Injectors and Annotations

Robert E. Filman
Research Institute for Advanced Computer Science
NASA Ames Research Center, MS/269-2
Moffett Field, CA 94035
rfilman@mail.arc.nasa.gov

Introduction

In [7], we presented the Object Infrastructure Framework. The goal of that system is to simplify the creation of distributed applications. The primary claim of that work is that non-functional "ilities" could be achieved by controlling and manipulating the communications between components, thereby simplifying the development of distributed systems. A secondary element of that paper is to argue for extending the conventional distributed objects model in two important ways:

1. The ability to insert *injectors* (filters, wrappers) into the communication path between components.
2. The ability to *annotate* communications with additional information, and to propagate these annotations through an application.

Here we express the descriptions of that paper.

Injectors

The ideas of OIF are demonstrated in a system built around CORBA and Java. OIF has a modified a CORBA IDL compiler. This compiler generates both client and server proxies that store, for each method on each object, a table of injectors. Between the application and marshalling, the new proxies successively execute the injectors associated with the called object and method (Figure 1).

Semantically, one would think one wouldn't need to wrap both the caller and the called function. After all, computationally, they're almost the same place. However, OIF places injectors on both the client and the server because

- In a distributed system, one may need behavior on both sides of the distributed divide. For example, security requires authenticating on the server credentials generated on the client.
- In a distributed system, there can be pragmatic difference depending on where something is done. For example, keeping a cache of recently-seen-values on the server is pointless.
- Using the annotation mechanism, injectors on each side shared some context with application objects.

Injectors in OIF are by object/method. Each instance proxy and each method on that object can have a distinct sequence of injectors. Newly created proxies that supported a particular interface use the default injector sequence defined for each method on that interface. The system also includes a language (Pragma) for specifying these defaults.

