Middleware Support For Crosscutting Features in Distributed, Heterogeneous Systems

Eric Wohlstadter[1] and Premkumar Devanbu[2]

[1] University of British Columbia, Vancouver BC, Canada
wohlstad@cs.ubc.ca
[2] University of California, Davis CA, USA
devanbu@cs.ucdavis.edu

Abstract. Some "non-" or "extra-functional" features, such as reliability, security, and tracing, defy modularization mechanisms in programming languages. This makes such features hard to design, implement, and maintain. Implementing such features within a single platform, using a single language, is hard enough. With distributed, heterogeneous (DH) systems, these features induce complex implementations which crosscut different languages, OSs, and hardware platforms, while still needing to share data and events. Worse still, the precise requirements for such features are often locality-dependent and discovered late (e.g., security policies). The DADO[3] approach helps program crosscutting features in standard CORBA based DH middleware software through an aspect-oriented approach. A DADO *service* comprises pairs of *adaptlets* which are explicitly modeled in IDL. Adaplets may be implemented in any language compatible with the target application, and attached to stubs and skeletons of application objects in a variety of ways. DADO supports flexible and type-checked interactions (using generated stubs and skeletons) between adaptlets and between objects and adaptlets.

1 Introduction

This paper addresses the problem of supporting the development of late-bound, crosscutting features in distributed heterogeneous systems.

Crosscutting features[4] stubbornly resist confinement within the bounds of modules. It is well known that features such as logging, transactions, security and fault-tolerance typically have implementations that straddle module boundaries even within the most sensible decompositions of systems [1–4]; we present a sample security feature in the next section which provides yet another example. The scattered implementation of such features makes them difficult to develop, understand and maintain. To worsen matters, the requirements of such features are often *late bound*: locality dependent, discovered late, and change often—security policies again being a prime example. Programmers are thus confronted with the difficult challenge of making a scattered set of changes to a broad set of modules, often late in the game.

Distributed Heterogeneous systems (abbreviated DH) are part of the IT infra-structure in many organizations: many needed software functions are provided by systems assembled

[3] DADO: Distributed Adaplets for Distributed Objects. We also note that a "dado" is a carpenter's tool for making cuts across the grain.

[4] We use the term *feature* to denote a user perceivable property of a software implementation, *concern* to describe a property of interest during software development or maintainance, and *aspect* to describe a particular software modularization strategy. Different interpretations can be found in the literature and we make no claim that these definitions are canonical.

from pieces running on different platforms and programmed in different languages. Distribution arises from pressures such as globalization and mobility. Heterogeneity arises from considerations such as performance, legacy systems, weight, size, vendor specialization, and energy consumption. Feature implementations are scattered across different languages, operating systems and hardware platforms. Feature implementation elements in one platform need to correctly exchange information with existing application code, and with such elements on other platforms. In a WAN context, the presence of different, incompatible features (*e.g.* different security policies) may even cause the application to fail. Some platforms may be too resource-limited or performance-constrained to support some types of software evolution techniques (*e.g.*, reflection). In some cases, source code may not be available for modification , so middleware-based wrapping might have to be used. However, since feature implementations may crosscut platforms, all these different techniques of software evolution should be allowed to co-exist, and inter-operate.

In this paper, we describe DADO, an approach to developing features that require code changes, in a *DH* setting, to both client- and server-side of a remote interaction. The specification of crosscutting features is separated into a language independent interface definition; platform specific compilers can generate transformations through a variety of techniques.

The paper begins with a motivating example in Section 2. We then survey the surrounding area in Section 3. Section 4 presents our research goals in more detail. Section 5 describes the current status of our experimental implementation of DADO (including the run-time, code-generation and deployment tools), which is based on the OMG CORBA standard. Section 6 lays out the details of the motivating example in a concrete case study. In section 7 we describe closely related projects. Finally we conclude with an overall view of the work, the current limitations, and our future plans.

2 Security Example

We begin our example with the description of an application, The Remote Collaboration Tool [**?**] (*RCT*), where a new security concern must be addressed. RCT is a distributed multi-user interactive environment for academic and research settings. The architecture is based on client-server remote procedure-call (RPC) through the CORBA object-oriented middleware. Users on the RCT can self-organize into different groups to engage in private chat-room sessions or file-sharing. Groups can be used to support classroom settings where each student is a member of a group for each registered class. Multimedia functionality such as interactive voice and collaborative workspaces is supported.

Some components of the RCT are prone to denial-of-service attacks (DoS) attacks because of the amount of computation required by the components. A DoS attack occurs when a malicious client (or set of malicious clients) overloads a service with computationally expensive requests, hindering timely response to legitimate clients. When deploying RCT in an environment where some clients may not always be trusted, it is necessary to provide protection from such attacks. Also consider that in certain classroom settings it is not difficult for malicious users to gain unauthorized access. The client puzzle protocol [5] (CPP) is a possible feature which could be used to mitigate this security concern. CPP protects a component by intercepting client requests and refusing service until the client provides a solution to a small mathematical problem. We discuss more details in Section 6.

The CPP protocol must be enabled to protect the DoS prone RCT components. Adding this feature requires changes to many different server components in the RCT implementation and also to the client. The changes are crosscutting: programs running on different platforms and

in different languages might need changing; the RCT client is written in Java and the server is written in C++ using ORBs from two different CORBA vendors (resp. JacORB and TAO). Since some platforms may have performance or battery limitations (e.g., PDAs or laptops) or be remotely located, different evolution strategies should be allowed and allowed to interoperate. Changes to different elements must be made consistently to ensure correct interaction. Changes must be properly deployed in the different elements, otherwise versioning errors may result. Since the functions for CPP may apply to other applications, it would be desirable to reuse the same implementation, should the platforms be compatible.

3 Current Approaches

There are a variety of approaches to dealing with crosscutting features. Our survey here is limited by space to be representative rather than exhaustive; no judgment of omitted or included work is implied.

Several *language-based techniques* have been proposed. Classical syntactic program transforms [6] were perhaps among the earliest to provide the capability of broad changes to programs. Reflection [7] provided means of introducing crosscutting changes at run-time in languages such as Smalltalk. Composition Filters [8] enables programmers to cleanly separate functional object behavior from code dealing with object interactions. Compile-time [9, 10] reflection in C++ (Open C++) and Java (OpenJava) has been developed and extended to load-time in Java using byte code editing [11]. Meta-Classes were exploited in the FRIENDS [12] project using OpenC++ in order to add fault tolerance and security to distributed programs. Mixin-layers [13] provide a way of adding features to methods in several different classes simultaneously. Implicit Context [14] is a method for separating extraneous embedded knowledge (EEK) (or crosscutting knowledge) from the design of a program, and re-weaving it back in later. Monads and monad transformers [15] have been used in lazy, pure functional languages to capture crosscutting features such as states and side-effects. They work by encapsulating the basic notion of a *computation*, and then allowing fundamental evaluation mechanisms such as value propagation to be overridden. Recently, approaches such as HyperJ [1], Aspectj [2], and Aspectual Components [16] provide differing approaches to implementing crosscutting features in Java. A detailed comparison (but see [17] for a comparison of compositional *vs.* aspectual views of program evolution mechanisms) of these differing approaches is beyond the scope of this paper; suffice to say we are interested in a DH setting, thus transcending language boundaries. While details vary, most of these languages provide two features: a *hook* or pattern, for describing where to insert crosscutting changes, and then a way to program the changes themselves. Since our approach uses the "hook" mechanism from AspectJ, we discuss it in more detail here.

AspectJ provides a pattern mechanism, called *pointcuts* for capturing groups of events, called *joinpoints* that may occur during a program's operation (such as method calls/receptions, constructor calls, field accesses, and exception events). The pattern-matching mechanism includes regular expression matching, with wild-carding over fragments of method names, function signatures, and types etc. Extra code, called *advice* can be associated with pointcuts, and is inserted by the AspectJ compiler into the join-points. Advice can inspect and modify data that are available at join-point events (*e.g.* method-call arguments and return values), and can create new data dynamically that is only shared with other advice. Our work uses these ideas for *modeling* crosscutting changes to distributed systems at the IDL level. However, the distribution, heterogeneity, and versioning problems that arise in our context, require new and different *implementations*.

