Injecting Management

Robert E. Filman*

Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, Texas 78759-6509
filman@mcc.com

Object Infrastructure Framework

The goal of the Microelectronics and Computer Technology Corporation's (MCC) Object Infrastructure Project (OIP) is to simplify the development and evolution of distributed, object-oriented applications. OIP is designing an architecture for distributed systems (a set of rules for distributed, component-based systems to follow) and implementing frameworks. (Such are examples of Object Infrastructure Frameworks, or OIF.) Particular frameworks are implemented for different computational environments.

OIP follows the CORBA model of organizing distributed systems. In CORBA, an object that wants to offer services to other objects describes these services in Interface Definition Language (IDL). IDL primarily specifies available methods and their type signatures. An IDL compiler compiles an application's IDL into the definition of a set of proxies.

In CORBA, a remote ("server") object is represented on a local address space by a proxy object (the "stub"). When a process on the local machine (the "client") wants to invoke a remote method on the server, it makes a call to the stub. The stub encodes the call to the server and pushes the encoding over the network towards the server. At the server end, there is a corresponding proxy object (the "skeleton") for decoding the remote calls and invoking the actual service. (The skeleton and stub cooperate in the corresponding manner for returning values.) This process is illustrated in Figure 1.

Conceptually, OIP extends this architecture by providing an additional language

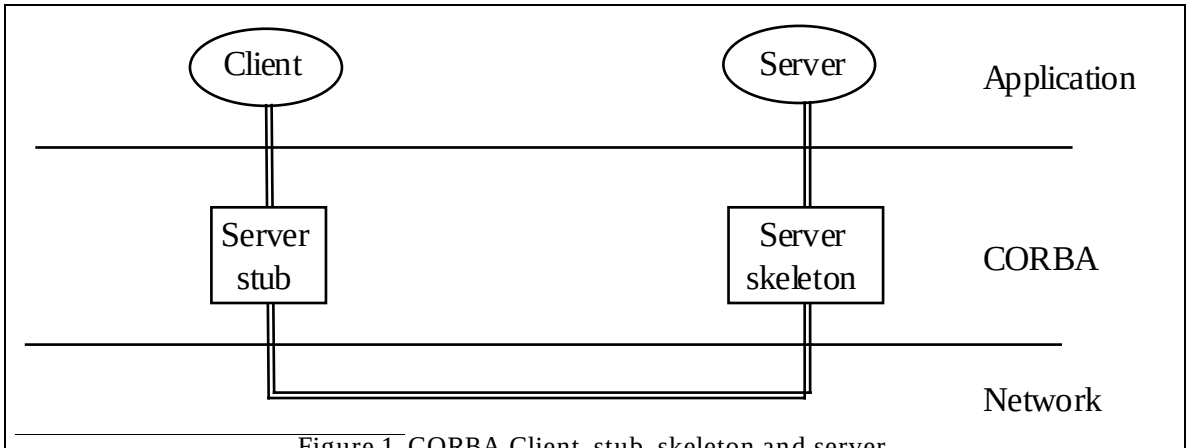

Figure 1. CORBA Client, stub, skeleton and server.

* This paper describes work performed at MCC while the author was on assignment from Lockheed Martin Missiles and Space. LMMS contact information: Advanced Technology Center; Lockheed Martin Missiles and Space; 3251 Hanover Street O/H1-43 B/255; Palo Alto, California 94304. Email: bob.filman@lmco.com.

that describes other desired behavior on the communication path. One possible implementation of an OIP framework compiles the concerns of this language into OIP stubs and skeletons which sit between the Client and Server components and the proxies. Figure 2 illustrates this relationship.

The OIF stub and skeleton realize the "mixing-in" of additional behavior on procedure calls, much as "mix-in" mechanisms in languages like Zeta-Lisp allowed the systematic addition of behavior to functions. In actual practice, there are a number of ways to accomplish such in-mixing, including: compiling stubs, as described above; taking advantage of built-in "interceptors" or "filters" on the communication path [1], as provided for in the CORBA security specification (or as implemented by several ORB vendors); or by actually transforming the application source code (e.g., [2-4]).

Proxy technology reifies the representation of each server on the client. That is, there is an object "representing" the server within the server address space, and one or more such representative objects in each client address space. By using one of the mechanisms above, we can control what gets executed in these proxies (beyond the functionality placed there by the IDL compiler.)

