An Aspect-oriented Approach to Bypassing Middleware Layers

Ömer Erdem Demir
Prémkumar Dévanbu
Dept. of Computer Science, UC Davis
devanbu@cs.ucdavis.edu

Eric Wohlstadter
University of British Columbia, Vancouver, Canada
wohlstad@cs.ubc.ca

Stefan Tai
IBM Research, Hawthorne, New York, USA
stai@us.ibm.com

Abstract

The layered architecture of middleware platforms (such as CORBA, SOAP, J2EE) is a mixed blessing. On the one hand, layers provide services such as demarshaling, session management, request despatching, quality-of-service (QoS) etc. In a typical middleware platform, every request passes through each layer, whether or not the services provided by that layer are *needed* for that specific request. This rigid layer processing can lower overall system throughput, and reduce availability and/or increase vulnerability to denial-of-service attacks. For use cases where the response is a simple function of the request input parameters, bypassing middleware layers may be permissible and highly advantageous. Unfortunately, if an application developer desires to selectively *bypass* the middleware, and process some requests in the lower layer, she has to write platform-specific, intricate low-level code. To evade this trap, we propose to extend the middleware platform with new aspect-oriented modeling syntax, code generation tools, and a development process for building bypassing implementations. Bypassing implementations provide better use of server's resources, leading to better overall client experience. Our core contribution is this idea: *aspect-oriented extensions to IDL, additional code generation, along with an enhanced run-time, can enable application developers to conveniently bypass middleware layers when they are not needed, thus improving the server's performance and providing more "operational headroom"*.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Constructs and Features; C.2.4 [*Distributed Systems*]: Distributed Applications

Keywords Middleware, cross-layer, model-driven, aspect oriented bypassing

1. Introduction

Middleware is a mixed blessing for application developers. On the one hand, middleware platforms provide convenient abstractions like distributed objects (in CORBA) or service endpoints (in Web Services); developers enjoy the convenience of high-level operations on remote interfaces. The implementation of these abstractions is a complicated matter, involving marshaling, dispatching, session maintenance, object activation, adaptation, etc. These intricacies are the exclusive concern of the middleware platform developers, and quite opaque to application developers. This is an example of the "separation of concerns" principle, pioneered by David Parnas [23, 36], and prized by software designers. Unfortunately, there is an undesirable flip side to this strict separation: changes to the lower layers can only be designed and built by the platform developers.

Suppose we have to program a task which can be completed by the server knowing only *some* of the input parameters, with little or no access to the application-level state. Such tasks include input data validation, filtering of hostile requests, and load balancing/distribution based on the request content. Another category of examples are related to Quality-of-service issues (QoS): access control, denial-of-service resistance [11], fail-over, cached request processing, fault-tolerance [34] etc. When the server state (and/or some of the input arguments) are not needed to process the request, it would be desirable to bypass some of the functional layers of the middleware, and service the request *before* the layers do their work. As we demonstrate in this paper, bypassing these layers can provide non-trivial performance improvements. As we shall argue below, bypassing can lead to more efficient use of a server's resources, which can significantly improve the overall experience of the whole user community, specially under conditions of stress [50]. Bypassing can also improve the availability of the server, which can be a key business advantage for service providers.

The difficulty here is this: to bypass these normally opaque middleware layers, the application developer must forsake convenience and hand-code the gory low-level details of request dispatching, argument extraction, and reply generation, without the benefit of IDL-driven code generation, type safety, platform run-time support *etc*. Worse still, any error she made in writing this intricate, low-level code could inadvertently affect other applications using the same middleware run-time environment.

We want to enable application developers to move part of their application to the socket-handling layer, effectively bypassing unwanted middleware layers. We refer to this approach as "bypassing" implementations of applications. Our goal is to make it easier for programmers to build this kind of layer-crosscutting applications. The paper describes the following:

- A new, aspect-oriented, IDL-driven, layer crosscutting approach to developing middleware applications, whereby developers can place pieces of the application in the socket-handling layer. The socket layer operations are modeled using an extended IDL.
- A code-generation toolset, and runtime environment, for the Orbacus and JacORB platforms, that allows socket-handling layer elements of applications to be built without having to resort to low-level programming.
- An evaluation of this approach using both several illustrative examples, and some performance measurements.

AOSD 07 March 12–16, 2007, Vancouver Canada
Copyright © 2007 ACM 1-59593-615-7/07/03…$5.00

We chose Orbacus because it is extremely fast, if not the fastest Java-based ORB (see Demarey *et al* [12] for data comparing the performance of several Java ORBs) and is available in source form for research purposes from the vendor[1]. If one can speed Orbacus up with this approach (despite it already being highly efficient) one can reasonably expect that other middleware platforms would also benefit. JacORB is another publicly available ORB (although not as fast as Orbacus) and we have an implementation for JacORB as well. We reported on JacORB implementation and its benefits in [13] but chose Orbacus as the target platform for this study because of the forementioned reasons. The JacORB implementation can be downloaded from `http://macbeth.cs.ucdavis.edu/bypass`. The Orbacus implementation is subject to licensing agreement from the vendor, and is not distributable.

The rest of the paper is organized as follows. In section 2 we discuss the general research context of our work. Next, in section 3 we discuss the goals of our research in more detail. In Section 4, we describe the overall process and tools that implement our approach. In section 4.3, we present the run-time architecture. In section 5, we present our results: several examples, with performance results. In section 6, we discuss the most closely related work, and then the paper concludes.

2. Related Work

The inflexibility arising from layered architectures has been extensively discussed in the literature. For reasons of space, we present only a representative sample of this research.

