Coping with Evolution: Aspects vs *Aspirin*?

Alex Brodsky[*] Dima Brodsky Ida Chan Yvonne Coady
Stephan Gudmundson[†] Jody Pomkoski Joon Suan Ong

Department of Computer Science
University of British Columbia

Abstract

Attempts to evolve a code base in an effective and comprehensible manner can give almost anyone a headache. For example, consider version 2 vs version 3 of FreeBSD's implementation of the Network File System (NFS) [5]. The v2 code base is approximately 10,000 lines, to which the integration of v3 adds over 100 small, scattered clusters of code. Although this code is differentiated from v2 by appropriate compiler-directives and system-wide identifiers, its crosscutting nature adds complexity to the original code, and introduces implicit coupling that poses further challenges for future evolution. This observation is consistent with Lehman and Belady's study showing structural deterioration over successive releases of OS/360 [4].

We are currently trying to determine the material impact aspect-oriented modularity has on the evolution of NFS code. Towards this end, we are studying implementations of versions 2 and 3, along with the specifications of 4 (not yet available in implementation). To date, we have designed an aspect that structures v3 client functionality relative to a v2 implementation for FreeBSD, and are working on OS neutral v4 features that we expect to be highly portable to both FreeBSD and Linux NFS implementations. This short paper presents some of our preliminary work, including (1) an example aspect-oriented implementation of a v4 related feature, replication, developed for a model of NFS, (2) a comparison of this implementation with an interposition approach, and (3) a characterization of how to build highly portable aspect-oriented implementations.

1 Introduction

Can an aspect-oriented modularity make evolution less of a headache? We are currently investigating this question in the context of successive versions of the long-lived Network File System (NFS) [5]. Our work to date has focused on steps in the evolutionary ladder corresponding to NFS versions 3 and 4.

So far, our study has yielded mixed results. For example the integration of v3 into FreeBSD's NFS v2, a code base that is approximately 10,000 lines, adds over 100 scattered clusters of code. We have been able to structure these v3 changes as a single aspect in AspectC [1], with compiler directives capturing changes to data structures – but our current design requires refinement. Although we believe this to be a significant improvement over the original implementation, our intuition is that this single aspect will be better represented as a composition of smaller aspects, improving the clarity of individual features of the v3 implementation. Additionally, as overviewed in Section 6, we believe we may be able to make this aspect composition more portable between operating systems by careful refactoring.

This short paper focuses on a preliminary but representative example of evolution associated with an individual v4 feature, and extrapolates from our experience so far. As the Java prototype of NFSv4 [6] has not yet been released, we implemented our own Java model of NFSv3, *JNFS* [3], and evolved it. The feature we focus on is our own client-specific policy for fault tolerance built on top of v4's support for replication. The characteristics we extrapolate are those associated with aspects that we expect to be highly portable between operating systems.

We begin by providing a high level overview of NFS in Section 2, and follow this with an overview of fault tolerance in Section 3. Our replication aspect is described in Section 4 and compared with a layered approach in Section 5. Section 6 presents a brief characterization of what we feel is necessary to build highly portable aspects and Section 7 concludes with future work.

2 NFSv2: under the hood

The NFSv2 protocol is a simple and stateless protocol. A client uses *file-handles* to access files and directories on the remote server. Requests are typically configured to

[*]Supported by NSERC PGSB
[†]Supported by NSERC PGSA

have an upper bound of eight kilobytes, so high-level requests exceeding this maximum are broken into several lower-level requests. Due to the statelessness of NFS, there is no notion of remote file *open* or *close* operations in the protocol itself. Thus, to access a file a client needs only to have a valid file-handle.

NFS clients must first *mount* the specified remote file system before accessing files on an NFS server. During the mount request the server returns a file-handle for the root of the exported file system, fh_{root}. A file-handle is a 32 byte identifier in NFSv2 that uniquely identifies the file or directory on that server.

To access a file, the client must first perform a *lookup* operation for each component on the path. To access `x/y/z`[1] the client sends the server the root file-handle fh_{root} and `x`; it receives a file-handle for `x`, fh_x. Next it sends fh_x and `y`, and receives fh_y. One more *lookup* is performed with fh_y and `z`. Once the client obtains fh_z it can perform the standard set of file operations on file `z`.

The NFS server must honour the file-handles it has issued. To reduce the overhead of performing a local *open* and *close* on every operation, the server caches recently used file-handles. When a file-handle is evicted from the cache the associated file is closed. The server forgets about file-handles if they have not been accessed for an extended period of time; the amount of time is usually on the order of minutes. If a client queries the server with an unknown file-handle the server returns an error.