Although the diagrams show a single "box" for each proxy, the behavior of that box can be specialized for each server method. In our implementation of OIF proxies, the per method proxy behavior is dynamically configurable from sub-behaviors. Conceptually, this can be thought of as maintaining, for each server method, on each proxy, a sequence of *injector* objects. A call to the server goes through the injectors on the client stub, over the network, through the injectors on the server skeleton, into the application object, then back through the skeleton injectors, network and client stub injectors. This process is illustrated in Figure 3. Note that the three different methods on the server have different sequences of injectors. In OIP, we use an annotation mechanism to communicate information among injectors [1]. Uses of injectors include logging injectors that record details of calls; queue management injectors that prioritize calls; futures injectors that turn synchronous communication into a futures-based approach; choice

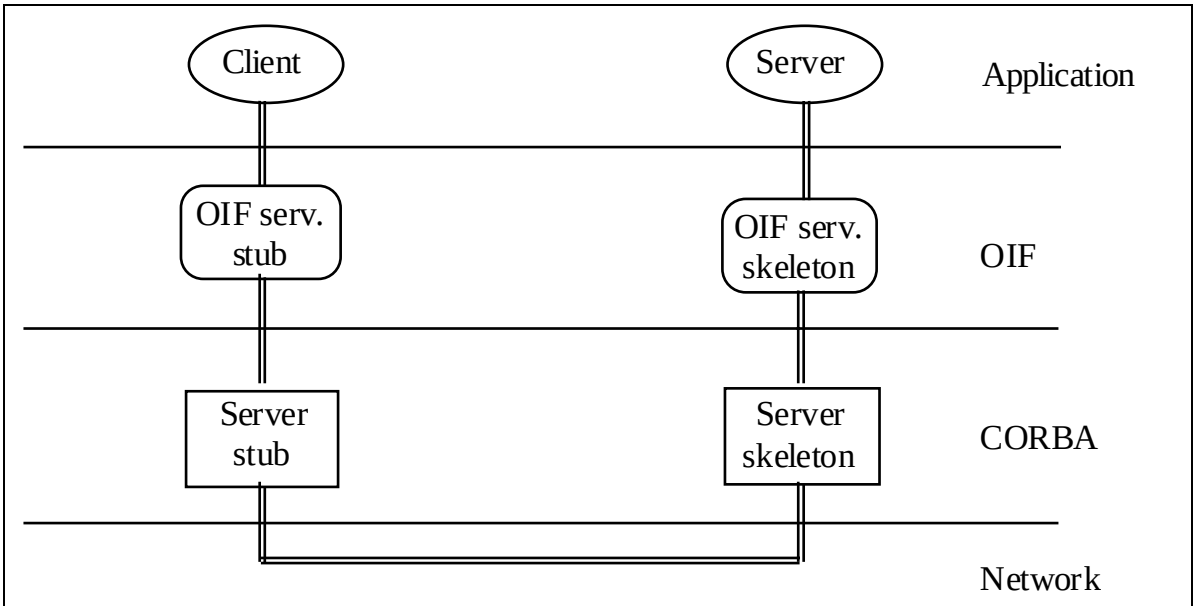

Figure 2. Client, OIF stub, CORBA stub, CORBA skeleton, OIF skeleton, and server

injectors, that, given a variety of ways of satisfying a request (e.g., multiple servers or alternative algorithms) can dynamically adjust calls based on current and historical performance; and caching injectors that (transparently to the application) return the value of cached function calls without the expense of remote communication.

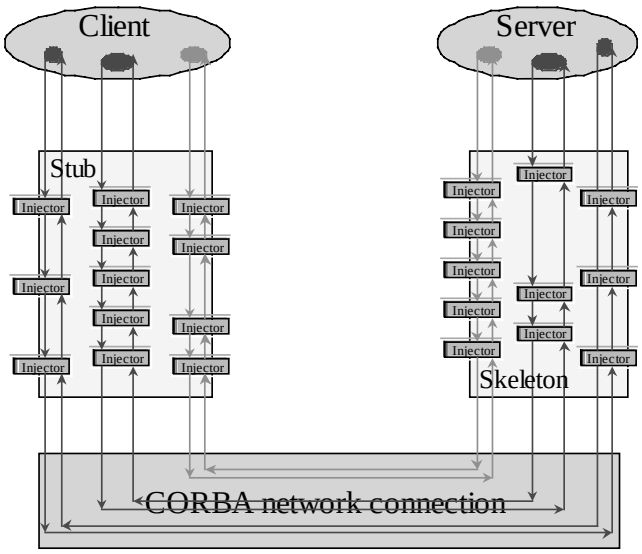

Figure 3: Injector invocation sequence

Managing Components

Let us illustrate the power of these ideas in the context of the dynamic management of component systems. ISO, in their standards for network management, identifies five sub-tasks to the activity of (network) management: performance measurement, accounting, failure analysis, intrusion detection, and configuration management. Managing distributed applications is very similar to network management. The clever use of communication injectors can be helpful in each of these tasks:

