3-2011

RestFS: Resources and Services are Filesystems, Too

Joseph P. Kaylor

Konstantin Läufer
Loyola University Chicago, klaeufer@gmail.com

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

Recommended Citation

Kaylor, Joseph P.; Läufer, Konstantin; and Thiruvathukal, George K.. RestFS: Resources and Services are Filesystems, Too. , Proceedings of Second International Workshop on RESTful Design, Hyderabad, India, http://dx.doi.org/10.1145/1967428.1967439.

RestFS: Resources and Services are Filesystems, Too

Joe Kaylor Konstantin Läufer George K. Thiruvathukal
Department of Computer Science
Loyola University Chicago
Chicago, IL 60611 USA
{jkaylor,laufer,gkt}@etl.luc.edu

ABSTRACT

We have designed and implemented RestFS, a software framework that provides a uniform, configurable connector layer for mapping remote web-based resources to local filesystem-based resources, recognizing the similarity between these two types of resources. Such mappings enable programmatic access to a resource, as well as composition of two or more resources, through the local operating system's standard filesystem application programming interface (API), scriptable file-based command-line utilities, and inter-process communication (IPC) mechanisms. The framework supports automatic and manual authentication. We include several examples intended to show the utility and practicality of our framework.

1. INTRODUCTION

The broader context for this paper 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 [3] and WebDAV [18]) and message-oriented protocols.

By exploiting 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 is 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 that this hierarchical compositionality of resources will facilitate the construction of complex, robust resource- and service-oriented software systems, and we hope further to substantiate our position by including several case studies.

Leveraging our prior work on the Naked Objects Filesystem (NOFS) [9], which exposes object-oriented domain model functionality as a Linux filesystem in user space (FUSE) [16], we have implemented RestFS, 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.

2. RELATED WORK

2.1 Representational State Transfer (REST)

Partly in response to the complexity of the W3C's WS-* web service specifications [2], resource-oriented approaches such as the representational state transfer (REST) architectural style [5] 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,

WS-REST 2011, March 2011; Hyderabad, India

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.

2.2 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 [10]. 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 [13] 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 [14, 15]. In our own research, we have implemented the Online Layered Filesystem (OLFS), which allows for a flexible layering and inheritance scheme through folder manipulation [8]. 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 Plan 9 operating system. Plan 9's filesystem layer, the 9P protocol, is used to represent user interface windows, processes, storage files, and network connections. In Plan 9, 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.

2.3 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 [17]. 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 [12]. 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 [16], ELFS [7], and Frigate [11].

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 v-nodes 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 OS X. 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.

3. COMPOSITION OF WEB SERVICES THROUGH THE FILESYSTEM

In this section, we study the role filesystems can play in the composition of web-based resources and services.

3.1 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, POST, PUT, and DELETE calls over HTTP to a web service URL. On a local filesystem, executable services can be invoked by loading and executing programs from the local filesystem.

3.2 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 which 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.

```
#!/bin/bash

URL_ARGS=`echo $@ | sed 's/ /%20/g'`
FILE_NAME=`echo $@ | sed 's/ /_/g'`
RESOURCE="ajax/services/search/web?v=1.0&q=$URL_ARGS"
RESOURCE=`echo $RESOURCE | \
     sed -e 's~&~\&amp;~g' \
         -e 's~<~\&lt;~g' \
         -e 's~>~\&gt;~g'`

touch $FILE_NAME
./configureResource .$FILE_NAME \
        fs:utime web:get \
        host:ajax.googleapis.com \
        resource:$RESOURCE
touch $FILE_NAME
cat $FILE_NAME
```

Figure 1: A script for issuing a Google search with RestFS

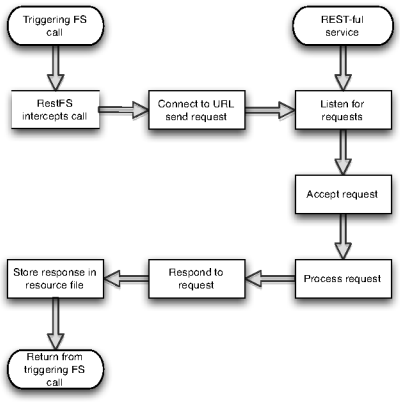

Figure 2: The timeline of a RestFS web service call

The Bash shell script shown in Figure 1 is responsible for configuring the resource, triggering the request, and parsing the results. The timeline of this type of interaction is shown in Figure 2.

Another example usage of RestFS is with the Yahoo! PlaceFinder service. This example is similar to the Google search example. The configuration file is set up with the URI for the web service, and the utime system call is used to trigger the web request. The shell script shown in Figure 3 configures the RestFS file, triggers the web service call, and parses the results.

