Aspect-Oriented Incremental Customization of Middleware Services

Alex Brodsky[*] Dima Brodsky Ida Chan Yvonne Coady Jody Pomkoski
Gregor Kiczales

Department of Computer Science,
University of British Columbia,
{*abrodsky,dima,idachan,ycoady,jodyp,gregor*}@*cs.ubc.ca*

Abstract

As distributed applications evolve, incremental customization of middleware services is often required; these customizations should be unpluggable, modular, and efficient. This is difficult to achieve because the customizations depend on both application-specific needs and the services provided. Although middleware allows programmers to separate application-specific functionality from lower-level details, traditional methods of customization do not allow efficient modularization.

Currently, making even minor changes to customize middleware is complicated by the lack of locality. Programmers may have to compromise between the two extremes: to interpose a simple, well-localized layer of functionality between the application and middleware, or to make a large number of small, poorly localized, invasive changes to all execution points which interact with middleware services. Although the invasive approach allows a more efficient customization, it is harder to ensure consistency, more tedious to implement, and exceedingly difficult to unplug. Thus, a common approach is to add an extra layer for systemic concerns such as robustness, caching, filtering, and security.

Aspect-oriented programming (AOP) offers a potential alternative between the interposition and invasive approaches by providing modular support for the implementation of crosscutting concerns. AOP enables the implementation of efficient customizations in a structured and unpluggable manner. We demonstrate this approach by comparing traditional and AOP customizations of fault tolerance in a distributed file system model, JNFS. Our results show that using AOP can reduce the amount of invasive code to almost zero, improve efficiency by leveraging the existing application behaviour, and facilitate incremental customization and extension of middleware services.

Keywords: customizing middleware, aspect-oriented programming

1 Introduction

Availability requirements for distributed applications often evolve over time. In such cases it is not always practical or necessary to make heavy-weight changes to the middleware platform. For example, bolstering the fault tolerance of an RMI-based application can be accomplished by adopting a new more sophisticated middleware; but making such a significant platform change requires extensive modification of the base application code involved, as well as a substantial investment in new middleware infrastructure.

A reasonable alternative is to "roll your own" customizations, perhaps even prototyping various middleware features before committing to a more heavy-weight commercial middleware platform. Currently, developing one's own customization involves a mix of two traditional approaches. In the *invasive* approach customization changes are made to the core application code while in the *interposition* approach the customizations are embedded in a layer that is interposed between the middleware layer and the application.

The advantage of the former approach is that application behaviour can be leveraged to yield efficient implementations. The problem is that even conceptually simple customizations can be a major undertaking due to their lack of locality – the implementation of these systemic customizations is inherently scattered throughout the core

[*]Supported by NSERC PGSB

functionality of the application involved.

The interposition approach is more modular than the invasive approach and as a result has the notable advantage of being unpluggable. Its main disadvantage is that this separate layer lacks the inherent advantages of the invasive approach – closer integration with the application code which leverages application behaviour to facilitate a more efficient implementation.

Ideally, we would like a modular, unpluggable solution that can leverage application behaviour.

Recently, the aspect-oriented programming (AOP) community [10] has focused attention on *crosscutting concerns*, which are elements of a system that cut through its primary modularity. They have proposed linguistic mechanisms allowing implementation of crosscutting concerns as first-class modules called *aspects* [2, 15, 12, 1]. We propose that an AOP approach to customization of distributed applications can yield the same level of modularity as the interposition approach and provide the ability to leverage application behaviour; thus it yields unpluggable, efficient, and modular implementations.

To validate our proposal we first developed an NFS-like distributed application (JNFS) with RMI as its middleware layer; we shall argue that JNFS is a reasonable model for testing our proposal. Second, we focus on one representative customization, fault tolerance, and argue that it is a reasonable representative of common customizations. Third, we present three different approaches to incremental customization of JNFS: the interposition approach, the invasive approach, and the AOP approach. Finally, we establish the relative impact these approaches have on the application base-code, and the efficiencies each can afford. The results of this preliminary study indicate that modular, non-invasive, and efficient customizations can be implemented using an AOP approach.