The evolution of attitudes towards layering and "division of knowledge" arises even in layers *below* the middleware layer, in the context of computer networking. The early design philosophy of the Internet was rooted in the "End-to-end" principle [42], which held that the networking component of a system should merely provide the abstraction of *dumb, fast packet transport* and all the intelligence should be at the ends. This view is enshrined in the design of the TCP/IP protocol architecture which has four layers, each playing a specific, well-defined role. In particular, the application layer, at the top, remained agnostic about the details of the lower layers. This design philosophy has since been sorely stressed by the rise of phenomena such as SPAM, fast-spreading viruses, billing issues, wireless networks [8], all of which create conflicts between context-specific needs, and the imperative to keep the networking protocol layers fixed, separate and simple. These design challenges [5] led to approaches such as application framing, integrated layer processing and synchronous up-calls [3, 6, 7]; all of these, arguably, optimized protocol stacks by mixing up the old, rigid roles of the protocol layers. "Cross-layer adaptation", another approach, advocates explicit cooperation between networking layers to provide better service to applications; this has been used with multimedia [32] and with ad-hoc application overlays like Gnutella [10]. Likewise, operating systems (OS) are an abstraction layer above raw hardware. OS design philosophy has also undergone a similar evolution, and explored designs for cross-layer optimization. Vertical Migration (VM) [46, 45] aims to improve software performance by migrating application primitives into lower layers of the OS, thus, mitigating the overhead of inter-layer calls. In VM, an application function can be migrated to a lower layer if it is not using the abstractions provided by the higher layer. Pu *et al* [38] generate specialized, context-dependent versions of system calls using partial evaluation which run faster. Webservers [24] have been optimized by including cacheing and routing operations in the networking layer. Researchers have also explored both manual [2] and automated [40] approaches bundling several system calls, to do more work with fewer protection-boundary crossings.

[1] See `http://www.orbacus.com/support/new_site/index.jsp`, last visited Sept 26, 2006.

In addition to these layer-breaching design approaches, researchers have also explored allowing explicit programming of lower layers. Networking researchers investigated programmable network infrastructures. Active networks [48] and open signaling [4, 19] are examples. The SPINE system [16] allows network adapters to be programmed to do some processing on each packet, (*e.g.,* for video handling) and thus off-load the CPU. Wang's Shield [49] allows customized, very fast packet filters to be placed in firewalls to protect vulnerable servers. OS researchers have also provided extensibility and customizability, with safety being always an issue of concern. Systems such as SPIN [20] allow users to write extensions, and guarantee safety using type-safe languages. Exokernel [15] allows applications to write their own hardware handlers; the monitor offers some level of protection. The Utah FluxOS work [17, 41] lets users customize the OS to their needs. In general, extending or customizing lower layers of networking (or OS) infrastructure involves writing tricky, intricate, low-level code which modifies the behaviour of schedulers, memory managers, file systems, network routers *etc.*. This is non-trivial, and (regardless of the actual mechanism used) must be approached with great caution.

Our work applies to the CORBA middleware platform, which is a layer *above* operating systems and the network; it is complementary to the work cited above. Our focus is to enable developers of middleware-based applications to conveniently adopt a *bypassing design pattern* (or *bypassing style*) (with good tool- and run-time support) to speed-up applications, without having to write intricate, low-level, inherently non-portable code. There is work in the middleware area that is more closely relevant; we defer this discussion of this work until after our work is presented, to section 6.

3. Research Goals

Normally, a CORBA request arrives at a server's socket layer, and then goes through several steps: demarshalling, service activation, despatching *etc*. For some requests however, all we really need to do is to identify the type of the object, the method being invoked, and some of the arguments. This can be done efficiently in the socket-handling layer (in general, this is the platform API which the middleware uses to send and receive messages: *e.g.,* the layer in the CORBA platform that interacts with the socket API; hereafter abbreviated as $\mathcal{SHL}$) bypassing the subsequent steps. Unfortunately, at the $\mathcal{SHL}$, all one has is the raw input request message in a buffer; therefore, to decode this message, and extract the needed contents, one has to write complex, intricate, low-level code. The developer who chooses this approach to writing a bypassing application faces several hurdles.

Forgoing useful abstractions Low-level code, written at the $\mathcal{SHL}$, cannot use middleware abstractions. The benefits of model-driven development are lost; one cannot enjoy such useful conveniences as IDL-level design models, code generation support, type-checked programming (including client-server type-agreement).

Manual de-multiplexing The type of the target CORBA object, and the method being invoked, must be extracted and decoded from a low-level request packet. Low-level code, perhaps using platform-specific low-level APIs, must be written to extract this information. Errors, easily made in such low-level code, can catastrophically affect any application running on the platform (not just the bypassing application).

Coding demarshaling/marshaling The wire format for the CORBA request message is efficient, but inconvenient. To extract request parameters with maximum efficiency, one must resort to cumbersome, intricate, fault-prone, low-level coding, as shown below. The

