3-2011

RestFS: The Filesystem as a Connector Abstraction for Flexible Resource and Service Composition

Joseph P. Kaylor

Konstantin Läufer
Loyola University Chicago, klaeufer@gmail.com

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

Author Name

Book title goes here

Foreward

I am delighted to introduce the first book on Multimedia Data Mining. When I came to know about this book project undertaken by two of the most active young researchers in the field, I was pleased that this book is coming in early stage of a field that will need it more than most fields do. In most emerging research fields, a book can play a significant role in bringing some maturity to the field. Research fields advance through research papers. In research papers, however, only a limited perspective could be provided about the field, its application potential, and the techniques required and already developed in the field. A book gives such a chance. I liked the idea that there will be a book that will try to unify the field by bringing in disparate topics already available in several papers that are not easy to find and understand. I was supportive of this book project even before I had seen any material on it. The project was a brilliant and a bold idea by two active researchers. Now that I have it on my screen, it appears to be even a better idea.

Multimedia started gaining recognition in 1990s as a field. Processing, storage, communication, and capture and display technologies had advanced enough that researchers and technologists started building approaches to combine information in multiple types of signals such as audio, images, video, and text. Multimedia computing and communication techniques recognize correlated information in multiple sources as well as insufficiency of information in any individual source. By properly selecting sources to provide complementary information, such systems aspire, much like human perception system, to create a holistic picture of a situation using only partial information from separate sources.

Data mining is a direct outgrowth of progress in data storage and processing speeds. When it became possible to store large volume of data and run different statistical computations to explore all possible and even unlikely correlations among data, the field of data mining was born. Data mining allowed people to hypothesize relationships among data entities and explore support for those. This field has been put to applications in many diverse domains and keeps getting more applications. In fact many new fields are direct outgrowth of data mining and it is likely to become a powerful computational tool.

Contributors

Michael Aftosmis
NASA Ames Research Center
Moffett Field, California

Pratul K. Agarwal
Oak Ridge National Laboratory
Oak Ridge, Tennessee

Sadaf R. Alam
Oak Ridge National Laboratory
Oak Ridge, Tennessee

Gabrielle Allen
Louisiana State University
Baton Rouge, Louisiana

Martin Sandve Alnæs
Simula Research Laboratory and
University of Oslo, Norway
Norway

Steven F. Ashby
Lawrence Livermore National
Laboratory
Livermore, California

David A. Bader
Georgia Institute of Technology
Atlanta, Georgia

Benjamin Bergen
Los Alamos National Laboratory
Los Alamos, New Mexico

Jonathan W. Berry
Sandia National Laboratories
Albuquerque, New Mexico

Martin Berzins
University of Utah
Salt Lake City, Utah

Abhinav Bhatele
University of Illinois
Urbana-Champaign, Illinois

Christian Bischof
RWTH Aachen University
Germany

Rupak Biswas
NASA Ames Research Center
Moffett Field, California

Eric Bohm
University of Illinois
Urbana-Champaign, Illinois

James Bordner
University of California, San Diego
San Diego, California

George Bosilca
University of Tennessee
Knoxville, Tennessee

Greg L. Bryan
Columbia University
New York, New York

Marian Bubak
AGH University of Science and
Technology

Kraków, Poland

Andrew Canning
Lawrence Berkeley National Laboratory
Berkeley, California

Jonathan Carter
Lawrence Berkeley National Laboratory
Berkeley, California

Zizhong Chen
Jacksonville State University
Jacksonville, Alabama

Joseph R. Crobak
Rutgers, The State University of New Jersey
Piscataway, New Jersey

Roxana E. Diaconescu
Yahoo! Inc.
Burbank, California

Peter Diener
Louisiana State University
Baton Rouge, Louisiana

Jack J. Dongarra
University of Tennessee, Knoxville, Oak Ridge National Laboratory, and
University of Manchester

John B. Drake
Oak Ridge National Laboratory
Oak Ridge, Tennessee

Kelvin K. Droegemeier
University of Oklahoma
Norman, Oklahoma

Stéphane Ethier
Princeton University
Princeton, New Jersey

Christoph Freundl
Friedrich–Alexander–Universität
Erlangen, Germany

Karl Fürlinger
University of Tennessee
Knoxville, Tennessee

Al Geist
Oak Ridge National Laboratory
Oak Ridge, Tennessee

Michael Gerndt
Technische Universität München
Munich, Germany

Tom Goodale
Louisiana State University
Baton Rouge, Louisiana

Tobias Gradl
Friedrich–Alexander–Universität
Erlangen, Germany

William D. Gropp
Argonne National Laboratory
Argonne, Illinois

Robert Harkness
University of California, San Diego
San Diego, California

Albert Hartono
Ohio State University
Columbus, Ohio

Thomas C. Henderson
University of Utah
Salt Lake City, Utah

Bruce A. Hendrickson
Sandia National Laboratories
Albuquerque, New Mexico

Alfons G. Hoekstra
University of Amsterdam
Amsterdam, The Netherlands

Philip W. Jones
Los Alamos National Laboratory
Los Alamos, New Mexico

Laxmikant Kalé
University of Illinois
Urbana-Champaign, Illinois

Shoaib Kamil
Lawrence Berkeley National Laboratory
Berkeley, California

Cetin Kiris
NASA Ames Research Center
Moffett Field, California

Uwe Küster
University of Stuttgart
Stuttgart, Germany

Julien Langou
University of Colorado
Denver, Colorado

Hans Petter Langtangen
Simula Research Laboratory and
University of Oslo, Norway

Michael Lijewski
Lawrence Berkeley National Laboratory
Berkeley, California

Anders Logg
Simula Research Laboratory and
University of Oslo, Norway

Justin Luitjens
University of Utah
Salt Lake City, Utah

Kamesh Madduri
Georgia Institute of Technology
Atlanta, Georgia

Kent-Andre Mardal
Simula Research Laboratory and
University of Oslo, Norway

Satoshi Matsuoka
Tokyo Institute of Technology
Tokyo, Japan

John M. May
Lawrence Livermore National Laboratory
Livermore, California

Celso L. Mendes
University of Illinois
Urbana-Champaign, Illinois

Dieter an Mey
RWTH Aachen University
Germany

Tetsu Narumi
Keio University
Japan

Michael L. Norman
University of California, San Diego
San Diego, California

Boyana Norris
Argonne National Laboratory
Argonne, Illinois

Yousuke Ohno
Institute of Physical and Chemical Research (RIKEN)
Kanagawa, Japan

Leonid Oliker
Lawrence Berkeley National Laboratory
Berkeley, California

Brian O'Shea
Los Alamos National Laboratory
Los Alamos, New Mexico

Christian D. Ott
University of Arizona
Tucson, Arizona

James C. Phillips
University of Illinois
Urbana-Champaign, Illinois

Simon Portegies Zwart
University of Amsterdam,
Amsterdam, The Netherlands

Thomas Radke
Albert-Einstein-Institut
Golm, Germany

Michael Resch
University of Stuttgart
Stuttgart, Germany

Daniel Reynolds
University of California, San Diego
San Diego, California

Ulrich Rüde
Friedrich–Alexander–Universität
Erlangen, Germany

Samuel Sarholz
RWTH Aachen University
Germany

Erik Schnetter
Louisiana State University
Baton Rouge, Louisiana

Klaus Schulten
University of Illinois
Urbana-Champaign, Illinois

Edward Seidel
Louisiana State University
Baton Rouge, Louisiana

John Shalf
Lawrence Berkeley National
Laboratory
Berkeley, California

Bo-Wen Shen
NASA Goddard Space Flight Center
Greenbelt, Maryland

Ola Skavhaug
Simula Research Laboratory and
University of Oslo, Norway

Peter M.A. Sloot
University of Amsterdam
Amsterdam, The Netherlands

Erich Strohmaier
Lawrence Berkeley National
Laboratory
Berkeley, California

Makoto Taiji
Institute of Physical and Chemical
Research (RIKEN)
Kanagawa, Japan

Christian Terboven
RWTH Aachen University,
Germany

Mariana Vertenstein
National Center for Atmospheric
Research
Boulder, Colorado

Rick Wagner
University of California, San Diego
San Diego, California

Daniel Weber
University of Oklahoma
Norman, Oklahoma

James B. White, III
Oak Ridge National Laboratory
Oak Ridge, Tennessee

Terry Wilmarth
University of Illinois
Urbana-Champaign, Illinois

List of Figures

List of Tables

Contents

Symbol Description

α	To solve the generator maintenance scheduling, in the past, several mathematical techniques have been applied.
σ^2	These include integer programming, integer linear programming, dynamic programming, branch and bound etc.
$\sum$	Several heuristic search algorithms have also been developed. In recent years expert systems,
abc	fuzzy approaches, simulated annealing and genetic algorithms have also been tested.
$\theta\sqrt{abc}$	This paper presents a survey of the literature
ζ	over the past fifteen years in the generator
∂	maintenance scheduling. The objective is to
sdf	present a clear picture of the available recent literature
ewq	of the problem, the constraints and the other aspects of
bvcn	the generator maintenance schedule.