Middleware-based approaches are certainly relevant. Some works exploit language-based reflection in the middleware [18] and other approaches use specially constructed reflective ORBs [19–21]. Communication reflection reifies the channels between client and server to address adaptation on a per-message level [22]. SOM [23] was an early approach to support reflection directly in the middleware. Interceptors [24, 25] and filters [26] provide a way of inserting extra functionality into every invocation that originates or arrives at a request broker; middleware-specific APIs provide means for interceptor code to reflect upon the details of the intercepted invocations. While these reflective methods are suitable for implementing cross-cutting services [27], (and for some highly dynamic services may be the only way to do it) the use of the low-level reflection APIs, along with the need for frequent use of type-casting makes programming difficult and error-prone; thus it would be preferable to use more statically checkable methods when possible. Proxies and wrappers [28, 29] are another approach. However, they are typically tailored for a specific application object interface; so thus, it would not be possible to reuse a wrapper to implement the same security policy on entirely different components.

Container models [30, 31] address this problem through code generation. They provide a *fixed* set of services (depending on the container vendor) to application components. Via configuration files and code-generation, services selected from a given set can be added to any component. Newer containers such as JBoss [32] application server provide aspect-oriented development and deployment of container services. In contrast to our work, these containers provide no support for matched client-server services.

Section 7 surveys several other closely related works, that are easier to relate to ours after DADO details have been presented.

4 DADO Overview

As illustrated in Section 2, late-bound, crosscutting functions such as security require extra functional elements (which in DADO we call *adaptlets*) to be located together with (potentially distributed) application software components. A client-server pair of adaptlets would constitute a distributed DADO service. We begin with a discussion of the main goals of our project. Then we describe the features of DADO that address these challenges.

4.1 Desiderata

Heterogeneity and Communication Adaplets may need to exchange information and co-ordinate with each other, and/or with the application components. While this is strongly analogous to AspectJ, adaptlets must communicate and co-ordinate in a *distributed heterogeneous* context. The adaptation mechanisms (source/binary transformation, runtime wrapping) may depend on the platform; even so, heterogeneous adaptlets should co-exist and inter-operate correctly. Just as IDL plays a central role in managing heterogeneity and communication in CORBA, we seek an abstract model of adaptlets.

Binding and Deployment It would be desirable to support *late binding* and *flexible deployment* of DADO services. Consider that container standards such as J2EE allow independent container developers to develop services that are customized for specific applications at deployment time. Likewise, we would like to allow vendors to build services consisting of DADO services, independently of application builders, and then allow deployment experts to combine services and applications to suit their needs. Here we also seek an abstract, platform-independent way of specifying binding and deployment.

Dynamic Service Recognition Several adaptlets, supporting different features, may be associated with an application component; clients and servers must deploy matching sets of adaptlets. In a dynamic, widely distributed context, clients may become aware only at run-time of the adaptlets associated with a server object. Thus adaptlets may be need to be deployed at run-time.
Flexible Communication and Co-ordination The interaction between a matched pair of client and server adaptlets may not be simple and monolithic. Under different circumstances, the client adaptlet may require and request different functions (with different parameters) that are supported by a server adaptlet (just as a distributed object can support several distinct methods). Likewise, the server adaptlet may request different post-processing functions on the client side. A client adaptlet can refer to it's server "partner" via the reserved name "`that`" (and vice versa). However, for efficiency, it would be better to have only a single invocation event through the middleware (e.g., a single CORBA synchronous call).

4.2 DADO Features

Modeling, Type-Checking, and Marshalling DADO employs an enhanced IDL and code-generation to support the following:

- Explicit interface-level modeling of adaptlets and their interaction with application components.
- Ability to implement adaptlets in different languages, while supporting:
- safer interaction (via static type-checking) between adaptlets, with automated generation of marshaling and instrumentation code.

Pointcut based Binding DADO separates services (which describe the interfaces supported by adaptlets) from a deployment description, which specifies the precise deployment context of a service (using a pointcut language and inheritance mechanism similar to AspectJ). This allows a deployment expert to tune the connection between DADO services and different application components. The binding language is agnostic with respect to the implementation; DADO adaptlets could be incorporated into the existing application using static transformations (binary or source) or dynamic wrapping, depending on available tools, performance issues, etc.
Multiple Contextual Invocations DADO allows adaptlets on the client and server side to communicate via messages. However, rather than inducing additional middleware invocations, multiple messages are piggy-backed within the single pre-existing application invocation.
Transparent Late binding DADO clients transparently (without additional programming) discover the services associated with a server, and deploy additional adaptlets as needed.

5 Dado

Currently, the process of building *DH* systems using middleware such as CORBA begins by describing the high-level interfaces using IDL. IDL specifications are then implemented by developers on different platforms and perhaps in different languages. When implementation is complete, the users of the distributed system can run ORBs on a network as suited to the application and organizational needs, and deploy the constituent application objects.

Dado brings three new roles into this process (see Figure 1): a *service architect*, *service programmer*, and a *service deployment expert*.

This service architect can design a *DH* service that implements a crosscutting feature. The process begins with the description of a Dado `service` in an enhanced IDL (known as DDL).

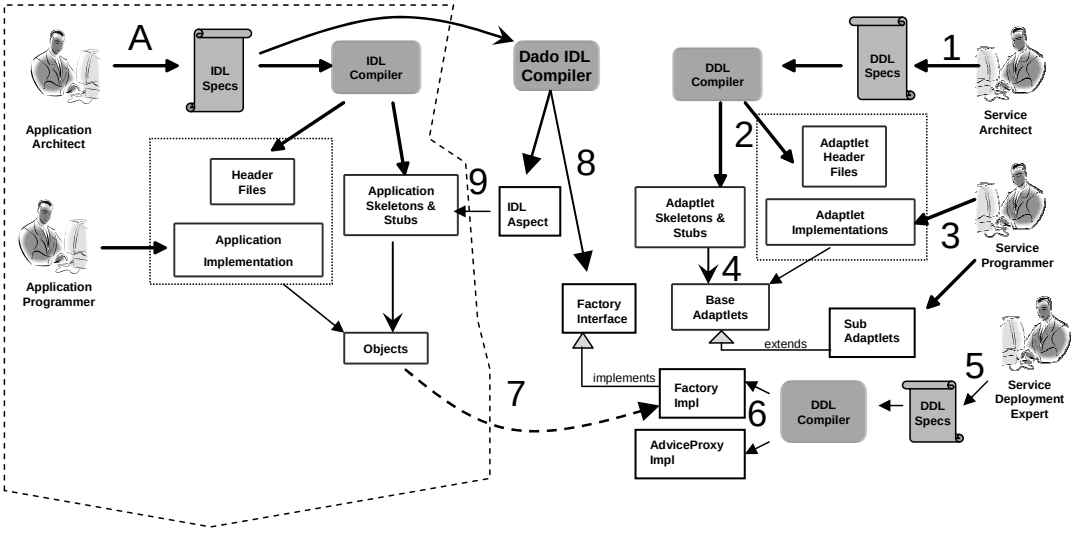

Fig. 1. *Dado Development Process* The left hand side (within the dotted lines) indicates the conventional CORBA process. On the right, the Dado service development begins (1) with modeling the interface of Dado adaptlets using DDL; from this the DDL compiler generates (2) marshaling code, and typing environments for adaptlet implementations. The programmer writes (3) the adaptlet implementations and links to get (4) the adaptlets. Now, the deployment expert extends adaptlets using (5) pointcuts, and these are used to generate (6) a FACTORY for obtaining `AdviceProxy` implementations at run-time (7). Applications are instrumented independently by the Dado IDL compiler. An AspectJ aspect or TAO PROXY is generated (8) and used to modify the existing application (9).

A service is a pair of client/server *adaptlet interface* descriptions, which consist of several operations, just like an IDL interface. These interfaces are then compiled using DDL compilers for a target implementation language (e.g., Java or C++), producing marshaling routines and typing environments. The implementation then proceeds by service programmers.

The deployment expert binds an implemented service to a given application by specifying bindings using a pointcut language. The deployment expert will need to understand both the application and the service, and select the bindings based on the specific installation. Currently, applications are instrumented in two different ways: through generated AspectJ code (for Java) or generation of middleware wrappers (for C++). A specialized CORBA IDL compiler (the Dado IDL compiler) is provided for this purpose.

