Managing large volumes of distributed scientific data.

Steven Johnston[a], Hans Fangohr[a], and Simon J. Cox,[a]

[a]Southampton Regional e-Science Centre,
University of Southampton, United Kingdom. Email: sjj698@zepler.org
{h.fangohr,s.j.cox}@soton.ac.uk

Abstract. The ability to store large volumes of data is increasing faster than processing power. Some existing data management methods often result in data loss, inaccessibility or repetition of scientific simulations. We propose a framework which promotes collaboration and simplifies data management. We propose an implementation independent framework to promote collaboration and data management across a distributed environment. The framework features are demonstrated using a .NET Framework implementation called the Storage and Processing Framework.
Key words: File Object Model, Data management, database

1 Introduction

It is generally accepted that processing power doubles every 18 – 24 months and data storage density every 12 – 18 months [1]. The result is that the cost per Gibibyte (GiB) to the end user is falling which indicates that the ratio of processing power to data storage will decrease. This opens up new opportunities to consider more efficient methods of data management.

Databases for example [2], provide a low level data management repository capable of storing and manipulating data. They are often considered too rigid for dynamic data and too complex for non-technical users.

Many users utilise proprietary or custom applications as an alternative to generic databases [3, 4]. These are usually designed for a particular task and tend to be more user friendly but often less flexible. For example the BioSimGrid project [5, 6] is establishing a worldwide repository for simulation results using Grid [7] technologies to distribute and manage the large volumes of data.

Filesystems are ideal for managing data but support for storing and searching the metadata is minimal [8].

Users prefer to keep data in a format with which they are familiar. File systems are capable of dealing with large volumes of data and databases are suitable for managing metadata about the data. There is a need to leverage the benefits of both advanced databases and robust filesystems to provide a simple and easy to use data management solution for scientific users.

Based on these challenges, we propose a method to assist with the collaboration and management of data, particularly scientific data. The proposed method

aims to transparently bring advanced database features to the inexperienced database user.

We propose a method of associating code with data files by treating them as objects. This provides users with additional functions or operations on a file, without having to customise code to process the file. We fully describe the proposed method and demonstrate a specific implementation of the proposal.

2 File Object Method

Most operating systems are capable of associating an application to a specific data file type, *e.g.* files with the extension `txt` are often associated with a text editor. This enables the user to open the file by 'double-clicking' an icon and relying on the operating system to open the appropriate application, capable of reading the file.

This mechanism is achieved either by associating the file extension, `txt` with an application or by associating the Multipurpose Internet Mail Extensions (MIME) [9] type with an application. Alternatively it is possible to inspect the content of the file to determine its type using a file signature [10].

When a user loads an application associated with a data file, the application can be thought of as a library of functions or methods which are capable of operating on that file. For example a `txt` file is opened using an editor which has features to count the words or change the data encoding.

We propose an infrastructure to extend this common functionality to support custom 'applications', in the form of user defined code. Instead of a data file having an associated application we propose that the data file has associated functions and methods which originate from user code.

We propose treating files like programming objects in an infrastructure called the File Object Method (FOM). The FOM associates code with files providing users with the ability to execute these routines as methods on file objects. The aim of the FOM is to extend the usefulness of flatfiles using object oriented programming techniques. This can then be used by hybrid database systems, as well as by users who manage data using flatfiles.

Extending files to appear as objects ensures users execute the correct methods on the correct file types, thus removing the responsibility for users to ensure they have the correct file format.

There are two ways with which a user can associate methods to files in the FOM. The first is to simply associate code with a single file *i.e.* its path and filename. The second is to associate a `type` with each method or set of methods, and then associate this with file types. This means that users can deposit a file into the FOM, and without any intervention be able to list and execute methods that are associated with that file type, using the data they have just deposited. In the text file example, when a user adds a new text file to the file system they will be able to see that there is a method to count the words in that file.

3 Implementation

We demonstrate the FOM using a prototype, called the Storage and Processing Framework (SPF) which demonstrates all the FOM features in a secure and distributed environment.