Part I

This is a Part

1

RestFS: The Filesystem as a Connector Abstraction for Flexible Resource and Service Composition

Joseph Kaylor

Department of Computer Science, Loyola University Chicago

Konstantin Läufer

Department of Computer Science, Loyola University Chicago

George K. Thiruvathukal

Department of Computer Science, Loyola University Chicago

CONTENTS

The broader context for this chapter comprises business scenarios requiring resource and/or service composition, such as (intra-company) enterprise application integration (EAI) and (inter-company) web service orchestration. The resources and services involved vary widely in terms of the protocols they support, which typically fall into remote procedure call (RPC) [1], resource-oriented (HTTP [6] and WEBDAV [22]) and message-oriented protocols.

By recognizing the similarity between web-based resources and the kind of resources exposed in the form of filesystems in operating systems, we have found it feasible to map the former to the latter using a uniform, configurable connector layer. Once a remote resource has been exposed in the form of a local filesystem, one can access the resource programmatically using the operating system's standard filesystem application programming interface (API). Taking this idea one step further, one can then aggregate or otherwise orchestrate two or more remote resources using the same standard API. Filesystem APIs are available in all major operating systems. Some of those, most notably, all flavors of UNIX including GNU/Linux, have a rich collection of small, flexible command-line utilities, as well as various inter-process communication (IPC) mechanisms. These tools can be used in scripts and programs that compose the various underlying resources in powerful ways.

Further explorations of the role of a filesystem-based connector layer in the enterprise application architecture have lead us to the question whether one can achieve a fully compositional, arbitrarily deep hierarchical architecture by re-exposing the aggregated resources as a single, composite resource that, in turn, can be accessed in the same form as the original resources. This is indeed possible in two flavors: 1) the composite resource can be exposed internally as a filesystem for further local composition; 2) the composite resource is exposed externally as a restful resource for further external composition. We expect the ability hierarchically to compose resources to facilitate the construction of complex, robust resource- and service-oriented software systems, and we hope that concrete case studies will further substantiate our position.

Leveraging our prior work on the Naked Objects Filesystem (NOFS) [12], which exposes object-oriented domain model functionality as a Linux filesystem in user space (FUSE) [20], we have implemented RestFS [11], a (dynamically re)configurable mechanism for exposing remote restful resources and as local filesystems. Several sample adapters specific to well-known services such as Yahoo! Placefinder and Twitter are already available. Authentication poses a challenge in that it cannot always be automated; in practice, when systems such as OAuth are used, it is often only the initial granting of authentication that must be manual, and the resulting authentication token can then be included in the connector configuration. As future work, we plan to develop plugins to support resources across a broader range of protocols, such as FTP, SFTP, or SMTP.

1.1 Related Work

There are various lines of related work, which we will discuss in this section.

1.1.1 Representational State Transfer (ReST)

Partly in response to the complexity of the W3C's WS-* web service specifications [3], resource-oriented approaches such as the representational state transfer (ReST) architectural style [7] have received growing attention during the second half of this decade. In ReST, addressable, interconnected resources, each with one or more possible representations, are usually exposed through the HTTP protocol, which is itself stateless, so that all state is located within the resources themselves. These resources share a uniform interface, where resource-specific functionality is mapped to the standard HTTP request methods GET, PUT, POST, DELETE, and several others. Clients of these resources can access them directly through HTTP, use a language-specific framework with ReST client support, or rely on resource- and language-specific client-side bindings.

1.1.2 Inter-Process Communication Through the Filesystem

Most methods of IPC can be represented in the filesystem namespace in many operating systems. Pipes, domain sockets and memory-mapped files can exist in the filesystem in UNIX [13]. While pipes are uni-directional, allowing one program to connect at each end point, other IPC methods such as UNIX domain sockets allow for multiple client connections and permit data to be written in both directions. With this capability, it is possible for output from several programs to be aggregated by one program instead of a 1:1 model as is allowed by pipes. Other methods of IPC, such as memory-mapped and regular files, allow several programs to collaborate through a common, named store of data.

Composition of the files in filesystems is also possible through layered or stackable filesystems. Mechanisms for this differ amongst operating systems. In 4.4BSD-Lite, Union Mounts [17] allowed for filesystems to be mounted in a linear hierarchy. Changes to files lower in the hierarchy would override files in the higher part of the hierarchy. The Plan 9 distributed operating system allowed for the filesystem namespace to be manipulated through the mount, unmount, and bind system calls [18, 19]. In our own research, we have implemented a layered filesystem, OLFS, which allowed for a flexible layering and inheritance scheme through folder manipulation [10]. Each of these approaches manipulates the filesystem namespace and consequently allows for changes in configuration and how IPC resources are located. This capability can help provide for new and interesting ways to share data between programs.

Although not as widespread, some operating systems implement more advanced IPC such as network connections, specific protocols such as HTTP or FTP, and other services through the filesystem namespace. An excellent example of this is the Plan9 operating system. Plan9's filesystem layer, the 9P protocol, is used to represent user interface windows, processes, storage files, and network connections. In Plan9, it is possible through filesystem calls to engage in IPC in a more uniform way on a local machine and across separate machines.

In terms of inter-machine file-based IPC, it has been possible for many years to coordinate and share data among processes by writing to files on network filesystems. As long as the network filesystem has adequate locking mechanisms and an adequate solution to the cache coherency problem, it is possible to perform IPC through file-based system calls over a network filesystem.

Other than coordination through network filesystems or specialized operating system mechanisms like 9P, much inter-machine IPC has been through abstractions on top of the network socket. Remote procedure call approaches such as RPC or RMI have provided a standard way for processes to share data and coordinate with each other. Other socket-based approaches include the HTTP protocol and abstractions on top of HTTP, such as SOAP and REST.

1.1.3 Recent Developments in File-Based IPC

Some more recent advances have been made in terms of inter-machine IPC over the filesystem. Application filesystems are being built on top of FUSE to act as clients for web services such as Flickr, IMAP email services, Amazon S3, and others. Instead of using the socket as the basis for IPC with these services, it has become possible to be able to interact with them through filesystem calls.

IPC through the filesystem offers some advantages. Although in UNIX-like operating systems, it is possible to redirect output to a socket through a program like socat, netcat, or nc, there are many network options and issues like datagram versus streaming to consider. File-based IPC often presents a simpler interface to work with and leaves many of the networking and protocol questions to the implementing filesystem. Another important advantage that it offers is that processes that interact with these application filesystems is transparency. The processes that interact with these application filesystems do not need to be aware of which service they are interacting with, which URL it is located at or what types of SOAP messages it requires to communicate with. With a Flickr filesystem, it is possible to use programs that simply interact with images aside from a web browser to interact with the Flickr photo service.

1.1.4 The Shift from Kernel Mode to User Mode Filesystem Development

In very early systems, development of new filesystem code was a challenge because of high coupling with storage device architecture and kernel code.

In the 1970s, with the introduction of MULTICS, UNIX, and other systems of the time, more structured systems with separated layers became more common. UNIX used a concept of i-nodes, which were a common data structure that described structures on the filesystem [21]. Different filesystem implementations within the same operating system kernel could share the i-node structure; this included on-disk and network filesystems. Early UNIX operating systems shared a common disc and filesystem cache and other structures related to making calls to the I/O layer that managed the discs and network interfaces.

Newer UNIX-like systems such as 4.2 BSD and SunOS included an updated architecture called v-nodes [15]. The goal was to split the filesystem's implementation-independent functionality in the kernel form the filesystem's implementation-dependent functionality. Mechanisms like path parsing, buffer cache, i-node tables, and other structures became more shareable. Also, operations based on v-nodes became reentrant, thereby allowing new behavior to be stacked on top of other filesystem code or to modify existing behavior. V-nodes also helped to simplify systems design and to make filesystems implementations more portable to other UNIX-like systems. Many modern UNIX-like systems have a v-nodes-like layer in their filesystems code.