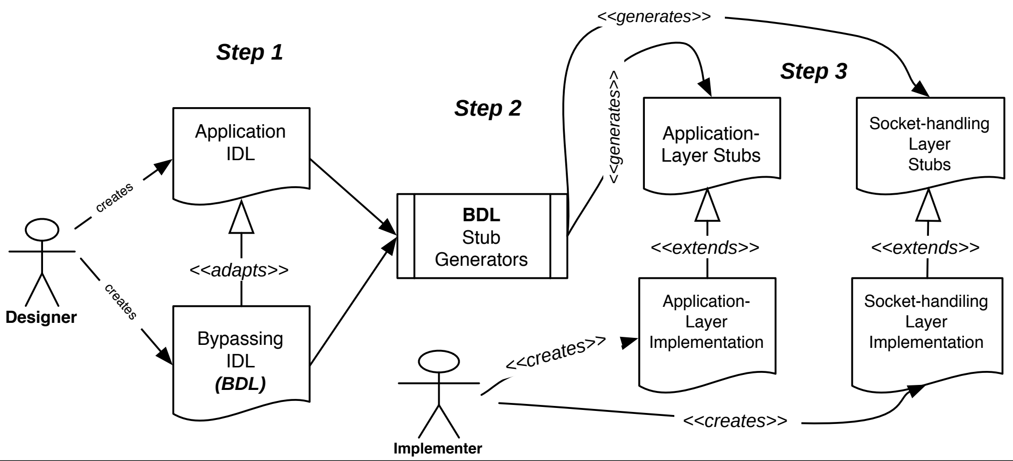

Figure 1. *Development Process: The normal model-driven development is extended, as designer also creates (Step 1 above) both normal interface and a model of the socket-layer bypass. Code generation tools now generate (Step 2) both application-layer and socket-layers stubs, in addition to data (not shown) for dispatch tables; developer (Step 3) implements application- and socket-layer stub separately.*

data must be read/written directly from/to the byte stream, at the lower layer. The programmer must carefully take note of the exact position lines (2) and type of the input and/or output data, and then read it (line 3) or write it (line 14). She also must ensure that the stream is repositioned (line 11) in case the bypass is not terminated in the $\mathcal{SHL}$, and the byte stream is read again in the upper layer. These intricacies can naturally lead to programmer error, in critical lower-layer code. This can have unintended consequences, even on applications that are not subject to bypassing.

```
0    ...
1    /* First extract data  */
2    stream._OB_mark();
3    String name = stream.read_string();
4    /* Create Return place holder */
5    BooleanHolder result = new BooleanHolder();
6    /* call application-specific code */
7    if (!password_lookup.lookup(name,result))
8    {
9          stream.needs_reply_advice=true;
10         /* Reposition for later  */
11         stream._OB_rewind();
12         return false;
13   }
14   write_reply(buffer,result);
15   ...
```

Also, if a request is completely processed at the $\mathcal{SHL}$, the response data must also similarly (and laboriously) be marshaled into the wire format.

Platform-specific activation The socket-level component of bypassing applications must be activated when these applications bind to the middleware. This code, which “hooks” the bypassing logic to the socket layer of the middleware, depends on the specific run-time implementation (the ORB), and must be written for each implementation.

Generality of our approach Our context is distributed systems that build applications using middleware-supported abstractions such as RMI (Java), distributed objects (CORBA) and service endpoints (SOAP). In all these cases, middleware is built on a communication layer such as sockets. Our approach applies to these middleware systems. However, porting our approach to a new type of middleware requires new stub generators. On the other hand, porting to a new implementation (e.g., another ORB) is much easier. When we ported from Orbacus to JacORB, we were able to reuse the code for bypass interface compiler and code generator which is a significant component.

3.1 Our Goals:

Our approach to bypassing implementation has the following specific goals:

Model-based and Type-Checked: Socket-handling-layer ($\mathcal{SHL}$) implementations will be model-based, using IDL-level design models for clarity, traceability, and code-generation tool support. Socket-handling layer implementations will have type-checked access to application level arguments, exceptions etc.

Abstract: $\mathcal{SHL}$ implementations can make full use of application-layer abstractions; (de-)multiplexing, marshaling will be handled transparently. Raw message data will simply not be in scope, thus reducing the risk of inadvertent programming errors affecting other applications.

Portable: Application developers should not have to rewrite their $\mathcal{SHL}$ bypassing code for a different implementation of the same middleware platform (if the platform supports bypassing in the standard manner we describe).

4. Overview of Approach

We describe how applications that exploit bypassing are written, and how they are executed at run-time.

4.1 Developing Bypassing Applications

The development process of bypassing server application begins (as with traditional middleware) with the designer creating a set of application interfaces, written in an interface definition language (in our case it is CORBA IDL, but it also could be WSDL for Web Services, *etc.*) (Figure 1). This is shown in the figure as application IDL. In addition, the design also models (using an extended IDL) advice that are to be implemented in the socket-handling layer (shown as BDL at step 1 of Figure 1). BDL models bypassing semantics of particular joinpoints, such as a specific operation specified in the IDL. The IDL and BDL are processed by code-generation tools (step 2 in Figure 1) to generate both application layer and $\mathcal{SHL}$ stubs. The application-layer (hereafter $\mathcal{AL}$) stubs