This section presents the details of the Dado features outlined in previous section, using the running example of an invocation timing feature. The current Dado implementation includes the DDL language for adaptlets, a DDL compiler for C++ and Java, and the Dado IDL compiler to modify applications for two different open-source ORBs (JacORB [**?**] and the TAO [**?**] ORB).

5.1 Timing Example

In this section we use the running example of a remote invocation timing monitor. This simple example is used to ground the discussion of newly introduced concepts; we return to the security example in Section 6. Here we discuss the high-level requirements for the timing implementation.

One could write code (e.g., using interceptors [25, 33] or aspects) to attach to the client that will record the time each invocation leaves and each response arrives. However, the client may also want the invocation arrival-time at the server, and the reply sending-time in order to compute the actual processing time. This scenario demands more coordination between interacting client and server adaptations.

This section, through the example, demonstrates how Dado provides four critical elements. First, clients must be able to ask the server for timing statistics; only some clients will request this feature. Second, servers may return data through a type-checked interface. Third, clients need some way to modify existing software to add logic for requesting timing statistics; different means should be allowed. Finally, client and server adaptations should be coordinated; clients will not request timing statistics from servers unable to provide them.

5.2 Dado Description Language

Dado adopts the philosophy that IDL-level models provide an excellent software engineering methodology for distributed systems. In addition to promoting better conceptualization of the design, one can construct tools to generate useful marshaling code and typing environments for static type-checking.

To gain an understanding of how a service architect works with DDL, it is useful to examine the elements of the DDL language presented in Figure 2. DDL introduces the notion of a *service* (line 1) that refers to a crosscutting feature. A service comprises a client and/or server *adaptlet* (lines 3 and 4).

A client and server adaptlet can be described together in a shared scope by declaring both inside the `service` element. The service can be viewed as a role-based collaboration [**?**] of client and server adaptlets. The client and server adaptlets described in this way share a common name, used to coordinate run-time deployment. The scope also provides a way to coordinate client and server adaptlets at the interface level. This may be needed t deployment time,

(1) Adaptlets :: **service** Name Inherits? { Pointcut* Client? Server? }**;**
(2) Inherits :: **:** Name*
(3) Client :: **client** { Element* }**;**
(4) Server :: **server** { Element* }**;**
(5) Element :: Pointcut | AdviceOp | Request | Context | AdviceApp
(6) Pointcut :: **pointcut** Name **(** FormalArg* **) :** PCExpr **;**
(7) PCExpr :: PCD | PCD **and** PCD | **!**PCD | PCD **or** PCD
(8) PCD :: OpPCD | PCApp
(9) OpPCD :: **oneway**? *TypeMatch TypeMatch* **::**(Name | Wildcard)**(** PCDParam* **)**
(10) PCApp :: Name **(** Name* **)**
(11) PCDParam :: **..** | *TypeMatch* | Name
(12) AdviceOp :: Around | Operation
(13) Around :: **around** *IDLType* Operation
(14) Operation :: Name **(** *FormalArg** **);**
(17) AdviceApp :: (**before** | **after** | **around**) **(** *FormalArg** **) :** PCApp { *OpApp* }
(18) OpApp :: Name **(** *Name** **);**
(15) Request :: **request** Operation
(16) Context :: **context** Operation
(19) FormalArg :: (**in** | **out** | **inout**) *IDLType* Name
(20) Name :: *String*

Fig. 2. Dado Description Language (DDL) EBNF : Used by Service Architects (item 1) and Deployment Experts (item 5) in Figure 1. Bold font represents terminals; for simplification, non-terminals in italics are not shown; delimiters (such as commas) have been elided. TypeMatch (not shown) provides quantification over IDL types.

through pointcuts (lines 6-11) and advice (lines 12-18), and at run-time, through `request` and `context` messages (lines 15-16).

5.3 Adaptlet Inheritance

Adaptlet interfaces can be extended through object-oriented multiple interface inheritance (line 2). Client adaptlets in a sub-`service`[5] implicitly inherit from the client adaptlets in the super-services; server adaptlets are similarly extended[6].

Inheritance provides for the description of generic services that are subsequently tailored for specific applications. The service-architect will write interfaces for client and server adaptlets, which are then implemented by service-programmers. A deployment expert can then add an adaptlet to an application by extending either the adaptlet interfaces (through DDL multiple inheritance), adaptlet implementations (through Java single inheritance or C++ multiple inheritance), or both. The inheritance model expands the notion of AspectJ's `aspect` inheritance to client/server adaptlet collaborations.

[5] We use the terms sub-,super-, and base-service/adaptlet in analogy to sub, super, and base classes.
[6] This is analogous to the inheritance provided by mixin layers where classes in one layer simultaneously extend from classes in another layer, but specialized for the case of client/server interface definitions.

5.4 Adaptlet Operations

Each adaptlet may support several operations, which may be of three different kinds. First, `before` or `after` advice operations (line 14) are specified as standard CORBA operations (with a void return type) or optionally tagged as `around` (line 13) (including a specific return type). Advice operations (Section 5.7) provide only the signature of an advice; the body is implemented in a programming language binding. Advice operations may be bound via `pointcut` patterns to application interfaces. They basically provide additional code that is run *every time* certain operations defined in an IDL interface are invoked. Code in the distinguished `around` advice have full access to modify intercepted application information, whereas other advice do not (as in AspectJ). Second, DDL services can also include `request` operations (line 15). These are asynchronous events that may be raised by any adaptlet advice causing an operation to be invoked on a matched adaptlet partner (Section 5.8). Third, `context` operations (line 16) provide similar functionality but must be polled for by an adaptlet partner, rather than triggering operation invocation (Section 5.9).

```
(1) service Timing {
(2)    client {
(3)       request timeResult(long received, long sent);
(4)       void timedOperation();
(5)    };
(6)
(7)    server {
(8)       request timeRequest();
(9)    };
(10)};
```

Fig. 3. Timing `service` : `client` and `server` adaptlets. Written by Service Architect as in item 1, Figure 1.

DDL for Timing Example In Figure 3, we see the base-service for the `client` and `server` Timing adaptlets. Each adaptlet includes one `request` operation.

When `timeRequest` (line 8) is invoked by the client adaptlet, a `request` message is added to the application invocation and dispatched to the server-side adaptlet. The server adaptlet can respond to a `timeRequest` by taking two timing measurements to determine the actual execution time for that application method invocation. It can then report the results back to the client, using the client request `timeResult` (line 3).

In order to trigger this exchange of `request` messages some advice (line 4) must be added to the client side. Later Section 5.7 shows how a deployment expert can extend the client adaptlet to include the advice.

5.5 DDL Language Mapping

The DDL compiler generates typing environments, as well as stub and skeleton functions. The generated typing environments (C++ header files or Java interfaces) ensure that the pieces of

a `service` can safely inter-operate. Adaptlets can currently be implemented in either C++ or Java but must be written in the same language as the application. This is primarily for performance reasons; if adaptlets are in a different language, it would be necessary to endure an expensive traversal of middleware to get from an application object to an adaptlet.

```
(1) class Timing_Client_Impl implements Timing_Client {
(2)
(3)    Timing_Server_Server_Stub _that;
(4)    void initialize(TimeStockServer_Server_Stub that) {
(5)       _that = that;
(6)    }
(7)    void timedOperation() {
(8)       _that.timingRequest();
(9)    }
(10)   void timeResult(long received, long sent) {
(11)      System.out.println(recevied + " " + sent);
(12)   }
(13)}
```

Fig. 4. Timing Implementation. Written by Service Programmer as in item 3, Figure 1.

Each adaptlet interface in a `service` is used to generate the appropriate programming language interfaces and the specialized communication code for `request` and `context` operations. These are then used by service programmers to build the adaptlet implementations. Figure 4 shows the Java implementation code for the client base-adaptlet.

From the `service` in Figure 3, the DDL compiler can generate the interface `Timing_Client`[7] which is referenced in Figure 4, line 1. The declaration of the `interface` (not shown, however implied by the implementation) includes three methods.