With the advent of micro-kernel architectures, filesystems being built as user-mode applications became more common and popular even in operating systems with monolithic kernel architectures. Several systems with different design philosophies have been built. We describe three of these systems that are most closely related to NOFS: FUSE [20], ELFS [9], and Frigate [14].

The Extensible File System (ELFS hereafter) is an object-oriented framework built on top of the filesystem that is used to simplify and enhance the performance of the interaction between applications and the filesystem. ELFS uses class definitions to generate code that takes advantage of pre-fetching and caching techniques.ELFS also allows developers to automatically take advantage of parallel storage systems by using multiple worker threads to perform reads and writes. Also, since ELFS has the definition of the data structures, it can build efficient read and write plans. The novelty of ELFS is that the developer can use an object-oriented architecture and allow ELFS to take care of the details.

Frigate is a framework that allows developers to inject behavioral changes into the filesystem code of an operating system. Modules built in Frigate are run as user-mode servers that are called to by a module that exists in the operating system's kernel. Frigate takes advantage of the reentrant structure of vnodes in UNIX-like operating systems to allow the Frigate module developer to layer behavior on top of existing filesystem code. Frigate also allows the

developer to tag certain files with additional metadata so that different Frigate modules can automatically work with different types of files. The novelty of Frigate is that developers do not need to understand operating-systems development to modify the capabilities of filesystem code, and they can test and debug their modules as user-mode applications. But they still need to be aware of the UNIX filesystem structures and functions.

File Systems in Userspace (FUSE hereafter) is a user mode filesystems framework. FUSE is supported by many UNIX-like operating systems such as Linux, FreeBSD, NetBSD, OpenSolaris, and Mac OSX. The interface supported by FUSE is very similar to the set of UNIX system calls that are available for file and folder operations. Aside from the ability to make calls into the host operating system, there is less sharing with the operating system than with v-nodes such as path parsing. FUSE has helped many filesystem implementations such as NTFS and ZFS to be portable to many operating systems. Since FUSE filesystems are built as user-land programs, they can be easier to develop in languages other than C or C++, easier to unit test, and easier to debug. Accordingly, FUSE has become a popular platform for implementing application-specific filesystems.

1.2 Composition of Web Services Through the Filesystem

Filesystems can play different roles in the composition of web-based resources and services. We will now study these in more detail.

1.2.1 Commonalities Between Web Resources and the Filesystem

We believe that there are clear commonalities between web services and the filesystem. Both systems have a concept of a URI. In web services, this can be an HTTP URL. In the filesystem this can be a file or folder path. In both systems there are protocol actions that can be used to send and retrieve data. In web services this can be accomplished through HTTP GET and POST. In filesystems, this can be accomplished through read() and write() system calls. In both systems it is possible to invoke executable elements. In web services this can be performed with GET and POST calls and the use of SOAP messages to web service URLs. On a local filesystem, executable services can be invoked by loading and executing programs from the local filesystem.

In our exploration we believe that there are three candidates for how to build the filesystem layer to expose resources from the web. The first way is through application filesystems built with the Naked Object Filesystem (NOFS) framework. The second way is to use the filesystem as a connector

layer to abstract and re-expose web resources to the local system. The third way is to use a combination of the filesystem as a connector layer and the filesystem as an application. We have explored this second route with RestFS, which has been implemented using the NOFS framework. In each of these methodologies we demonstrate how to map concepts from web services onto the filesystem. We will also explain the advantages and disadvantages to each approach.

1.2.2 The Filesystem as a Connector Layer

In our exploration of filesystems, we questioned whether a filesystem could be used as a connector layer for web services. We also questioned whether that connector layer could be used to compose web services with local and other web services and then expose those web services externally as a new web service. RestFS is our attempt to implement such a filesystem.

RestFS is an application filesystem implemented with the NOFS framework. RestFS uses files to model interaction with web services. When a file is created in RestFS, two files are created: a configuration file and a resource file. The configuration file contains an XML document that can be updated to contain a web service URI, web method, authentication information, and a triggering filesystem method. Once configured, the resource file can be interacted with on the local machine to interact with a web service.

One example of the usage of RestFS is to create a file that can perform a Google Search. In this example, the file is configured with the Google APIs server and the web search service. Web requests are sent with the GET HTTP method and are triggered by the utime filesystem call. When a user of the filesystem issues a 'touch' command on the resource file, a GET request is issued by RestFS to the Google API server and the response from that server is written back to the resource file, which will be available for subsequent reads. In this example, the task of configuring the resource, triggering the request, and parsing the results are left to a Bash shell script.

Another example usage of RestFS is with the Yahoo! PlaceFinder service. This example is similar to the Google search example. The configuration file is setup with the URI for the web service, and the utime system call is used to trigger the web request. Also, in this example, a shell script is used to configure the RestFS file, trigger the web service call, and to parse the results.

With our implementation of resource files in RestFS, remote web resources can be interacted with in a similar way as other local file based IPC. The local nature of the resource files allows for programs that read from and write to the resource files to be unaware of the web service that RestFS is communicating with. For example, it is possible to use programs such as grep, sed, or perl to search, transform, and manipulate the data in the resource file. In each of these cases, these programs do not need to be aware that the data they are working with has been transparently read from or written to a remote web service.

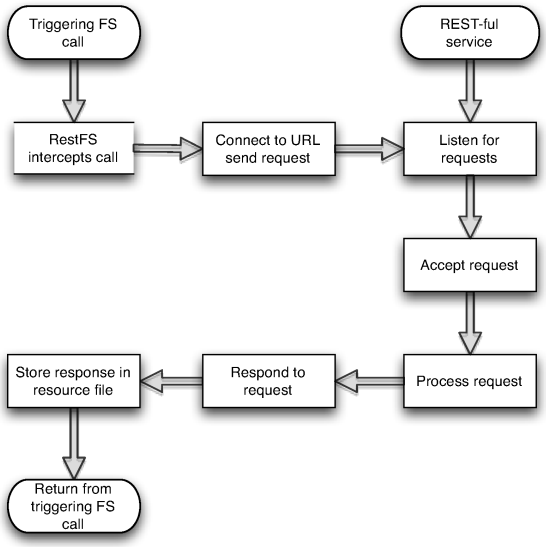

FIGURE 1.1
The timeline of a RestFS web service call

Because RestFS acts as only a connector layer and provides no additional interpretation or filtering of requests or responses, external programs are required to read and write the structured data that is necessary for interact with configured web services. In the Google Search and Yahoo! PlaceFinder examples, the task of writing a structured request and parsing the response was left to a shell script that took advantage of UNIX command line tools like sed, grep, and others. These scripts had to be aware of the structure of both the requests and response needed by the web service. It is possible to filter, translate, and load data from the resource files with any local program that can accept data from a file or a UNIX pipe. As a consequence, it is possible to augment the value added of the web service with local programs in several possible combinations.

The connector model presented by RestFS in combination with other IPC mechanisms on the local operating system makes it possible to compose the data from several web services with each other in a flexible and reconfigurable way. One possible example of this would be to setup several resource files for RSS news feeds across the internet. A script could be implemented to parse each of those news sources for specific topics, aggregate them, and then write them to another resource file that could represent a submission form and service for creating articles on a blog. The same system then could have several resource files setup to watch Twitter accounts for comments on the article and post responses on Twitter to the blog site. If new news sources

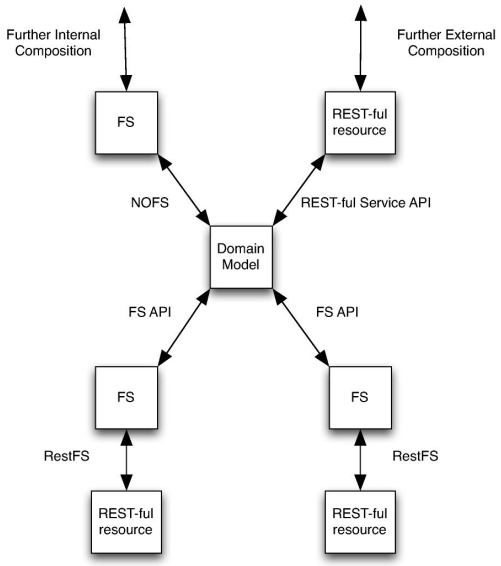

FIGURE 1.2
The flexible internal and external composition possible with RestFS

become important or new Twitter accounts are necessary, new resource files and alterations to scripts can be made to expand and reconfigure the system. It is possible to do all of this with a series of scripts and small programs on a UNIX operating system that use RestFS as a connector layer.