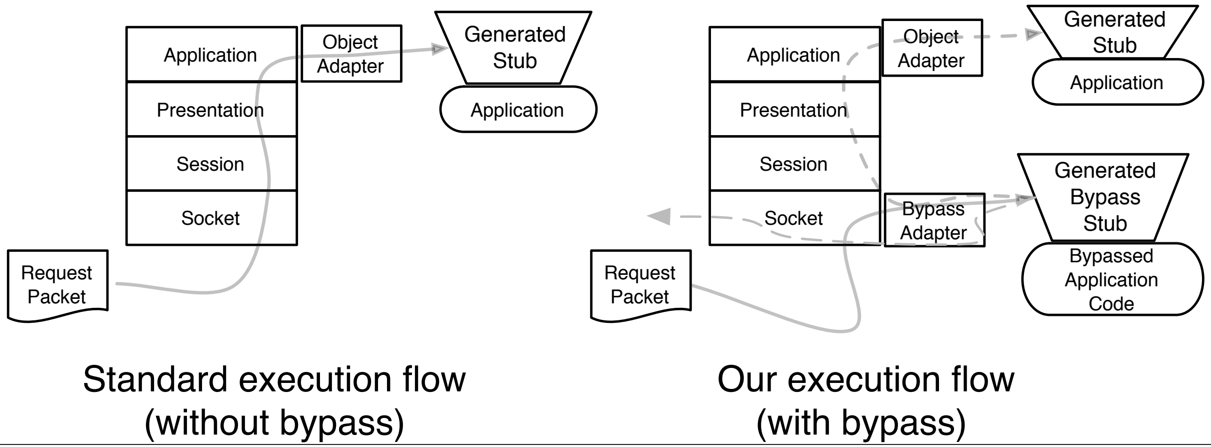

Figure 2. *With standard middleware, an arriving request packet triggers a series of computations through the layers (standard execution flow, grey arrow), terminating in the application code, via the stub. We check if a bypass is present for a specific request (our execution flow, bypass adapter box). If so the bypassed code is executed; then, execution may return directly or proceed to the upper layer (our execution flow, dashed arrows). If no bypass exists, execution proceeds as usual.*

play the same role as traditional middleware stubs. The $\mathcal{SHL}$ stubs enable application developers to program the BDL advice without having to resort to complex, intricate, low-level code. These stubs handle despatching and de-marshaling for the $\mathcal{SHL}$ bypass implementation, and provide a typing environment (*i.e.*, a collection of declared parameters that support checking for type consistency with the invoking clients). The stubs also handle the marshaling of responses (both return values and exceptions; `out` arguments will be done shortly). Using both the standard application stubs (step 3 in Figure 1) and the new $\mathcal{SHL}$ stub, the developer implements the application logic and $\mathcal{SHL}$ advice. With the $\mathcal{SHL}$ stub, and the extended run-time which dispatches these stubs, the developer has her "layered cake" (some of the convenience of the upper-layer abstraction) and can eat it too (bypass those layers of the middleware that are not needed).

We now discuss the modeling, tools, and run-time enhancements, that support the bypassing style for middleware apps.

4.2 BDL: Bypass Modeling

We illustrate interface modeling in our approach with an example of a simple stock quoting application, which accepts a string (indicating a security) and returns a quote. The IDL interface is:

```
interface StockServer {
     float get(in string sym);
};
```

A server implementing this interface can use a *bypassing cache*, whereby previously computed quotes are cached for a fixed period of time in a $\mathcal{SHL}$ cache. A request for a quote on, *e.g.*, "IONA" is intercepted at the $\mathcal{SHL}$, before further processing; if a fresh entry for it, in a $\mathcal{SHL}$ cache datastructure is found, it is returned[2]. Otherwise, execution proceeds to the upper layer, where it is executed normally. When the value is returned, the return path is intercepted in the $\mathcal{SHL}$, and the cache is updated to reflect the new value. The implementation of the bypassing cache begins as shown in Figure 1 with an interface description of the bypassing code. This description is written in a *bypass definition language* (BDL), as shown below.

```
bypass perinstance automatic interface Cache{
    float returnCached(in string sym);
    void saveInCache(in string sym, in float res);
    pointcut servicecall(in string sym):
             exec(float StockServer.get(sym));
    before servicecall(sym): returnCached(sym);
    after servicecall(sym) returning(res):
             saveInCache(sym,res);
};
```

This BDL interface models the advice implemented in the $\mathcal{SHL}$ cache. The focus of BDL modeling is on exactly which $\mathcal{AL}$ operations are bypassed, and how they are despatched in the $\mathcal{SHL}$. The syntax is similar to IDL, with a few differences. The "`perinstance automatic`" refers to bypass activation, and is discussed later. The first line models the ($\mathcal{SHL}$) method `returnCached` that looks up a symbol in a cache and returns the value, before the invocation is sent up through to the $\mathcal{AL}$ code. Next is the method `saveInCache` which is invoked in the $\mathcal{SHL}$ code to update the cache with a fresh new value computed in the $\mathcal{AL}$. The next line defines a *pointcut* that captures the execution of the `StockServer.get` method, whose results are being cached. The pointcut/advice concept from AspectJ [26] is used here to model the interface of bypassing code. In our case the pointcut `servicecall` is defined above to bind to the `get` method, and to also bind the parameter `sym`.

The `before` binding causes the method `returnCached` to execute before the `servicecall` pointcut. The `sym` variable is used for the lookup in the cache, before the call is passed up to the $\mathcal{AL}$. Note that the `before` binding is different than the corresponding binding in AspectJ: to pass the call to the $\mathcal{AL}$ the programmer calls `proceed` in the advice body, otherwise the call terminates at the $\mathcal{SHL}$. The `after` binding is used to update the cache value on the return path, using the value computed in the $\mathcal{AL}$. We refer to the code that is executed before the upcall as the *before-bypass advice* and the code that is executed after as the *after-bypass advice*. The code the programmer writes to implement the before-bypass advice is illustrated below (in simplified form, details elided for brevity):

```
public class MyCache extends CacheBP {
  protected float returnCached(String sym) {
       if (table.contains(sym))
           {
```

[2] Cacheing could also be done on the client side, for example using DADO [52]; however the bypassing approach allows the server to more efficiently support a larger number of client requests, even from clients that don't cache.

```
            return table.get(sym);
         }
         else {
            /* Continue to next advice or AL */
            proceed() ;
         }
   }
   ... definition of saveInCache ...
}
```

Bypass advice implementation is straightforward, and can exploit both the procedure call abstraction and typed, symbolic argument access. When bypasses don't terminate the execution (here, when there is cache miss) the bypass signals to the runtime environment to proceed with the call, by callling `proceed()`; our tools translate the proceed, at load-time, into bytecodes that continue the execution by returning a special value. This value signals the bypass stub to continue through to the $\mathcal{AL}$.