The first, `initialize`, allows the implementation to obtain a reference to the server side stub of the matched partner. It is important to note that this is not a standard remote object stub but a custom stub generated for `request` and `context` operation semantics. The reference can be used to communicate through operations declared on the server side, such as `timeRequest` (Figure 3, line 8).

A second operation, `timedOperation` has been added to trigger the timing `requests` for particular client operations. This is achieved on line 8 by invoking the `timeRequest` operation on the provided server adaptlet stub. However, the actual pointcut is left for a deployment expert to fill in later.

The third method defined in the interface is `timeResult` (lines 10-12). This method will be invoked whenever the server-side adaptlet adds a `timeResult request` by using a `client` adaptlet stub. Here, the implementation simply prints out the results.

So far, in this section we provided an introduction to the adaptlet interface language called DDL. We saw how the DDL is used to generate interfaces implemented by `service` programmers. The implementation could make use of stubs that refer to the matched `client` or `server` adaptlet. The details presented were motivated by three concepts: that adaptlet implementation should be provided a statically typed-environment for communication, that com-

[7] The suffix _Client is appended for a client adaptlet and _Server for a server adaptlet.

munication details would be automatically generated, and that generic and application specific components could be separated through inheritance.

5.6 Application Instrumentation

In order to trigger adaptlet behavior at runtime, application code must somehow be modified, or execution intercepted to capture the right events. A wide range of binary and source-code, static and dynamic instrumentation mechanisms have been reported [9, **?**,2, **?**]. Middleware, also, can support highly dynamic reflective mechanisms [34, 35]. In keeping with the *DH* philosophy, we have explored heterogeneity in the implementation of the triggering mechanism. Thus, while the pointcut specifies the high-level design of the binding, different implementation strategies are possible.

Applications are prepared statically, for adaptation by adaptlets at run-time. This is done in a generic fashion. The preparation steps do not depend on the adaptlets that will be added. Instrumentation takes place on the client and server CORBA stubs and skeletons. During the instantiation of stubs or skeletons, a singleton factory is consulted to obtain a reference to a list of decorators which wrap the stub or skeleton. Each decorator also acts as an adapter from the interface of an application to a particular adaptlet. The decorators are hard-coded to know what advice should execute and how to map parameters from the application to the adaptlet operations. Adaptlets can be added and removed at run-time through management of decorators, however in this paper we only focus on the case where clients must add adaptlets to match those of a particular server.

For C++, we make use of two mechanisms available in the TAO ORB: the TAO smart proxies [24] and the Object Reference Factory [33]. Similar mechanisms do not exist in JacORB, so AspectJ was a natural alternative.

Both the smart proxy and the Object Reference Factory essentially provide the same functionality for our purpose; details vary in the implementation. The smart proxy acts on the client and the Object Reference Factory acts on the server. Both allow stubs or skeletons to be wrapped by decorators. Equivalent code that was implemented in the C++ decorators is implemented as generated AspectJ advice for the Java implementation.

Naturally, client- and server-side adaptlets, even if using different languages or different instrumentation mechanisms, are fully inter-operable. The adaptlet programmer remains agnostic with respect to the actual instrumentation mechanism that is used to trigger the adaptlet.

5.7 Advice Operations

Recall that Dado introduces several new service-related roles into the software process: a *service architect*, *service programmer*, and a *service deployment expert*. When a service architect decides that some additional behavior on the client or server of a distributed application is desirable, she can add an advice operation to the interface of an adaptlet. The service programmer has the obligation to implement each advice. The deployment expert can then add the behavior specified by the advice interface to a specific application object by writing an appropriate pointcut. Additionally, the deployment expert can specialize the advice implementation by overriding any advice using a sub-adaptlet.

In the example, to deploy `Timing` adaptlets for a given application object, the server-side would make the `service` available by registering a server `Timing` adaptlet. When clients become aware of those server objects, the Dado run-time will automatically deploy client adaptlets based on the client's pointcuts (Figure 5). Thus, for matched client/server adaptlets

```
(1) service TimeStockServer : Timing {
(2)    client {
(3)       before call(float StockServer.getQuote(string)) :
(4)              timedOperation();
(5)    };
(6)};
```

Fig. 5. Timing deploy. Written by Deployment Expert as in item 5, Figure 1.

each client-side pointcut is implicitly conditional on the deployment of the particular server-side adaptlet for each server object. Depending on which server object an invocation is destined for, helps determine whether the pointcut applies. Server adaptlets are only made aware of the existence of a matched client-side adaptlet when a `request` or `context` message is found. This assymteric setup follows the CORBA distributed object philosophy where client processes do not export any referenceable identity.

In this deployment (Figure 5) the client would like invocations to the `getQuote` operation to be intercepted by the advice `timedOperation`. Here, a client deployment expert extends the `Timing` adaptlet interface independently from the server. The server side is unaware of the pointcut binding on lines 3-4. The server-side does not need to specify any additional pointcut instructions, as the `request` operation `timeRequest` is invoked dynamically by the client-side adaptlet.

In this section, we detailed how advice operations would be used by the different roles in the Dado development process and how they are implemented in our prototype middleware tools. Next, we see how adaptlets communicate by modifying client/server application communication using `request` messages.

5.8 Adaptlet Requests

Some services can be implemented simply by executing advice on the client- or server-side. However, in some cases, additional information may need to be sent along from the client to the server side adaptlet (or vice versa).

In our running example we presented a service where a client-side adaptlet can request that a matching server adaptlet calculate server processing time for specific invocations, and then communicate this information back to the client adaptlet. This additional information conveyed between client and server adaptlets is contextual. It must be associated with some original CORBA invocation; in other words, it would not make sense to communicate this information ”out-of-band” by introducing new operations into the application IDL. So, the timing behavior by the server adaptlet must occur before and after the processing of the invocation for which the client adaptlet requested measurements.

The service architect can include operations tagged with the `request` modifier keyword to provide an extra communication path between `client` and `server` adaptlets that is associated with the current CORBA invocation. The body of `client` and `server` advice can be programmed to add `request` messages by using a reference which exposes the interface of request operations available to a client adaptlet by the server adaptlet and vice versa. A client-side `request` acts as a conditional `after` advice that is triggered by a server-side adaptlet, which provides the actual arguments. Likewise, a server-side `request` acts as a conditional `around` advice, triggered by the client-side.

Adaptlet requests give service developers more ways of programming interactions between client-side and server-side adaptlets. Consider that a client adaptlet may require different types of actions to be taken at the server side. As a very simple example, a per-use payment service adaptlet attached to a server object might accept e-cash payments, or a credit card. Another example is authentication. It could be based on Kerberos-style tokens, or on a simple password. We could include both options as possible parameters, in a single operation signature, along with an extra flag to indicate the active choice; this leads to poorly modularized operations with many arguments. Rather, we take the "distributed object" philosophy of supporting different requests at a single server object; we allow adaptlets on either side to support several different requests.

Adaptlet requests are implemented as one-way, asynchronous "piggy-backed" message that are sent along with an invocation (from client to server) or a response (vice versa). Since multiple services can be present simultaneously, the requests are queued on each client and packaged with the original invocation using the CORBA Service Context. This Service Context is an extensibilty element in the CORBA IIOP protocol.

Advice and `request` play different roles in adapting the execution of a distributed application. Advice operations are used to add code at points in the program determined by pointcut based deployment. Pointcuts create a connection between client programs and client adaptlets or server objects and server adaptlets only. The connection between client adaptlets and server adaptlets is made through request messages and is completely dynamic. The request messages serve both to convey additional information and invoke behavior to process the information. In essence, request messages provide a form of dynamic per-invocation adaptation [36] while supporting type-checked interactions and modular design through IDL declaration.

A timing service such as the one presented could be programmed using the Service Context API and Portable Interceptor API already available in the CORBA standard; however such an approach lacks the clarity of an explicit IDL model, static type-checking, and generated communication code for varying invocation semantics. These are properties that middleware programmers rely upon and we have attempted to build Dado as a natural extension of middleware for aspect-oriented software development.

5.9 Context Operations

In some circumstances a service may require extra data to be shared by the client and server adaptlets. Previously, we described how this was achieved using a `request` message. When a `request` message arrived at a matched adaptlet, some operation was invoked that matched the signature of the message. This provided the semantics we required for conditionally executing timing requests.