The organization of the paper is as follows: Section 2 describes the model and argues its validity, Section 3 describes and compares the traditional approaches, Section 4 describes our AOP approach, Section 5 describes our evaluation of the approach, Section 6 describes related work, Section 7 discusses open issues and future work, and Section 8 concludes.

2 The Model

To demonstrate how aspects can be applied in the middleware domain, we chose a model that closely resembles a real system.

We built a simple NFS-like distributed application [14], JNFS, which uses RMI for its middleware. NFS is a network file system that permits users to perform remote reads and writes on a centralized file server. The NFS protocol is a simple get/put segment interface. Like NFS, JNFS is composed of two parts, a server and a client.

We believe JNFS is a reasonable model to validate our preliminary proposal for several reasons. First, it is a client/server architecture like many other applications. Second, JNFS has a static and strict interface that is not easily extensible. Interfaces for many client/server systems tend to be fixed so that a wide variety of clients and servers can inter-operate. Third, the main functionality of JNFS is to perform remote operations, such as reading and writing data. Other applications such as distributed calendars, network information services (NIS), and video streaming also have these three traits. Because it generalizes to a common class of middleware applications, we feel that the JNFS application is a reasonable model.

The goal of many common customizations is to improve the robustness of an application. Customizations such as security, caching, and filtering also improve the usability of an application. Fault tolerance also falls into this category. Ideally such customizations should be unpluggable, modular, and efficient because different customizations are required for different environments. Thus, given that fault tolerance has similar traits as other common customizations we feel it is a good representative of a common customization.

2.1 The NFS Protocol

In modern operating systems the NFS protocol is implemented using XDR [21] and RPC [22]. XDR/RPC is a low level and mostly featureless type of middleware. We used Java and RMI because it has considerably more features as an environment for middleware; thus, it is closer to modern middleware. It was also far easier to develop a simple NFS server/client in Java since AspectJ already exists, although a tool that can do aspects in *C* is forthcoming[1].

We used RMI to implement version 2 of the NFS protocol [5]. The remote interface consisted of all the functions in NFS version 2 and the necessary functions in the mount protocol.

2.1.1 Under The Hood

The NFS protocol is a simple and stateless protocol. A client uses *file-handles* to access files and directories on the remote server. All requests usually have an upper bound of eight kilobytes. Due to the statelessness of NFS there is no notion of file *open* or *close* operations in the protocol itself. The nature of *open* and *close* impart state

[1] AspectC is currently being developed at UBC (`http://www.cs.ubc.ca/labs/spl/aspects/aspectc.html`).

Function	Description
`mount( remote, local )`	Attach a remote file system into a clients local directory structure.
`umount( path )`	Detach an attached remote file system from a client.
`fd open( path )`	Open the file specified by the path for file operations.
`close( fd )`	Close the file for file operations pointed to by fd.
`read( fd, buffer, len )`	Read len bytes into the buffer from a file pointed to by fd.
`write( fd, buffer, len )`	Write len bytes from the buffer to a file pointed to by fd.
`seek( fd, offset )`	Seek to position offset in the file pointed to by fd.
`mkdir( path )`	Make a directory pointed to by the path.
`rmdir( path )`	Remove the directory pointed to by the path.

Table 1: The nine functions exported by the client to test and evaluate it.

in the same way as an on/off switch. Thus, to access a file a client needs only to obtain a valid file-handle for it.

To access files on an NFS server a client must first *mount* the specified remote file system. During the mount request the server returns a file-handle for the root of the exported file system, $\mathrm{fh}_{\mathrm{root}}$. A file-handle is a 32 byte identifier in NFS version 2 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`[2] the client sends the server the root file-handle $\mathrm{fh}_{\mathrm{root}}$ and `x`; it receives a file-handle for `x`, $\mathrm{fh}_{\mathtt{x}}$. Next it sends $\mathrm{fh}_{\mathtt{x}}$ and `y`, and receives $\mathrm{fh}_{\mathtt{y}}$. One more *lookup* is performed with $\mathrm{fh}_{\mathtt{y}}$ and `z`. Once the client obtains $\mathrm{fh}_{\mathtt{z}}$ it can perform the standard set of file operations on file `z`.

The server must honour the file-handles it has issued. To reduce the overhead of performing an *open* and *close* on every operation, the server also 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.