It should be noted that the cache lookup adds an overhead to every request, whether or not the $\mathcal{SHL}$ cache has a usable value. Thus, the cache would perform better than the base non-passing implementation only with higher hit rates. We present specific performance figures later in Section 5. We now describe the details of the operation of the bypassing code, specifically by comparing with non-bypassing implementations.

4.3 Bypass Execution

On the left of Figure 2 we show execution flow with standard middleware (which remains largely unchanged in our approach for applications that don't bypass). A request packet triggers a series of activations through the layers of the middleware: the $\mathcal{SHL}$ assembles a datastructure containing relevant information about the request, and dispatches it up. The request goes through the session, presentation and application layers. Finally, the object adapter performs such functions as demultiplexing, activation, and thread management; then, the stubs decode the arguments of the request, and invoke the actual advice. All this logic is executed for every request. Execution flow for a bypassing application is shown on the right on Figure 2. The request packet is processed in the $\mathcal{SHL}$, which we extend. First, we check if there is a bypass for the type of object and the operation being invoked. If so, the request is forwarded through the $\mathcal{SHL}$ bypass adapter, to the $\mathcal{SHL}$ stub, which executes the actual bypassed advice.

Implementation Details To execute the bypassed advice, we have added some lightweight machinery to the $\mathcal{SHL}$. This machinery handles *bypass recognition*, *demultiplexing*, and *marshaling*. Finally, the *activation and deactivation* of bypassing raises some special coordination concerns, which we discuss below when we describe bypass activation.

Bypass Recognition: Request packets arriving at the $\mathcal{SHL}$ bear an object identifier. We use a hashmap (the *object ID hash table*, or OIHT) at the $\mathcal{SHL}$ to track if an object-id is associated with a bypass; if not the request is passed through. If a bypass exists, the lookup recovers two things: the object's IDL type from the object identifiers, and an operation demultiplex table (ODT), which is uses perfect hashing to quickly lookup the operation and despatch the execution. This information is then passed on to the $\mathcal{SHL}$ adapter for the next step. We note that the check for the existence of a bypass is a constant overhead, (because of the hash lookup) regardless of how many object implementations are bypassed.

Bypass Demultiplexing: The $\mathcal{SHL}$ adapter then recovers the operation name, and checks if this operation is associated to a bypass by a pointcut. The precise stubs to be executed in the next step are determined by the pointcuts given in the bypass interface definition. The stubs are found in the operation demultiplex table (ODT) which uses a fast perfect hashing lookup [39, 43] to map the operation name to the bypass stubs that should be despatched for this request. Perfect hashing is pre-determined by matching pointcuts in the $\mathcal{SHL}$ BDLs to $\mathcal{AL}$ IDLs. If many bypasses are associated with one particular operation, the corresponding stubs linked up in a DECORATOR pattern [18]. The ordering of the stubs in the chain is determined by the order in which bypass interfaces are given in the input to the toolset. In the future, BDL will allow the explicit specification of pointcut ordering for each operation.

De-marshaling & Dispatching: The selected stub will then extract the needed data (as specified in the advice and the pointcut) from the request packet, and make the upcall to the $\mathcal{SHL}$ bypass implementation. This stub is generated, and is built for speed rather than readability: efficient low-level code extracts just those arguments that are needed from the request packet[3].

Bypassing Application Activation: Activating and de-activating bypassing applications requires some additional book-keeping. Bypasses come in several flavors. They can be *statically scoped* (applicable to all instances) or *per-instance*; they can be *automatic* (activated when the application-layer servant is activated) or *manual* (activated under application control). Currently, we support only the following types of bypasses: `static automatic`, `perinstance automatic`, and `perinstance manual`.

Suppose that an instance of the IDL type `StockServer` (discussed above) activates, and announces to the middleware run-time that it is ready to accept requests. If this application requires automatic bypass activation we must activate the `Cache` bypass implementation in the socket layer. The constructor for the bypass implementation class is called; this allows the bypass developer to perform required initialization.

As mentioned above, our run-time keeps data about active bypasses in the OIHT, which maps object-ids to types and then (via the ODT) to applicable bypass advice. This table is updated when a `StockServer` instance binds to the ORB at the application layer.

It should be noted that there is a corresponding set of dispatch tables in the application layer, to resolve object-ids within incoming requests to actual stubs. These tables must be consistent with the socket-layer tables. CORBA runtimes operate in a multi-threaded environment, so we must ensure that an arriving request will never find $\mathcal{AL}$ and $\mathcal{SHL}$ dispatch tables in an inconsistent state. Updates to *both* socket layer and upper-layer dispatch tables only occur when an instance of an object that includes a bypassing implementation is activated; in this case, we synchronize the two updates into one atomic operation. We use a readers/writers locking protocol to allow concurrent reads, but exclusive writes, to the dispatch tables.

Manual Activation: As we discuss below, bypassing is most advantageous under certain run-time conditions. For this reason, we also allow the programmer (within application logic) to activate and deactivate the bypassing socket-layer code under different operating conditions. Currently this is allowed only for bypasses activated on a per-instance basis; we hope to allow manual activation for static bypasses, as well, later. There are two different types of manual activation interfaces.

1. *Asynchronous Bypass Activation:* When this type of activation is requested, the socket layer bypass is activated immediately, and will be in effect for the next arriving request packet; it will not affect any request packets already processed and forwarded. This activation is risky but simple. it can be used when the bypass layer implements just a non-functional feature that does not alter the application semantics. Examples, such as cacheing, and attack filtering, are given in Section 5.

