Can AOP Support Extensibility in Client-Server Architectures?

Yvonne Coady, Alex Brodsky, Dima Brodsky, Jody Pomkoski,
Stephan Gudmundson, Joon Suan Ong, Gregor Kiczales
University of British Columbia

Abstract

Extensible client-server software requires a clear separation of core services from those that are customizable. This separation is difficult, as these customizable features tend to crosscut the primary functionality of the core services. We believe that an aspect-oriented approach to client-server architectures supports extensibility in a way that is more flexible than traditional approaches. Our experiment focuses on clients within a distributed file system that dynamically negotiate with servers for adaptive delayed-write and prefetching behaviour.

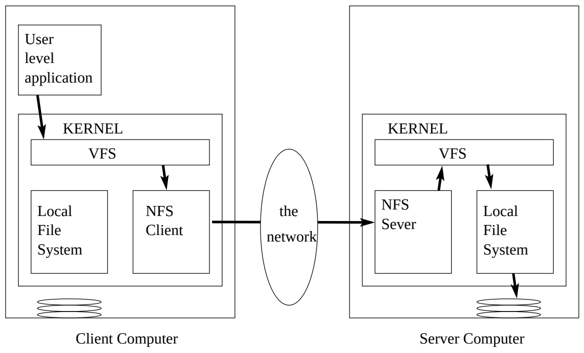

Figure 1: NFS software architecture.

1 Introduction

Client-server architectures boil down to *getting* or *putting* information between a client node and a server node. Associated with simply getting and putting however, are issues of performance and concurrency. These issues introduce complexity to core client-server services because their implementation is crosscutting. As a result, they are not readily amenable to change.

For example, Sun's Network File System (NFS) has provided a sufficient and stable set of core distributed file system services since the mid-80s. Even though NFS is often considered a *de facto* standard, it is well known that it does not address scalability issues that are increasingly important in modern distributed applications. Although these issues, such as performance and concurrency control, are not considered part of core functionality, they ultimately set a practical-limit on the size of NFS-based distributed file systems.

Extending client-server architectures to respond to growing demands, however, adds considerable complexity because these extensions involve invasive, crosscutting changes to what are now stable, core functions. We believe that better support for separating crosscutting concerns greatly facilitates extensibility in client-server architectures because essential core services can remain stable while extensions are independently developed. To investigate this further, we have experimented with designing aspects to structure some of these kinds of concerns in a simple model of NFS.

This paper describes our aspect-oriented design of NFS. We show how to structure the implementation of cache consistency and prefetching using aspects, and how AOP allows us to extend these services in a number of significant ways that would otherwise require invasive changes to core functionality.

1.1 NFS overview

NFS provides transparent access to remote files; application code accesses a remote file in exactly the same manner as it accesses a local file. The common interface used by both local and remote file systems is defined by the the Virtual File System (VFS) layer in the kernel.

At a high level, NFS is divided into two parts: client and server. Typically, both client and server modules are installed within the operating system kernel. Each computer in an NFS network can act both as client and server, accessing remote files and exporting local files for remote access respectively.

As shown in Figure 1, VFS requests referring to files in a remote file system are translated by the NFS client module and communicated to the NFS server module on the computer holding the relevant file. The client module is thus responsible for transferring blocks of files to and from the server and caching blocks within the shared *file buffer cache* of the client system. Major performance enhancing operations associated with this cache are described below.

1.2 Caching: prefetching, delayed-write and write-through

The performance of an application's request to read or write is greatly enhanced if it can use the file buffer cache instead of an expensive disk request. It is important to note that in NFS there are two caches involved: the cache on the client and the cache on the server. Major cache related activities include *prefetching*, *delayed-write* and *write-through* operations.

Prefetching predicts read accesses and fetches disk blocks into the buffer cache in advance of any explicit request. Delayed-write postpones writing changed blocks to disk until it is more cost effective, or necessary, to do so. Write-through, the alternative to delayed-write, transfers each write request immediately to disk. In NFS, the client cache uses a delayed-write strategy to improve performance, but write-through is necessary at the server because a failure of the server could result in undetected loss of client data.

1.3 Consistency and performance

Cache consistency can be a problem when multiple clients are concurrently accessing a shared file. On a single machine, there is a single cache maintained by the operating system. With NFS, several clients on different machines may simultaneously access the same remote file, creating independently cached copies of portions of files. This can lead to consistency problems, as writes by one client do not update cached copies at other clients.