2.1 NFSv4

NFSv4 is currently being designed to: (1) improve access and performance on the Internet, (2) strengthen security with negotiation built into the protocol, (3) increase cross-platform interoperability, and (4) facilitate protocol extensions. Although the replication aspect presented in this paper is related to the first goal on this list, improved access, the long-term goal of our work is most akin to the last item on this list – extensibility. That is, we hope to establish the material impact that aspect-oriented implementations of concerns (1) through (3) have on the ability to realize (4).

3 Fault Tolerance

Our approach to fault tolerance uses replicas to mirror all file system writes to a set of servers, instead of just a single server as is the default in NFSv2/v3. Though we do not have a true NFSv4 implementation to build upon, our understanding is that v4 potentially provides support for this brand of fault tolerance by giving clients the ability to obtain a list of servers storing a given file.

[1]We use Unix path syntax. The path delimiters are `/` and the path components are `x`, `y`, and `z`; `x` and `y` are directories and `z` is a file.

Ideally, we would like to make the client impervious to server crashes. Therefore, we further introduce a mechanism that automatically switches to an alternate server should the primary server go down. This switch-over is transparent to the client.

Essentially, to implement this client-specific policy for fault-tolerance, we need to make changes to all functions that read and write from/to the server. This applies to both data and metadata operations. All reads must include the functionality to switch-over in case of failure, and all writes must include this switch-over in addition to the ability to relay the writes to replica servers.

It is important to note that in our model, the client is responsible for replicating the data. If servers crash and come back, it is quite probable that the state between the servers will differ. We do not yet attempt to maintain consistency between the servers.

4 The Replication Aspect

The details of this implementation are described in terms of our current prototype *JNFS* client, with customizations introduced using AspectJ [2]. In our JNFS model, the client supports a standard, OS neutral, NFS interface.

The first part of our replication aspect, shown in Figure 1, introduces data structures and common functionality required to support replication. The two helper methods are used by advice in the aspect. The first helper method, `remap_server` establishes a live replica if the primary server fails while the system is running. The second helper method, `set_servers`, is used during set-up to establish which servers are designated as replicas.

AspectJ pointcuts provide an abstraction for specifying points in an executing program when advice is to run, and parameters available to that advice. We need replication code to affect the flow of events when an *nfsiod daemon*, a thread running on behalf of an NFS client, is processing a request.

The first of these pointcut declarations, `nfsiod_op_cflow`, identifies all points in the executing program that are in the control flow (or *cflow*) of functions whose signatures match the expression `void nfsiod_*( request req )`. Given the naming conventions in our code, this captures execution points when `nfsiod` functions are on the runtime stack. This pointcut is shared by both the read and write operations in the aspect.

Additionally, we specify that these functions take a single parameter of type *request*, which will be bound to `req` in the advice body. We use the request object to extract file handle information.

```
aspect Replication {

    public Vector      fd_entry.fhs;

    public boolean     nfsiod.dead[];
    Vector             nfsiod.replicas;
    int                nfsiod.primary   = 0;

    boolean nfsiod.remap_server(fd_entry f) {
       // select replica and set nfsiod.primary index to it.
    }

    public void nfsiod.set_servers( Vector servers ) {
       // set the vector of servers provided as replicas
    }

    pointcut nfsiod_op_cflow( request req ):
         cflow( calls( void nfsiod_*( req )));
```

Figure 1: Replication aspect

```
    pointcut reads(nfsiod n, JNFS_arg args, request req):
        nfsiod_op_cflow( req ) && ( read_*(n, args) );

    pointcut read_getattrs(nfsiod n, JNFS_getattr_arg args):
        within(n) && calls(public  * JNFS_getattr(args));

    pointcut read_reads(nfsiod n, JNFS_read_arg args):
        within(n) && calls(public  * JNFS_read(args));
```

Figure 2: Read pointcuts

4.1 Reading

The first part of the aspect code associated solely with reading simply establishes the points in the executing code where advice will apply, shown in the three pointcut declarations in Figure 2.

The `reads` pointcut uses `nfsiod_op_cflow` in conjunction with the other two pointcuts to capture points in the program when reading operations are performed on behalf of `nfsiod` methods. Metadata reads are captured by the pointcut `read_getattrs`, and data reads are captured by the pointcut `read_reads`.

The around advice that uses the `reads` pointcut, shown in Figure 3, attaches to all the places where reading activity takes place, and handles the remote exception raised by a failed server by remapping.

In the body of this advice, we use the keyword `proceed` both within the `try` clause and the `catch` clause of the remote exception handling. Proceed continues with the execution of the primary function to which the advice is attached. We use the `try` to catch the exception raised by a failed server, at which point we use the `remap_server` method introduced earlier. The `catch` allows the intended read operation to continue on the newly designated primary server.