In other situations, adaptlet advice will be required that should not be executed conditionally, but still needs to share data with it's partner. For example, consider an access control advice which will check client credentials and throws an exception when encountering an invalid password. For this situation we provide the `context` operation which provides for the flow of context directly between advice.

Instead of being dispatched to a method site, context messages are unmarshaled and stored as thread local variables. The provided actual arguments of the context operation can be polled for by invoking an operation of the same name on its matched adaptlet stub. This operation returns either true or false, indicating whether the message is available. We further show how `context` is useful in the implementation of the CPP server adaptlet.

5.10 Transparent Late Service Binding

In a WAN environment such as the Internet, where servers are discovered at run-time, clients cannot predict the set of services provided by (or required by) a particular server until it is located. Static approaches that install new services based only on type information cannot easily provide this kind of late binding.

When server objects are associated with a Dado service (this happens at deployment time via a configuration file), they are assigned an external object reference that is used by the client side run-time to detect the applicable services.

Essentially, the reference encodes information about the adaptlets associated with this object. Our implemenation uses the Tagged IOR[8] Components which is an extensibilty element of the CORBA IOR format. This information is used by the Dado interception logic on the client-side to transparently engage the corresponding client-side adaptlets.

The process is graphically illustrated in Figure 6. When an application object registers itself with a naming service, the reference encodes all active services (Arrow 1). Subsequently, a retrieved reference (2) is intercepted by the Dado runtime, which decodes the applicable service identifiers from the reference. For each operation in the interface, it injects the appropriate advice into the execution path of invocations originating from the client.

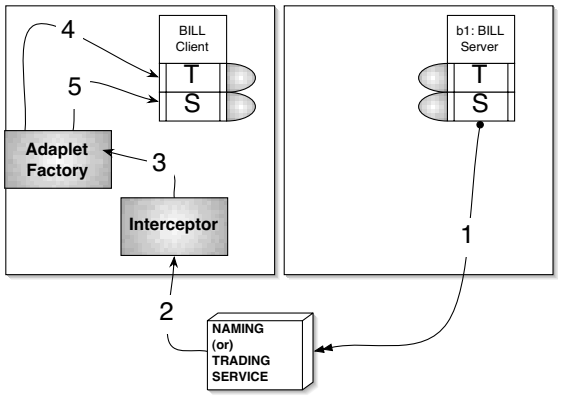

Fig. 6. *Late-binding service adaptations* (1) Server object, with Security and Transaction adaptlets, named "b1" of type "Bill" is registered with a Naming service. The identifiers "Transaction" and "Security" are tagged to the external object reference. When client looks up object named "b1", the returned object reference (2) is intercepted by Dado component. Dado attempts (3) to find client-side adaptlets for "Transaction" and "Authentication" from client-side factory. Factory creates and binds transactions (T) and security (S) adaptlets to client application object.

The late-binding process described in this section ensures that client adaptlets are deployed to match adaptlets on the server and removes the responsibility of detecting applicable services for each object from the service programmer.

5.11 Measurements

The data presented is in the style of micro-benchmarks: we measure the incremental effect of the actual additional marshaling work induced by the new communication code (generated by

[8] Interoperable Object Reference

DDL compiler), as well as for dispatching adaptlet advice and request. For this reason, we use null advice and request methods that do no computation, so that we can focus primarily on the actual overhead of the runtime.

The measurements were taken for a single client server pair. The client machine was a 1.80 GHz Intel Pentium with 1GB main memory running Linux 7.1. The client middleware was JacORB 1.4 on JDK 1.4. The server machine was an 800 Mhz Intel Pentium Laptop with 512MB main memory running Microsoft Windows 2000. Server software used TAO 1.2 compiled in C++ Visual Studio. The DDL interface to the adaptlet used for performance measurement is shown in Figure 7.

```
service Test{
  client {
    void grabArg(string arg);
    before call(* StockServer.getQuote(arg)) :
           grabArg(arg);
  };

  server {
    request putArg(string arg);
  };
};
```

Fig. 7. Test Adaptlets

As can be seen, there is one client-side advice and one server side request. The client-side advice is bound to a method call (with a single argument of type `string`) by the pointcut. In our implementation, the client-side advice simply captures the string argument from the invocation and calls the server side request, passing along the string argument. So the overhead we are measuring (beyond the normal CORBA invocation overhead) includes the additional cost of: intercepting the invocation on the client-side, dispatching the client-side advice, executing the client-side request stub, marshaling the additional data, transmitting the additional data over the wire, unmarshaling the data on the server side, and dispatching and executing the request implementation on the server side. All measurements given here are for round-trip delays for a simple invocation that sends a “hello world” string. The data is averaged over 1000 invocations, and is given in milliseconds.

Experiment	**100 Base-T RPC (ms)**
1. Plain CORBA	0.65
2. with 1 advice execution, 1 request	1
3. with 10 advice executions, No request	0.68
4. with 10 advice executions, 10 request	1.52
5. Plain CORBA with equivalent raw data Payload for 10 requests	1.38

The first row is the plain unloaded CORBA call, as a baseline for comparison. The second row is a CORBA call with one adaptlet advice, and one additional request. In the third row, we show the effect of applying the advice 10 times. The fourth row shows the effect of executing the advice shown on the second row 10 times, and attaching a request for each advice. The critical *fifth* row shows an interesting comparison: it measures the plain CORBA call, with additional "piggy-backed" data, *exactly equivalent to 10 request messages*, without any adaptlet code whatsoever. This row corresponds to the precise straw-man comparison for sending data without Dado, and corresponds to the way interceptor-based services (such as Transactions and Real-Time, as per [33], page 30 of Chap. 13) are currently programmed.

As can be seen, the advice itself, which does not send any data, does not induce very large overheads (comparing rows 1 and 3, it is about 5% in both cases for 10 advice invocations). We believe the overhead for sending *request*s is largely due to the base cost of marshaling and adding our "piggybacked" messages. Addition of messages is achieved through calls to a middleware-level CORBA API.

6 Client Puzzle-Protocol

As Section 2 discussed, CPP defends against cpu-bound DoS attacks. The protocol works by intercepting client requests and refusing service until the client software provides a solution to a mathematical problem. The time it takes to solve the problems is predictable; fresh problem instances are created for each request. The need to solve puzzles throttles back the client, preventing it from overloading the server. Typically the puzzle involves finding a collision in a hash function, e.g., finding an input string that hashes to a given n bit value modulo 2^m, for $n > m$. Such puzzles are very easy to generate and require about 2^m times as much effort to solve, given a collision-resistant hash function. Here, m is called the difficulty, size, or length of the puzzle. The advantage of CPP over traditional authentication techniques is that CPP defends against DoS even in the face of forged or stolen credentials; additionally, CPP can work in an environment where clients are anonymous. This example is made more concrete through Figure 8.

```
(1) struct PuzzleChallenge {
(2)    long puzzleID;
(3)    short puzzleLength;
(4)    Octet64 puzzlePreImage;
(5)    Octet16 puzzleHash;
(6) };

(7) exception PuzzleException {
(8)    PuzzleChallenge challenge;
(9) };

(10) struct PuzzleResponse {
(11)   long puzzleID;
(12)   Octet64 puzzleSolution;
(13) };
```

Fig. 8. Client-Puzzle Protocol Messages

A server can generate a `PuzzleChallenge`, as in line 1. The `PuzzleChallenge` is delivered to a client through the `PuzzleException` (line 7) which encapsulates the challenge. The server will not honor the client's invocation until it has solved the puzzle. It can provide a solution through the structure on line 10. Now, we discuss the rationale for this protocol.

By providing a solution to the puzzle a client has *proven* to the server that it has performed a certain computational task. In other words, the client has *expended* some amount of CPU resources. Pragmatically, no client can ever obtain an infinite amount of CPU resources. The CPP is designed so that legitimate clients expend a minor amount of resources. However, since a server can increase the difficulty of puzzles as it becomes overload, launching a DoS attack would require more resources than is practical.

6.1 CPP Adaptlets

```
(1) service CPP {
(2)    client {
(3)       around void catchPuzzle();
(4)    };
(5)    server {
(6)       around void CPP() raises PuzzleException;
(7)       context puzzleSolution(PuzzleResponse solution);
(8)    };
(9) };
```

Fig. 9. Generic Cache Adaptlets

Figure 9 shows the DDL for the CPP adaptlets. The client side has one advice used to bracket the execution of the application invocation. Clients receive puzzle challenges through the exceptions that are caught by this code.

