is only moved when it is required, allowing the references to be passed through many services without having to move very large amounts of data through with them. Second, the use of references means that data transfer can be instigated by a more efficient method than SOAP [11]. As a result the data transfer model is abstracted so that the protocol used to transfer the data is separate from the implementation of the rest of the middleware, with lazy evaluation [22] being used to minimise the data transfer overhead.

In addition to the use of references to pass data, references are used to pass functions. This allows functions to be first class values that can be passed into and used in other functions. This applies both to functions that have been submitted to the middleware using the Martlet language and to the base functions that are deployed to the system by administrators.

The middleware also has a comprehensive locking, provenance and data transfer model that allows multiple functions to run at the same time, and functions to be queued without having to worry about forgetting data or generating race conditions. The system that performs locking is extended to keep records so that a provenance system is built into the middleware so keeping a complete record of the life of each piece of data.

4.1 Topology

The implementation is divided into three logical units: data stores, data processors and process coordinators.

Data Stores provide a set of methods for accessing the data stored at a given location. This unit is deliberately lightweight and only capable of generating a data structure from stored data.

Data Processors ingest, store and run Martlet abstract syntax trees on datasets, which they either have locally or retrieve from another data processor or a data store.

Process Coordinators are the components that users interact with. They handle access to the rest of the middleware. This is consequently where most of the complexity of the project occurs, since this is where the generic trees that represent submitted functions are adjusted to fit the arguments on which the function has been called and then are broken up and scheduled across the data processors. This is the only component to have any knowledge of other nodes in the system.

All three components share common functionality for transferring, discarding, and publishing data. They can be grouped together at will on servers, so it is possible to have both a data processor and a data store on the same machine. A possible configuration of five servers is shown in Figure 1. It is worth noting that many different process coordinators can use each data processor and data store concurrently and each process coordinator does not need to be aware of all the other nodes.

This topology has similarities with the architecture of the Monet project [16], with the process coordinator mapping to the Monet *broker* and the data store mapping to the Monet *mathematical object store*. Although elements of the architecture are similar, this work serves an entirely different purpose since Monet was aimed at automatically combining services for mathematical functions, not at the abstraction of parallelisation from analysis of distributed data.

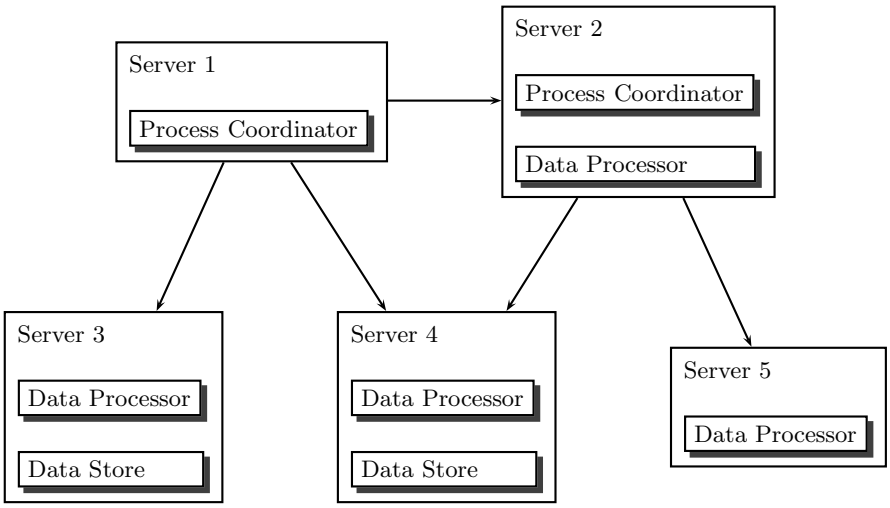

Fig. 1. An example of how five servers could be configured. Note that more than one Process Coordinator can use each Data Store and Data Processor and the Process Coordinator does not need to know about all available Data Processors.

4.2 Shared Features

Before looking at the implementation of the logical units, we will examine some of the design issues relating to the implementation of the shared functionality.