Reads and writes are performed in chunks of eight kilobytes. Thus, to read a ten kilobyte file would require two NFS read operations.

2.2 The Server

Our model server design closely resembles that of the userlevel NFS server for Linux [19]. The server implements the remote interface, including the mount protocol, and only excluding links and softlinks. We use a Java `Hashtable` for the file-handle cache and use the hashcode generated by the `File::hashCode` function for the file-handle. This approach would not work in practice because the hashcode is computed on the file path. Therefore, there is a possibility that the hashcode will not be unique, but it is sufficient for our purposes.

[2] 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.

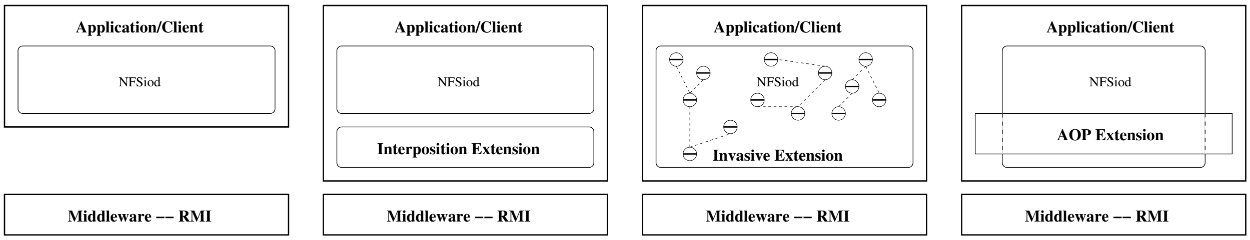
2.3 The Client

We modelled the JNFS client as a typical NFS client running in an operating system kernel. The `nfsiod` portion of the client uses RMI and the defined remote interface to communicate to the server. The client exports a unix-like interface, enabling us to test and evaluate it. There were nine functions that enabled us to create, read, and write files and create and delete directories. The nine functions are listed in Table 1.

Like the server, the client also has limitations and that is why it is a model rather then a real world application. JNFS is a network file system. In the real world the client would be located within the system so that users are able to access it like a local file system. Currently JNFS does not interface with any known system. Therefore, it is difficult to use except through a scripting interface built for testing purposes.

2.4 The Customization

Fault tolerance has several components but we focus on two, reliability and availability. Both components are similar but address slightly different issues. They rely on each other for completeness; moreover, replication is a necessary condition for availability. Reliability encompasses the problem of data loss and ensuring that none occur. We implement reliability through a replication extension that mirrors all file system writes to a set of servers. Availability addresses the issues that arise when servers crash. Ideally, we would like the client to be impervious to server crashes. Therefore, we handle availability by having a mechanism that automatically switches to an alternate server should the primary server go down. This



(a) No customization

(b) Interposition - Modular but inefficient implementation due to the inability to leverage application behaviour.

(c) Invasive - The implementation is able to leverage application behaviour and thus is efficient but it is not modular or unpluggable.

(d) Aspect-Oriented - The implementation is modular, unpluggable, and efficient because it is able to leverage application behaviour.

Figure 1: The four implementations of the JNFS client.

switch-over is transparent to the client.

To implement fault tolerance the configuration of the entire system must also change. Initially the system has a star topology and uses unicast. When fault tolerance is introduced both the topology and the communication protocol changes. Fault tolerance requires that the client broadcast all operations that modify data to a set of replication servers. Thus the topology becomes one-to-many and the communication mechanism ideally changes to multicast for efficiency.

Replication and availability are mostly client-centric since it is the client that requires these features. In our model no modifications were necessary on the server side. Modifications would be necessary if the server partook a more active role in the customizations; for example providing consistency between servers.

In our model, the client is responsible for replicating the data. This mirrors the fact that in in a real system the client would also be potentially responsible for ensuring that all the replication servers have a single consistent view of the world. If servers crash and then come back then it is quite probable that the state between the servers will differ. We do not attempt to maintain consistency between the servers because we are just demonstrating replication customizations.