In the event that remapping fails, the aspect uses a local function to dispatch the error handling according to the signature of the primary function involved. This dispatching specializes the return type of the advice, defined here to be `Object`.

4.2 Writing

Structure-wise, the code in our replication aspect associated with write is similar to the read. It first establishes the points in the executing program where the advice applies, shown by the pointcut `writes` in Figure 4. Just as in the `reads` pointcut, `writes` uses `nfsiod_op_cflow` in a conjunction with a list of other more specific write-related pointcuts, the complete declarations of which are not shown here.

The around advice for writes, shown in Figure 5, is more complicated than in the case of reads. It has to han-

```
around( nfsiod n, JNFS_arg args, request req ) returns Object:
    reads(n, args, req) {
        // map to proper primary server

        try { // attempt invocation
            return proceed( n, args, req );
        } catch ( RemoteException r ) {
            // if n.remap_server, then proceed
            // else build and return an error.
        }
    }
```

Figure 3: Read advice

```
pointcut writes(nfsiod n, JNFS_arg args, request r):
     nfsiod_op_cflow( r ) && ( write_*(n, args) );

pointcut write_mnt(nfsiod n, JNFS_mnt_arg args):
     within(n) && calls(public * JNFS.JNFS_mnt(args));

pointcut write_setattrs(nfsiod n, JNFS_setattr_arg args):
     within(n) && calls(public  * JNFS.JNFS_setattr(args));

...

pointcut write_lookups(nfsiod n, JNFS_lookup_arg args):
     within(n) && calls(public  * JNFS.JNFS_lookup(args));
```

Figure 4: Write pointcuts

dle remapping, shown by the first `try/catch` in the body, in the same way as the around advice on `reads` does. It also must handle propagation of the write requests to the replicas involved. A helper function dispatches the explicit write request to a given replica according to the signature of the primary function involved.

5 The Interposition Alternative

In addition to providing this functionality as an aspect, we also explored a layered approach. Layering is a common way to extend system interfaces. The approach involves interposing the layer between the interface and the application. All calls from the application are intercepted by the layer and eventually forwarded to the interface after some mutation of the arguments. The values returned by the interface may be further mutated before being returned to the caller. Usually, the layer utilizes additional state to perform the modifications.

For our purposes the fault tolerance mechanism must perform two distinct but related tasks: the replication of all mutator operations (operations that have side effects) and the transparent remapping of servers and file-handles upon failure of a primary server. Using the interposition approach we achieved these requirements in the following manner.

The `JNFSReplicator` class implements the same interface (`JNFS`) as the `JNFSServer`; the client is passed a replicator object instead of a server object. Each remote invocation is intercepted by the replicator object, the request is modified and finally forwarded to the primary server. Additionally, mutator requests, like writes and file creation, are forwarded to the replicas as well. If a primary server fails in the course of a request, a `RemoteException` is caught by the replicator, the replicator selects a server from the set of replicas to act as the primary and dynamically remaps the file-handles. The file-handles exchanged between the client and the primary server need to be mapped to the file-handles of the new primary server as well as the replicas.

The replicator layer accomplishes the first function (replication) by maintaining a vector of replicas and a map that translates a primary handle (file-handles returned by the primary server) to a vector of replica handles (file-handles returned by the replica servers). On an invocation of a mutator request, the request is first forwarded to the

```
around( nfsiod n, JNFS_arg args, request req ) returns Object:
     writes(n, args, req) {
         // map to proper primary server

         try {
             rs = (JNFS_res) proceed(n, args, req);
         } catch ( RemoteException r ) {
             // try to remap servers and restart invocation
             // otherwise build and return an error.
         }

         for( i = 0; i < size; i++ ) {
             // map arguments' handles to proper replica

             try {
                 res = delegate_to(server, ..., (JNFS_arg)args);
                 // if operation fails, replica is out of synch
                 //    so mark it as dead
                 // if result returns a new file handle, add it to the map
             } catch ( RemoteException r ) {
                 // if replica dies, mark it as dead, and don't use it.
             }
         }
         return( rs );
     }
```

Figure 5: Write advice

primary server. Upon the successful completion of the request the primary file-handles embedded in the request are mapped to the replica file-handles and the request is then forwarded to the corresponding replica. If the request generates a new file-handle, like a *mount* or a *create* request, a new mapping is created. The second function (transparent remapping) makes use of the same data structure.

The replicator layer intercepts all `RemoteException` exceptions thus providing transparent fault handling. When an invocation to the primary server fails, the server is marked as dead and one of the replicas is chosen in its stead. Since all file-handles issued by the defunct server must be honoured, an additional mapping must occur. Before the request is forwarded to the primary server, the file-handles embedded in the request must be mapped to the file-handles of the current primary server; the same map that is used for replication suffices. Unfortunately, the map is both necessary and expensive.