- **Performance measurement injectors** can record the time of events (e.g., the initiation of a request, the receipt of an answer, the initiation of the server-side processing of a request, the production by the server of the answer). Appropriately coalesced, this data is the basis for performance evaluation. Of course, this is giving us information about the performance of black-box components, not the performance interior to the components.
- **Accounting injectors** can verify account status, log the information required for payment, and even cause the appropriate billing transactions to begin. All this can be done independent of the original application. However, accounting injectors cannot account with respect to the interior activities of an application. That is, an accounting injector can charge for use of a database per query, by complexity of the query, by the number of records returned by the query or

even by the amount of time it took to process the query. But it can't charge by the number of records accessed in the processing the query.

- **Failure analysis** is a primary activity of programmers and a major concern of administrators. Trace injectors can report on calls and returned values. Break injectors can interrupt processing to allow human (or softbot) examination of the call and return values. Heartbeat injectors can verify connectivity and report its demise. These things once again are the appropriate-level tools for examining behavior between components, but not within components.
- **Intrusion detection injectors** can be programmed to recognize and report on intrusion patterns [5]. Note that such injectors can react to intrusions dynamically (even blocking certain kinds of access which a system is under attack) rather than merely writing logs for retrospective examination.
- **Configuration injectors** can check and update component configurations. To the extent that the compilation of new versions of a component can be done with respect to a new version of a "version" injector, injectors can be used to make sure that appropriate component versions are being used together. Properly endowed injectors can even dynamically update components. However, all these (dynamically configurable) injectors on proxies themselves present a configuration problem. That problem is likely to be more severe than simply configuring monolithic components.

Two common themes run through this litany, that of injectors that report and broadcast interesting events and of additional cleverness in the definition of "interesting." In our OIP work we are developing enhanced versions of "event channels," where applications can more selectively subscribe to appropriate information. Similarly, injectors can apply more complex, historical-pattern tests in deciding which events to publish. These patterns can be based not only on the actual arguments and results of calls, but also on a common language of message annotation [1]. By selective reporting (and being able to dynamically configure what is reported), injector-based mechanisms can avoid information flood.

Concluding remarks

We have argued that controlling the inter-component communications gives a good handle on dynamic component management. Our initial experiments have lent credence to this hypothesis: we have developed a system for inserting injectors into CORBA applications and have been particularly pleased with the use of reporting injectors in debugging applications. We are currently working on extending this demonstration to all aspects of system management and security.

Acknowledgments

The ideas expressed in this paper have emerged from the work of the MCC Object Infrastructure Project, particularly Stu Barrett, Carol Burt, Deborah Cobb, Tw Cook, Phillip Foster, Diana Lee, Barry Leiner, Ted Linden, David Milgram, Gabor Seymour, Doug Stuart and Craig Thompson.

My thanks to Tw Cook, Diana Lee, Ted Linden, Dave Milgram and Tom Shields for comments on the drafts of this paper.

References

[1] Robert E. Filman, "Injecting Ilities," *Workshop on Aspect Oriented Programming*, ICSE-20, Kyoto, April 1998.

[2] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin "Aspect-Oriented Programming, " Xerox *PARC Technical Report, February 97, SPL97-008 P9710042.*
http://www.parc.xerox.com/spl/projects/aop/tr-aop.htm

[3] Cristina Videira Lopes, and Gregor Kiczales, "D: A Language Framework For Distributed Programming," Xerox PARC Technical report, February 97, SPL97-010 P9710047.
http://www.parc.xerox.com/spl/projects/aop/tr-d.htm

[4] Robert E. Filman, "Applying AI to Software Renovation," *Automated Software Engineering, Vol. 4*, 1997, pp. 341-360.

[5] Robert Filman and Ted Linden, "Communicating Security Agents," The Fifth IEEE-Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises---International Workshop on Enterprise Security, Stanford, California, June 1996, pp. 86-91.