Figure 1 shows four versions of the JNFS client. The first implementation, Figure 1a, is of a client with no customizations. The second implementation uses the interposition approach to add the fault tolerance customization. Figure 1b shows the same customization as a separate layer in the client. The customization is relatively clean but inefficient because state has to be duplicated and retained by the interposition layer. A more efficient implementation is shown in Figure 1c, done using the invasive approach. However, the code is not modular and difficult to unplug. Figure 1d shows an implementation that is done using aspects. This implementation is efficient but still retains its modularity and unpluggability.

	Interposition	Invasive
Unpluggable	yes	no
Leverage Application Behaviour	no	yes
Modular	yes	no

Table 2: Comparing the two approaches.

3 Traditional Approaches

There are two commonly used approaches for customizing middleware: the invasive approach and the interposition approach; both approaches have their advantages and disadvantages (see Table 2). Ideally, we would like an approach that has all the benefits of the two approaches with none of the disadvantages.

3.1 The Interposition Approach

Interposition is one of the traditional approaches for extending interfaces. The approach involves interposing a 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 as well 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 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.

3.2 The Invasive Approach

Mirroring the file-handles is an inviolate requirement of the interposition layer. However, given the unidirectional nature of function invocation, it is difficult to mirror the file-handle state. Without additional mechanisms the layer must store a set of file-handles that is significantly greater than the actual working set.

For example, the client acquires many file-handles during a lookup operation that are used only once and then discarded. A client disposes of a file-handle by simply forgetting it, e.g., freeing the memory, or annihilating it along with the stack frame. Unfortunately, the mapping layer can have no knowledge of such actions unless it is explicitly notified by the client. As a result, the layer must use a conservative policy and retain all file-handles; this is expensive.

One solution is to leverage application behaviour by placing explicit code in the core client code to notify the layer every time a file-handle is destroyed. Thus, the layer actually mirrors the client in the bijective sense of the word. However, this type of leveraging is invasive and requires extreme care during core modifications. Thus, the invasive approach affords better efficiency in exchange for higher complexity.

Initially both fault handling and replication are performed on the client side. However, it is quite conceivable that replication may be a server side operation. In this case the same map data structure would have to be duplicated on both hosts. One possible solution is to observe that since servers crash rarely, trading time for space during the remapping is reasonable option. Instead of storing the full map, a client need only store the map for the root file-handles. When the primary server fails, the client can acquire new file-handles from the new primary server. The remapping time would become proportional to the number of file-handles owned by the client, however, the required map would shrink to a negligible size. Such a solution would require additional core code that not only kept track of the origin of each file-handle and caught server failure exceptions, but reloaded file-handles, i.e., repeated lookup operations upon server failure.

As a hybrid solution, the client's map can be treated as a cache. If a file-handle map is evicted from the cache and then requested, the same code to re-acquire file-handles from a new primary server can be used to reload the cache with the required mapping; as before, we trade a decrease in space utilization for increased request latency.

As we shall see, implementing any of the preceding mechanisms would require significant modification to the core code and would be impossible if we were to use an interposition library. Subsequent modifications to the current customization and subsequent customizations would become correspondingly more complex because of the non-local scattering of the customization code.

The former approach, interposition, suffers from inefficiency due to its inability to leverage application behaviour. The invasive approach solves this problem by embedding code into the core. However, the resulting code becomes scattered, non-modular, and hard to maintain. An ideal solution would preserve the modular properties of the interposition approach, while providing the ability to leverage application behaviour.

4 Aspect-Oriented Customization

We developed the AOP version of these customizations to combine the advantages of the interposed and invasive techniques: it is both unpluggable and closely integrated with the application base-code. This section starts with a brief introduction to AspectJ, the AOP language we used in our implementation and proceeds on to the implementation details for this approach.

4.1 AspectJ Overview

The aspect-oriented implementation of replication presented here uses AspectJ – a simple AOP extension to Java. These extensions support modular implementation of crosscutting functionality by allowing code that would otherwise be spread across several functions to be localized and to share context. It also permits the addition of new elements to classes. In both cases, AspectJ allows us to modify the base-code non-invasively, i.e. without the programmer seeing extra code interwoven throughout the source. A precompiler known as a weaver automatically places the intruding code at specified locations defined by *pointcuts*, an AspectJ construct. Therefore, the intruding code remains a separate module in the source, only becoming scattered and invasive at the byte-code level (which the programmer does not need to read).

