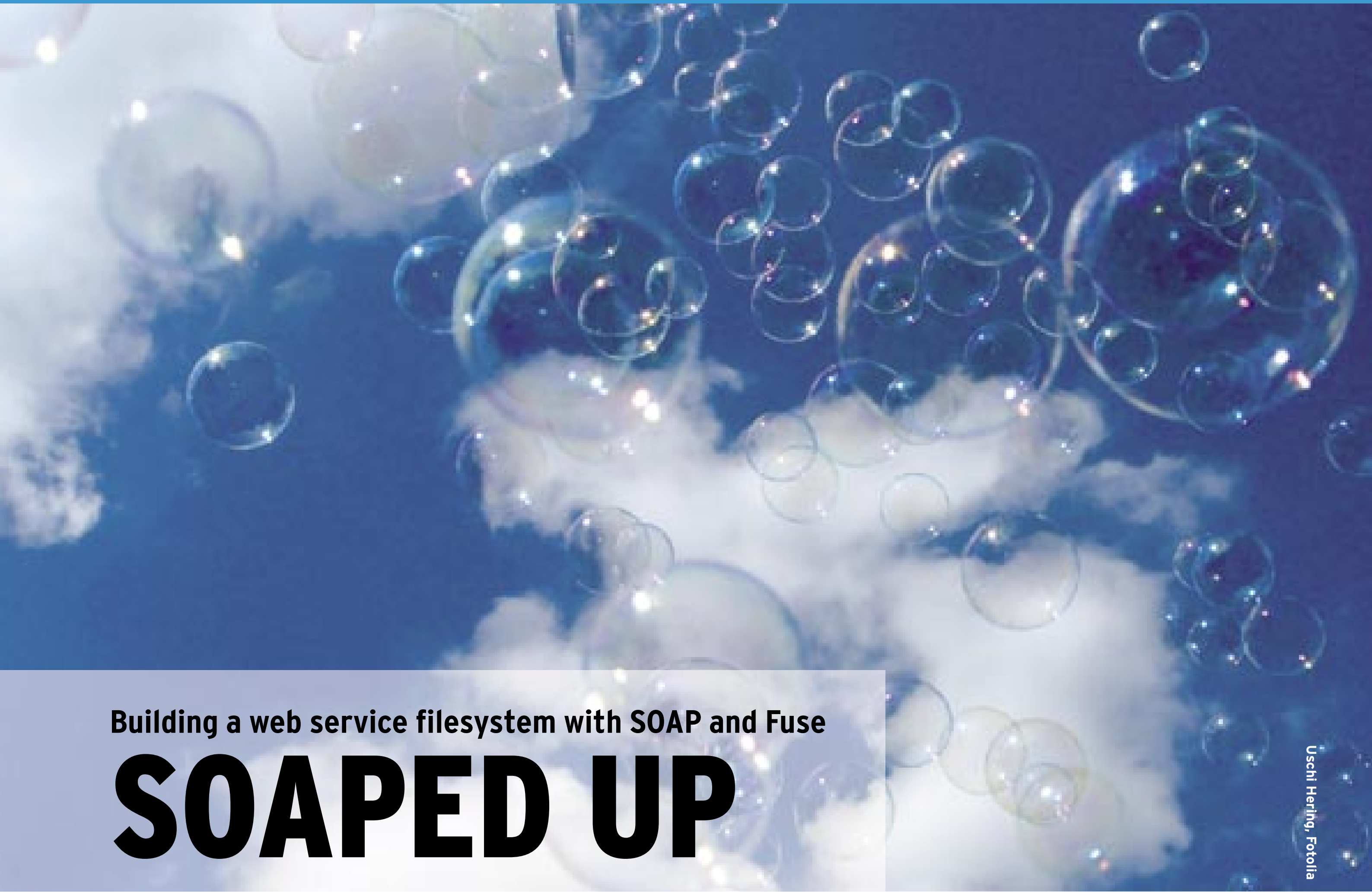

Building a web service filesystem with SOAP and Fuse

SOAPED UP

The Fuse kernel module lets developers implement even the most idiosyncratic of filesystems. We'll show you how to build a filesystem that relies on SOAP to publish data over web services.

