DADO: A Novel Programming Model for Distributed, Heterogenous, Late-Bound QoS Implementations

Eric A. Wohlstadter and Premkumar T. Devanbu

Software Systems Research Laboratory
Department of Computer Science
University of California at Davis
Davis, CA 95616, U.S.A.
{devanbu,wohlstad}@cs.ucdavis.edu

Abstract. Quality of service implementations, such as security and reliability, are notoriously difficult for software engineers. They have fragmented, cross-cutting implementations, with elements required in application components on both client and server sides. Heterogeniety and distribution make things even more difficult. To cap it all, precise requirements, specially for security, are often deployment-specific. DADO is a new programming model, with roots in aspect-oriented programming, that aims to improve matters. The goal of DADO is to provide a CORBA-like programming model to the developers of distributed, heterogenous QoS features. DADO comprises a modeling language, a deployment language, code generators, and a run-time environment. DADO allows QoS features to be modeled at the IDL level. Communication between QoS elements is explicitly modeled, as is the interaction between QoS elements and applications (*viz.,* advice). Model-driven code-generation handles the mundane details of marshaling, despatch, and heterogeniety. DADO has a separate deployment language that allows QoS services to be deployed against applications. The DADO run-time also has several interesting features: clients can dynamically respond to QoS features at the server by deploying matching client-side service elements at run-time. DADO supports various interception mechanisms, such as proxies, interceptors, and code transformation, and all mechanisms are inter-operable. DADO currently works on TAO and JacORB, and supports QoS implementations in both Java and C++.

1 Introduction

Consider a distributed medical application, wherein clients obtain prescriptions from doctors. The attendant security policy requires that a client must obtain a credential from a credential server (as in Kerberos; this eliminates the need for the client to reveal his password to anyone but the credential server, and also does not require the doctors/clients to reveal the security policy to the authentication server) and present this credential to authenticate themselves to

R. Meersman and Z. Tari (Eds.): OTM Workshops 2003, LNCS 2889, pp. 926–933, 2003.

a doctor. To use the doctor, the clients must first register themselves with a doctor for a period of time (this is to prevent clients from seeking duplicate prescriptions from multiple doctors). While registered with a doctor, a client may not use other doctors. The client can then obtain prescriptions.

Such a policy presents serious challenges for developers using middleware such as CORBA. The policy requires modifications to both client-side (to obtain credentials) and server-side (to validate credentials/registration, and also register clients), and requires an extra bit of data (the credential) to be sent across. Existing options for implementing such a service are unattractive. Modifying the interface, and weaving the policy code into the application tightly binds the security policy into the application and tangles application code, vitiates separation of concerns, precludes the reuse of the policy code elsewhere, and complicates (likely) future modifications to the security policy. Reflective implementations ([1,2,9] do allow separation of concerns, and a componentizable, reusable implementation; however, the use of reflection reduces the utillity of static type-checking, and usually requires hand-written marshaling, demarshaling and dispatching code.

DADO provides a different approach to programming security and other QoS features: the design philosophy is *to bring to service developers the same convenient programming model that CORBA brings to developers of distributed heterogeneous software.*

2 DADO

The Dado programming model comprises languages, tools, and run-time enhancements.

Languages. There are two langauges: the DADO interface description language, or *DAIDL*, is used to describe the interface between the service elements and the application components. The interfaces of client-side and server-side service elements are modeled separately. DAIDL models QoS methods, called *advice*, that interact with application methods. The communications between client- and server-side is also explicitly modeled, as *requests*. The second language, DADO deployment language, or *DDL*, is based on Aspectj [7] pointcuts; this language specifies precisely how a QoS feature interacts with an underlying application. For example, we can specify which methods are intercepted by the security QoS advice, and (if need be) which arguments of these methods are to be captured and sent to the security policy implementation.

Tools. Dado has two sets of tools, for *development time* and *deployment time*. At development time, code generators produce stubs and skeletons from the DAIDL descriptions These provide multi-lingual type environments (currently, for C++ and Java) that allow service developers to write client-side and server-side QoS code in a type safe manner. Marhalling and demarshalling code for QoS data exchange are also generated. The deployment time tools match the QoS DAIDLS against the application IDLs, and based

on the DDL specifications, generate despatch tables that control the invocation of DAIDL advice.

Run-Time Libraries. DADO run-time libraries handle despatch of advice, and of requests. The run-time also handles the dynamic matching (on the client-side) of client QoS features to QoS features deployed at the server.

Figure 1 (reproduced from [15]) describes the typical software process workflow when using DADO to develop and deploy QoS features. The caption on the figure explains the process. We emphasize here that DADO aspires to the CORBA "hourglass" philosophy: a simple, flexible, powerful programming model that can be implemented in a variety of different ways. In particular, DADO allows QoS features to be implemented in a variety of programming languages (currently, we support Java and C++), and allows QoS elements to be inserted (instrumented) into the application code in a variety of different ways. Certainly, in a heterogeneous environment, different implementation and instrumentation technologies can co-exist.