Data Structures There are two classes of data structure supported by the middleware, 'local' and 'distributed'. Local data structures are any data structure that always resides on a single server. Distributed data structures are abstractly a list of local data structures. So for example a distributed matrix is a list of matrices, which, if concatenated together in the order they appear in the list, would produce a single matrix equivalent to the distributed matrix. This means that, when a new data structure is added, its distributed counterpart is automatically generated. The encoding in the handles used for the list is an array of reference URIs.

Resource Lookup All data and functions are held in handles stored in a lookup table indexed by URIs, which are created when handles are added. The use of handles facilitates the storage of meta-data about the data structure or function to be seamlessly passed around with the data structure or function. This meta-data consists of both object specific information and implementation specific information. Examples of this data include, the number of processes currently using a data structure, if a data structure is locked to prevent race conditions and when the data structure's existence can no longer be guaranteed.[3] The inclusion of resource lifetimes is similar to the ideas in the WS-RF [18] specification where resources are given a updatable life time. This means that if a data structure is lost because of either the failure of a server or the forgetfulness of the user, the resources used will be recovered automatically.

URIs A range of information is encoded within the URIs used for referencing handles in the lookup table. This information includes the URL of the holding server, a unique identifier within the domain of the holding server, the type of the item referenced and the group it belongs to. Through the combination of the first two elements, all URIs are guaranteed to be globally unique. The encoding of the URL into the URI also allows servers to dynamically discover other servers as and when required, thus reducing the amount of state required in the system.

We chose to encode this much data into the URI because the desired system architecture required this information to be provided. The only other sensible alternative currently available would be the use of WS-Addressing [23], in which this information is encoded into an XML document that is used instead of the URI. We felt that this added unnecessary overhead parsing the document. A suitable level of abstraction has been included so that this decision can be changed later for reasons such as interoperability.

[3] If there is space to continue storing a data structure there is no reason to actively destroy it. Instead this can be delayed until the garbage collector is invoked.

Security In order that the security model remains orthogonal to the work on the underlying design, we decided that the security should be implemented by having a security handler on every server. Messages have to pass through this handler before reaching the underlying framework. If the messages are allowed through then it is assumed that the request is legal and it will be executed. It is possible for the handler to check a range of issues including the source of the message and the permissions of the sender. This fits in well with the model that Apache Axis [12] uses for service construction, in which services are made from chains of handlers, with each handler responsible for a specific task.

If the URIs are changed so that the security handler lets a request through, the request will fail to map to a handle in the resource lookup, and will fail before it can do anything harmful. By doing this, the security can be abstracted away from the rest of the middleware while still being taken into account in the design. The group field in the URIs was added to enable users and sets of handles to be grouped, thereby simplifying the administration of the security model.

Sending Messages The communication of messages to other components of the middleware is handled by a set of classes, one for each different type of component. This abstracts the details of communication away from the workings of the middleware to a single point. Coupled with the use of Axis handlers, this allows for the handling of errors thus providing a reliable means of communication that can automatically handle a range of errors. Unrecoverable errors will propagate through for the calling infrastructure to handle. These will however, either have been thrown by the receiving service, or because the communication remains impossible after the communication class has tried all available options.

Data Transfer Data transfer is initiated by a service provider that requires data, making an asynchronous request to the provider that holds the data. This request contains the URI of the required data and the location where it would like the data to be placed. The data is encoded into a file and transferred before an asynchronous call is made that informs the requesting service provider that it has arrived. This model was chosen as SOAP is not a good format for sending large datasets due to its verbose nature [24]. Currently the data is transferred by SFTP. This choice was made for convenience and is deliberately abstracted so it can be changed to a different protocol at a later stage if desired.

Since the receiving server initiates the data transfer, the overall model for transferring the data is a pull model, though the underlying protocol to transfer the data uses a push model, in which the sending processor initiates the transfer. This is done because it reduces the amount of inter-node signalling that is required to organise an effective cleanup of transient data structures. The potentially huge sizes of the datasets makes it necessary for transfers to be asynchronous to avoid timeout issues occurring while the nodes are marshaling the data. Figure 2 shows a generic outline of the sequence of events for data transfer.[4]

[4] The function call is not restricted to being from a process coordinator.

Each data type requires an encoder and decoder, which can be made by extending a class which provides the base functionality. So for example matrices are encoded using the netCDF [25] file format. The choice to use netCDF for the matrix transfer was made because this is one of the formats that is going to be supported by the NERC DataGrid project [26,27], it is the native format of the returned climate data and it is supported by libraries in a wide range of languages.