The SPF is implemented using the .NET framework and is based on web services. The underlying infrastructure supports a secure and distributed file system upon which we demonstrate the FOM features using two examples. The examples are written in different .NET languages and used to demonstrate the multi-language support of the SPF.

The SPF has three key components: the storage service, the storage manager and the client layer.

The storage service manages the data, mapping it to a physical resource. The SPF can have many storage services each controlling a single resource. All the storage services are controlled by a single storage manager. The storage manager is the point of entry for the client layer which exposes all the SPF features to the end users. As each storage service has to register with the storage manager, the user layer can locate any data in the SPF.

3.1 Storage layer

The key objective of the storage layer is to provide a mechanism for accessing files on a given machine via a web service. By implementing the storage layer it is possible to show how this FOM model can be used to perform calculations in a distributed environment. This storage layer is not intended to be a substitute for a distributed file system and is merely a testing platform for the SPF.

The storage layer is responsible for making the files transparently accessible to the client layer, regardless of location. The storage layer is built up with many storage services and a single storage manager, both of which are web services.

3.2 Storage service

Each machine has one storage service which manages the data stored on that resource. The storage service is responsible for taking files and storing them on the storage space provided by a resource. The service then responds to requests for files and information about files. The storage service instance has two key components: i) the storage API and ii) the DLL manager.

The storage API maps the SPF file requests to local files and is responsible for invoking the DLL manager. When a file is deposited, it is stored in the local file system and the name and file type are registered with the local SQL database.

The users .NET code is compiled into an assembly called a Dynamically Linked Library (DLL). The DLL manager stores all the user's code and maps it to files and file types. When a file is selected, either programmatically or through the user interface, the DLL manager provides information about the user's code associated with that file and what methods are available. When a user executes

a method the DLL manager locates the code and executes the constructor using the local file as a parameter.

The DLL manager caches the results returned by an SPF method. These results are stored in the DLL SQL database and are used to speed up SPF execution. The DLL cache stores the last modified time of all SPF files. When an SPF method is invoked the timestamps are compared. If an SPF file has been altered the SPF cache is flushed.

3.3 Storage manager

Each storage service has to register itself with the storage manager which is responsible for receiving file requests from the client layer and returning a list of storage web services that store the requested data file.

There are many storage services: one per machine and one storage manager in the SPF. The FOM architecture can support more than one storage manager to allow users to have more than a single point of entry. This can assist with load balancing although the SPF implementation utilises a single storage manager.

The storage manager's function is to provide a point of entry for the client service. When a client requests a file it first asks the storage manager to return the file providing the location of the client web service which requires the file. The storage manager is a lightweight index of the files stored in the SPF. If a requested file is unknown the storage manager polls all the known storage services requesting the file. The location of files are cached in the storage manager and the client layer.

3.4 File objects

The storage layer identifies files using a SPF Uniform Resource Indicator (URI). Each storage service has its own root folder where all the SPF data files are located. The SPF URI is relative to the root folder. Thus each file does not need to have the same location on each storage service.

The client layer treats a replicated file as a single file object. When a user requests data from the file, the most appropriate storage service is selected. This is based on Central Processing Unit (CPU) utilisation but can be substituted for other machine parameters *e.g.* memory or storage requirements. If the file replications are not synchronised, the client layer flags the file as dirty. If the user chooses to ignore this then the file with the most recent time stamp is used. When the client layer marks a file as dirty it notifies all the out-of-date replicas of the storage service where the most recent data is stored. The replica storage services then synchronises the data files.

The client layer can query all known storage services for a list of files and directories contained within a specific SPF folder.

3.5 Client layer

All the client features are accessible through the client Application Programming Interface (API). This interface provides a .NET library upon which all the client layers are built.

The client API must have a point of entry into the SPF. This is accomplished by supplying the location of one storage manager web service. Once this is established, the API can query the SPF for files, names, locations and associated user code. If the API does not know of the storage service where a data file is located, it first asks the storage manager to supply a list of valid storage services. The locations of files are cached in the API to speed up frequent access. The API can then query one of the storage services to retrieve information about the associated code.

The client API provides methods to retrieve data files as well as query the associated user code. This provides a list of associated user classes which can in turn provide a list of associated user methods.