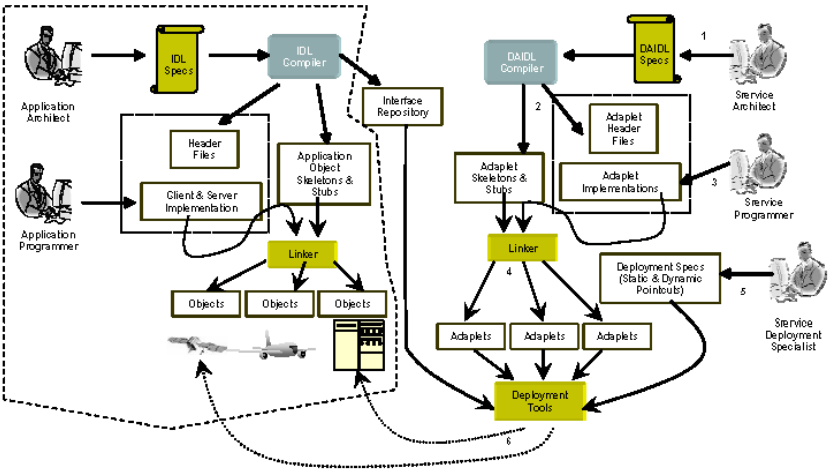

Fig. 1. *DADO Development Process.* The left hand side (within the dotted lines) indicates the conventional CORBA process. On the right, the DADO service development begins (1) with modeling the interfaces to DADO adaplets using DAIDL; from this the DAIDL compiler generates (2) plumbing code, and typing contexts for adaplet implementations. The programmer writes (3) the adaplet implementations and links to get (4) the adaplets. Now, the development specialist prod uces (5) deployment specs, and these are used by deployment tools to install (6) the adaplets at the proper application object locations. Deployment can occur at compile time, link time, or run-time, depending on the instrumentation technology used (only run-time insertion is illustated in the figure).

3 Security Example

We now turn to the security policy example presented at the beginning of this paper. The Dado interface description of this QoS feature is shown below. We note that this description simply provides the interface; as in the case of CORBA (and as described above in figure 1) this interface is processed by tools to provide stubs and skeletons; the skeletons then have to be implemented by service programmers.

```
adaplet Wall {
  server {
    request authenticInfo(in Authentic::Token tok);
    request register(in Authentic::Token tok,
                     in long duration);
    advice void check()
                     raises(NotRegistered,NoAccess);
  };
  client {
    advice contactAuthentic();
  };
};
```

A QoS feature in DADO (specifically, DAIDL) is termed an *adaplet*. An adaplet has two interfaces, one for the client side, and one for the server side. In this case, the client-side adaplet interface comprises one *advice* method, which intercepts the client-side of application methods that require security policy enactment. The implementation of the advice `contactAuthentic` in this case will get the authentication token, and communicate it to the server side. The server-side adaplet will have the logic to check this token, and also register the client with a particular server implementation. These two operations (receiving and checking the token, and registering a client) are modeled in the server side adaplet interface as *requests*: `authenticInfo` does the former, and `register` the latter. Finally, the advice operation `check` will intercede on application methods requiring security policy enactment.

4 Performance (Cacheing) Example

We present a cacheing example, whereby a client requests string-indexed values (*e.g.* stock quotes) from a server. These values are assumed to periodically change, as the server receives updates from a notification ("push") service. Here, a client-side adaplet maintains a string-indexed cache of values. If a request can be answered from the cache, the server is never contacted. To complicate matters slightly, we also assume that the client side adaplet can request the server side for a time-to-live (TTL) value. The server side adaplet maintains TTL values for various string indices by intercepting updates and observing update frequency for various string indices (*e.g.,* the "NYSE:HAL" ticker symbol might be updated more often when an oil-producing country is invaded).

```
adaplet Cache
{       client⟨ A ⟩ {
                    request void set_TTL(in long ttl);
                    around A get(in string hashcode);
        };
        server      {
                    request void ask_for_TTL();
                    before void update_TTL();
        };};
```

We start with an abstract interface to the client and server side adaplet components necessary to carry out the caching service. The client-side includes a request operation to receive TTL values that are "piggybacked"[1]. on server responses. Additionally, an around advice is specified to short circuit client requests for which cached data is available. The cache is keyed through the string index. This index is obtained from the application that the cache is deployed on. The type of data stored by the cache can be polymorphic and is bound though the type parameter `A`.

On the server side there is also one request and one advice operation. The request `ask_for_TTL` allows the client side adaplet to signal the server side that it is planning to cache the data in the current invocation response. The before advice `update_TTL` should be triggered whenever application events occur on the server side that invalidate cache data. This allows the server to heuristically adjust its forecasted TTL values. The DDL specification to bind this service to a specific application is shown below. Notice the instantiation of type parameter `A` with `float`.

```
adaplet ClientSideStockCache : Cache
{
    client: Cache ⟨ float ⟩ {
          pointcut cachedOperation(in string x) :
                   call(float StockServer::getQuote(x));
          around cachedOperation(x) :
                   get(in string x);
    };
};
```

Now, the abstract interface must be specialized for deployment on both client and server. Specialization is acheived through object-oriented inheritance. In order to cache the result of the StockServer::getQuote operation the client side cache type parameter A is instantiated as float. The pointcut cachedOperation matches the invocation to StockServer::getQuote and captures the argument to key the cache. Finally this pointcut is applied to the around advice that we want to intercept these invocations.