BY MATTHIAS FÜLLER, WILLI NÜSSER, AND DANIEL ROSOWSKI

Most Linux users have had some experience with the legacy Network File System (NFS [1]). NFS relies on executing remote procedure calls (RPCs) to give a local machine access to remote data. Unfortunately, RPCs can cause trouble in modern IT landscapes, especially in the interoperability field. For example, RPCs are normally blocked by firewalls, and the procedural programming techniques associated with RPCs aren't state of the art.

Web service technology appears to be the natural successor to RPCs. Web services are object oriented and rely on open standards such as XML. Web services are also typically based on HTTP, and firewalls don't try to block them.

A web service filesystem (WSFS) can run in either kernel space or user space. Because Fuse (Filesystem in Userspace) [2] provides a tried-and-trusted interface for implementing filesystems in user-space applications, we decided to use Fuse as the underpinnings for a WSFS. The goal is to provide a tangible example of web service technologies in Linux. Of

Listing 1: Server Interface WsFsInterface.java

```
package wsfs;
...
public interface WsFsInterface {
        public void chmod(String path, int mode) throws WsFsException;
        public WsFsStat getattr(String path) throws WsFsException;
        public long open(String path, int flags) throws WsFsException;
        public byte[] read(String path, long fh, long offset, int
    size) throws WsFsException;
        public void mkdir(String path, int mode) throws WsFsException;
        ...
}
```

Table 1: Filesystem Functions

access	chmod
chown	create
destroy	flush
fsync	ftruncate
getattr	init
link	mkdir
mknod	open
opendir	read
readdir	readlink
release	rename
rmdir	statfs
symlink	truncate
unlink	utime
write	

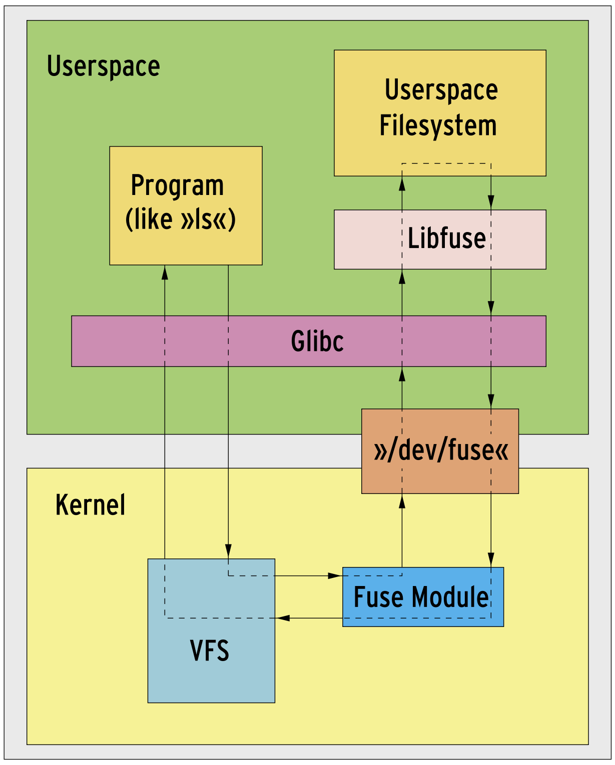

Figure 1: Fuse gives programmers enormous potential for implementing a filesystem in user space. The Fuse architecture is the basis for the web-service-based filesystem in this article.

course, several very robust and useful Linux filesystems already exist. You should treat this example as a proof of concept rather than relying on this filesystem for production use.

How Fuse Works

Figure 1 shows the general structure of a filesystem that relies on Fuse. The latter includes two major components: the Fuse library, Libfuse, and the kernel module, *fuse.ko*. The kernel module helps the virtual filesystem layer recognize Fuse as an independent filesystem type and passes requests for this filesystem to *fuse.ko*.

A single kernel module is all you need for a filesystem like Ext3, which resides totally within the kernel. Fuse additionally needs to communicate with the filesystem implementation, which runs in user space. Libfuse and the */dev/fuse* device file handle this.

Libfuse acts as an abstraction layer that removes the need for direct contact between the filesystem implementation and the device file.

Before you can harness the power of Fuse for your own development work, you need to do some things: install the kernel module, create a user-space implementation of the filesystem (typically by running a C or Java application), then link the application with Libfuse, call an entry method, and run it.

In a C environment, the function name is *fuse_main()*; in Java, the *FuseMount.mount()* method handles initialization. In the background, both of these calls tell Fuse to execute the familiar Linux system call *mount()*. Pass in the third argument, the type of filesystem to mount, to "*fuse*", and your user-space filesystem should be ready for use.