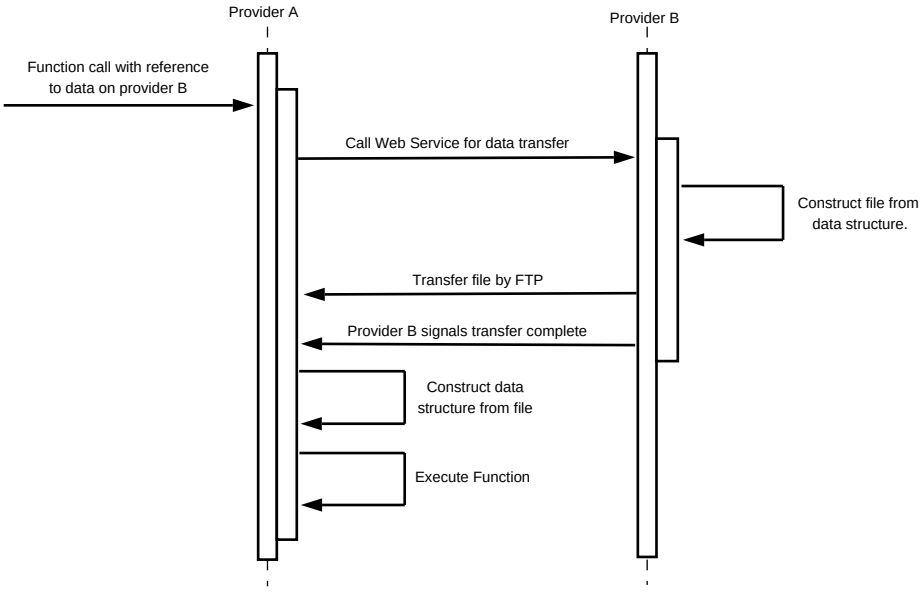

Fig. 2. Process diagram of non-local reference retrieval.

In order to save on communication times data transfer occurs if, and only if, a node is handed a reference to a handle on a different server and it tries to read the data from it. This is instigated by the server's recognition that it is not a local URI, so creating a lazy [22] handle. This lazy handle has all the information it requires to transfer the data, but only executes this operation if `get` is called on the handle to retrieve the data it holds. If a `set` call is made, the lazy handle will remove the old copy of the data from the remote server, making the local version the only copy. Otherwise the local copy is removed when the function terminates in order to ensure that only one copy of the data exists at any time. The decisions about which copy of the data to remove were made in order to keep the amount of network traffic to a minimum.

If copied data is unmodified, the copy of the data will be deleted after the server has finished with it, so the user URI will remain unchanged. However, if data is modified the original is deleted leaving only the copy. The user is informed of the URI of this new copy when the modifying function terminates.

The original URI will no longer point to anything. As such it is not safe to share the URI of mutable data with other users, as there is no guarantee that it will not be changed leaving no resource referenced by the shared URI. This restriction can however, be avoided by the taking a copy of the data to be shared and continuing any further work with the original.

Resource Locking As the environment will require multiple threads in order to provide acceptable performance, data locking is required to prevent race conditions and ensure only valid answers are produced. One solution would be to make all data immutable after creation. However since this would result in large time and space complexities it is not a practical solution. The chosen design principle was the addition of information to the handles that can only be altered by a single monitor. This enables large numbers of handles to be locked atomically, thereby protecting against internal deadlock. Deadlock during internode communication is a problem since atomic locking across all nodes is neither practical nor possible. To prevent deadlock occurring it is necessary to restrict the user in such a way that if a function is going to write to a reference, while that function is being executed, all other functions that use the same reference must be submitted through the same process coordinator.

The monitor is constructed using a static synchronised method to lock an array of handles as an atomic action, else return the index of the handle in the array that was unlockable. Each handle has a method that the monitor can call in order to wait on until the handle becomes free, at which point the monitor can try to get a lock again. Each lock can be either *unlocked*, *read only*, or *write*. The type of lock that each handle requires to perform a given action is read in from the called function before any attempt is made to get a lock. This is abstracted away from the programmer by an abstract class called a "function object", which is extended by all objects that read or write to data structures. This class contains a boolean array stating which arguments are read only and which are write, along with four methods for controlling access to the arguments; *ready()*, *finished()*, *invoke()* and *run()*. These methods perform the following tasks:

Ready returns with the service provider in a state where all handles held by the object are locked by the object making it safe to proceed with the functions execution.