There are some instances where the connection layer concept has some difficulties in our exploration. When trying to compose some web services that are built around human interaction through rich user interfaces, it can be difficult to create a program that can interact with these services in a simple way.

One example of this is the CAPTCHA human test. To reduce "spam" in the form of email and as entries on blogs, many websites incorporate a form that requests the user perform a small test such as recognizing a sound or interpreting letters on an image to prove to the system that the user of the web service is in fact a human. Often, after these initial interactions, it is possible for simple interaction with RestFS, but because of them it is not always straightforward to automate the entire interaction with a web service. Other forms of non machine readable interactions such as the use of images, sounds, or video can present complications for composing web services with RestFS.

Another example would be web services that make use of the user interface for complex validation or additional business rules. While not an ideal design, such web services still exist on the internet. Because local programs will in-

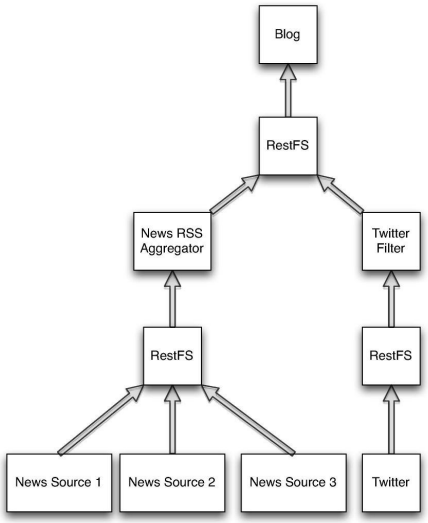

FIGURE 1.3
A sample composition of a blog, news sources, and Twitter

teract with the application tier and not the presentation tier of a web service, any logic that exists in that presentation tier that is necessary for proper communication with the application tier must be duplicated in whatever local composition is made of the web service.

1.2.3 The Filesystem as an Application and Abstraction

While exploring the possibilities for using filesystems to interact with web services, we observed the emergence of application oriented filesystems such as WikipediaFS, IMAPFS, and FlickrFS. Each of these filesystems demonstrate different web services represented as different components on filesystems. In several email oriented filesystems, folders available in IMAP accounts are represented as folders on the local filesystem and individual email messages as files. In photo-sharing-oriented filesystems such as FlickrFS, photos are categorized into folders and exposed as standard image files. In each of these application filesystems, normal file operations work as expected. Copying and deleting files in FlickrFS completes the expected operation of downloading and uploading photos with a user's Flickr account.

After our own experiences with implementing storage oriented filesystems in FUSE, we felt that application filesystems would benefit from a different abstraction than what is presented by FUSE. To that end, we implemented the Naked Objects Filesystem (NOFS). NOFS allows a developer to implement an application filesystem by annotating Java classes in an application

domain model. Through inspection of these domain objects and associated annotations, NOFS presents a filesystem composed of files, folders, and executable scripts to the user through FUSE to interact with the domain model. We will explore in detail the architecture and internal workings of NOFS in a later section.

With the NOFS framework, we were able to implement application filesystems in a more rapid fashion with less filesystem glue code needed. This helped reduce the necessary components to expose a web service such as the Flickr photo service as a filesystem (Figures 1.4, 1.5) to the interaction with the REST-ful web service and the construction of an adequate domain model to represent the structure of the service and filesystem. Our implementation of a simple Flickr filesystem took 484 lines of Java code. An existing Python implementation of the Flickr filesystem that uses FUSE directly took 2144 lines of code. About half of the Python implementation was code used to glue FUSE to the Flickr photo service. The remainder of the code was related to handling the Flickr photo service.

Another example of an application filesystem built with NOFS is the Yahoo! Finance stock ticker filesystem. We were able to implement the entire filesystem with just 155 lines of code in two Java classes (see Figures 1.6, 1.7)

Application filesystems like those that can be built with NOFS are very useful for user interaction. Actions that make sense in a photo library service have excellent mappings to filesystem actions. The fundamental unit in the service, the photo, maps well to a file. Collections and categories of photos map well to folder structures. In this particular case, for the sake of user interaction, the structure of the web service calls and their mapping into a connector layer like RestFS would not be a convenient structure for user interaction. The application filesystem allows for a better mapping of the business unit / domain model that is presented by the web service.

Application filesystems built through NOFS also are able to handle action validation and interaction in a simpler way than is possible with RestFS like systems. If an action on the domain model for an NOFS filesystem is in some way invalid, an exception can be raised so that the filesystem call that triggered the action can return an error code. In this way, NOFS domain models can restrict copy, delete, read, write or other filesystem operations to those that are considered valid by the domain model. Resource files in RestFS expect that data written to and read from the resource files is in a valid format.

Application filesystems are not as well suited for simple re-configuration or changes in composition as RestFS is. To introduce changes in an application filesystem, either facilities for dynamically adding plugins must be introduced, or the system must be unmounted, modified and mounted as a filesystem again.

```
@DomainObject(CanWrite=false)
public class FlickrPhoto implements IProvidesUnstructuredData {
   private byte[] _data;
   public void setData(byte[] data) {
      _data = data;
   }

   public FlickrPhoto() {}

   private String _name;
   @ProvidesName
   public String getName() { return _name; }
   @ProvidesName
   public void setName(String name) { _name = name; }

   public boolean Cacheable() { return false; }
   public long DataSize() { return _data.length; }
   public void Read(ByteBuffer buffer, long offset, long length) {
      for(long i = offset; i < offset + length && i < _data.length;
               i++) {
         buffer.put(_data[(int)i]);
      }
   }
   public void Truncate(long length) { }
   public void Write(ByteBuffer buffer, long offset,
      long length) { }
}
```

FIGURE 1.4
The FlickrPhoto domain object from FlickrFS

```
@FolderObject(CanAdd=false, CanRemove=false)
@DomainObject
public class FlickrUser {
   private List<FlickrPhoto> _photos =
      new LinkedList<FlickrPhoto>();
   public FlickrUser() {}

   private String _name;
   @ProvidesName
   public String getName() { return _name; }
   @ProvidesName
   public void setName(String name) { _name = name; }

   private IDomainObjectContainerManager _manager;
   @NeedsContainerManager
   public void setContainerManager(IDomainObjectContainerManager
         manager) {
      _manager = manager;
   }

   private long _lastGet = 0;
   @FolderObject(CanAdd=false, CanRemove=false)
   public List<FlickrPhoto> getPhotos() throws Exception {
      if(_lastGet == 0 || System.currentTimeMillis() - 10000 >
            _lastGet) {
         UpdatePhotos();
         _lastGet = System.currentTimeMillis();
      }
      return _photos;
   }

   private void UpdatePhotos() throws Exception {
      _photos = new LinkedList<FlickrPhoto>();
      FlickrFacade facade = new FlickrFacade();
      for(PhotoSet set : facade.getPhotoSets(_name)) {
         for(Photo photo : facade.getPhotosInASet(set, 100)) {
            FlickrPhoto newPhoto = _manager
                .GetContainer(FlickrPhoto.class)
                .NewPersistentInstance();
            newPhoto.setName(photo.getTitle() +".jpg");
            newPhoto.setData(facade.getDataForPhoto(photo));
            _photos.add(newPhoto);
            _manager.GetContainer(FlickrPhoto.class)
               .ObjectChanged(newPhoto);
         }
      }
      _manager.GetContainer(FlickrUser.class).ObjectChanged(this);
   }
}
```