[1] The DADO runtime packages QoS related messages from client to server (or vice versa) in the *service context* field in the IIOP RPC message

```
adaplet ServerSideStockCache : Cache
{
   server{
         pointcut timedOperation() :
                  call(void StockServer::setQuotes(in QuoteList));
         before timedOperation(): update_TTL();
   };
};
```

The server side deployment consists of dispatching the update_TTL advice for operations where TTL calculation can be updated. The pointcut timedOperation matches the operation StockServer::setQuotes and is applied so update_TTL will be dispatched before those invocations.

5 Current Status

Currently, DADO works on both Linux and Windows. On both platforms, we support both TAO and JacORB. DADO tooling can generate both C++ (Gnu and Microsoft) and Java. Performance measurements indicate that the overhead of our marshalling and dispatching machinery is reasonably modest. Details are available in [15]. Several sample services have been implemented; including the above two, we have also implemented a generic failover service. We have also experimented with generic adaptations to prevent denial-of-service attacks, and generic implementations of P2P architectural styles. Work in progress includes development of a DADO-like approach for web services, as well as dynamic, coordinated deployment of QoS services in complex, distributed, feature-rich and feature-interacting settings.

6 Related Work

There is related work on handling cross-cutting adaptations in heterogenous enviornments. We have already compared DADO with reflective ORBS [14,8, 3]. Interceptors [14,9], and filters [11] can intercept every method, but require marshalling and dispatch to be hand-constructed without the benefit of static typechecking. Proxies and wrappers [5,12] are specific to applications. Containers [10,13] generalize by exploiting code generation, but require new code generators for each new QoS approach. In addition, client-side QoS adaptations are not supported.

Duclos, Estublier, and Marat [4] describe the component virtual machine, (CVM). CVM provides a meta-object protocol for components that allows component adaptation, specific as AspectJ-style pointcuts. DADO is similar in spirit, but complements CVM by allowing component adaptations on both client and server sides. CQoS [6] allows the construction of generic QoS components using generic QoS components, and application/platform dependent interceptors. Cactus provides consistent coordination adaptation of distributed systems using

micro-protocols. DADO exploits on aspect-oriented model for adapting applications with QoS features. CQoS allows per-object QoS bindings, whereas the current implementation of DADO only allows per-POA bindings.

7 Conclusion

DADO provides a convenient programming model for building QoS features such as security, fault-tolerance and billing/usage tracking in a distributed, heterogenous, setting. DADO can conveniently handle split-context QoS features, and supports piggy-backed messaging from client to server side and vice versa. DADO supports a variety of different instrumentation techniques interoperably. DADO comprises a modeling language, code generation tools, run-time enhancements, and a deployment language. DADO is currently available for both Java and C++ on JacORB and TAO, and works on both Linux and Windows. An unsupported version is available at `rickshaw.cs.ucdavis.edu`

References

1. G. Blair and R. Campbell, editors. *Reflective Middleware*, 2000.
2. L. Capra, W. Emmerich, and C. Mascolo. Reflective middleware solutions for context-aware applications. *Lecture Notes in Computer Science*, 2192: 126–??, 2001.
3. N. Coskun and R. Sessions. Class objects in som. *IBM Personal Systems Developer*, Summer 1992.
4. F. Duclos, J. Estublier, and P. Morat. Describing and using non functional aspects in component based applications. In *International Conference on Aspect-Oriented Software Development*, 2002.
5. T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, pages 2–16, 1999.
6. J. He, M. A. Hiltunen, M. Rajagopalan, and R. D. Schlichting. Providing qos customization in distributed object systems. In *Middleware 2001 : IFIP/ACM International Conference on Distributed Systems Platforms*. Springer-Verlag Heidelberg, January 2001.
7. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072: 327–355, 2001.
8. F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, and R. H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*
9. P. Narasimhan, L. Moser, and P. Mellior-Smith. Using interceptors to enhance CORBA. *IEEE Computer*, July 1999.
10. E. Roman, S. Ambler, and T. Jewell. *Mastering Enterprise JavaBeans*. Wiley, 2001.
11. J. Siegel. *CORBA 3 Fundamentals and Programming*. Wiley Press, 2000.
12. T. S. Souder and S. Mancoridis. A tool for securely integrating legacy systems into a distributed environment. In *Working Conference on Reverse Engineering*, pages 47–55, 1999.

13. A. Troelsen. *C# and the .NET Platform*. Apress, 2001.
14. N. Wang, K. Parameswaran, and D. Schmidt. The design and performance of meta-programming mechanisms for object request broker middleware, 2000.
15. E. Wohlstadter, S. Jackson, and P. Devanbu. Dado: Enhancing middleware to support crosscutting services. In *Proceedings of the International Conference on Software Engineering*, Portland, USA, 2003. IEEE.