Finished unlocks all the handles, notifying any function objects waiting on them that they are now free. It then signals any processes listening for the completion of the function before returning.

Invoke is the abstract method that must be extended with the function a class is going to perform.

Run spawns a new thread, therein providing the required asynchronous nature. This method calls *ready()*, *invoke()* and finally *finish()*.

Additionally, handles hold locking information to control when the data they hold can be copied safely. Since data transfer only reads from a single handle,

it is sufficient to make each handle a monitor for its own lock. The necessity for the second lock is created by the scenario where a composite function calls a function on a different node, passing references to local data. The data is not in a final state at this point, so it must not be used by other functions, but the data transfer request must not be locked out. Making data requests ignore locking would also be dangerous since a different part of the function may be altering the data at the time the transfer request arrives.

Provenance Provenance [28] is one of the pieces of meta-data held in handles. It is of importance to this project since the reproducibility of results is vital to any scientific work and, with the ability to publish URIs online, it is not necessarily sufficient for the user to record what he has done. Moreover, it is possible for such a system to record information about any operations performed on the data with greater detail, accuracy and fidelity than a user could manage.

The provenance data is stored as an acyclic directed graph, in which each node represents a function call, and the vertices from the node lead to the nodes that represent the inputs to the function. Each handle holds the node that represents the last operation to happen to its data. As such, each handle starts with a node listing the dataset it was constructed from. When an operation is performed on this handle the node is replaced with an operation node that has the old node as one of its children. From this graph it is then possible to construct a list of operations that have been applied to the queried data structure to transform it into its current state. The next node is constructed by the functions in the abstract class function object. Through the adjustment of this class, it is possible to include the storage of a wide range of information, such as execution time, middleware overhead, the server the execution occurred on, the date, and so on.

Publishing All nodes have a *publish* method which takes a reference to a data structure, and returns the URL of a web page where it has been published. For local data structures this done by encoding the data structure into a file and placing it into the directory structure of a web server along with a corresponding web page. For distributed data structures, the publish call is propagated to all the nodes holding parts of the structure. The returned URLs are then added to a web page constructed on the server holding the distributed handle and the URL of this page returned to the user.

4.3 Data Store

The Data Store is the logical unit that retrieves data from the stored datasets and manipulates it into a data structure for analysis. It adds just one method to the base functionality. This method takes a dataset descriptor and a range descriptor. It uses these to retrieve the data described by the range descriptor from the dataset described by the dataset descriptor. From this data it constructs a data structure and places it in a handle, returning its URI reference. For

example the range descriptor may state that "only the temperature of Europe is required" and the dataset descriptor may state that "this should only be taken from returned models that had pressure at mean sea level set to 1013mb when the model was started".

The process coordinator constructs the range descriptor from a description submitted by the user when requesting the construction of the data structure. The dataset descriptor, on the other hand must be constructed earlier by a separate call to the process coordinator. This is required because the global dataset is constantly growing as more model runs are returned and, if two subsets are going to be compared it will be necessary for them to be taken from the same dataset i.e. data structure 1 must be constructed from the same returned models as data structure 2. By the use of the dataset descriptor a fixed point in time is created from which comparable data structures can be created. When the dataset descriptor is created from the database of returned runs, it not only contains the returned results from which the data structure is going to be built, but also the servers these model runs are stored on. As a result, the process coordinator needs only to know the location of the database for the project, not the location of every data store used by the project.

4.4 Data Processor

Data processors perform analysis on the data structures generated by data stores. To do this they store a set of "base functions", which have been approved for use. Base functions are constructed by wrapping a function through the extension of the abstract class function object and building a factory to construct these wrapped functions through the extension of the class "function constructor". The use of wrapping like this is important as it allows for the integration of legacy code into the set of base functions. The information about these functions is added to an XML document and placed on a supporting website so that any process coordinator that wants to use the data processor is aware of the supported functions. The decision to host the XML locally and not use advertisements elsewhere was made since the additional level of indirection is not required and restrictions on the structure of the adverts would restrict the design if an inappropriate choice were made. However a sufficient level of abstraction has been maintained so that this decision can be changed later if desired.