The call to the Fuse library triggers the library to poll the */dev/fuse* file for commands. Every operation on this filesystem – *ls* and *mkdir*, for example – is now passed to the Fuse kernel module via GLibc and the VFS. The VFS stores requests for the */dev/fuse* file, allowing Libfuse to pass them on to the implementation for processing. The results of processing – a directory listing, for example – retrace these steps to their point of origin.

Web Services

After this short introduction to Fuse, it is now time to mount the planned filesystem with the use of a web service. Web services fulfill the main task that RPCs are normally asked to perform: supporting simple communications between distributed applications. However, web service technologies do far more than RPCs in many respects, explicitly addressing applications that use the Internet to communicate. To allow this to happen, they need an open, cross-platform data format on the one hand and open data transmission methods on the other.

Listing 2: Server Implementation WsFs.java

```
package wsfs;
...
public class WsFs implements WsFsInterface {

  final String mountpath = "/tmp";

  public WsFsStat getattr(String path) throws WsFsException {
    File file = new File(mountpath + path);
    if(file.exists()) {
      WsFsStat stat = new WsFsStat();
      stat.mode = file.isDirectory() ? FuseFtype.TYPE_DIR | 0755 : FuseFtype.TYPE_FILE | 0644;
      stat.nlink = 1;
      stat.uid = 500;
      stat.gid = 500;
      stat.size = file.length();
      stat.atime = stat.mtime = stat.ctime = (int)(file.lastModified()/1000L);
      stat.blocks = (int)((stat.size + 511L)/512L);

      return stat;
    }
    throw new WsFsException("No Such Entry").initWsFsErrno(FuseException.ENOENT);
  }

  ...

  public void mkdir(String path, int mode) throws WsFsException {
    File f = new File(mountpath + path);
    if(!f.exists())
      f.mkdir();
  }

}
```

Web service standardizing bodies opted for XML to fulfill the first of these requirements, with HTTP as the typical transport protocol. Programmers do not need to restrict themselves to HTTP, however; they could just as easily opt for a different transport mechanism, such as SMTP or even pure TCP [4].

Web service communications typically start by encoding the data in the XML format specified by the SOAP standard before transmitting. The second step is HTTP-based transport initiated by a *POST* request. The receiving end unpacks the data and processes them. To allow this to happen, both communication partners talk the same language, allowing the receiver to correctly interpret the data sent to it.

The description of the data and methods provided by the receiver is typically explicit. The Web Service Description Language (WSDL) provides a tried-and-trusted XML-based tool to handle this. Figure 2 shows a simplified overview of this relationship. The entity that sends the request is typically referred to as a consumer because it uses the services provided by the receiver (producer).

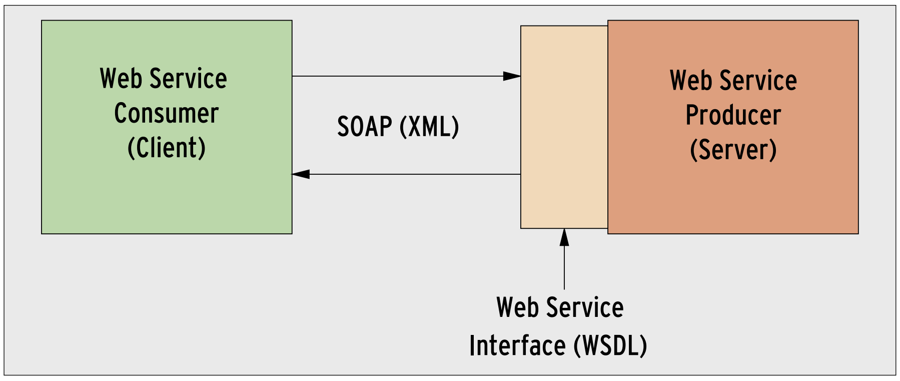
Web Service Consumer (Client)
SOAP (XML)
Web Service Producer (Server)
Web Service Interface (WSDL)

Figure 2: In a web service app, the consumer relies on services provided by a producer. Both sides use SOAP to encode the data in XML, and WSDL to negotiate the correct interpretation.

Beyond the two standards referred to in Figure 2 – SOAP and WSDL – many other standards for web services have developed. Methods exist for the use of transactions [5] or to ensure secure data transmission [6]. None of these additional standards has any effect on the issues under consideration in this article.