[3] "`Out`" arguments are not handled yet, but will be implemented in the normal way, using handles.

2. *Co-ordinated Bypass Activation:* This approach is used when the bypass layer activation must be co-ordinated with a corresponding change in the application layer logic. This is a two phase process, involving an API call to activate the bypass code, and a call back from the middleware to signal the application to change its logic to reflect the active bypass. When the developer requests activation, the activation API call returns immediately. Next the run-time environment activates the bypass code, by making an entry in the object-id hash table (OIHT); synchronously, it makes an entry in the table indicating that the first subsequent $\mathcal{AL}$ request to this object-id will be preceded by a call to the activation callback in the $\mathcal{AL}$[4]. This design guarantees that the application code gets the call back before any actual client operation invocations that have been processed by the bypass layer; it also reduces need for synchronization between upper and lower layers. Deactivation is handled similarly[5].

5. Bypassing Case Studies

In this section, we present some case studies of applications that use bypassing, and discuss the implementation effort required, and some early performance results.

The performance studies were done using Orbacus version, running on Java 1.5. For hardware, we used an asymmetric client/server arrangement, with a beefy client (Dual-processor, Intel Xeon 2GHz with 1GB RAM) and a 1 Ghz Pentium Laptop with 512 MB RAM as server; connected over a 100BaseT Ethernet. With this set-up, we were able to drive the laptop-based server into saturation, in virtually every setting we tried; some bypassing implementations were so fast that this was the only way we could do it! The resulting maximum throughput measurement is used as our basis for comparing performance. In every trial, the configuration was allowed to run until the performance was stable before the measurements were taken; the average saturation rate (invocations processed per unit time) was measured over a period of 5 minutes, and recorded when stable. All the examples discussed in this section work for both Orbacus and JacORB, and show similar performance improvements in each.

Microbenchmarks: To evaluate the raw overhead of our run-time enhancements, we measured the execution time of an empty application (no arguments, no return values, just a method call) with and without an empty bypass. The empty base application, without any of our bypass run-time, processes 9810 requests per second. The empty application, with our bypass code, can process 9620 req/s, about 1.9% slowdown. An empty bypass, that finishes in the socket layer and returns without passing to the application layer can process 13,850 req/s. The processing *rate* is increased by about a factor of 41%.

It is important to note that this example somewhat pessimistically ignores the effect of parameter demarshaling. Application operations usually have one or more arguments, all of which are demarshaled within the middleware. In most cases, bypassing code will benefit from selectively demarshaling *some* of the arguments for processing, whereas the application level code always demarshals every argument.

[4] The full details are omitted here for brevity: but essentially requests arriving after bypass activation have a special field in the request context object. This signals the POA to maintain a separate queue for the requests that arrive after bypass activation; the POA waits until all pre-bypass requests are flushed; then it triggers the activation, to signal the application layer code to prepare itself, and begins work on the post-bypass queue.

[5] In case the programmer wishes to activate multiple bypasses she must do it serially; she can activate a successive bypass only after the activation call back is received from the previous activation request.

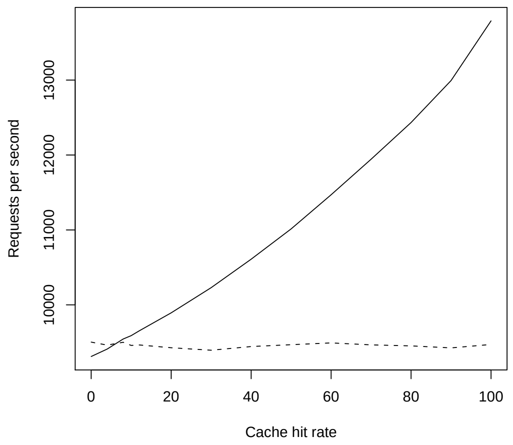

Figure 3. *Relative performance of an application layer (dotted line) and bypassing (solid line) implementation of cacheing. Note that with bypassing cache, the request is passed through to application layer in case of a cache miss.*

The raw overhead of the middleware layers causes bypassing invocations to enjoy a 40% or so advantage. However, the resulting checks in the lower layer increase the load by about 1.9% for requests that do not use the bypass. This suggests that while bypassing is generally quite a bit faster, it is sometimes very slightly slower; we discuss the implications of this further, below.

Bypassing Cache: To evaluate the bypassing cache, we implemented the cache in the $\mathcal{SHL}$ (using the BDL as shown in Section 4.2, and using the generated stubs). We used a hashtable based implementation for the bypassing cache. In the bypassing implementation, if the lookup in the $\mathcal{SHL}$ failed to produce a current value, the execution proceeds to the $\mathcal{AL}$ where a similar hashtable lookup is done, which is guaranteed to produce a value. This value is then filled in the bypassing cache on the return path. This assumption might seem rather pessimistic (since actual application-layer lookups are likely to be slower database accesses). However, it provides a reasonable basis to measure the *actual benefit* of bypassing the application layer, by comparing apples with apples: a hash-cache in the socket layer with a hash-cache in the application layer.

In saturation, the base application (sans bypassing) processes in about 9500 req/s. By comparison, the bypass cache implementation processes 13800 req/s with 100% hit rate, and 9310 req/s with 100% miss rate. Note that the case of a cache miss in the bypassing code is somewhat slower than the implementation without a bypass, and the case of a cache hit is almost 50% faster. When the hit rate exceeds 5-6%, the bypassing cache implementation becomes faster than the non-bypassing implementation.