The data processor has three additional core methods that extend the base functionality:

Function Invocation takes a reference for the function and an array of references for arguments. It retrieves the handle containing the function constructor and uses it to construct an object that will perform the function on the arguments provided. Creating function constructors as factories for function objects simplifies the management of multiple concurrent calls to the same function, leaving the function object to deal with locking the handles and executing the individual request.

Function Submission allows for the submission of new functions described with a Martlet abstract syntax tree. When submitted, the abstract syntax tree is parsed into a tree of function constructors that can be placed in a handle and used to construct function objects. This method does not affect the published XML documents, since the submitted methods are not base functions but parts of a function currently being executed by a process coordinator. As such they are not guaranteed to be self-contained, have a very short lifetime and are not appropriate for use in other functions.

Creating New Handles is a simple method that creates empty handles of a given type for storing the output of computations. It returns a corresponding URI after creating the handle.

4.5 Process Coordinator

This is the part of the system with which the user will interact. It coordinates requests across all other parts of the system, deals with locating servers, adjusting functions to match the splitting of data and scheduling jobs to compute the adjusted functions.

Locating Servers As a process coordinator is not simply provided with a list of servers, it is necessary to locate servers. This is performed in two separate ways.

The simpler of these is the method used to locate data stores. This is achieved by giving the process coordinator the address of a web service that provides an interface to a database storing the location of the returned runs. Through querying this web service, it is possible to get a list of all the servers that are required to supply data for a given computation, along with a description of what data they must supply.

The second method is the method used to locate data processors and keep track of what services they provide. This is done by maintaining a set of URLs of active data processors. For each URL the process coordinator periodically queries the XML document published by the data processor to see what functions it supports. From these documents, the process coordinator is able to maintain for each base function a list of servers that support it. Each of these lists is referenced by a URI to enable users to use the base function in Martlet programs. When such a program is run, the URIs will be substituted with URIs from the lists they reference. A diagram showing the linking for three servers supporting two functions and their corresponding handles is shown in Figure 3.

If a URI is used that references a different process coordinator, then the referenced process coordinator can be queried about its set of data processors, thereby increasing the number of data processors each process coordinator can use.

If a server is removed in a controlled way, it will empty its XML document of functions a predefined time before it goes offline, allowing process coordinators to update their records. If, on the other hand, it fails, this will be detected and

reported when attempting to invoke functions on it, triggering its removal and the rescheduling of jobs.

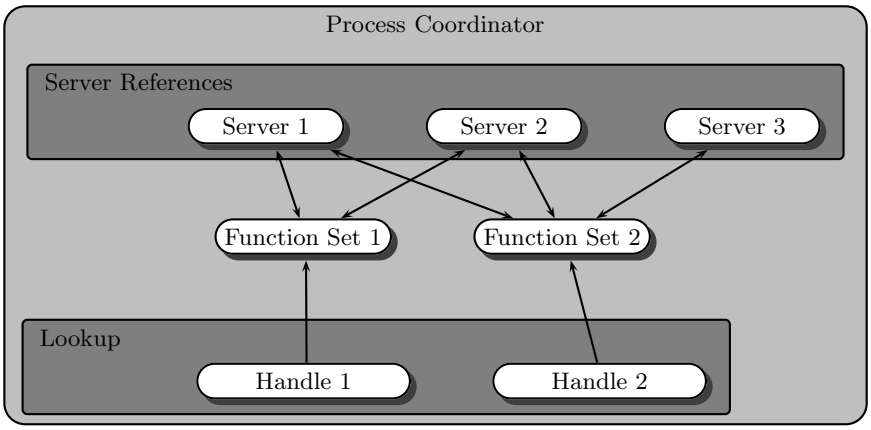

Fig. 3. The objects and linking for three data processors, which collectively support two functions. Note that Sever 3 only supports one function.