Java, C/C++, and scripting languages such as Python are a typical choice of programming language for the implementation. The consumer typically runs as a standalone application and the producer as a service on a SOAP engine such as Apache Axis [7] or Codehaus XFire [8]. Axis and XFire both run as web applications on an application server such as, say, Tomcat. Because today's SOAP engines also are governed by a large number of additional web service standards, they are referred to as web service stacks.

Structure of a WSFS

To implement a web-based filesystem, we have to combine Fuse with web service technology. To do so, we used Java because both Apache Axis and Codehaus XFire are native Java solutions.

The Fuse-Java connector, Fuse-J [3], comprises the *libjavafs.so* library, which uses the Java Native Interface (JNI) to

Interfaces and Java Tie-In

Table 1 shows the methods a fully implemented filesystem needs to possess. They include simple display methods such as *getattr*, which outputs information such as size, owner, and access times, as well as methods for modifying metadata, such as *chmod*, and data access methods such as *read* and *write*.

The Java link Fuse-J [3], which we use as an example of a WSFS in this article, does not support all the calls listed in Table 1. For example, it does not include *access*; the Fuse-J developers have bundled other calls (*getdir* is simply a succession of calls to *opendir*, *readdir*, and *closedir*).

Listing 3: Startup Code for XFire Server

```
import org.codehaus.xfire.XFire;
import org.codehaus.xfire.XFireFactory;
import org.codehaus.xfire.server.http.XFireHttpServer;
import org.codehaus.xfire.service.*;

import wsfs.WsFs;
import wsfs.WsFsInterface;
...

   WsFs wsfs = new WsFs();

   // Create service ...
   ObjectServiceFactory serviceFact = new ObjectServiceFactory();
   Service service = serviceFact.create(WsFsInterface.class);
   service.setInvoker(new BeanInvoker(wsfs));

   // ... and register
   XFire xfire = XFireFactory.newInstance().getXFire();
   xfire.getServiceRegistry().register(service);

   // Start server
   XFireHttpServer server = new XFireHttpServer();
   server.setPort(8191);
   server.start();
...
```

Speeds Compared

Read and write speed is a critical factor in opting for or against a filesystem. A couple of benchmarks revealed some of WSFS's major speed deficits.

A single copy operation within the WSFS via the loopback device achieved a speed of 0.5MBps; a comparable copy with an NFS configuration achieves speeds of 7 to 10MBps, about 20 times the performance. In a production environment, the network would add another restricting factor. The values are reflected in the known overhead of an RPC call compared with a web service call – the difference is typically at least a factor of 10.

support access by Java applications to the C code of Fuse, and the interface description for a Java filesystem. This description includes the Java interface, *fuse.Filesystem*, with methods such as *open()*, *getattr()*, and *read()*.

Figure 3 clarifies the relationship between Fuse, Fuse-J, and the web application. The user-space implementation of the filesystem includes a local Java application, which acts as the Java client for the web service component and handles mounting at one end. The second component, the web service itself, resides on a remote computer.

The web service receives requests from the Java client in the form of SOAP messages. The web service processes the requests and returns the results as SOAP messages.

Implementing the Server

The WSFS here is based on XFire, which is less well known than Apache Axis but an interesting and easy-to-use alternative all the same. The basic approach is the same in both environments, so we concentrated on XFire in this article.

Developing Web Services in general can be done from two different starting points. The first approach calls for the developer to define a publicly visible interface to the web service. This definition includes the structure of the data to be exchanged and the available operations. The definition is normally stored in a WSDL file.

Tools are created from the WSDL file code framework, which programmers can then fill with implementation logic. This approach is often known as *contract first*, meaning that the WSDL description (the contract) is the first item on the roadmap.

The alternative is the pragmatic *code first* approach, which many developers prefer. The first step is to code Java interfaces and their implementations in the normal way and then use the engines to create a WSDL file as needed. The next step is to apply the WSDL file to create clients.

For this article, we will use the code first approach because Fuse and Fuse-J already have all the definitions and interfaces we need. The first task is to create the web service that will wait for and process filesystem requests via SOAP. Listing 1 shows an excerpt from this interface that implements the web service and complies with the filesystem interfaces described previously.