In AspectJ, aspect code known as *advice* interacts with primary functionality at function call boundaries. Advice is declared to run *before*, *after* or *around* a designated call. New fields and methods can be attached to existing classes through the use of AspectJ's introduction mechanism. These new elements behave exactly like those of the existing components, as if they were defined originally in the class declaration. However, only code that knows these extensions exist can make use of them. This normally would only be aspect code, as these elements are defined in the aspect. The central elements of the language are a means for designating and attaching advice to particular function calls, for accessing parameters of those calls, and for introducing new elements into existing classes.

4.2 Aspect-Oriented Implementation

Overall, only a small portion of our AOP implementation of these customizations relies on AspectJ extensions. The rest is ordinary Java code from the original implementation, and can be thought of as a straight refactoring of the interposed and invasive approaches described in Section 3. The distinctive feature of this implementation is purely structural, not functional. In the invasive code, fault tolerance is tightly integrated and scattered among the core functions involved to improve efficiency. The AOP code implements the same tightly integrated, efficient approach, but as a modular, unpluggable aspect– normally achieved only by the interposition method.

This section focuses on the structural details of our AOP implementation. In the interest of space, the complete code for helper functions is not shown, but overviewed instead.

4.3 Remapping

The first part of our replication aspect (see Figure 2) implements the additional functionality required to handle remapping the primary server to one of the backup replicas.

To maintain state, the aspect introduces all the data structures required to support replication efficiently, as opposed to the hash tables used in traditional interposition techniques. Therefore, we leverage application behaviour like in the invasive approach– the data structures are thrown away by the client on *close* operations. Although this state can be thought of as being added to other classes, such as `fd_entry` and `nfsiod`, the non-aspect code cannot access anything introduced by the aspect, because it does not know about these new elements. Thus these changes are unpluggable, even though they involve modification of base classes.

The first method, `remap_server`, is a helper method used to establish a live replica in the event that the primary server fails. It resets important nfsiod state, such as `serv`, which always identifies the primary server. This method is essentially a helper function, called only by advice within this aspect (see Figures 3 and 4).

The aspect also introduces a new method `set_servers` into the `nfsiod` class. This allows the applications that

```
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
    }
```

Figure 2: Replication aspect

```
    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 ) {
                // try to remap servers and restart invocation
                // otherwise build and return an error.
            }
        }
```

Figure 3: Read advice

use fault tolerance and replication functionality to inform the aspect which servers are designated as replicas.

4.4 Reading

Essentially, the crucial element of efficient fault tolerance is the seamless remapping to a replica in the case of a crashed primary server. The additional state and methods introduced by the aspect will enable us to remap efficiently.

The first part of the aspect code associated directly with reading simply establishes the points in the executing code where this extension will apply. In AspectJ, this is done with the four *pointcut* declarations shown in Figure 5.

AspectJ pointcuts are used to access parameter lists as well as precisely specify when to run advice. We only want the replication code to affect the flow of events when the nfsiod 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 r )`. Given the naming conventions in our code, this captures execution points when nfsiod functions are on the runtime stack. Also, we specify that these functions take a single parameter of type *request*, which will be bound to `r` in the advice body. We look at the request object `r` to extract file descriptor information. This pointcut is shared by both the read and write operations in the aspect.