The server side includes an advice operation `CPP` (line 6) that should be bound to application operations protected by the CPP. The `context` operation `puzzleSolution` allows clients who have been presented with a puzzle challenge to return their solution.

6.2 CPP Implementations

The implementation of the CPP is completely application agnostic. The only interaction with the application is through modification of the client/server control flow. Server responses are preceded by the issuing of a `PuzzleException` and subsequent verification of the `PuzzleResponse`. First, we present the client implementation where `PuzzleException`s are caught and used to construct the `PuzzleResponse`.

The client implementation, Figure 10, consists of one operation `catchPuzzle`. This code is placed around invocations that may potentially raise a `PuzzleException`. In Dado, a special argument (shown on line 2) is provided to every `around` advice. It provides the same functionality as the `proceed` keyword in the AspectJ language, which invokes execution of the original application code. However, in Dado, no special language is used for the implementation of adaptlets. The `proceed` object encapsulates a reference to the next adaptlet

```
(1) CPP_ClientImpl implements CPP_Client {
(2)    void catchPuzzle(catchPuzzle_Proceed proceed) {
(3)       try {
(4)           proceed.proceed();
(5)       }
(6)       catch(PuzzleException pc) {
(7)           try {
(8)               PuzzleSolution ps;
(9)               ps.puzzleID = pc.challenge.puzzleID;
(10)              recursiveSolver(pc.challenge.length,
(11)                              pc.challenge.puzzlePreImage);
(12)              ps.puzzleSolution = pc.challenge.puzzlePreImage;
(13)              _that.puzzleSolution(ps);
(14)              proceed.proceed();
(15)          }
(16)          catch(PuzzleException pc2) {
(17)              throw new RemoteException("CPP Error");
(18)          }
(19)      }
(20)   }
(21)}
```

Fig. 10. CPP_ClientImpl : Declaration of `_that` and `initialize` not shown.

decorator on the list. When the function `proceed` is invoked, it indirectly causes any other adaptlet advice to be invoked.

We see this in lines (3-5). An attempt is made to contact the server by `proceed`ing with the invocation. In case an exception is raised, it is handled in lines 7-15.

First, the ID of the puzzle is copied (line 9) so that the server may correlate challenges and responses. We note that it is not feasible for a client to guess a solution by forging an ID. Second, a "helper" method, `recursiveSolver` (not shown), is called to solve the puzzle (line 10). Once the puzzle is solved it is added to the invocation through the `puzzleSolution` context message (line 13). Once again the invocation is attempted on line 14. If failure occurs for the second time, the client adaptlet gives up and throws an exception to the application (line 17).

```
(1) CPP_ServerImpl implements CPP_Server {
(2)    void CPP(CPP_Proceed proceed) throws PuzzleException {
(3)       PuzzleResponseHolder response;
(4)       if(_that.puzzleSolution(response) && checkResponse(response)) {
(5)          proceed.proceed();
(6)       else {
(7)          PuzzleChallenge challenge = createChallenge(16);
(8)          throw new PuzzleException(challenge);
(9)       }
(10)   }
(11)}
```

Fig. 11. CPP_ServerImpl : Declaration of `_that` and `initialize` not shown.

Now we turn to the CPP_Server implementation in Figure 11. Here also, only one operation is required to be implemented since the context operation is only a vehicle for communicating PuzzleResponses to the existing advice. The advice is declared on line 2. It must first check for an incoming response (line 4). If one is available, it must also verify its validity (also line 4) by a call to the helper method `checkResponse` (not shown). At this point the invocation is allowed to `proceed`. Otherwise, a new challenge will be created (line 7) by calling `createChallenge` (also not shown). The challenge is delivered to the client through an exception. If client and server deployment have been properly coordinated, the advice on the client side will catch the exception. Otherwise, the exception will propagate to the application. One of our motivations for explicitly describing matched client and server adaptlets is to ensure this proper coordination.

6.3 RCT Implementation

In order to test the compatibility of our implemented Dado software, we set out to deploy an example on a realistic CORBA application. This was done to see if we had made some unrealistic assumptions in our design choices. Although a single case study cannot serve to completely validate our implementation, it has proven useful in understanding some CORBA use-cases that were not previously anticipated. Here we present our example deployment using the CPP adaptlets and the RCT application.

As was discussed in Section 2.2, the RCT is a client/server application running over the CORBA middleware. Since this software was developed by an outside third-party we felt it made a good candidate for our case-study. An initial problem involved the server-side software implemented in C++. This software utilized the Omni ORB, while our software was written for the TAO ORB. Omni ORB does not support the mechanisms we required for transparent wrapping of client and server side objects (through the Object Reference Factory). After an initial review of the RCT implementation we judged that it would be straightforward to "port" the software to the TAO ORB. At this time we became familiar with the implementation details of the RCT server software. On the client side we had very little difficulty. Our AspectJ implementation depends only on the details of the generated CORBA stubs and skeletons. For Java, the interface of this code is standardized by the CORBA Portable Stubs/Server specification.

Previously, we had implemented the generic CPP Java and C++ adaptlets. We hypothesized that some applications would require protection on *some* but not *all* of the operations made available to clients. In these cases pointcuts could be used by a deployment expert to seamlessly apply the CPP protocol across those operations.

After reviewing the server software we noticed that a subset of the operations across the RCT interfaces had code that interacted with the server-local relational database. We identified this as a crosscutting concern. These operations were seen to be highly CPU intensive. Each query to the database involves communication across processes and execution of the relational query engine. The other operations (those not part of the database concern) simply return some values stored in the CORBA objects.

This database concern that we identified serves as the basis for the example deployment of the CPP adaptlets.

CPP Deployment As shown in Figure 12, the pointcut `databaseConcern` is declared inside the `service` scope and shared by both the client and server adaptlets. This allows coordination of advice that throws puzzle exceptions and advice that catches puzzle exceptions. The pointcut matches 76 operations in the interfaces of the RCT application except those declared in either the `Server` or `PingServer` interface.

```
(1) service CPP_RCT : CPP {
(2)    pointcut databaseConcern() : !(call(* Server.*(..)) ||
(3)                                   call(* PingServer.*(..)));
(4)    client {
(5)       around databaseConcern() : catchPuzzle();
(6)    };
(7)    server {
(8)       around databaseConcern() : CPP();
(9)    };
(10)};
```

Fig. 12. CPP_RCT

It is often claimed that a pointcut, such as the one shown here, is more concisely able to capture the *intention* of the developer modifying an application. Here, the databaseConcern is captured and ameliorated by applying the CPP protocol.

6.4 CPP Measurements

To measure the end-to-end overhead of our generated code, we compare our adaptlet tool modified RCT application with a hand-modified RCT application. As we described earlier, our current implementation has not been optimized for performance and these results can help to highlight pitfalls in implementation and to focus further implementation efforts. We measure the round-trip latency for a synchronous remote procedure call for eight different application configurations. Measurements were performed in two different environments. The first environment was a single 1.6Ghz Pentium M laptop computer with 1GB of main memory running both client and server processes in Windows XP Professional. The second environment placed the client process on a 2Ghz Pentium IV computer running Linux at UCDavis and the server process on the aforementioned laptop computer at the University of British Columbia. First we present the details and measurements for the traditional application configurations where no tool support is used. Then we compare these results with the configuration and measurements of the adaptlet modified application.

The baseline "App" configuration measures the round-trip latency for a call to the `get_id_from_user_alias` function on the RCT server. This was chosen as a representative application function for convenience because its execution presumes no particular application state. We hand-modified the original Java RCT 1.0 client application to take timing measurements using the `java.lang.System.currentTime` function before and after 10,000 calls in the local environment and 100 calls in the Internet environment. The values presented are the average over the 10,000 (100) calls in milliseconds. We performed ten trials of the 10,000 (100) calls and took the average of each measurement. The standard deviation for the local measurements was quite small; for example, .0014 in measurement 1. In the case of the Internet measurements, the deviation was more significant; for example, 4.19 in measurement 3. These measurements were performed with and without the original server function body present. The server code is the RCT 1.0 application running on the TAO CORBA ORB compiled under Microsoft Visual Studio for C++. Measurements where the function body was removed are labeled as "App(null)" and are presented to see just the overhead of communication through the middleware.