NFS attempts to minimize inconsistencies, but does not guarantee the same semantics as local file systems. This is because forcing this distributed system to behave as a centralized one would penalize performance, in most cases, unnecessarily.

To improve performance, sometimes at the cost of consistency, NFS clients rely on prefetching and delayed-write strategies to maximize the use of caches. Requests to prefetch from the server and write blocks to the server are asynchronously handled by special *daemon* processes[1] that provide this kind of 'behind-the-scenes' client-server communication.

1.4 Inflexibility of current structure

The use of separate daemon processes to implement asynchronous prefetching and delayed-write behaviour accomplishes a coarse granularity of separation in NFS:

- In the case of reading, a daemon is notified after each read request and prefetches blocks from the server into the client cache.
- In the case of writing, a daemon delays sending blocks to the server until they are filled.

Work performed by the daemons is not absolutely necessary for correct operation, but it reduces the chances of inconsistency and is required for satisfactory performance.

Although this separation supports some degree of independent development, extensibility is limited by the fact that the interaction between core functionality and daemon processes is embedded deep within the core functionality of the client, where daemon activity is invoked. This structure is thus inflexible because extensibility of non-core concerns requires invasive changes to core NFS implementation. As a result, it is exceedingly difficult to fine-tune these increasingly important concerns in NFS-like distributed applications.

2 Aspect-Oriented NFS

We are currently sketching out aspects for an NFS-based client-server architecture using an AspectJ-like AOP language, AspectC [3, 1]. Specifically, we want to find out if can design aspects that effectively isolate issues associated with consistency and performance – potentially allowing us to extend these implementations without requiring invasive changes to core functionality.

In the long term, we are particularly interested in growing the client population to include many different approaches for delayed-write and prefetching operations. That is, we envision some 'smart' clients dynamically negotiating with servers for adaptive application of performance and consistency requirements. To date however, we have focused on high level design issues, presented here.

2.1 The experiment

We focused on two specific extensions to explore issues of consistency and performance:

1. **Consistency extension:** shorten the interval in which caches may be inconsistent.
2. **Performance extension:** aggressively prefetch from disk into the server cache when sequential access is detected on the client.

The following sections describe each of these extensions, and contrasts the original implementation with an aspect-oriented approach for each.

3 Consistency extension

In order to ensure some level of cache consistency in NFS, a timestamp-based heuristic is used to invalidate cached

[1] a user-level process that performs system tasks

```
aspect validation_check{

  pointcut validation_check_points( file_id fid ):
      calls( int nfs_client_read( fid, ..) ) || calls( int nfs_client_write( fid, ..) ) ||
      calls( int nfs_client_cache_check( fid, ..) ) || calls( void daemon_write( fid, ..) ) ||
      calls( void daemon_prefetch( fid, ..);

  before( file_id fid ): val_check_points( fid ) {
    get_fresh_timestamp( fid );
  }
}
```

Figure 2: AspectC pseudo-code for client-based consistency extension.

```
aspect active_invalidation{

  after( file_id fid ): calls( int nfs_server_read( fid, .. )) {
    update_state_info( fid->state_info );
  }

  after( file_id fid ): calls( int nfs_server_write( fid, .. )) {
    check_state_and_invalidate( fid->state_info );
  }
}
```

Figure 3: AspectC pseudo-code for server-based consistency extension.

blocks. Each client holds a timestamp indicating when their copy of the file was last modified at the server. If the server's last modified time is more recent than the timestamp held by a client, the cached blocks at the client are invalidated and must be freshly retrieved from the server when they are next requested. Validation is performed as an auxiliary request whenever the client contacts the server to read a new block.

It is important to note that there exists an interval of time between one client's writing to a file and another client's invalidation of cached blocks. Within this interval, client caches can become inconsistent. It is not unreasonable to assume that in some applications, certain clients would rather trade performance for consistency. That is, we would like to be able to extend this validation checking in a way that will at least shorten, if not eliminate, this interval. We have explored two different approaches for this consistency extension, one client-based and one server-based. In both cases, we believe an aspect-oriented approach supports these extensions better than the original implementation.

3.1 Client-based consistency extension

One way to make this kind of extension is to trigger validation from within a wider variety of NFS client activity. That is, to go beyond just attaching validation to client-based read requests and to attach it to client writing, client cache checking, daemon writing and daemon prefetching. In the original implementation, this would involve inserting validation requests into each of these NFS functions.