These methods match the methods referred to as *Filesystem* by the central Fuse-J interface, the only modification being that they use byte arrays instead of *ByteBuffer*, for the simple reason that byte arrays are easier to implement as web service calls [9]. The method definitions are to be found in the *WsFs* class; its basic structure is shown in Listing 2.

The implementation in Listing 2 uses the *WsFsStat* class, which can be viewed as an example for the various classes re-

Listing 4: WSFS Client WsFsClient.java

```
...
  public static void main(String[] args) {
      new WsFsClient(args);
  }

  public WsFsClient(String[] args) {

      ObjectServiceFactory serviceFactory = new ObjectServiceFactory();
      Service serviceModel = serviceFactory.create(WsFsInterface.class);

      XFireProxyFactory proxyFactory = new XFireProxyFactory();
      WsFsInterface wsfs = (WsFsInterface) proxyFactory.create(serviceModel,"http://SERVER:8191/WsFsInterface");

      FuseMount.mount(args, new ClientFS(wsfs));
      ...
  }
```

Listing 5: Client Proxy ClientFS.java

```
  ...
  public FuseDirEnt[] getdir(String path) throws FuseException {
      return wsfs.getdir(path);
  }

  public void read(String path, long fh, ByteBuffer buf, long offset) throws FuseException {
      byte[] b = wsfs.read(path,fh,offset,buf.limit());
      buf.put(b);
  }
```

INFO

[1] NFS overview: *http://nfs.sourceforge.net*

[2] Fuse project homepage: *http://fuse.sourceforge.net*

[3] Fuse-J project homepage: *http://sourceforge.net/projects/fuse-j*

[4] Alonso, G., F. Casati, H. Kuno, and V. Machiraju. *Web Services*. Springer, 2004.

[5] Web Services Transactions specifications: *http://www-128.ibm.com/developerworks/library/specification/ws-tx/*

[6] WS-Security: *http://www-128.ibm.com/developerworks/library/specification/ws-secure/*

[7] Apache Axis (versions 1.x and 2): *http://ws.apache.org*

[8] XFire: *http://xfire.codehaus.org*

[9] XFire User Guide, serialization in XFire: *http://xfire.codehaus.org/Aegis+Binding*

[10] Java Native Interface: *http://java.sun.com/docs/books/jni/*

[11] XFire User Guide, Embedded HTTPServer: *http://xfire.codehaus.org/Embedded+XFire+HTTP+Service*

[12] Tomcat: *http://tomcat.apache.org*

[13] XFire User Guide, reference to services.xml: *http://xfire.codehaus.org/services.xml+Reference*

[14] IBM Developerworks on web service performance: *http://www-128.ibm.com/developerworks/webservices/library/ws-best9/*

[15] SOAP Message Transmission Optimization Mechanism (MTOM): *http://www.w3.org/TR/2004/WD-soap12-mtom-20040209/*

quired to extend the corresponding Fuse-J classes. They include additional setter and getter methods for trouble-free use in a web service environment, especially for serializing Java objects in XML streams. More or less all web service tools expect this Bean-style behavior.

The server implementation is fairly easy. The mount path is hard coded in the class definition (*/tmp* in this case). Almost all required file access can be implemented with the use of just the Java standard classes. However, Java does not give us the ability to read file attributes directly. Although JNI [10] solves this problem, this is not the focus of this article, which explains why the code in Listing 2 implements a dummy that allocates fixed values for user privileges.

Finally, to make the whole enchilada available as a web service, the implementation needs to be embedded in XFire. The latter runs as a framework in a servlet container. To allow this to happen, XFire includes the Jetty server [11]; as an alternative, you could use Apache Tomcat [12]. Because the XFire + Jetty variant is easier to configure, we opted to stick with it for this article.

Deploying the WSFS service typically involves several steps, including staging the compiled code and creating a couple of XML configuration files. The files include *web.xml*, which you will know from servlet containers, and an XFire-specific configuration file called *services.xml* [13]. If you opt for Jetty, the configuration is handled by Java code. Start by launching the HTTP daemon on the server; Listing 3 will handle this.

The service factory automatically creates the WSDL file from the *WsFsInterface* passed to it and calls the methods from the *WsFs* implementation class passed to it. After compiling and launching the Java code, the server now listens on port 8191. The server should display the WSDL description of the WSFS in response to a URL of *http://HOST:8191/WsFsInterface?wsdl*. Now it's time to start developing the client.