With our implementation of resource files in RestFS, interaction with remote web resources is similar to 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.

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

```
#!/bin/bash

URL_ARGS=‘echo $@ | sed ’s/ /%20/g’‘
FILE_NAME=‘echo $@ | sed ’s/ /_/g’‘
RESOURCE="ajax/services/search/web?v=1.0&q=$URL_ARGS"
RESOURCE=‘echo $RESOURCE | \
     sed -e ’s~&~\&amp;~g’ \
         -e ’s~<~\&lt;~g’ \
         -e ’s~>~\&gt;~g’‘

touch $FILE_NAME
./configureResource .$FILE_NAME \
        fs:utime web:get \
        host:where.yahooapis.com \
        resource:$RESOURCE
touch $FILE_NAME

LATITUDE = ‘cat $FILE_NAME \
     sed -e ’s/.*<latitude>//g’ \
         -e ’s/<\/latitude>//g’ \
         -e ’s/<\?.*\?>//g’‘
LONGITUDE = ‘cat $FILE_NAME \
     sed -e ’s/.*<longitude>//g’ \
         -e ’s/<\/longitude>//g’ \
         -e ’s/<\?.*\?>//g’‘

echo "($LATITUDE,$LONGITUDE)"
```

Figure 3: A script for issuing requests to Yahoo! PlaceFinder

the structured data that is necessary for interaction 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 as seen in Figure 4.

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 use for this would be a flexible system to send e-mail alerts for an investment portfolio. One or more resource files can be configured to access RSS news feeds. The results from requests to the resource files can then be filtered by a small script for terms related to a list of stocks or investments that the user is interested in. Additionally, another resource file could be configured to query Yahoo! Finance for up-to-date values for the same list of stocks. A script can review the results for price changes that the user may be interested in. The results of both of these local compositions could be e-mailed to one or more e-mail accounts on an hourly basis. If the user wants to change the RSS feeds involved, they only need to update the RestFS configuration file. Figure 5 illustrates how to accomplish 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 con-

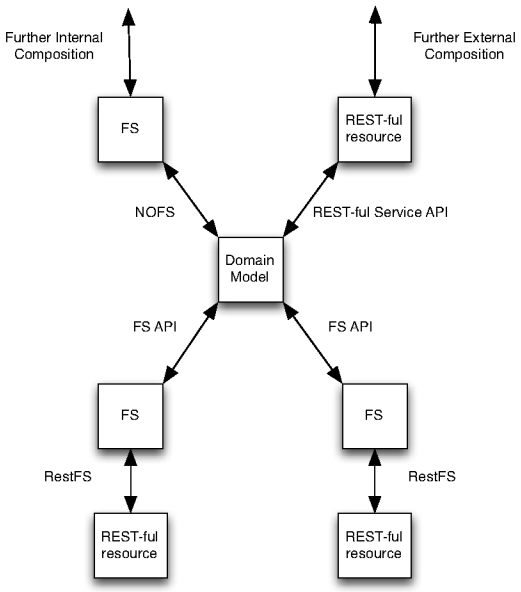

Figure 4: RestFS supports flexible internal and external composition

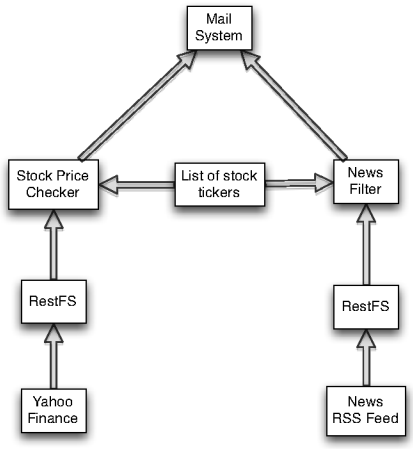

Figure 5: A sample composition of Yahoo! Finance, RSS, and e-mail

cept presents some challenges 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 interact with the application tier and not the presentation tier of a web service, any logic that exists in the presentation tier required for proper communication with the application tier must be duplicated in whatever local composition is made of the web service.

4. ARCHITECTURE OF RESTFS

RestFS was inspired by two other bodies of work: Plan 9's 9P protocol and netfs [14], and Representational State Transfer or REST [5]. 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 RESTful services onto the filesystem. Another important observation that we made at the time is how other forms of inter-process 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 Unix, network communication is performed through system calls like *accept*, *connect*, *listen*, *send* or *recv*. By contrast, in Plan 9, network communication is 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.

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.

```
<?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;amp;q=Brett%Favre
  </Resource>
  <Host>ajax.googleapis.com</Host>
  <Port>80</Port>
  <OAuthTokenPath></OAuthTokenPath>
</RestfulSetting>
```

Figure 6: An example RestFS configuration file for a Google Search

4.1 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.