FIGURE 1.5
The FlickrUser domain object from FlickrFS

```
@RootFolderObject
@DomainObject
@FolderObject(CanAdd=false, CanRemove=false)
public class Portfolio {
   private IDomainObjectContainerManager _manager;
   private List<Stock> _stocks = new LinkedList<Stock>();

   @NeedsContainerManager
   public void setContainerManager(IDomainObjectContainerManager
         manager) {
      _manager = manager;
   }

   @FolderObject(CanAdd=true, CanRemove=true)
   public List<Stock> getStocks() throws Exception {
      UpdateStockData();
      return _stocks;
   }

   private void UpdateStockData() throws Exception {
      String url = BuildURL();
      List<String> dataLines = getDataFromURL(url);
      for(Stock stock : _stocks) {
         String dataLine = null;
         for(String line : dataLines) {
            if(line.startsWith("\"" + stock.getTicker())) {
               dataLine = line;
                break;
            }
         }
         if(dataLine != null) {
            stock.UpdateData(dataLine);
         }
      }
   }

   private String BuildURL() { .... }
   private List<String> getDataFromURL(String url) { .... }

   @Executable
   public void AddAStock(String ticker) throws Exception {
      Stock stock = _manager.GetContainer(Stock.class)
         .NewPersistentInstance();
      stock.setTicker(tocker);
      _stocks.add(stock);
      _manager.GetContainer(Stock.class).ObjectChanged(stock);
      _manager.GetContainer(Portfolio.class).ObjectChanged(this);
   }
}
```

FIGURE 1.6
The Portfolio class for the stock ticker filesystem

```
@DomainObject(CanWrite=false)
public class Stock {
   private String _ticker;
   private string _data;

   public Stock(String ticker) {
      _ticker = ticker;
   }

   @ProvidesName
   public String getTicker() { return _ticker; }

   public void UpdateData(String data) { _data = data; }

   public String getPrice() {
      return _data.split(",")[1];
   }

   public String getDate() {
      return _data.split(",")[2];
   }

   public String getTime() {
      return _data.split(",")[3];
   }
}
```

FIGURE 1.7
The Stock class for the stock ticker filesystem

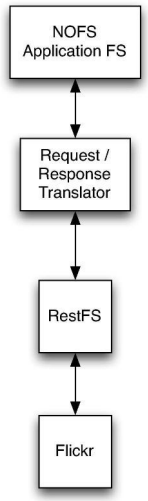

FIGURE 1.8
FlickrFS with both RestFS and NOFS

1.2.4 Combining the Approaches: Using the RestFS Connector Layer in a NOFS Application Filesystem

It is also possible to use the filesystem as an application and the filesystem as a connector layer to form service compositions. The positive aspects of both approaches can be combined to derive the advantages of each system.

One of the important disadvantages of a filesystem as an application is that extra code must be added to the implementation to accommodate changing configurations and compositions of external resources. If this extra code is not present, then to realize changes, a filesystem must be unmounted, modified and then mounted again. With the filesystem as a connector layer, adding complex validation and advanced user interaction semantics is difficult. When both approaches are combined, these disadvantages are no longer present.

To demonstrate a possible use of both technologies, consider a photo service such as Flickr that you wish to represent as a filesystem. One possible way to construct a filesystem is to use both RestFS and an application filesystem built with NOFS. A domain model similar to the one in the FlickrFS example discussed earlier can be constructed. In this case, instead of using a library to interact with Flickr in the application filesystem, the application filesystem could use a RestFS resource file and a small script that translates requests and replies from the Flickr photo service into representations that conform to the domain model of the application filesystem.

This composition is more flexible to change than it would be implemented only as an application filesystem. For example, if an additional photo service

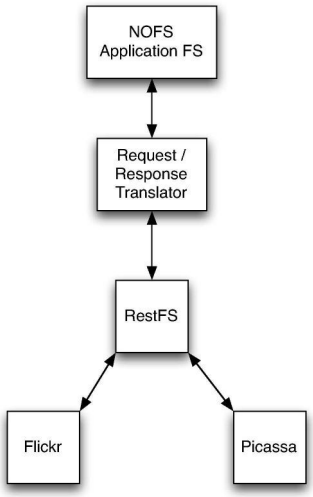

FIGURE 1.9
A photo filesystem composed of multiple photo services

were added, it would involve creating a second resource file in RestFS that the NOFS application filesystem would interact with. All that would be needed is to implement a small script that could translate requests and replies from the new web service into a form that could be consumed by the application filesystem's domain model.

1.3 Building Application Filesystems with the Naked Object Filesystem (NOFS)

The capabilities, role and development process of the filesystem have evolved throughout the years. Early on, filesystems were developed as tightly integrated operating system kernel components. Kernel mode filesystems require a complex understanding of systems programming, systems programming languages, and the underlying operating system. There are fewer people who have this skill set as object-oriented frameworks and languages are becoming more and more popular. As user mode programs are more suited for loading and launching programs dynamically, a kernel mode component often has to take additional steps to support being unloadable or configurable at run time. Also because operating system kernels cannot easily depend upon user mode libraries, it is difficult to reuse software components within the operating system and by extension in filesystem implementations. Because of this, there is

much code that has already been developed using the patterns available and common to enterprise application frameworks that either cannot be used or are difficult to reuse in systems development. Two important advancements needed over kernel mode filesystems development are the ability to implement filesystems as user-mode programs and frameworks that allow enterprise development techniques and patterns to be applied to filesystems development. The answer to the user mode problem has been user-mode filesystem frameworks such as FUSE for UNIX-like operating systems and Dokan for the Windows operating systems. Our answer to provide an enterprise-patterns-friendly framework is the NOFS framework.

1.3.1 An Explanation of Naked Objects

Naked Objects [16] is the term used to describe the design philosophy of using plain object-oriented domain models to build entire applications. In the realm of desktop applications, Naked Object frameworks remove the concern of the developer in implementing user interfaces, model-view-controller patterns, and persistence layers. These components are generated for the domain model by the Naked Objects framework automatically either through the use of reflection or through additional metadata supplied with the domain model.

A characteristic feature of Naked Object frameworks is that they present an object-oriented user interface. Applications where the user is treated more as a problem solver than as a process follower benefit from an object oriented user interface [16, p41]. For many applications, processes are very important and an object-oriented user interface is not the best fit. We believe that the interface presented to the programmer and to the user of a filesystem is also object-oriented. In a filesystem, the components are not exposed to the user to facilitate the moving, reading, writing, creation, or deletion of files and folders. These actions are accomplished with external programs and references to the actual objects as command line parameters. The user interaction with filesystems is a noun-verb style of interaction and not a verb-noun interaction, which is more common with typical desktop applications. Like the Naked Object user interfaces, filesystems "provide the user with a set of tools which to operate and does not dictate ... the users sequence of actions" [16, p41].

1.3.2 The Naked Object Filesystem (NOFS)

There are three important contributions made by the NOFS framework. The first is that NOFS demonstrates the filesystem can be used as an object-oriented user interface in a Naked Objects framework and that the Naked Objects design principle can be applied successfully to filesystems development. The second contribution is that NOFS inverts and simplifies the normal filesystem development contract. In FUSE and operating system kernels, there are a series of functions to implement and data structures to work with. With the NOFS framework, a domain model is inspected to produce a filesys-

tem user interface. Domain models for NOFS do not implement filesystem contracts or work with filesystem structures. Instead, they are described with metadata that is used by NOFS to allow the domain model to interact with the FUSE filesystem framework. In this way, NOFS follows the dependency inversion principle in that the higher level domain model does not depend upon the lower level file system model. The third contribution made by the NOFS framework is that by providing an object-oriented framework to develop filesystems, we allow developers who are unfamiliar with systems or UNIX programming to more easily and rapidly implement experimental or lightweight filesystems. With this object-oriented framework, it becomes easier to unit test a filesystem implementation because details of the operating system do not need to be stubbed or mocked out; only the domain model needs to be verified.

1.3.3 Implementing a Domain Model with NOFS

Here we will explore developing a domain model with NOFS. We will explore three domain models: an address book domain model that was developed for presentation purposes, a Flickr domain model for manipulating photos on the Flickr photo service, and a stock ticker tracking filesystem for Yahoo! Finance.

1.3.3.1 Implementing Files and Folders in NOFS

In NOFS, files are modeled as plain classes that are described with metadata. The methods on the class are not constrained to any specific interface but are used to model the structure of the data in a file. There are two ways for classes to expose their data: through translation of the return values of public methods to structured XML files or by defining the structure of these files by implementing an interface with read and write methods.

In the example in Figure 1.10, the class Contact marks itself as a file object by using the @DomainObject Java annotation. The class also tells NOFS that it manages its own file name with the @ProvidesName annotation on the getName accessor and the setName mutator methods. The persistence mechanism of NOS is injected upon construction of the Contact class through the setContainer method, which is marked by the @NeedsContainer method. An example representation of the Contact class as a file in the NOFS filesystem is as follows in Figure 1.11.