Program Submission and Execution When Martlet programs are submitted they are converted into abstract syntax trees and stored, since at this point there is insufficient information to do anything more. When they are executed with a full set of arguments, however, all the required information is present, allowing the following chain of events to take place:

1. The handles of the arguments are locked and a copy of the tree made to protect the structure of the original and leave it available for other invocations.
2. The abstract syntax trees of any function calls to other Martlet programs are retrieved and recursively integrated into the tree, so that the only remaining function calls are calls to base functions.
3. By looking at the distributed data structures, the tree is expanded to remove the functional constructs, replacing them with a set of procedural constructs that represents their function over the given distributed data structures.
4. Since the full tree is now present, the base functions are scheduled to run on specific data servers chosen from the function sets.
5. With the tree now full quantified, it is automatically broken into pieces and distributed to the appropriate data processors.
6. The root node is invoked. Once the execution completes, the subtrees are disposed of, leaving just the returned results.

To ensure that deadlock does not occur and to guarantee simultaneously that the results are correct, it is necessary for all execution requests, which write to a given data structure concurrently to be submitted to the same process coordinator. For example, suppose there is a process `f` that adds the value in its first argument to its second argument, and the call `f(a,b);` is made to process coordinator `x`, then other requests using `b` can be made. However, until the `f` completes, these requests are bound to being submitted to `x` as well, even if it only reads `b`.

Scheduling As the development of an advanced scheduler was not one of the aims of this project and the requirements of a scheduler may change dramatically according to the environment in which it is operating, the current scheduler is very rudimentary. To facilitate the construction and use of custom schedulers, an interface is provided for schedulers to implement in order to interact with the process coordinator. In addition to this a "visitor" interface is provided allowing the construction of custom visitors for traversing the expanded abstract syntax tree. As the tree is capable of handling an arbitrary number of visitors and the visitor controls its movement through the tree a huge degree of freedom is afforded to schedulers wishing to analyse and manipulate these trees.

5 Further Work

The project so far has produced a middleware running to over 10,000 lines of Java code spread across 160 classes supporting the Martlet language allowing for abstracted parallel functions to be implemented. All three components of the infrastructure having been deployed and tested on the data generated by Climate*prediction*.net and we are currently working towards a position where this can be deployed across all our servers. This will provide us with a dataset measured in terabytes on which to examine the scalability of this architecture. These experiments so far have highlighted areas in which there is room for this prototype to be improved. So we shall now discuss where future work in this project may take us.

5.1 Just In Time Compliers

As stated in Subsection 4.5, once the abstract syntax tree is expanded, it no longer contains any functional constructs. In fact it no longer contains any constructs that are not supported by a wide range of existing workflow languages running on a wide range of existing middlewares and benefiting from higher levels of support and much wider user bases. With this in mind, the best way to increase the availability of the functionality provided by this project would be to construct JIT compilers allowing the process coordinator to sit on top of existing grid engines. Then, when a function is executed, it would expand the tree, then instead of sending it to the scheduler and distributing it amongst the data processors, it would convert the tree into the required workflow language

and submit it to the underlying grid engine. Due to the varying nature of the different grid engines such work would need to be done on a case-by-case bases, but the abstraction provided through the "function objects" and "function constructors" should make this a fairly trivial task.

5.2 Class of Algorithms

While it is possible to describe a wide range of algorithms using Martlet, including algorithms for calculating the leading p vectors of a distributed singular value decomposition [29], it is not currently possible to describe algorithms that are required to share data between parts of a distributed data structure before the reduction step. For instance a naive QR factorisation [29] on a distributed matrix that is too big to be placed on a single computer would not be possible. As the size of the data structure is too big to store on a single computer such operations may never be required, since operations that go on to use such data would probably be too closely coupled to be of use in the distributed environment at which this work is aimed. It is not safe however, simply to say that such operations will never be required over the distributed datasets described here. And it will be important to observe and consider the possibility of adding additional constructs to Martlet and the supporting middleware in order to loosen the constraints on the user. Such constructs may include statements that iterate $n-1$ times for a data structure split into n pieces, and constructs that allow the interaction of local data structures with their adjacent data structures within a distributed data structure.

5.3 Move to OMII

The OMII are working on integrating a range of grid middleware into a single package on which to build new services. They now have several successful stable releases of both their client and server packages providing a range of grid services and, though we originally decided to use Apache Axis and Jakarta Tomcat as a platform to construct this project, because we did not want to be affected by an immature platform or restricted by their design decisions. Now that our architecture is well defined and the OMII's platform is both stable and available it is a sensible time to migrate. In addition to being on a widely used grid platform, on which we can construct JIT compilers to extend the reach of this project, it would also allow us to take advantage of the additional functionality provided by OMII including WS-Security [21].