The second pointcut, `reads` uses `nfsiod_op_cflow` in conjunction with the final two pointcuts to clearly capture

```
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 4: Write advice

points in the program when reading operations are performed on behalf of nfsiod methods. Metadata reads are captured by the third pointcut, `read_getattrs`, and data reads are captured by the fourth pointcut, `read_reads`.

The around advice that uses the `reads` pointcut is shown in Figure 3. Essentially, this advice attaches to all the places where reading activity takes place, and handles the remote exception raised by a dead server by remapping. We can say that around *intercepts* the calls to the primary functionality and performs additional tasks, which is exactly like how an interposition library operates.

Within 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 restarts the execution of primary function to which the advice is attached. In the try, it allows us to catch the exception, using the remap_server method introduced earlier. In the catch, it allows us to then restart the intended read operation on the newly designated primary server.

In the event that the remapping fails, the aspect uses a local function (not shown here) to multiplex the error handling according to the signature of the primary function involved in order to return the right type (note that the return type for the around is simply Object).

4.5 Write

Structure-wise, the write operation is very similar to the read. It first establishes the points in the executing program where the advice applies, shown by the pointcut `writes` in Figure 6. 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 declarations of which are not shown here.

The around advice for writes shown in Figure 4 is somewhat more complicated than in the case of reads. It has to handle remapping, as 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 all of the replicas involved. The helper function `delegate_to` (not shown here) essentially dispatches the explicit write request to a given replica according to the signature of the primary function involved.

4.6 Summary

Our AOP implementation has best features of both the invasive and interposition methods. Since we can introduce additional state and code, we can leverage application behaviour, as afforded by invasive techniques. Since all new

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

pointcut reads(nfsiod n, JNFS_arg args, request r):
    nfsiod_op_cflow( r) &&
    ( read_getattrs(n, args) || read_reads(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 5: Read pointcuts