Once the user's method has been selected the API can call the storage service to invoke the associated code on the selected SPF file. The results are then transferred back to the API layer where users are free to manipulate the data. The results of a user method invocation are returned as .NET objects which the users can then use programmatically in future code.

The Graphical User Interface (GUI) provides a visual representation of the underlying API capabilities and is intended as an example application of the SPF. All the features exposed in the GUI can be programmatically utilised by the user.

4 Example Scenario

To demonstrate the FOM capabilities we provide two examples.

A C# class which provides metadata about a specific file. The `AdvancedFileInfo` class takes a file name as a constructor parameter and provides three methods to return information about the selected file, using the following methods.

The `contact` method returns a string indicating whom the user can contact for further information about the class. The `getFullLocalPath` method returns the full path and file name of the selected file. This is useful in a distributed environment as the SPF file name masks the real location of the data files. The `getLastAccessTime` method returns a `DateTime` object showing when the file was last accessed.

The second example, written in Visual Basic, the `WordCount` class takes a filename as a constructor parameter and provides the following methods to return information about the selected file:

The `countWords` method returns the number of words in the selected text file. The `countLines` method returns the number of lines in the selected text file. The `countCharacters` method returns the number of characters in the selected text file

Both .NET classes are compiled into a .NET library (DLL) which are then imported into the SPF so they become available to all users.

4.1 Selecting an SPF method

The SPF windows GUI shown in figure 1 is divided into five regions. The lower region displays the properties of the currently selected object.

The upper four regions are used to find and select methods associated with data files. For example, to select the `GetLastAccessedTime` method on the `readme.txt` file, the user's four steps are shown in figure 1.

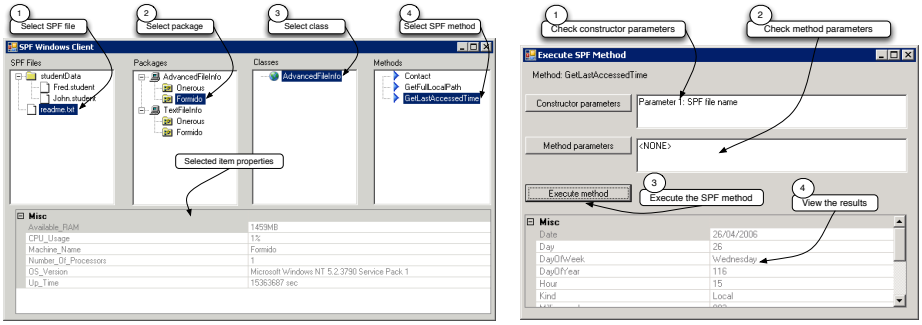

Fig. 1. The SPF client GUI (*Left*). The GUI to execute an SPF method *Right*.

Step 1: When the user interface loads, the client layer connects to a storage manager web service. The client then requests that all storage services return a list of all files and folders stored within the root SPF directory. These files are then collated and displayed in the *SPF file* tree view of the user interface. In this example the user has selected the `readme.txt` file.

Step 2: When a file is selected in the *SPF file* tree, the *packages* tree is then populated. This provides a list of all the packages (collections of classes) which are associated with the selected file. In this example the `readme.txt` file has two associated packages, the `AdvancedFileInfo` and the `TextInfo` package. Not all storage services have the capability to run all user code as the DLL may not exist on a particular resource. The client layer will automatically select a storage service to execute a method base on the machine's load. In the GUI a user can see which machines are capable of running particular user code. In this example the user has selected the `AdvancedFileInfo` package.

Step 3: When a package is selected in the *packages* tree, the *classes* tree is populated. This is a list of all the classes contained within the selected package. In this example there is just one class, `AdvancedFileInfo`.

Step 4: When a class is selected in the *classes* tree, the *methods* tree is populated. When the `AdvancedFileInfo` class is selected the three methods are displayed. The user can then select a method and execute it.

4.2 Executing an SPF method

Once a user has selected the SPF method to execute, it can then be invoked and the results returned. When a user selects a method in the *SPF Windows client* shown in figure 1 the *Execute SPF method* window appears (figure 1).