5.4 Type Checking

The current underlying infrastructure that the middleware is built on will detect type errors, so they will not harm the system as a whole. It cannot however, detect these at the time of submission. The addition of type checking on submitted functions could be implemented by augmenting the information provided about

base functions with information about the types of their arguments. It will be possible to check from this that the types of the arguments for each function whether primitive or composite, are correct.

6 Conclusions

We have introduced in this paper a class of problems that falls under the definition of Grid computing, but fails to be addressed by the current Grid computing projects. We have then gone on to show how these problems can be addressed by slightly lowering the level of abstraction that the underlying services have from the middleware, and by creating two classes of data structures, distributed and local, which enable the middleware to reason about the handled data. It is then possible to produce a middleware that allows the user to have a higher level of abstraction from the problems faced when working with distributed algorithms over a dataset, such as that of Climate*prediction*.net. This middleware has been produced in keeping with many of the principles of existing projects, but takes advantage of a slightly more restrictive, higher level of abstraction, provided through the use of the work-flow language Martlet, which was inspired by functional programming.

The middleware draws upon ideas put forward by a wide range of distributed computing projects, including MONET [16], WS-GAF [19] and WS-RF [18]. While none of these addresses the specific problem addressed here, they provided insight into and solutions for many of the sub-problems that required solving in order to implement such a middleware. In addition, it is built on top of widely used components [12,13] used in other grid solutions such as the OMII [14], placing it in a strong position to complement these existing packages in providing a more rounded set of tools for grid computing. The performance of functions running on this framework is dependant to a large degree on the function in question. Appropriately rough grained functions should suffer a relatively small overhead compared to the time taken to process such large datasets.

For the ideas presented here to be taken up on wider scale it will to be necessary to become interoperable with other grid engines. With a move to a the OMII platform and the construction of JIT compilers we hope to facilitate this wider uptake. Though this model is not appropriate to all grid problems, it is a good solution to a thus far under-visited part of the grid picture that has to be addresses to produce a complete grid solution. It is capable of being used on a large number of more traditional grid problems and it is hoped that the ideas and principles presented in the Martlet language will be absorbed into the next generation of workflow languages. While this potentially means that this prototype language and middleware themselves are not widely used, the knowledge gained through its construction should prove invaluable and will need to be absorbed into the middleware constructed to support this next generation of languages, if they are to address this type of grid problem.