```
pointcut writes(nfsiod n, JNFS_arg args, request r):
      nfsiod_op_cflow( r ) &&
     ( write_setattrs(n, args) || ... || write_lookups(n, args) );
```

Figure 6: Write pointcuts

	Interposition	Invasive	AOP
Unpluggable	yes	no	yes
Leverage Application Behaviour	no	yes	yes
Modular	yes	no	yes

Table 3: Summarizing the approaches.

elements can only be used by code with knowledge of the extension, i.e., only local code, our approach involves writing modular, tightly bound, easily unpluggable code characteristic of interposition methods.

5 Analysis

In this section we evaluate the success of the AOP approach versus the traditional approaches. Our evaluation is based on the criteria listed in Table 2: unpluggability, ability to leverage application behaviour, and modularity (see Table 3); leveraging application behaviour facilitates efficient implementation of customizations. As a prelude, we look at the number of lines of code required by each of the approaches to confirm that the AOP approach is competitive in that respect also. We discuss each of these in turn.

5.1 Amount of Code

The interposition approach required approximately 375 lines of code (not including comments) all contained within a single file `JNFSReplicator.java`. An additional couple of lines of code in the start up code was required to pass the replica locations to the replicator layer; the invasive approach requires a comparable amount of code. The AOP approach (see `Replicator.java`) required about 300 lines of code. These results support our intuition that the customizations in question require at least 300 lines of code regardless of what approach is used; the code in all the approaches is strikingly similar.

From the preceding numbers we conclude that in terms of code size the AOP approach is no more expensive than any of the other approaches and in our case actually reduced the code size by about 25%. The question is how unpluggable, efficient, and modular are the approaches.

5.2 Unpluggability

A customization is unpluggable if it is localized, modular, and well defined, i.e., few code modifications are required to remove the customization from the code base. In the case of the interposition approach a one line code change would be required to remove the customization from the core functionality. Put another way, a programmer need not be familiar with the code in order to remove the customization. In the case of the invasive approach, we identified 4 locations in 2 classes in `nfsiod.java` and `fd_entry.java` where fields would need to be removed, and 33 locations in 20 functions in `nfsiod.java` where code would need to be deleted. Hence, the interposition approach facilitates unpluggable customizations while the invasive approach does not. We note, that in our model the client (`nfsiod.java`) is only 520 lines of code. Thus, the changes would affect about 8% of the code.

The AOP approach also requires only one line of code changed (to specify the replicas) in order to plug/unplug the customization. Apart from this, core code is not modified in any way and no modifications are required to return it to its original state; recompiling the code without the aspects is all that is necessary.

5.3 Leveraging Application Behaviour and Efficiency

The ability to leverage application behaviour greatly affects the efficiency of the implementation. The interposition approach lacks this ability while the invasive approach and the AOP approach support it. As a result the space efficiency of our implementations varied greatly.

The amount of additional state varies greatly between the two traditional approaches. In all cases, all valid file-handles must be honoured by the fault handling and replication layer. Without the use of invasive reaper functions, called by the application, to inform the layer about discarded handles, the layer must continue to store file-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, without additional invasive functions, is extremely space inefficient.

The invasive approach uses the existing file descriptors 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.

The AOP approach also leverages the file descriptors. Hence, the same benefits derived from the invasive approach are reaped in the AOP approach. The AOP approach is as space efficient as the invasive approach. Hence, having the ability to leverage application behaviour is extremely beneficial.

5.4 Modularity

We analyze the three approaches in terms of the benefits traditionally associated with modular programming [17, 8].

5.4.1 Independent Development

In both the interposition and AOP approaches, the interface between the primary functionality of the client and the fault tolerant functionality is static and abstract. We can easily determine what functions in the main code the customization knows about and what arguments are passed. Since the interface is static and abstract, it is possible to develop the main code and the customization independently.

While the aspect also depends on the application code, in order to leverage internal data structures, it depends on the existence of the data structures rather than their structure.

On the other hand, the invasive approach does not facilitate independent development. Both the core code and the customization are interleaved within the same file, which makes independent development impossible.

5.4.2 Comprehensibility

The interposition approach allows us to easily reason about the behaviour of both the core functionality and the customization. Each part of the system can be reasoned about separately since the flow of control in a layered system is well understood.

In the AOP approach decomposing the invasive implementation into the main client functionality and fault tolerance aspect allows us to reason about the different parts and their respective interaction separately. Since the aspects do not modify core application state and do not affect the control flow of the core code, reasoning about each part separately is possible.

In the invasive approach, the customization is part of the main code. Although the customization does not affect the core code flow, because it is interwoven with the main code, reasoning about each part separately is impossible.

6 Related Work

We have shown that AOP can be used as a method to provide application specific customization of middleware. Our work stems directly from the approach to separation of concerns supported by the language extensions developed by the AspectJ project [2]. A number of general approaches to separation of concerns in complex systems have emerged in the last few years. Work on subject-oriented programming [16] and hyperspaces [15] is aimed at composing hierarchies of concerns and focuses on multiple dimensions of concerns. Composition filters [1] separate objects into internal parts and interfaces to which filters can be applied. The SADES project [18] presents work showing that for complex systems a hybrid approach using different separation of concerns methods may be better than a single method.

Quarterware [20] also provides for application specific customization by first abstracting the individual middleware services and then allowing application specific implementations or extensions. The Quarterware architecture does not however use any separation of concerns techniques. Our approach did not re-implement RMI but rather extended it at the client side.