This skeleton is now populated manually to contact a specific web service. In the example shown in Figure 6, the resource file has been configured to contact the Google search service and perform a GET HTTP request when the utime 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) [6] has been proposed as a RESTful counterpart to the Web Service Definition Language (WSDL) [2]. We are currently investigating ways to use WADL in conjunction with RestFS, in particular, to populate RestFS configuration files from WADL service descriptions.

4.2 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 RestfulSet-

```
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 7: RestFS resource file triggering handler

ting. NOFS manages passing values to and from the methods of this class through reflection. The settings objects in the XML file are managed by the resource files that we will discuss shortly.

4.3 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 handler method in Figure 7 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" : "student", "name": "Joe"} would translate to the URI http://host/service?description=student&name=joe.

After the triggering call, the response from the web service is placed in the resource file. The file can then be read from or written to until another triggering call is made and another response is stored in the file. If the application requires the contents of the resource file to always be up to date for reads, the triggering call can be set to update the file before each read. If the contents of the file need to be sent after each write, the triggering call can be set to to issue the web request after each write.

4.4 Authentication in RestFS

As many RESTful 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; this structure is shown in Figure 8). The config file, shown in Figure 9, 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 RESTful web service.

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

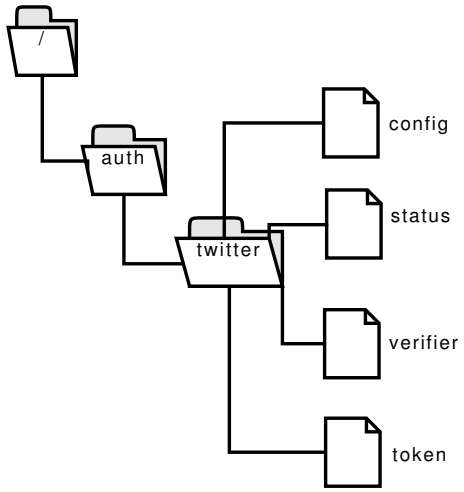

Figure 8: An example of an OAuth configuration in RestFS

```
<?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 9: An example OAuth configuration file for Twitter

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

Figure 10: An example OAuth Token file

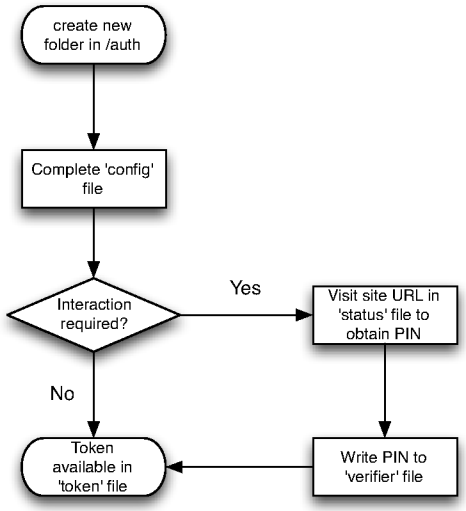

Figure 11: The RestFS authentication process

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 10.

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 in Figure 11.

5. SUMMARY

With RestFS, we have demonstrated how RESTful 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 that translates filesystem calls into REST web service requests and supports local and external composition of web services. We discussed the challenges of translating web service authentication to the filesystem interface, the separation of configuration and resource files, and best uses of RestFS to expose web services through external programs or scripts.

6. FUTURE WORK

For the next revision of RestFS, we intend to add functionality to help to simplify the interaction with and configuration of RestFS.

One improvement we are investigating is to allow WSDL and WADL files to be imported into the filesystem to help automate the creation of several configuration and resource files in one simple operation.

We are also investigating the possibility of allowing plugins or to allow the user to specify an XSLT to transform requests and responses. This improvement will help conform the format of resource files for different web services where convenient. For example, it may be useful to transform requests and responses from one service to be more similar to those of another service so that existing software will be compatible.

In addition to improvements on the filesystem interface, we are also investigating allowing protocols other than HTTP to be used for RestFS. Some possibilities include FTP and SMTP.

We are also considering ways to incorporate HATEOAS [4] into RestFS in a way that would conform with normal filesystem operations. Among others, we are investigating the use of symbolic links and folders to represent the links contained in web service responses.

7. REFERENCES

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

[2] 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.

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

[4] Roy Fielding. REST APIs must be hypertext-driven. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven, October 2008.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] Brian W. Kernighan and Rob Pike. *The UNIX*

Programming Environment. Prentice Hall Professional Technical Reference, 1983.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

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

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

[18] J. Whitehead and Y. A. Goland. WebDAV: A network protocol for remote collaborative authoring on the web. In *Proc. 6th European Conference on Computer-Supported Cooperative Work (ECSCW)*, 1999.