In this example the class FlickrPhoto (Figure 1.4) marks itself as a file object by using the @DomainObject Java annotation. It tells NOFS that it is immutable by setting the CanWrite member of the DomainObject annotation to false. IFlickrPhoto's responsibility is to model a graphical image from the Flickr photo sharing website. Since it is convenient to expose to the filesystem these photos as an image file and not as an XML file, FlickrPhoto provides read and write methods as defined by the IProvidesUnstructuredData NOFS interface.

```
@DomainObject
public class Contact {
   private String _name;
   private String _phoneNumber;
   private IDomainObjectContainer<Contact> _container;

   @ProvidesName
   public String getName() { return _name; }

   @ProvidesName
   public void setName(String name) { _name = name; }

   public String getPhoneNumber() { return _phoneNumber; }
   public void setPhoneNumber(String value) {
      _phoneNumber = value;
   }

   @NeedsContainer
   public void setContainer(IDomainObjectContainer<Contact>
         container) {
      _container = container;
   }
}
```

FIGURE 1.10
The Contact NOFS Domain Object

```
<?xml version="1.0"?>
<Contact>
   <PhoneNumber>555-5555</PhoneNumber>
<Contact>
```

FIGURE 1.11
Representation on the filesystem of the Contact domain object

```
@DomainObject
@FolderObject(CanAdd=true, CanRemove=true)
public class Category extends LinkedList<Contact> {
   private String _name;

   @ProvidesName
   public void setName(String name) { _name = name; }

   @ProvidesName
   public String getName() { return _name; }
}
```

FIGURE 1.12
The Category NOFS Domain Object

In the example in Figure 1.6, the class Portfolio marks itself as a folder object by using the @DomainObject and the @FolderObject Java annotations. The FolderObject annotation sets CanAdd and CanRemove to false to tell NOFS that the user of the filesystem cannot add or remove files from the folder. The Portfolio class exposes two objects to NOFS, a folder called Stocks through the getStocks() method and an executable script through the AddAStock method. NOFS can tell that getStocks() is a folder because its return type is a collection and because of the FolderObject annotation on the method declaration. NOFS can tell that the AddAStock method is to be exposed as an executable script because of the Executable annotation on the method declaration. The script that will appear in the Portfolio object's folder will be an automatically generated Perl script that will accept one argument and pass it back to NOFS, which will in turn pass it to the correct domain object instance based upon path. In this way, NOFS domain objects can expose additional executable behavior to the filesystem interface.

Another way to implement a folder is through extending a collection type such as LinkedList. The Category class in Figure 1.12, which is a part of the address-book filesystem, takes advantage of this approach. Instead of statically defining the components of a folder as was done in the Portfolio example, the Category folder's components will be defined by what is present in the collection.

1.3.4 Architecture of NOFS

There are two important aspects to the architecture of NOFS. The first is its place and role in the filesystem architecture and the second is how domain objects are mapped to FUSE calls. Firstly, the overall architecture of FUSE is not changed by NOFS. NOFS exists as an additional layer on top of FUSE. A diagram of this relationship is available in Figure 1.13.

The existing context switches between user-mode programs with the kernel

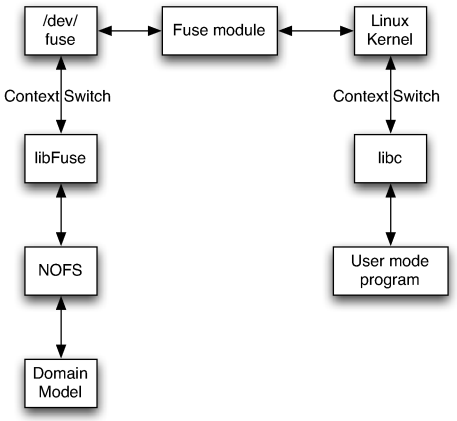

FIGURE 1.13
The relationship between NOFS, FUSE, and the Linux kernel

and between filesystem implementations with FUSE still exist with NOFS. No new context switches are created by the NOFS framework. The reader is encouraged to consult literature and documentation on FUSE to explore additional details of FUSE and its implementations (see also 1.1.4 above).

The way domain models are mapped to fuse calls can be split into two important parts: how paths are translated to domain objects and how domain objects are translated to different file object types.

Domain objects are translated to files, folders, root-folders, and executable scripts through the use of Java annotations. Depending upon the annotation, classes or methods are scanned to see if there are matching annotations. If a class or method is marked as a file, then that class instance or the return value of that method is exposed as a file on the filesystem. The same is true of folders. If a class is marked as a folder and if it is also a list, then the class is exposed as a folder and the contained objects in the list are exposed as children of that folder. If the class is marked as a folder and is not also a list, then the member methods of the class are exposed as children of the folder. If a particular method is encountered and marked as executable, NOFS generates a Perl script that accepts as arguments a list matching the parameters of the method. Executable methods will be explored in more detail soon.

Paths are translated with the algorithms in Figures 1.14 and 1.15. The algorithm basically finds the root of the filesystem by searching for an object instance of type root and then traverses the path from that instance until it encounters a mismatch or runs out of segments in the path and returns a matching object.

Additional path and type translation is involved in methods that are exposed as executable scripts in NOFS. If an method has as parameters just

```
translate_path(path) {
   current = find_root();
   for-each(segment in path) {
      if(current IsA folder) {
         if(current IsA list) {
            current = current[segment];
         } else if(current HasA member whose name matches
               segment) {
            current = current.members[segment];
         } else {
            raise exception "invalid path";
         }
      } else {
         raise exception "invalid path";
      }
   }
   return current;
}
```

FIGURE 1.14
The NOFS path translation algorithm

```
find_root() {
   List roots = new List();
   for-each(instance in all_instances) {
      if(instance IsA root-folder) {
         roots.add(instance);
      }
   }
   if(roots.count() == 0)  {
      raise exception "no roots found";
   } else if(roots.count() > 1) {
      raise exception "more than one root found";
   }
   return roots[0];
}
```

FIGURE 1.15
The NOFS root discovery algorithm

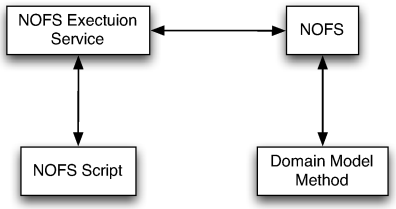

FIGURE 1.16
The communication path for executable scripts in NOFS

```
translate_arguments(arg_list, method) {
   for(int i = 0; i < arg_list.length; i++) {
      if(method.parameters[i] IsA NOFS-domain-object) {
         args_list[i] = translate_path(arg_list[i]);
      }
   }
}
```

FIGURE 1.17
The NOFS argument translation algorithm

primitive or string types, then NOFS has no additional translation work to perform and just passes values as they are to a method from the script. If a method parameter is of one of the domain model's types, then the script will accept a path as a valid argument and NOFS will translate the path to an object reference that is then passed to the method (see Figure 1.17). In this way, it is possible to pass by value or by reference to methods on NOFS domain classes.

With path to object translation, filesystem calls like getdir(), mkdir(), mknod(), unlink() and similar calls map pretty well into path translation and object creation and deletion actions. Next, we will discuss how calls such as read(), write(), open(), and close() work.

In NOFS, there are three ways that a file object's data is managed. The first way is if the file happens to be an executable script. If a method is determined to be an executable script, NOFS will generate Perl code to wrap a call back into NOFS and make file that the Perl code is placed in read-only. The second way data is managed is through the IProvidesUnstructuredData interface. This interface was mentioned earlier in the FlickrPhoto example. If NOFS encounters a file object that implements this interface, it will pass read and write calls directly to the object. The final way data is managed is if the domain object exposes public members. In this case, NOFS will examine the members and translate all primitive members into XML elements. If a non

```
represent_as_xml(object) {
   for-each(member in object.class_definition) {
      if(member IsA primitive) {
         emit element with value of primitive;
      } else {
         represent_as_xml(member);
      }
   }
}
```

FIGURE 1.18
The NOFS XML serialization algorithm

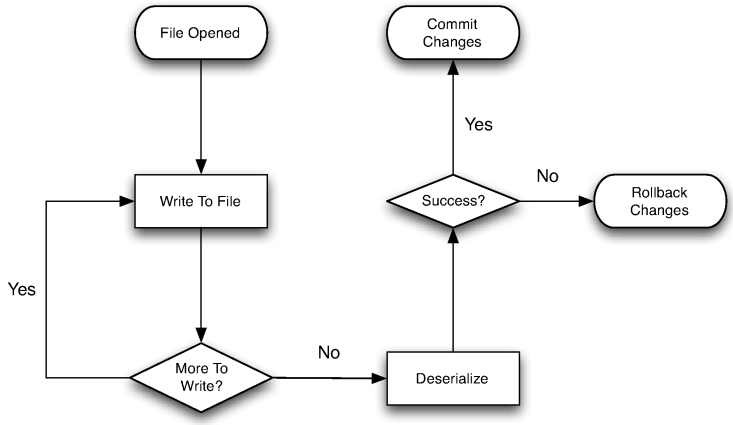

FIGURE 1.19
The NOFS cache and serialization relationship

primitive type is encountered an element will be emitted and it will also be serialized into XML. The algorithm is available in Figure 1.18.