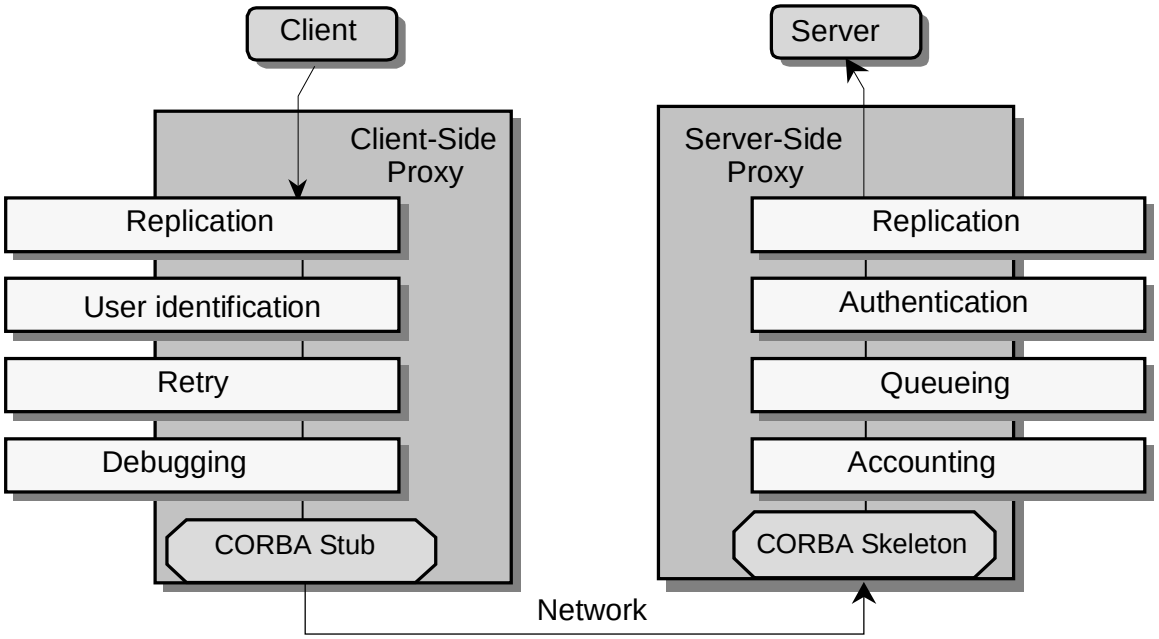


Figure 1: OIF inserts injectors between the application and the network

Injectors in OIF are dynamic. There are interfaces for changing the injectors on a given proxy and for changing the default injectors for a newly created proxy's type. More formally, one specifies *factories* of injectors and parameters for the factories for defaults. This allows creating injectors that obey patterns such as singleton (one injector object for every proxy), full multiplicity (each object got its own injector), or other possibilities in between. The factory objects are called with the newly created proxies and additional parametric information. Dynamic injectors allow techniques such as runtime, interactive placement of debugging and monitoring probes; and creating software that detects its own obsolescence and updates itself. Of course, one would only want to expose this interface to trusted applications.

Injectors are called with the CORBA request object and an object representing the *continuation*—the rest of the injector sequence to be called. From the request object, injectors can manipulate the call's target, method name, arguments, annotations, and returned value. Injectors are objects, and themselves can do arbitrary other computations, including publishing their distributed-object address and making remote calls of their own.

The last action of a normally terminating injector is to invoke the next injector in the continuation. It gives that injector the request object and the remainder of the continuation sequence. (This behavior is wrapped up in a helper function on the "list" object that forms the continuation.) This allows injectors to catch exceptions and to forgo or change the continuation sequence. Catching exceptions allows error handling by injectors—for example, retargeting a failing request to another server [9].

We also use exceptions as a control mechanism. For example, in the authentication injector pair, the server-side authentication injector, if unsatisfied with a request's credentials, throws an exception, to be caught by the client-side authentication injector. The client injector obtains the user's credentials (in our example, using the famed iButton

Ility	Injector	Action
Security	Authentication	Determines the identity of a user.
	Access control	Decides if a user has the privileges for a specific operation.
	Encryption	Encodes messages between correspondents.
	Intrusion detection	Recognizes attacks on the system.
Reliability	Replication	Replicates a database.
	Error retry	Catches network timeouts and repeats call.
	Rebind	Notices broken connections and opens connections to alternative servers.
	Voting	Transmits the same request to multiple servers (in sequence or parallel) combining the results by temporal or majority criteria.
	Transactions	Coordinates the behavior of multiple servers to all commit or fail together. Requires additional interface on application objects.
Quality of service	Queue-manager	Provides priority-based service.
	Side-door	Provides socket-based communication transparently to application.
	Futures	Provides futures transparently to the application.
	Caching	Caches results of invariant services.
Manageability	Logging	Reports dynamically on system behavior.
	Accounting	Reports to accounting system on incurred costs.
	Status	Accrues status information and reports when requested.
	Configuration management	Dynamically test for incompatible versions and automatically updates software.

Table 1: Injector applications

Java ring [3]) and reinvokes the process.

Retargeting is one example of changing the continuation. A simpler one is the injector that implements a cache—if the desired value is already in the cache, then that injector skips calling the remainder of the injector sequence and simply returns the marked value. Caching can be used for services that are functional, or to create "by need" objects. Table 1, from [6], list some possible applications of injectors. Reference [8] discusses several examples of injectors in greater depth.

Annotations

Injectors need to communicate among themselves. For example, an authentication injector needs to know the identity and credentials of a service requestor. Other examples of tasks aided by inter-injector communication include sending process priority, accounting data, debugging interests, version information, and transactions.

Annotations provide a language for applications and injectors to communicate about requests. That is, they are a meta-language for statements about requests and the processing state. Annotations can express notions like "This request is to be done at high priority," "Here are the user's credentials for this request," and "Here is the cybercash to pay for this request."

In OIF, annotations are a set of arbitrary name-value pairs. The names are strings and the values, CORBA ANY types. Using ANY types allows object references as annotation values. (Of course, if we were to rebuild the system today, fashion would demand

that the annotations be encoded in XML. Conceptually, this would make little difference.)