We believe an aspect-oriented approach better supports extensibility because these validation checks can be attached to a wide range of NFS core functions by identifying an appropriate pointcut and basing validation on that definition, as outlined in Figure 2. In addition to client-side reading, the pointcut in this aspect identifies 4 additional client-side functions that will trigger validation checking. The before advice attaches the validation code to each of the functions listed in the pointcut. This approach offers the potential to support development of validation strategies and associated triggers independent from core functionality.

3.2 Server-based consistency extension

Another way to approach this extension is to actively invalidate appropriate client caches when relevant writes are received at the server. This would require maintaining state information at the server[2]. Introducing this support however, would require considerable crosscutting changes to core NFS server functionality. Given that this added complexity would only be used to customize service for special-case clients that dynamically negotiate it, this option unnecessarily compromises stability of core functionality for the majority of clients.

An aspect-oriented approach could be structured to accomplish this 'active invalidation' without invasive changes to core NFS server functionality. At a high-level, attaching state update operations to read requests and attaching invalidation notices to associated write requests appears to be the natural structure of this extension. This simple approach is outlined in Figure 3.

[2]NFS versions 2 and 3 servers are stateless, but all this about to change with the latest release of NFS version 4.

```
aspect sequential_prefetch{

  /* client-side advice */

  before( file_id fid, block_num bnum ): calls( int nfs_client_read( fid, bnum, ..) ) {
    update_pattern_of_access( fid, bnum );
  }

  after( file_id fid, block_num bnum ): calls( int nfs_client_read( fid, bnum,..) ) {
    if ( fid->access_pattern == SEQUENTIAL )
      nfs_daemon_prefetch( fid, bnum+1 );
  }

  /* server-side advice */

  after( file_id fid, block_num bnum ): cflow ( calls ( void nfs_daemon_prefetch(..) )) &&
      calls ( int nfs_server_read( fid, bnum, ..) ) {
    aggressive_prefetch_into_server_cache( fid, bnum+1, bnum+MAX_PREFETCH );
  }
}
```

Figure 4: AspectC pseudo-code for sequential prefetching.

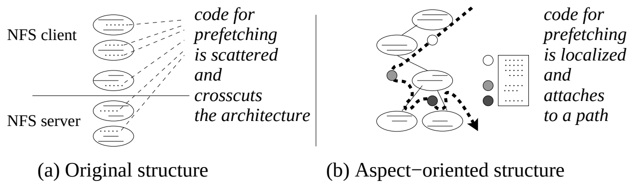
NFS client
NFS server
code for prefetching is scattered and crosscuts the architecture
code for prefetching is localized and attaches to a path
(a) Original structure
(b) Aspect−oriented structure

Figure 5: Prefetching and primary functionality.

4 Prefetching extension

Figure 4 outlines a possible implementation of a prefetching aspect for sequential access. The client-side advice keeps track of access behaviour, noting if it is sequential, and invoking the daemon to prefetch if it is. The server-side after advice aggressively prefetches into the server cache if the request to read originated from a daemon's prefetching request on the client.

Figure 5 shows how we envision prefetching to be structured according to client-aware path-specific customizations. This work is a continuation of our work with prefetching in the local file system [2]. The key difference here is that we need to carry the *cflow* mechanism outside of local context and through to a remote machine.

5 Evaluation

Based on this design, we believe an aspect-oriented approach to extensibility allows us to addresses several key issues better than a traditional approach:

- Pluggable functionality - enhanced consistency and performance strategies can be optionally included in NFS make files.
- Independent development - extensions are not invasive to core functionality.
- Comprehensibility - the internal structure of extensions are localized within a shared context, as well as a complete description of their interaction with primary functionality.

6 Implementation considerations

We believe this exploration of an aspect-oriented approach to extensibility in NFS holds promise and are now proceeding to implement this design. In particular, we will be interested in understanding the impact some of these design decisions have on performance. We recognize that these extensions must not introduce a significant performance penalty for clients that continue to rely only on core NFS services.

7 Conclusion

We have shown how to use AOP to structure extensions in a way that makes them easy to develop and apply independently. We believe that aspects support extensibility better than a corresponding traditional approach for two reasons: (1) the core functionality is stable, (2) the extensions are modular.

References

[1] www.cs.ubc.ca/spider/ycoady/aspectc.html.

[2] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong. Structuring system aspects. *Communications of the ACM*, October 2001. To appear.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. AspectJ home page. http://www.aspectj.org.