If transparent fault handling is to be achieved, any file-handle issued by a primary server must be honoured regardless of the state of the server that issued it. Hence, a map must be maintained that translates file-handles from a previously working primary server to the file-handles of the current primary server. Given the number of different files that can be requested by a typical client, the amount of state necessary to implement such a mapping can be quite large. The fact that the interposing layer must mirror all the file-handles held by the client implies that such layers have a large memory overhead.

Additionally, the layer relies not only on the interfaces of the server, but also on the structure of the arguments being passed to the server. This is an endemic problem because such layers must inevitably mutate the contents of the messages before forwarding them to the servers. Any changes to the interface or the internal structure of objects requires modifications to the interposed layer as well.

5.1 Approach Comparison

The amount of additional state varies greatly between the two approaches. In both cases, all valid file-handles must be honoured by the fault handling and replication layer. Without a mechanism to inform the layer about discarded handles, the layer must continue to store handles and file-handle maps for every file-handle issued. To say that the corresponding state is large would be an understatement. Hence, the interposition method is extremely space inefficient.

The aspect-oriented approach uses the existing file state to store the replication maps required for fault handling and replication. When the file descriptor and the file-handle are discarded, the corresponding replica map is discarded simultaneously. The amount of additional state is proportional to the number of replicas and the number of currently open files; the number of files open at any time is orders of magnitude less than the total number of files.

6 Building Highly Portable Aspects

Layers are portable – should we expect the same from aspects? Aspects that target functionality adhering to a strict interface, like NFS, can capture structure in such a way that applies well to all operating systems that support the interface. In order to build highly portable aspects, we believe it is important to: (1) target the right interface, (2) establish appropriate aspect/component boundaries, and (3) think ahead in terms of composition.

6.1 Targeting the right interface

Below the NFS client interface, the implementations of NFSv2/v3 in FreeBSD and Linux are very different. The OS neutral quality of our replication aspect is that its pointcut interface is essentially built above the only common denominator between these implementations. Although we have not yet specialized this general implementation for either OS, we do not anticipate making significant structural changes when doing so.

6.2 Establishing appropriate boundaries

The replication aspect is not a complete implementation of fault tolerance. As mentioned in Section 3, making a resurrected server consistent is an important piece of associated functionality not implemented by the aspect. We envision this to be an component of our brand of fault tolerance, integrated with the rest of the system through the replication aspect. Differentiating this kind of aspect/component functionality has not always proven to be trivial in our experience.

6.3 Thinking ahead in terms of composition

Evolution can involve (1) sets of extensions to existing features, and (2) new features. Structuring a set of changes to extend a major feature, F, as a single aspect (as we did initially for v3) may fail to individualize important elements of this implementation. We believe this subtle differentiation between a single large aspect and a closely-knit composition of small aspects to be of particular importance with respect to portability – where feature F is subject to subtle variations between OS implementations.

7 Future Work and Conclusions

Though we believe an aspect-oriented modularity can facilitate evolutionary changes relative to a traditional approach – our results are not yet conclusive. We do not expect significant performance degradation with an AOP approach, but to date we have only focussed on proof-of-concept within a minimal model. Establishing the impact aspect compositions have on the overall conceptual complexity of the system is also a high priority.

Beyond v4 features, other important evolutionary changes that we believe may benefit from an aspect-oriented modularity include: client-side peer-to-peer cooporative caching, in which clients acquire additional cache resources by negotiating with other clients; and client-side encryption and forwarding, where clients in a local LAN are considered secure, but the link to the server is not, thus idle clients are used to encrypt and forward data to the server. The need to support this kind of high-level diversity is increasingly important as today's LAN environments breed more client-specific constraints. The question of whether support will come in the form of structured crosscutting – versus something with a child-proof cap – remains to be seen.

References

[1] AspectC. www.cs.ubc.ca/labs/spl/aspects/aspectc.html.

[2] AspectJ. www.aspectj.org.

[3] Alex Brodsky, Dima Brodsky, Ida Chan, Yvonne Coady, Jody Pomkoski, and Gregor Kiczales. Aspect-oriented incremental customization of middleware services. Technical Report TR-2001-06, University of British Columbia, 2001.

[4] L.L. Lehman and L.A. Belady. Program evolution. *APIC Studies in Data Processing*, (27), 1985.

[5] A. Osadzinski. The network file system (NFS). *Computer Standards & Interfaces, North Holland*, 8:45–48, 1988.

[6] Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson, and Robert Thurlow. The nfs version 4 protocol. In *International SANE 2000 (System Administration and Networking) Conference*, May 2000.