If the hit rate in the cache cannot be predetermined, one strategy might be to keep a cache in the application layer (which would provide some improvement in performance, by avoiding a database lookup), and activate the bypassing cache when the hit rates becomes high. In the real world stock market, conditions such as “flash crowds” do arise, and some stocks can become very interesting to a great many people. A dynamically activated bypass layer will fill its cache very quickly, and boost server performance and availability under this kind of stress.

Credit Card Validation: Consider a credit-card payment service. It uses credit card number validation, which is a simple checksum operation[6] which can be done entirely in the socket layer. We consider a service of the form shown below, which includes two relevant operations (others elided for clarity): `Validate`, to validate a credit card, and `Charge`, to make a charge to the card; `Validate` returns `true` or `false` indicating the validity of the card.

```
interface CCard {
          exception ChargeFailed {Details Elided} ;
          boolean Validate(in string card,
                           in string Expiry,
                           in string secret-code);
          void    Charge( ...)
                  raises(ChargeFailed);
}
```

Both operations shown above begin with a check of the credit card number, using the checksum rules. If the check fails, operation returns `false`. We want to move this operation to the socket layer, since it's a simple idempotent operation. The BDL model is shown below:

```
bypass static automatic interface CCvalidate {
       boolean check(in string Card,
                     in string Expiry);

       pointcut validate(in string Card, in string Expiry):
         (exec(CCard.Validate(Card,Date,...))
          || exec(CCard.Charge(Card,Date,...)))

       before validate(C, D): check(C,D);
}
```

Note the method `check`, which models the validity check. If the check fails, `check` returns `false`. The pointcut `validate` declares that this check is to be done before the operations `Validate` and `Charge` modeled in the `CCard` interface. The pointcut also binds the two needed variables from the scope of the operation invocations. Finally, we bind the `validate` pointcut to the `check` advice, passing on the extracted operation arguments.

In this implementation, a substantial amount of work is required in the $\mathcal{SHL}$: the run-time environment has to extract two string arguments, and construct a reply message if the credit-card is invalid (if it's valid, the execution is passed up). Again for fairness, we ignore other application functions (such as actually updating business objects) since that part remains unchanged; we seek to factor out the actual benefit of bypassing. In the base case, without bypassing the base application can process about 9330 client requests per second for invalid cards, and 9310 req/s for valid cards. With the bypassing in place, the figures are 13,240 req/s for invalid cards, and 9130 for good cards. Since the bypassing implementation first checks before passing it on to the application layer, we get a slowdown for the valid cards with bypassing.

Based on these numbers, we estimate bypassing will be advantageous if about 5.8% of the requests are with invalid cards. While this may not be true under normal circumstances, this type of input validation can be extremely effective in warding off concerted application layer attacks, to guess valid inputs, such as credit card numbers or dictionary attacks on passwords. In this case, as mentioned earlier, an application developer can also dynamically engage the socket layer adaptation, however, the co-ordinated activation (See Section 4.3, page 6) should be used in this case, since the bypassing directly affects the application semantics.

[6] Although validation can be also performed using client-side scripting, the services might prefer to check numbers submitted from untrusted clients before proceeding.

Defending the Servant: An adversary mounts a *denial-of-service attack*[7] (DOS) on a system by attempting to overwhelm it with input, most often spurious or malformed input. A *distributed* denial-of-service attack (DDOS) occurs when a large group of machines, widely distributed across the internet are orchestrated to mount a DOS attack on a target. A fast-spreading virus can have the same effect as a DDOS. The virus compromises machines as it spreads, and these machines attack others; soon, any given machine might be under attack by several other infected machines.

These attacks can occur at any level of the communications protocol—from low-level attacks based on flooding with SYN packets, to upper-layer attacks using malformed argument values (*e.g.,* SQL injection attacks [21, 47]). There are different viruses that have exploited vulnerabilities at different (both lower and upper) layers of the protocol, spread rapidly on the internet and overwhelmed a large number of servers. These attacks essentially threaten to consume the target's resources. The more of a target's resources an adversary can consume, the more damaging the attack. The target, on the other hand, should spend as little resources as possible on requests that are part of an attack, and try to recognize (and reject) them as quickly as possible. Otherwise, the services provided to legitimate clients may degrade, which would be unacceptable in mission-critical settings. For example, clients attempting to make purchases at a website that is under attack may find it sluggish and get discouraged. Therefore, application developers and security engineers seek to identify and reject attacks as quickly as possible. Network-layer (e.g., firewall) adaptations can be used to defeat DDOS attacks; however, application-layer attacks can depend on the inputs in non-trival and changing ways. These can be difficult to detect based purely on network-level data packets. Firewall based filtering of such attacks would be difficult, risky, and hard to evolve.

DDOS would certainly apply to middleware-based applications. There is therefore a strong case to implement defenses against possible attacks using bypassing; if an attacking request can be detected and rejected in the socket layer itself, the resources consumed by the adversarial request would be reduced. For example, consider an SQL injection attack on a request that takes two strings, a login and a password. Typically the application forms a "SELECT . . . WHERE . . ." query which searches for a user and password in a database, and the password WHERE condition is the last one in the generated query. The adversary may inject spurious SQL syntax, (such as a tautology[8]) into the password field, and gain access to the system. Alternatively, an adversary may inject expressions that force complicated computations like joins, to consume the victim's resources. We consider a case where repeated SQL injection attempts (either by one adversary or by a distributed group of compromised hosts) are being made.

It would be desirable to quickly reject malformed input strings by filtering the strings [47] in a bypassing implementation. If an input password string is invalid, there is no need to proceed with the login, and the request can be rejected. To measure the effectiveness of bypassing in this case, we have built two versions of simple input filtering, with and without bypassing. Then, we subjected each version to increasing levels of attack. Now, *while under attack*, we measured the response time of servers to *legitimate client requests*. For this set-up, we used an "adversarial" machine which attacks at a fixed rate, while an "innocent" machine submits successive requests as fast as possible. The resulting processing rate for the

[7] See `http://www.cert.org/tech_tips/denial_of_service.html` for general information and references on denial-of-service attacks

[8] For example, using the comment character "`--`" or quotations in an input field, attackers can alter the semantics of a query constructed from input data.