The use of arbitrary name-value pairs, like weak-typing, has advantages and disadvantages. Arbitrary pairs require implicit agreement among processes as to the meaning and data types of annotations. On the other hand, arbitrary pairs allow easily augmenting the system with additional information without having to reprogram every associated element to know about the new kind of annotation. The framework defines certain common annotations, including session identification, request priority, sending and due dates, version and configuration, cyber wallet, public key, sender identity and conversational thread. Programs can rely on the common meanings of these annotations. Arbitrary fields, with no prohibition against unknown fields, are a feature of many application interfaces, including http and mail headers, certain XML DTD forms, and property sets in Lisp and Java.

OIF implements annotations by turning them into a sub rosa additional call argument.

One of the defining requirements of the OIF system was that it be invisible to users who weren't using it. (We were trying, after all, to simplify distributed computing [5]). Nevertheless, applications stilled need to communicate with the injector mechanism. For example, a quality-of-service injector may want to process requests in order of their priority, but the only reasonable source of request priority is the client application. We mediate this issue by giving each application thread a *thread context*. Thread contexts are the same stuff as annotations—arbitrary maps from strings to values. On creating a new request, the framework populates the annotations of that request with the annotations of the calling thread context. These annotations are passed to the server side, where the thread context of the thread serving the request is initialized to these values. At return time, the process is inverted—the annotations of the server thread are copied to the reply object and passed to the framework on the client side. The client uses these values to update the annotations of the originally calling thread.

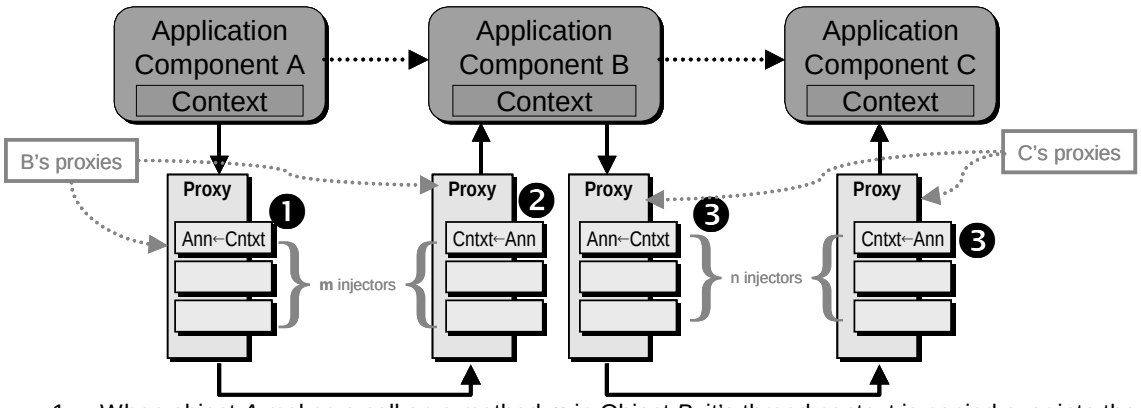
This scheme has the feature of propagating context through a chain of calls: client A's call of B at priority *x* becomes B's context's priority of *x*. B's request of C (in furtherance of A's call) goes out with priority *x*. Figure 2 illustrates this pattern.

Thread contexts have the advantage of permitting client/injector communication without modifying the application interfaces. They have the disadvantages that (1) newly spawned threads need to copy or share the context of their parents and (2) there is no primitive linguistic mechanism for neatly "block structuring" a change to a thread's context—for example, allowing a thread to simply timeshare among tasks. I note that J2EE also associates "user identity" with thread, and fails for non-system spawned threads.

In OIF, declarations can control annotation copying. For example, the number of times the (client) retry-on-failure injector retries is purely local to the client and is not sent downstream. Similarly, we do not want a server to be able to update a client's user identification. The default behavior is to copy, enabling creating and propagating new annotations without modifying existing application code.