In the case of XML files being written back to, all writes are cached by NOFS until the file handle is closed. When the file handle is closed, NOFS will perform a similar algorithm as represent_as_xml except to deserialize the XML back into the domain object. If there is a mismatch in the XML structure with respect to the domain object or if the deserialization process causes the domain object to throw an exception, the change to the domain is rolled back entirely and the contents of the XML file are reverted to their state before any write occurred. The cache management algorithm can be found in Figure 1.19.

The final set of calls mapped to FUSE by NOFS are metadata calls such as getxattr, getattr, chown, chmod, and other related calls. There are two ways that these are managed. The first way is if a method has any of the ProvidesGID, ProvidesUID, ProvidesMode, ProvidesLastAccessTime, ProvidesLastModifiedTime, or ProvidesCreateTime annotations. For any class that has methods with these annotation, NOFS assumes that the domain object maintains this metadata. For each case where one of these annotations is not encountered, NOFS will provide a default implementation and store appropriate metadata in a small db4o database for each instance of a domain object.

It is sometimes useful for domain models to manage this additional metadata in a non-default way. One important reason is if the data is a legitimate part of the domain model. One good example would be a web service that provides online document editing. The domain object that models a document should also retrieve attributes like creation, modification, and access times from the server. For other domain models, such as the stock ticker domain model presented earlier, this information is less important to the domain model and can be adequately handled by the NOFS default implementation. These two possibilities allow the creator of the domain model to model only attributes that they are concerned with and nothing more.

The domain object persistence mechanism used in NOFS is straightforward and natural in the way it maps annotated class definitions to XML elements at run time. A thorough evaluation of this approach and its alternatives is still needed. One alternative is our earlier work on simple XML data bindings and linearized external representations of XML data [2]. Other choices include more complex, schema-based XML data binding frameworks such as JAXB [5] and XStream [23], as well as non-XML formats such as JSON [4]. In addition, we plan to allow domain classes in future versions of NOFS to choose alternate representations through their own serializers or XSLT transformations.

1.4 Architecture and Details of RestFS

Our work on RestFS was inspired by two other bodies of work: Plan 9's 9P protocol and netfs [18], and Representational State Transfer or REST [7]. While exploring REST, we realized that the GET, PUT, POST, and DELETE HTTP methods mapped well into filesystem operations and that there were a few ways that we might map REST-ful services onto the filesystem. Another important observation that we made at the time is how other forms of interprocess communication and especially sockets have been the basis for composing programs and services. We felt after our exploration of layered filesystems research with the OLFS filesystem that the filesystem held the possibility to mediate the composition of web services. With these observations in hand and

with the NOFS filesystem framework we set about developing a filesystem to support communication with and composition of web services.

In Plan9, network communication is not performed through the use of system calls like accept, connect, listen, send or recv. Network communications are performed through file operations in netfs under a special folder '/net' in the Plan 9 filesystem. In addition to folders separating types of network connections into UDP and TCP, there are two types of folders in netfs: connection / configuration files and stream files. Connection / configuration files contained details about IP addresses, port numbers, and socket options. Once fully configured it is possible to read from and write to the special stream files in netfs to send and receive data from a remote computer.

1.4.1 RestFS's approach

The use of files for networking and the separation of files into configuration and streams offer very important advantages over the family of calls used in UNIX and other operating systems for networking. The first advantage is that no additional system calls other than the ones necessary for filesystem interaction are needed to work with the network. Calls like connect, listen, send, recv, accept, and others are not necessary when the network can be managed through the filesystem. The other important advantage is in the separation of responsibility between the files. With the separation, it is possible for one process to manage configuration of the network connection while another process is responsible for reading and writing to the connection as if it were a normal file. In this way, software that is capable of working with just file I/O calls does not need to be extended to support networking code; it need only be supplemented with some prior configuration. Another important advantage of using the filesystem for network communication is that it allows for network connections to be named in a namespace that has a longer lifetime than programs that may take advantage of a network connection. For example, a program may read from and write to a network file and work correctly for some time. If that program crashes, it can be re-launched and resume working with the network file without having to re-establish any connections. This capability also allows the programs on either end point of the connection to change over time without resetting the connection.

1.4.1.1 Configuration Files in RestFS

In RestFS, when a file is created, it is created as a pair consisting of a resource and a configuration file that are bound to each other. For example, if a file called "GoogleSearch" is created, then a companion configuration file called ".GoogleSearch" will also be created in skeleton form.

Next, this skeleton is populated manually to contact a specific web service. In the example shown in 1.20, the resource file has been configured to contact the Google search service and perform a GET HTTP request when the utime

```
<?xml version="1.0" encoding="UTF-8"?>
<RestfulSetting>
  <FsMethod>utime</FsMethod>
  <WebMethod>get</WebMethod>
  <FormName></FormName>
  <Resource>ajax/services/search/web?v=1.0&amp;q=Brett%Favre
  </Resource>
  <Host>ajax.googleapis.com</Host>
  <Port>80</Port>
  <OAuthTokenPath></OAuthTokenPath>
</RestfulSetting>
```

FIGURE 1.20
An example RestFS configuration file for a Google Search

filesystem call is performed on the GoogleSearch file. When this occurs, RestFS will make a call to the web service and place the results in the resource file.

The Web Application Description Language (WADL) [8] has been proposed as a REST-ful counterpart to the Web Service Definition Language (WSDL) [3]. We are currently investigating ways to use WADL in conjunction with RestFS, in particular, to populate RestFS configuration files from WADL service descriptions.

1.4.1.2 Implementation of Configuration Files in RestFS

Since RestFS is implemented as a NOFS application filesystem, implementing files that are represented as XML is straightforward. The individual elements are implemented as accessors and mutators in a Java class called RestfulSetting in Figure 1.21. These settings objects are managed by the resource files that we will discuss shortly.

1.4.1.3 Resource Files in RestFS

As stated before, resource files in RestFS contain the state of a current request or response with a web service. Resource files can be configured to be triggered to respond to web service calls upon being opened, before deletion, when the resource file's timestamp is updated, before the resource file is read from, and after the resource file has been written to. This triggering capability is accomplished through the implementation of the NOFS IListensToEvents interface. With this interface, the RestFS resource file is notified by NOFS when actual calls to FUSE are encountered. Once a triggering call is encountered, the algorithm in Figure 1.22 is run.