In the "App with CPP" configurations we implemented the CPP protocol over the original application by adding a new application function, `get_challenge`, and modifying the signature of the `get_id_from_user_alias` function. A call to `get_challenge` is made before each call to the original functions to obtain a `PuzzleChallenge`. The signature of

the application function was widened to include a parameter for the `PuzzleResponse`. We include measurements where the `PuzzleChallenge` generation and `PuzzleResponse` solving code are removed in order to show just the communication costs. These measurements are labeled as "CPP(null)". These configuration serve as a baseline for including the CPP feature but not using any support from Dado.

Experiment	**RPC latency (ms) local**	**RCP latency (ms) Internet**
1. App(null)	.1274	77.40
2. App(null) with CPP(null)	.2467	148.6
3. App with CPP(null)	1.207	151.6
4. App with CPP	132.3	349.4

Fig. 13. Traditional Application Measurements

By comparing measurements 1 and 2 in Figure 13 we see that the overhead for CPP communication is about 93% (local) and 92% (Internet) . Recall that the "App with CPP" configurations include an additional function call so two RPC calls are required instead of one. Comparison of local measurements 2 and 3 shows the cost for the actual `get_id_from_user_alias` function, .9603ms. Measurement 4 includes the cost for the application and CPP protocol functions. The CPP generation and solving functions add two orders of magnitude to the local end-to-end latency (131.3ms) and more than doubles the latency over the Internet.

Four more measurements were made in order to see the overhead from our tool generated code. The first measurement shows the cost of 10,000 (100) RPC calls hand-modified to use the PortableInterceptor API [25] to piggyback a `PuzzleResponse` to the client request and a `PuzzleChallenge` to the server response. Our generated code uses these PortableInterceptor API calls to package adaptlet `request` and `context` messages. This measurement was performed to see what percentage of overhead is due to the use of this API. This measurement is labeled as "App(null) with Interceptors".

The final three measurements apply the CPP adaptlets previously presented in this section automatically to the RCT application using our tool support. We include measurements where the adaptlets simply pass dummy challenges and responses, "CPP Adaptlets(null)", in order to highlight the cost of the generated code.

Experiment	**RPC latency (ms) local**	**RPC latency (ms) Internet**
5. App(null) with Interceptors	.3398	77.64
6. App(null) with CPP Adaptlets(null)	.5419	150.9
7. App with CPP Adaptlets(null)	1.641	154.5
8. App with CPP Adaptlets	133.2	349.6

Fig. 14. CPP Adaptlet Measurements

A comparison of local measurements 1 from Figure 13 and 5 from Figure 14 reveals the overhead due to the use of the PortableInterceptor API. As was detailed in Chapter 5 this cost can be high in a local setting, accounting for a 166% overhead or .2124ms. By factoring this cost from measurement 6 we are left with .3295ms which compared to measurement 2 gives us a 33% overhead for our remaining generated code in the local setting. Factoring the PortableInterceptor API out of measurements 7 and 8 gives us an overhead of 18% and .004% respectively when compared to measurements 3 and 4.

In the Internet setting a comparison of measurements 1 through 4 to measurements 5 through 8 shows a smaller overhead. For example, comparing measurement 3 to 7 gives a 2% total overhead. This variation falls within the range of the standard deviation for measurement 3 (4.19ms) and so is fairly insignificant.

Here we conclude with some remarks about the performance of our current tools. Our dependence on the PortableInterceptor API has an obvious drawback in terms of performance. We made the decision to use this approach because it does not require tightly integrating our tools with a particular CORBA middleware. Tighter integration could probably improve performance at the cost of portability. Even still, there is a significant performance overhead for local measurements 6 and 7. Thus we conclude that our current unoptimized implementation may not be suitable for many high performance embedded applications where communication and application latency is low.

7 Closely Related Work

In this section, we discuss closely related work and compare them in greater detail with DADO.

Dassault Systèmes CVM The Dassault Systèmes Component Virtual Machine (DS CVM) [35] is targeted at container-based systems, and allows the implementation of custom container services. The DSCVM comprises an efficient, flexible CVM that essentially supports a meta-object protocol which can be used for instrumentation of middleware-mediated events. This allows the CVM to support the triggering of advice when the CVM executes specific events such component method invocations. Pointcut "trigger" specifications are implemented using the DSCVM events. Advice can be bound to patterns of these events, and thus be used to implement services.

DADO is complementary to DS CVM in that DADO allows elements of crosscutting services to be placed on the client site in a coordinated manner, for reasons argued earlier. DADO also operates outside of a component/container model in "bare-bones" CORBA; thus it must (and does) allow heterogeneity in the implementation of triggering mechanisms such as source transformations, binary editing etc. (See Section **??**). The heterogeneity assumption also influences our design of type-checked information exchange between client and server adaptlets, using generated stubs and skeletons (Section 5.2).

QuO The Quality of Objects (QuO) [37, 38] project aims to provide consistent availability and performance guarantees for distributed objects in the face of limited or unreliable computation and network resources. QuO introduces the notion of a "system condition", which is a user-definable measure of the system, such as load, network delay etc. System conditions can transition between "operating region"s which are monitored by the run-time environments. The novelty in QuO is that adaptations can be conditionally run to respond not only to normal middleware events, but also to region transitions.

QuO's version of adaptlets are confined to a single system. Unlike DADO, Quo provides no special support for communicating information from a client-side adaptlet to a server-side adaptlet.

AspectIX The AspectIX [21] middleware for Java is motivated by similiar goals as the QuO project. The use of *object fragments* allows the placement of adaptive code transparently at the client, server, or at an intermediate location. Policies are used by the client at run-time to determine the composition of fragments, allowing client specific QoS. Unlike DADO, AspectIX does not utilize an interface level model for declarative service/object bindings. Also, generic services or heterogeneity in instrumentation are not considered.

Lasagne Lasagne[36] is a framework for dynamic and selective combination of extensions in component based applications. Each component can be wrapped with a set of decorators to refine the interaction behavior with other components. Every decorator layer is tagged with an extension identifier. Clients can dynamically request servers to use different sets of decorators at run-time. An innovative aspect of Lasagne is the usage of extension identifiers to consistently turn on and off adaptive behavior. However, the use of the decorator-style constrains all extensions to have the same interface. Any additional extension-specific information must be communicated using a "context" object, without the benefits of typechecking or automated marshaling.

Cactus The Cactus[39, 40] framework provides customizations in distributed systems and has been applied in the area of distributed object computing. Software implementing customizations is known as an adaptive component (AC). ACs are injected into a system using the proxy technique. These ACs encapsulate alternative implementations of a specific service ala the Strategy pattern. When important state changes in the system occur ACs can respond by swapping in and out different strategies. ACs may communicate in order to reach consensus on strategy changes however no one particular consensus approach is dictated by their model. Cactus does not utilize static IDL level information for detecting application level events. Also, information shared between ACs is not described in an interface as are Adaplet request messages. Thus Cactus follows the reflective, dynamically typed approach to adaptation.

Software Architecture In software architecture, connectors [41–43] have proven to be a powerful and useful modeling device. Connectors reify the concern of interaction between components, and are a natural foci for some crosscutting concerns. Implementations of architectural connectors have also been proposed [44–47]. Some of these provide specific services [45–47] over DH middleware, such as security. Our work can be viewed as providing a convenient implementation vehicle for different connector-like services in a heterogeneous environment. The DDL language and compiler allow service builders to write client and server adaptlets that provide many kinds of "connector-style" functionality, while the DDL "plumbing" handles the communication details. Furthermore, the pointcut language allows a flexible way of binding this functionality to components, using pattern matching to bind events to adaptlets.

8 Conclusion

We conclude here with several observations about DADO, it's limitations, and our plans for future work.