In the GUI interface, users can supply primitive parameters using the constructor parameters option. In this example the class only has one constructor which takes the file name, thus the default options are used.

The default behaviour of the SPF is to execute a method without any parameters. Users can provide parameters by selecting the method parameter option.

Once a user has set the optional constructor and method parameters it can be invoked using the execute method button. This will cause the storage service where the file is located to request the DLL Manager to create an instance of the class and invoke the selected method. The results from this method are then serialised and passed back to the client layer using web services. The client layer returns the results object to the calling interface, in this example the GUI.

The results of the method are returned as an object, displayed the properties of the object in the results view as show in figure 1. It is expected that a user will take the return object and utilise it in a calling application. The GUI is intended to demonstrate the capabilities of the underlying API.

5 File Object Method Features

All the storage services run on the host machine's file system and the structure of the files remain unchanged. Simulation and experimental data can be written directly into the storage service using the user's preferred method.

User codes must be compiled into a class library (DLL) which is common practice in the .NET framework.

When a class is selected the associated files and file types are displayed. To associate additional files with the currently selected class, users can add full SPF file names. Users can associate the selected class with any file of a particular type. In this example entering `*.txt` will associate the `AdvancedFileInfo` class with all text files.

Directories and files are treated the same in the SPF, thus users can associate code on a one-to-one basis with directories.

The example shown in section 4 demonstrates how users can discover methods using the client API. All the methods associated with a data file can be listed, queried and executed.

When an SPF method is executed the results are returned as .NET objects. These objects allow the user to programmatically integrate the SPF into existing applications.

Users cannot directly return SPF method results as file objects. If the SPF method creates a file when executed this will then appear in the same directory as the data file. This mechanism can be used to retrieve data from SPF methods as file objects.

5.1 Load balancing

The load balancing is managed in the API layer. When an SPF method is called the API looks at all the storage services and selects the one with the lowest CPU load. The rules which determine how a machine is selected are stored in a single class. Further SPF optimisations can include additional rules to manage the load balancing. For example, rules which take the available RAM, network speed and available storage can be added to the load balancing rules.

5.2 Security

All the SPF user accounts are created and managed by the Windows operating system and Active direcotry (AD). The SPF security is managed at the operating system level ensuring that the data and hosting machines remain robust against malicious users. When data is deposited it is marked as read-only for all users and read-write for the depositing user. Other users can be given permission to alter data files using the OS user permission settings. Users can change the file permissions to suit their task using the methods currently used.

All the code executed in the SPF framework runs as a restricted user. The SPF, by default only supports managed code. This reduces the users ability to execute malicious code on any of the SPF storage services.

Users can see all the methods available on any data file, the users code is restricted by the OS user account under which it is executed.

Users have the ability to turn on transport security features in the SPF. For example the Message Transmission Optimisation Mechanism (MTOM) data encryption can be used to encrypt data sent to the storage services. The .NET framework supports Simple Object Access Protocol (SOAP) extensions to encrypt web service data.

In addition the option to run reliable message transacted messages is available using the web service standards.

5.3 Method results cache

When an SPF method is executed the results are requested from a storage service. Each storage service is completely autonomous and is responsible for invoking the DLL Manager. This provides a good opportunity to cache previously computed results.

Every time an SPF method is executed the calling parameters and the results are stored by the DLL manager. If the method is executed with the same parameters the cached results are returned.

The cache is only valid for a single storage service to ensure that data is kept consistent. If the data in a file is changed all the cached method results are removed.

6 Discussion

The ability of the FOM to support many programming languages adds complexity to the project. Wrapping or integrating different programming languages can often be impossible or prone to errors. It is for this reason that the FOM will never support *all* programming languages.

To ensure that the FOM is non-intrusive and of benefit to users we aim to support as many of the key programming languages as possible. It is important that users are free to write FOM methods in their programming language of preference. This is enhanced with the use of the .NET Framework which supports many programming languages.

The issue of code quality will always be in dispute as it is not possible to check all the code submitted. This feature remains in the FOM specification as the aim is to provide some assurances, i) user's code cannot compromise the host system, ii) user's code cannot corrupt the data stored in the FOM.