When the triggering call is made on the resource file, RestFS will check the current contents of the file. If the file contains a JSON object, the object will be parsed and passed as arguments to the web service call. For example, the JSON object {"description" : "stu-

```
@DomainObject
public class RestfulSetting extends BaseFileObject {
   private String _method;
   public String getMethod() { return _method; }
   public void setMethod(String value) { _method = value; }

   private String _formName;
   public String getFormName() { return _formName; }
   public void setFormName(String value) { _formName = value; }

   private String _port = "";
   public String getPort() { return _port; }
   public void setPort(String value) { _port = value; }

   private String _host = "";
   public String getHost() { return _host; }
   public void setHost(String value) { _host = value; }

   private String _resource = "";
   public String getResource() { return _resource; }
   public void setResource(String value) { _resource = value; }

   private String _oauthTokenPath = "";
   public String getOAuthTokenPath() { return _oauthTokenPath; }
   public void setOAuthTokenPath(String value) {
      _oauthTokenPath = value;
   }
}
```

FIGURE 1.21
The RestfulSetting NOFS domain object

```
RespondToEvent(event_type, settings, current_file_data) {
   if(settings.triggering_call == event_type) {
      response = IssueWebRequest(settings.URI,
         settings.WebMethod, current_file_data);
      SetCurrentFileData(response);
   }
}
```

FIGURE 1.22
RestFS resource file triggering algorithm

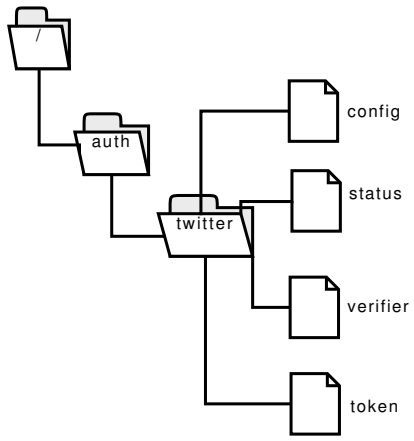

FIGURE 1.23
An example of an OAuth configuration in RestFS

```
<?xml version="1.0" encoding="UTF-8"?>
<OAuthConfigFile>
  <Key>asdf3244dsf</Key>
  <AccessTokenURL>https://api.twitter.com/oauth/access_token
  </AccessTokenURL>
  <UserAuthURL>https://api.twitter.com/auth/authorize
  </UserAuthURL>
  <RequestTokenURL>https://api.twitter.com/oauth/request_token
  </RequestTokenURL>
  <Secret>147sdfkek</Secret>
</OAuthConfigFile>
```

FIGURE 1.24
An example OAuth configuration file for Twitter

dent", "name": "Joe"} would translate to the URI http://host/service?description=student&name=joe.

1.4.1.4 Authentication in RestFS

As many REST-ful web services support the OAuth authentication model, we decided to add special OAuth file and folder types to assist in establishing authorization for web services. In RestFS, there is one special folder '/auth' in the root of every mounted RestFS filesystem. When a folder is created in the '/auth' folder, a config, status, verifier, and token file are created. The config file takes the OAuth API-Key, secret, and set of URLs to communicate with to establish an authorization token. These fields are typically provided by the service provider for a REST-ful web service.

```
<OAuthTokenFile>
  <AccessToken>2534534asdf2348</AccessToken>
  <RequestToken>aql2343</RequestToken>
  <TokenSecret>adfjdsl24522</TokenSecret>
</OAuthTokenFile>
```

FIGURE 1.25
An example OAuth Token file

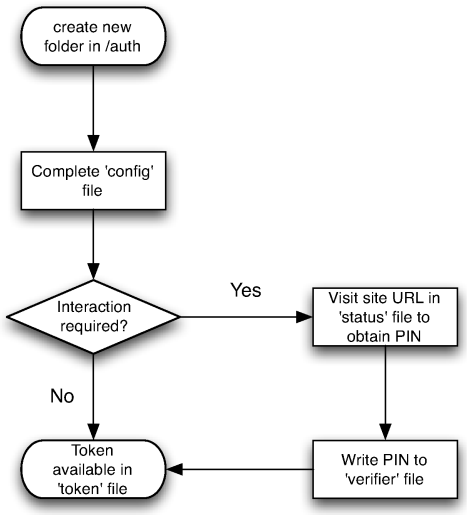

FIGURE 1.26
The RestFS authentication process

Once all of the appropriate fields are written to the configuration file, RestFS will contact the web service to obtain authorization. Depending upon the implementation there are a few possibilities. If the service requires human interaction to accept a PIN or pass a CAPTCHA test, the URL for that step will be written to the 'status' file. If the service provides a PIN, it should be written to the 'verifier' file. Once this process is complete, the 'token' file will be populated with the OAuth access and request tokens for use in further communications. An example of this token file can be seen in Figure 1.25.

Once authorization is successful, the token file can be referred to in any configuration file by path reference in the OAuthTokenPath element. If the configuration file contains a valid token file, RestFS will handle any call to the resource file using the appropriate OAuth token. The user of the resource file then, does not need to worry about authentication any further. This process is summarized by figure 1.26.

1.4.1.5 Putting it All Together

With these three types of files: authentication, configuration, and resource, it is possible to connect to and work with a web service through filesystem calls. If several resource files are created, it is possible to work with several web services and to send multiple requests and compose multiple responses locally using UNIX command line tools or through small programs.

1.5 Summary

With RestFS and NOFS, we have demonstrated how web services can be abstracted and composed in an arbitrarily deep hierarchy through the implementation and use of filesystems. We have shown how the filesystem can be used as a connector layer to translate filesystem calls into web service calls and how this can allow for local and external composition of web services. We have also shown how application filesystems can be used to provide a user-friendly interface for web services to provide validation and more complex structure. Finally, we have shown how the two approaches can be combined to provide effective representations of web services through the filesystem interface.

In our deeper exploration of NOFS, we discussed how the Naked Objects design principles can be used to build filesystems and how the dependency inversion approach simplifies filesystem design. We also explored several example filesystems and explained how NOFS handles translating requests from FUSE to operations on a domain model.

While exploring RestFS, we discussed the challenges of translating web service authentication to the filesystem interface, how configuration and resource files are separated, and how best to use RestFS to expose web services through external programs or scripts.

Bibliography

[1] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2:39–59, 1984.

[2] Matt Bone, Peter F. Nabicht, Konstantin Läufer, and George K. Thiruvathukal. Taming XML: Objects first, then markup. In *Proc. IEEE Intl. Conf. on Electro/Information Technology (EIT)*, May 2008.

[3] R Chinnici, J-J Moreau, A Ryman, and S Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language. W3C Recommendation, June 2007. Available from http://www.w3.org/TR/wsdl20.

[4] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006.

[5] Joe Fialli and Sekhar Vajjhala. Java architecture for XML binding (JAXB) 2.0. Java Specification Request (JSR) 222, October 2005.

[6] R. Fielding, H. Frystyk, Tim Berners-Lee, J. Gettys, and J. C. Mogul. Hypertext transfer protocol - HTTP/1.1, 1996.

[7] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[8] Marc J. Hadley. Web application description language (WADL). Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2006.

[9] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Extensible file system (ELFS): an object-oriented approach to high performance file I/O. In *OOPSLA '94: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 191–204, New York, NY, USA, 1994. ACM.

[10] Joe Kaylor, Konstantin Läufer, and George K. Thiruvathukal. Online layered file system (OLFS): A layered and versioned filesystem and performance analysis. In *Proc. IEEE Intl. Conf. on Electro/Information Technology (EIT)*, May 2010.

[11] Joe Kaylor, Konstantin Läufer, and George K. Thiruvathukal. RestFS: A FUSE filesystem to expose REST-ful services. http://restfs.googlecode.com/, 2010–2011.

[12] Joe Kaylor, George K. Thiruvathukal, and Konstantin Läufer. Naked object file system (NOFS): A framework to expose an object-oriented domain model as a filesystem. Technical report, Loyola University Chicago, May 2010.

[13] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment.* Prentice Hall Professional Technical Reference, 1983.

[14] Ted H. Kim and Gerald J. Popek. Frigate: an object-oriented file system for ordinary users. In *COOTS'97: Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 9–9, Berkeley, CA, USA, 1997. USENIX Association.

[15] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proc. Summer USENIX Technical Conf.*, pages 238–247, 1986.

[16] R. Pawson. *Naked Objects.* PhD thesis, Trinity College, Dublin, Ireland, 2004.

[17] Jan-Simon Pendry and Marshall Kirk McKusick. Union mounts in 4.4BSD-lite. In *TCON'95: Proc. of the USENIX 1995 Technical Conf.*, pages 3–3, Berkeley, CA, USA, 1995. USENIX Association.

[18] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[19] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in Plan 9. *SIGOPS Oper. Syst. Rev.*, 27(2):72–76, 1993.

[20] M. Szeredi. Filesystem in userspace. http://fuse.sourceforge.net, February 2005.

[21] K Thompson. *UNIX implementation*, pages 26–41. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[22] J. Whitehead and Y. A. Goland. WebDAV: A network protocol for remote collaborative authoring on the web. In *ECSCW 1999*, 1999.

[23] Eugene Y. C. Wong, Alvin T. S. Chan, and Hong Va Leong. Xstream: A middleware for streaming XML contents over wireless environments. *IEEE Trans. Softw. Eng.*, 30:918–935, December 2004.