Design Choices. The design space of a convenient framework to implement DH crosscutting services is quite large, comprising many dimensions such as synchronization mechanisms, scope of data, and the handling of exceptions. The current implementation of DDL has made some reasonable choices, but other choices will need to be explored as other application demands are confronted. Some examples: "cflow-scoped" state, i.e., state that is implicitly shared

between adaptlets across a distributed thread of invocations; services whose scope transcends a matched stub-skeleton adaptlets; other (*e.g.*, synchronous) interactions between adaptlets. We would like to implement an adaptlet-per-object instance policy as well. Currently , only run-time exceptions are supported for adaptlets—in the Java mapping, better static checking would be desirable.

Implementation Limitations. Currently our implementation has some limits. As outlined earlier, the marshaling and dispatch functions need further optimization. Additionally, further tests are necessary using a more realistic client work-load.

Service interactions. Feature interactions are a difficult issue that DADO services must deal with. We note here that it is currently possible to program interactions between two DADO services: one can write a third service that pointcuts adaptlets in each, and responds to the triggering of both by preventing one from running, changing argument values, return values etc. Recently, we have designed a feature mediation protocol for crosscutting distributed features[48]. Feature policies from both client and server are interpreted by middleware to dynamically choose applicable features.

In conclusion, DADO is a model-driven approach to programming crosscutting concerns in distributed heterogeneous systems based on placing "adaptlets" at the points where the application interacts with the middleware. It supports heterogeneous implementation and triggering of adaptlets, allows client- and server- adaptlets to communicate in a type-checked environment using automated marshaling, provides flexibility in communication between adaptlets, allows flexible binding, and late deployment of adaptlets on to application objects. These qualities are made possible through code generation from a platform independent model.

9 Acknowledgments

References

1. Tarr, P.L., Ossher, H., Harrison, W.H., Jr., S.M.S.: N degrees of separation: Multi-dimensional separation of concerns. In: International Conference on Software Engineering. (1999)
2. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: European Conference on Object Oriented Programming. (2001)
3. Murphy, G.C., Lai, A., Walker, R.J., Robillard, M.P.: Separating features in source code: An exploratory study. In: International Conference on Software Engineering. (2001)
4. Coady, Y., Brodsky, A., Brodsky, D., Pomkoski, J., Gudmundson, S., Ong, J.S., Kiczales, G.: Can AOP support extensibility in client-server architectures? In: Proceedings, ECOOP Aspect-Oriented Programming Workshop. (2001)
5. Dean, D., Stubblefield, A.: Using Client Puzzles to Protect TLS. In: Proc. of USENIX Security Symposium. (2001)
6. Balzer, B.: Transformational implementation: An example. IEEE Transactions on Software Engineering **7** (1981)
7. Rivard, F.: Smalltalk: a reflective language. In: Proceedings, Reflection. (96)
8. Aksit, M., Wakita, K., Bosch, J., Bergmans, L., Yonezawa, A.: Abstracting Object Interactions Using Composition Filters. (In: Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming)
9. Chiba, S.: A metaobject protocol for C++. In: Object Oriented Programming, Systems, Languages, and Applications (OOPSLA). (1995)
10. Tatsubori, M., Chiba, S., Itano, K., Killijian, M.O.: Openjava: A class-based macro system for java. In: OOPSLA Workshop on Reflection and Software Engineering. (1999)
11. Chiba, S.: Load-time structural reflection in Java. In: European Conference on Object Oriented Programming. (2000)

12. Fabre, J.C., Perennou, T.: A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach. IEEE Transactions on Computers **47** (1998)
13. Smaragdakis, Y., Batory, D.: Implementing layered designs with mixin layers. In: European Conference on Object Oriented Programming. (1998)
14. Walker, R.J., Murphy, G.C.: Implicit context: easing software evolution and reuse. In: Foundations of Software Engineering. (2000)
15. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Symposium on Principles of Programming Languages. (1995)
16. Lieberherr, K., Lorenz, D., Mezini, M.: Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA (1999)
17. Harrison, W., Ossher, H., Tarr, P.: Symmetrically and assymmetrically organized paradigms of program transformation. Unpublished manuscript (2002)
18. Pawlak, R., Seinturier, L., Duchien, L., Florin, G.: Jac: A flexible framework for aop in java. In: International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection). (2001)
19. Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L.C., Campbell, R.H.: Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In: IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware). (2000)
20. Clarke, M., Blair, G., Coulson, G., Parlavantzas, N.: An efficient component model for the construction of adaptive middleware. In: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware). (2001)
21. Hauck, F., Becker, U., Geier, M., Meier, E., Rastofer, U., Steckermeier, M.: Aspectix: a quality-aware, object-based middleware architecture. In: Proc. of the 3rd IFIP Int. Conf. on Distrib. Appl. and Interoperable Sys. (2001)
22. Cazzola, W., Ancona, M.: (mChaRM: a reflective middleware for communication-based reflection. technical report disi-tr-00-09, disi, universit degli studi di genova, 2000)
23. Coskun, N., Sessions, R.: Class objects in SOM. IBM Personal Systems Developer (1992)
24. Wang, N., Parameswaran, K., Schmidt, D.: The design and performance of meta-programming mechanisms for object request broker middleware. In: Conference on Object-Oriented Technologies and Systems (COOTS). (2000)
25. Narasimhan, P., Moser, L., Mellior-Smith, P.: Using interceptors to enhance CORBA. IEEE Computer (1999)
26. Siegel, J.: CORBA 3 Fundamentals and Programming. Wiley Press (2000)
27. Capra, L., Emmerich, W., Mascolo, C.: Reflective middleware solutions for context-aware applications. In: International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection). (2001)
28. Fraser, T., Badger, L., Feldman, M.: Hardening COTS software with generic software wrappers. In: IEEE Symposium on Security and Privacy. (1999)
29. Souder, T.S., Mancoridis, S.: A tool for securely integrating legacy systems into a distributed environment. In: Working Conference on Reverse Engineering. (1999)
30. Roman, E., Ambler, S., Jewell, T.: Mastering Enterprise JavaBeans. Wiley (2001)
31. Troelsen, A.: C# and the .NET Platform. Apress (2001)
32. http://www.jboss.org: JBoss. (4.0 edn.)
33. Object Management Group: CORBA 3.0 Specification. (3.0 edn.)
34. Blair, G., Campbell, R., eds.: Reflective Middleware. (2000)
35. Duclos, F., Estublier, J., Morat, P.: Describing and using non functional aspects in component based applications. In: International Conference on Aspect-Oriented Software Development. (2002)
36. Truyen, E., Vanhaute, B., Joosen, W., Verbaeten, P., Jorgensen, B.N.: Dynamic and selective combination of extensions in component-based applications. In: International Conference on Software Engineering. (2001)
37. Loyall, J., Bakken, D., Schantz, R., Zinky, J., Karr, D., Vanegas, R., Anderson, K.: QuO Aspect languages and their runtime integration. In: Proceedings of the Fourth Workshop on Languages, Compilers and Runtime Systems for Scalable Components. (1998)

38. Vanegas, R., Zinky, J., Loyall, J., Karr, D., Schantz, R., Bakken, D.: Quo's runtime support for quality of service in distributed objects. In: International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware). (1998)
39. Chen, W.K., Hiltunen, M., Schlichting, R.: Constructing adaptive software in distributed systems. In: International Conference on Distributed Computing Systems. (2001)
40. He, J., Hiltunen, M.A., Rajagopalan, M., Schlichting, R.D.: Providing qos customization in distributed object systems. (In: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001))
41. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: International Conference on Software engineering. (2000)
42. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes **17** (1992)
43. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology **6** (1997)
44. Ducasse, S., Richner, T.: Executable connectors: towards reusable design elements. In: Foundation of Software Engineering. (1997)
45. Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., Zelesnik, G.: Abstractions for software architecture and tools to support them. Software Engineering **21** (1995)
46. Dashofy, E.M., Medvidovic, N., Taylor, R.N.: Using off-the-shelf middleware to implement connectors in distributed architectures. In: International Conference on Software Engineering. (1999)
47. Spitznagel, B., Garlan, D.: A compositional approach to constructing connectors. In: Working IEEE/IFIP Conference on Software Architecture (WISCA). (2001)
48. Wohlstadter, E., Tai, S., Mikalsen, T., Rouvellou, I., Devanbu, P.: GlueQoS: Middleware to Sweeten Quality-of-Service Policy Conflicts. Under review for ICSE (2004. http://rickshaw.cs.ucdavis.edu/npaper.pdf)