The ORB architecture described in [9] provides a descriptive language that clients use to describe application specific policies. The components of the ORB have multiple implementations that publish service guarantees using the same language. The ORB adjusts its implementation at run-time to implement the policy. In [4] a reflective architecture uses a per-object meta-space and meta-models to compose a middleware implementation. These are in contrast to our use of AOP and compilable code that implements the needed changes at the client without modifying the middleware implementation.

CORBA ORBs provide for a limited amount of application specific customization through Object Adaptors, interceptors, filters, smart proxies, and smart stubs, depending on the ORB implementation. Since many of these extensions are vendor specific, portability of CORBA applications between CORBA ORBs is reduced. Our customizations should work in any java RMI compliant layer.

This work is also related to work in the OS community that explores methods of incremental customization. The degree to which the structure of OS code can support incremental customization was one of the fundamental questions posed [7] but left unanswered by research in extensible systems such as SPIN [3] and Kea [23]. Efforts to customize policy through reflection range from all OS policy in Apertos [24], to file system services [13] and virtual memory management [11] using metaobject protocols. Most recently, we have been able to show how an AOP approach can be used to modularize path-specific customization of prefetching code in FreeBSD [6].

7 Open Issues and Future Work

Efficiency and scalability are some essential open issues of AOP's general utility in customization of middleware services. Although we do not expect significant performance differences between the invasive and AOP approaches, to date we have only focussed on proof-of-concept within a minimal model. We have not yet tested a more sophisticated model to be conclusive.

Composibility is another important issue we have yet to address in our single aspect implementation. We plan to investigate the complexities introduced by the composition of multiple aspects, e.g., how composed aspects catch the same exception. We intend to research this further by building a modular, sophisticated structure of aspects aimed at supporting the development of essentially middleware independent applications.

8 Conclusion

To improve application robustness, "roll your own" customizations are commonly used to extend middleware. Traditionally, a mix of interposition and invasive approaches is the de facto implementation strategy. While the former offers unpluggability, simplicity and good modularity, it can be grossly inefficient. On the other hand the invasive approach facilitates efficient implementation by leveraging the existing state of the application. We have proposed a third approach, aspect-orient programming, which allows us to create a modular and unpluggable implementation that at the same time leverages the application state reaping the efficiency of the invasive approach.

To test our proposal we constructed an NFS like system (JNFS) on top of RMI, which is representative of a large class of distributed applications. Using the different approaches we added a fault tolerance customization that implemented replication and fault handling; both are good representatives of a large class of customizations that increase system robustness.

The amount of code (300 to 400 lines of code) required to implement customization was comparable in all cases; the AOP approach yielded the fewest lines of code. The interposition approach yielded a clean but inefficient implementation, requiring orders of magnitude more state than the invasive approach. On the other hand, the invasive approach would yield an efficient but hard to maintain and non-modular implementation, i.e., the approach required 37 different locations in the core code to be modified.

The AOP approach yielded a modular and unpluggable implementation with the efficiency of the invasive approach. A single modification to the original source is required and the level of familiarity with application code need not be as great as in the case of the invasive approach. The same approach can be applied to a myriad of customizations including: logging, filtering, tunneling, security, compression, caching, prefetching, and fault detection.

Acknowledgments

Many thanks to Brian de Alwis, Christina Green, Stephan Gudmundson, Norm Hutchinson, Vibha Sazawal, and Andrew Warfield for their insightful comments on drafts of this paper.

References

[1] M. Aksit and B. Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. In *OOPSLA AOP'98 workshop position paper*, 1998.

[2] AspectJ. www.aspectj.org.

[3] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, 1996.

[4] Gordon S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, September 1998.

[5] Brent Callaghan. *NFS Illustrated*. Addison-Wesley, Reading, MA, USA, 2000.

[6] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on th Foundations of Software Engineering (FSE-9)*, 2001. To appear.

[7] Peter Druschel. Efficient support for incremental customization of OS services. In *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*, December 1993.

[8] Stevens et al. Structured design. *IBM Systems Journal*, 13, 1974.

[9] B. Jrgensen, E. Truyen, F. Matthijs, and W. Joosen. Customization of object request brokers by application specific policies. In *Proceedings of the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing (Middleware2000)*, November 2000.

[10] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[11] Keith Krueger, David Loftesness, Amin Vahdat, and Thomas Anderson. Tools for the developement of application-specific virtual memory management. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 1993.

[12] K.J. Lieberherr. *Adaptive Object-Oriented Software: the Demeter Method with Propagati on Patterns*. Boston: PWS Publishing Company, 1996.

[13] Chris Maeda. *Service Decomposition: A Structuring Principle for Flexible, High Performance Operating Systems*. PhD thesis, CMU, 1997.

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

[15] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the Hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2000.

[16] Harold Ossher, William Harrison, Frank Budinsky, and Ian Simmonds. Subject-oriented programming: Supporting decentralized development of objects. In *Proceedings of the 7th IBM Conference on Object-Oriented Technology*, 1994.

[17] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1972.

[18] Awais Rashid. A hybrid approach to separation of concerns: The story of sades. In *Proceedings of 3rd International Conference on Meta-Level Architectures and Seperation of Concerns Refelection 2001*, September 2001.

[19] Mark Shand, Donald Becker, Rick Sladkey, Orest Zborowski, Fred N. van Kempen, and Olaf Kirch. Universal nfs server for linux, 1999. ftp://linux.mathematik.tu-darmstadt.de/pub/linux/people/okir/.

[20] Ashish Singhai, Aamod Sane, and Roy H. Campbell. Quarterware for middleware. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS)*, May 1998.

[21] Sun Microsystems. *XDR: External data representation standard*. Sun Microsystems, June 1987. RFC1014.

[22] Sun Microsystems. *RPC: Remote Procedure Call Protocol Specification*. Sun Microsystems, 1988. RFC1050.

[23] Alistair C. Veitch and Norman C. Hutchinson. Kea - a dynamically extensible and configurable operating system kernel. In *Proceedings of the 1996 Third International Conference on Configurable Distributed Systems (ICCDS)*, 1996.

[24] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 1992.