The FOM data security comes from the underlying Operating System (OS), providing a user has permissions to change data the FOM permits the operation.

The FOM does not control the names that users provide for methods and classes, hence naming clashes are expected. This could be overcome using a namespace similar to that used in Extensible Markup Language (XML) or providing an internal FOM name. The end result has to ensure that the FOM is capable of dealing with methods of the same name from different users.

When users submit code it is copied to a single storage service. Multiple submissions are required if the user wishes to make the code available on many machines. The client API supports methods to copy user code to other machines. Currently this copies the user's compiled library to a different machine. It is possible to automatically copy all users' code to all storage services but this does not deal with code dependencies. It is for this reason that code exists only on the machine where it is deposited.

The SPF results cache is only flushed when a data file has been altered. Since an SPF method may return large volumes of data the cache can quickly grow. Currently the cache is limited by available storage space. It would be beneficial to change this cache so that only methods which are time consuming are cached. There needs to be a mechanism to limit the size of the cache and rules dictating which cached results are removed first.

7 Summary

We have proposed a concept, called FOM, where files are treated as objects, exposing users' code as methods on these file objects. The aim is to enhance the user's data management experience without drastic changes to the user's existing workflow.

The key objectives of the FOM are to preserve the user's data format, and the ability to reuse existing code such that the FOM is non-intrusive to the user.

The FOM provides operations which enable users to deposit and retrieve data into the repository. Once the data is stored in the FOM, users have the ability to submit and execute code on the data.

The FOM objectives and operations are discussed along with an implementation independent description of the FOM. The FOM provides a series of features which are then described in the context of the SPF implementation.

We demonstrate all the FOM features using a fully-functional prototype called the SPF. This prototype is implemented using the .NET framework and integrates data across a distributed environment.

We demonstrate the SPF capabilities using two user examples written in different programming languages. The SPF provides the ability to execute methods on remote machines and returns the results as .NET objects. The capabilities of the SPF underlying client API are demonstrated with the use of a GUI.

Using this interface we show how users can locate data and view its associated methods. The GUI can invoke SPF methods and display the results.

Future works involves securing user code, leveraging existing distributed file systems and interoperability across heterogeneous systems.

References

1. Moore, G.: Cramming more components onto integrated circuits. Electronics **38**(8) (April 1965) 114–117
2. Brown, E.: An overview of sql server 2005 beta 2 for the database administrator. Journal (July 2004)
3. Benson, D.A., Karsch-Mizrachi, I., Lipman, D.J., Ostell, J., Wheeler, D.L.: Genbank: update. Nucleic Acids Research **32** (September 2004)
4. Berman, H., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T., Weissig, H., Shindyalov, I., Bourne, P.: The protein data bank. Nucleic Acids Research **28** (2000)
5. Ng, M.H., Johnston, S., Wu, B., Murdock, S.E., Tai, K., Fangohr, H., Cox, S.J., Essex, J.W., Sansom, M.S.P., Jeffreys, P.: Biosimgrid: grid-enabled biomolecular simulation data storage and analysis. Future Generation Computer Systems (2006) doi:10.1016/j.future.2005.10.005.
6. Tai, K., Murdock, S., Wu, B., Ng, M.H., Johnston, S., Fangohr, H., Cox, S.J., Jeffreys, P., Essex, J.W., Sansom, M.S.P.: Biosimgrid: towards a worldwide repository for biomolecular simulations. Organic & Biomolecular Chemistry **2** (2004) 3219–3221
7. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the grid: Enabling scalablevirtual organisations. International Journal on Supercomputer Applications (2001)
8. Apple: Spotlight overview. Developer Connection (April 2005)
9. Freed, N., Borenstein, N.: Multipurpose internet mail extensions. The Internet Engineering Task Force, RFC-2045 (November 1996) [Online; accessed 20-February-2006], `www.ietf.org/rfc/rfc2045.txt`.
10. Sammes, A.J., Jenkinson, B.: Forensic Computing: A Practitioner's Guide. Springer-Verlag London Ltd (January 2007)