Related Work

OIF is in the spirit of much work in Aspect-Oriented Programming [4]. It has particular similarity to the work on Composition Filters [1] and Aspect Frameworks [2]. The use of filters on communications has been taken up in the security interceptors of CORBA [10].



1. When object *A* makes a call on a method *m* in Object *B*, it's thread context is copied over into the annotation of the request
2. After creating the thread to serve *A*'s request, the annotations of the request are copied to that thread's context
3. *B* calls method *n* on *C*. This process is repeated for *B*'s calls when handling that request. Thus, an annotation (e.g., priority) set in *A* is carried over through *B* to *C*.

Figure 2. Propagating annotations through multiple calls

From our point of view, these interceptors are in the wrong place, after argument marshalling, where the contents of requests are too opaque to the filters. We could argue that per object / per method filtering is a richer mechanism than the singletons of typical CORBA implementations. The counter arguments would be that per object / per method filtering can be implemented in a singleton filter, and that somehow having so much information would be too expensive. The idea of filtering has also been applied by Thompson et al. to web services [11].

Concluding remarks

Distributed systems introduce additional complexity. Developing a distributed system is in itself a more difficult task because distributed systems imply non-determinism (and non-determinism is complex), distribution introduces many additional kinds of failures, distribution is naturally less secure, and distribution's inherent decentralization is inconvenient to manage. Distributed computing can be made simpler by making it look more like conventional programming and by providing and automatically invoking correct implementations of distributed and concurrent algorithms.

Injectors and annotations are part of a mechanism to get a handle on these problems. By providing a mechanism for introducing corrective action and a locus for expertise, injectors and annotations can be an element in achieving the simpler-distributed-computing goal.

References

[1] Bergmans, L., and Aksit, M. Composing crosscutting concerns using composition filters. *Comm. ACM 44*, 10 (Oct. 2001), pp. 51–57.

[2] Constantinides, C. A., and Elrad, T. Composing concerns with a framework approach. In *Proc. International Workshop on Distributed Dynamic Multiservice Architectures, 21st IEEE Int'l Conf. on Distributed Computing Systems, Workshops, Vol. 2* (Phoenix, Apr. 2001), pp. 133–140.

[3] Dallas Semiconductor Corporation. ibutton: Touch the future. http://www.ibutton.com/.

[4] Elrad, T., Filman, R. E., and Bader, A. Aspect-oriented programming. *Comm. ACM 44*, 10

(Oct. 2001), 29–32.

[5] Filman, R. E. Injecting ilities. In Int'l Workshop on Aspect Oriented Programming, ICSE, (Kyoto, Apr. 1998).

[6] Filman, R. E. Applying aspect-oriented programming to intelligent synthesis. In Workshop on Aspects and Dimensions of Concerns, ECOOP 2000 (Cannes, France, June 2000).

[7] Filman, R. E., Barrett, S., Lee, D. D., and Linden, T. Inserting ilities by controlling communications. *Comm. ACM 45*, 1 (Jan. 2002), 116–122.

[8] Filman, R. E., Korsmeyer, D. J., and Lee, D. D. A CORBA extension for intelligent software environments. Advances in Engineering Software, 31, 8–9 (2000), 727–732.

[9] Filman, R. E., and Lee, D. D. Redirecting by injector. In *Proc. International Workshop on Distributed Dynamic Multiservice Architectures, 21st IEEE Int'l Conf. on Distributed Computing Systems, Workshops, Vol. 2* (Phoenix, Apr. 2001), pp. 141–146.

[10] Siegel, J. *CORBA Fundamentals and Programming*. John Wiley & Sons, New York, 1996.

[11] Thompson, C., Pazandak, P., Vasudevan, V., Manola, F., Palmer, M., Hansen, G., and Bannon, T. Intermediary architecture: Interposing middleware object services between web client and server. ACM Computing Surveys 31, 2es (1999), 14.