References

1. Gagliardi, F., Jones, B., Reale, M., Burke, S.: European datagrid project: Experiences of deploying a large scale testbed for e-science applications. In: Performance Evaluation of Complex Systems: Techniques and Tools, Performance 2002, Tutorial Lectures, Springer-Verlag (2002) 480–500
2. EGEE: (EGEE: Enabling Grids for E-science in Europe) URL: http://egee-intranet.web.cern.ch/egee-intranet/gateway.html.
3. National Science Foundation: (US TeraGrid) URL: http://www.teragrid.org.
4. Andrews, T., Curbera, F., Doholakia, H., Goland, Y., Kiein, J., Leymann, F., Liu, K., Roller, D., Smitth, D., Thatte, S., Trickovic, I., Weerwarana, S.: BPEL4WS. Technical report, BEA Systems, IBM, Microsoft, SAP AG and Siebel Systems (2003)
5. Deelman, E., Blythe, J., Gil, Y., Kesselman, C.: Pegasus: Planning for execution in grids. Technical report, Information Sciences Institute (2002)
6. Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M.R., Wipat, A., Li, P.: Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics **20**(17) (2004) 3045–3054
7. Goodman, D.: Martlet; a scientific work-flow language for abstracted parallelisation. Technical Report RR-05-06, Oxford University Computing Lab (2005)
8. Goodman, D., Martin, A.: Grid style web services for climate*prediction*.net. In Newhouse, S., Parastatidis, S., eds.: GGF workshop on building Service-Based Grids, Global Grid Forum (2004)
9. Stainforth, D., Kettleborough, J., Martin, A., Simpson, A., Gillis, R., Akkas, A., Gault, R., Collins, M., Gavaghan, D., Allen, M.: Climate*prediction*.net: Design principles for public-resource modeling research. In: 14th IASTED International Conference Parallel and Distributed Computing and Systems. (2002)
10. Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: Seti@home: an experiment in public-resource computing. Commun. ACM **45**(11) (2002) 56–61
11. W3C: Simple Object Access Protocol (SOAP) 1.1. (2000) URL: http://www.w3c.org/TR/SOAP.
12. Apache Software Foundation: Apache Axis. (2005) URL: http://ws.apache.org/axis/.
13. Apache Software Foundation: The Apache Jakarta Project. (2005) URL: http://jakarta.apache.org/tomcat/.
14. OMII: The omii product roadmap. Technical report, OMII (2004) URL: http://www.omii.ac.uk/roadmap.htm.
15. Foster, I., Kesselman, C.: Globus: A metacomputing infrastructure toolkit. IJSA **11**(2) (1997) 115–128
16. Consortium, T.M.: Monet architecture overview. Technical report, The MONET Consortium (2003) URL: http://monet.nag.co.uk/cocoon/monet/.
17. Blower, J., Haines, K., Llewellin, E.: Data streaming, workflow and firewall-friendly Grid Services. In Cox, S.J., Walker, D.W., eds.: Proceedings of the UK e-Scienece All Hands Meeting 2005, EPSRC, EPSRC (2005) 858–865
18. Czajkowski, K., Ferguson, D.F., Foster, I., Frey, J., Graham, S., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W.: The WS resource framework. Technical report, Computer Associates International Inc and Fujitsu Limited, HP and IBM and The University of Chicago (2004)
19. Parastatidis, S., Webber, J., Watson, P., Rischbeck, T.: A Grid application framework based on Web Services specifications and practices. Technical report, North East Regional e-Science Centre (2003)

20. Bird, R.: Introduction to Functional Programming using Haskell. Second edn. Prentice Hall (1998)
21. Atkinson, B., Della-Libera, G., Hada, S., Hondo, M., Hallam-Baker, P., Klein, J., LaMacchia, B., Leach, P., Manferdelli, J., Maruyama, H., Nadalin, A., Nagaratnam, N., Prafullchandra, H., Shewchuk, J., Simon, D.: Web services security (ws-security). Technical report, W3C (2002) URL: http://www-106.ibm.com/developerworks/webservices/library/ws-secure/.
22. Henderson, P., James H. Morris, J.: A lazy evaluator. In: POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages, New York, NY, USA, ACM Press (1976) 95–103
23. Box, D., Christensen, E., Curbera, F., Ferguson, D., Frey, J., Hadley, M., Kaler, C., Langworthy, D., Leymann, F., Lovering, B., Lucco, S., Millet, S., Mukhi, N., Nottingham, M., Orchard, D., Shewchuk, J., Sindambiwe, E., Storey, T., Weerawarana, S., Winkler, S.: Web services addressing (ws-addressing). Technical report, W3C (2004)
24. Chiu, K., Govindaraju, M., Bramley, R.: Investigating the limits of soap performance for scientific computing. In: HPDC. (2002) 246–254
25. Rew, R., Davis, G., Emmerson, S., Davies, H.: NetCDF User's Guide, UCAR/Unidata Program Center P.O. Box 3000 Boulder, Colorado, USA 80307. 2.4 edn. (1996)
26. Lawrence, B., Boyd, D., van Dam, K.K., Lowry, R., Williams, D., Fiorino, M., Drach, B.: The NERC DataGrid. Technical report, CCLRC - Rutherford Appleton Laboratory (2002)
27. Lawrence, B., Cramer, R., Gutierrez, M., van Dam, K.K., Kondapalli, S., Latham, S., Lowry, R., O'Neill, K., Woolf, A.: The NERC DataGrid prototype. Technical report, CCLRC e-Science Centre, British Atmospheric Data Centre and British Oceanographic Data Centre (2003)
28. Buneman, P., Khanna, S., Tan, W.C.: Why and where: A characterization of data provenance. In: ICDT. (2001) 316–330
29. Golub, G.H., Loan, C.F.V.: Matrix Computations. 3 edn. The Johns Hopkins University Press (1996)