Implementing the Client

Listing 4 shows sample client code that uses XFire-specific calls. If you use Axis, the only call not to change is the central *FuseMount.mount(args, new ClientFS(wsfs));* call, which mounts the filesystem.

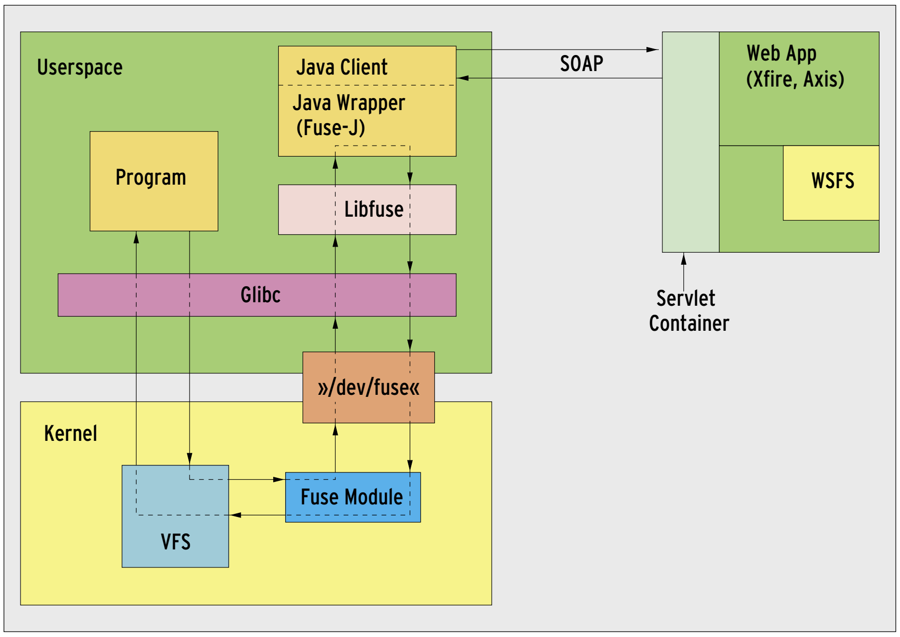

Figure 3: WSFS structure with Apache Axis or Codehaus XFire.

The *ClientFS* type object, which the client in Listing 4 instantiates, is a proxy for the web service.

ClientFS accepts requests and passes the requests in to the web service as SOAP messages. Listing 5 shows an example of this in the form of a modified wrapping to convert, say, *ByteBuffer* to simple byte arrays.

For developers who prefer to avoid complex XML processing, there is good news: The engine, XFire in our case, takes care of managing XML messages and the SOAP protocol. This convenient interface is one of the major reasons that engines such as Axis and XFire have become so widely accepted.

To wrap things up and test the WSFS, you can now launch the central client class, *WsFsClient*; the *mount* command should now show another filesystem. Commands such as *ls* will be passed to the *WsFs* server implementation via SOAP from now on, and a response to reflect the directories and files on the server should be returned.

Conclusions

Developing a simple web-service-based filesystem on Linux is fairly painless thanks to the existing tools. Fuse and Fuse-J provide interfaces for developing a user-space filesystem, and Linux has powerful environments such as XFire and Axis to considerably facilitate work with web service standards and remove the need for extensive coding.

This said, the example shown in this article is not meant to be more than a prototype for two reasons. First, it is fairly simple to implement read-only filesystems. A fully fledged filesystem that supports arbitrary write operations is far more sophisticated (just think about synchronizing access to shared data).

The second reason our implementation remains a prototype is the weakness of web services, as evidenced by the example of a filesystem. The prototype showed extremely slow response in data transfer via the loopback device – refer to the box titled "Speeds Compared."

These figures point to various known issues with web service performance [14]. Many minor system calls, as generated when we ran *ls*, are poison if they trigger extensive XML parsing, as in our case. On the other hand, extremely large data packets can also generate excessive load, especially when transported within the XML message and not in an attachment [15]. ■

THE AUTHOR

Willi Nüßer is Professor for Applied Computer Science at the Fachhochschule der Wirtschaft (FHDW, University of Applied Science) in Paderborn, Germany. Before this appointment, Willi worked for SAP for six years, where he was a developer in the SAP Linux Lab.

Matthias Füller and Daniel Rosowski are students of Technical Computer Science at the FHDW.