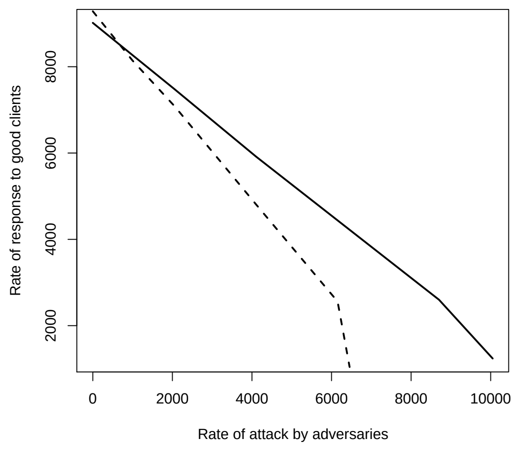

Figure 4. *Relative performance of an application layer (dashed line) and bypassing (solid line) implementation of a denial-of-service defense mechanism. Both attack rate and response rate are measured per second.*

"innocent" requests are measured. The results are shown in figure 4. The x-axis is the attack rate from the adversary; the y-axis is the processing rate of legitimate users' requests.

The steeper plot on the left is the performance curve for the implementation without bypassing, and the other curve uses bypassing. As can be seen, the $\mathcal{AL}$ implementation quickly degrades, providing lower processing rate for good clients as the attack rate increases, dropping to about 1000 req/s at about 6500 attacks/sec. In fact once the attack rate exceeds 6000/sec, the conventional implementation declines rapidly in performance, and the response time for legitimate clients declines. The bypassing implementation, on the other hand, can handle about a 55% higher load, while still providing a client request processing rate of 1240 req/s.

We note that for low attack rates (under 600/sec) the bypassing implementation is actually slower (for legitimate requests). In this case, as discussed earlier for the bypassing cache, it would be desirable to dynamically engage the bypassing implementation when the attack rate reaches a point that is empirically known to be the threshold where bypassing becomes advantageous.

Observations: Bypassing gains performance advantage by not executing several layers of the middleware, and also from not demarshaling all the arguments. This results in a *base overhead* reduction of about a half for bypassing code. Greater reductions can be obtained if bypassing code needs only to demarshal some arguments. The run-time machinery involved in bypassing involves a small additional overhead for applications that don't use bypassing, measured at about 1.9% in our experiments. Bypassing can thus provide developers with a way to make applications more efficient, in cases where a significant fraction of the requests can be processed within the bypass code. We have presented some several illustrative cases above. Finally, it should be noted that network delays will generally make these differences not noticeable directly to individual clients; however bypassing can lead to *more efficient use of the server's resources* [50], and thus more "head room" under heavy load, leading to a better experience for the entire client population. This is clearly evident from the last example, where the bypassing defense of the server allows legitimate users to get much better response.

The following table presents the size of the actual implementation in non-comment source lines for the above 3 examples (broken down into IDL, BDL, and size of the Java bypass implementation). The fourth column shows the size of the bypass stub code generated by from the BDL. In all cases, it can be seen that the actual manually written bypass implementation code is fairly compact. The large size of the generated code includes inlined demarshaling code, and the generated perfect hashing code. In one case (Defense) we actually had a previously hand-implemented version done not using our approach; this was done as a proof-of-principle of the bypassing approach. In this case, defense was done with 267 lines of hand-coded Java (not including perfect hashing code); this favorably compares with 11 lines written with the tool-generated stub, showing significant savings of programmer time. We also note that in the case of Defense, there is no measurable difference in performance between the manual and the (much shorter) version exploiting the our approach. In practice, since our tools generate highly optimized code, and exploit a carefully designed run-time, we can reasonably expect that bypassing with generated code will be faster than a completely hand-written bypass.

Application	IDL size	BDL size	Hand-written	Generated	Manual
Cache	7	10	17	581	N/A
Credit Card	8	8	22	540	N/A
Defense	7	8	11	523	267

6. Closely Related Work

In this section, we discuss work that allows programmers to customize the behavior of the middleware platforms to suit application needs.

There are specially constructed ORBs [28, 29, 27] that allow modification of the internal structure, to adapt to changes in the operating environment. DynamicTAO [29] is a reflective CORBA ORB designed on top of TAO [44]. It allows introspection of the internal components and component interactions, and allows changes to the components on-the-fly. In addition, to allow the users to incorporate QoS features (*e.g.* scheduling, reliability) adaptive middleware provides reflective programming models such as low-level system and library interceptors [33]. Adaptive middleware has also been used for adjusting quality of service for multimedia applications in response to bandwidth changes and energy constraints [37, 1]. The OpenCOM [9] middleware provides an efficient, dynamic and reflective, architecture for middleware adaptation. We believe bypassing implementations could be built using this architecture but would not benefit from an IDL based deployment. Our work is not aimed at actually modifying the behavior of the ORB, but rather allows application developers to bypass some functions of the ORB when they are not needed.

The DADO system [51] adopts aspect separation for programming crosscutting concerns into distributed heterogeneous systems. Adaptation services, which coordinate crosshost adaptations to standard components, are encapsulated into late-bound client and server-side components called adaptlets. DADO supports adaptlet communications in a type-safe environment. The AspectIX [22] middleware provides application customization using the notion of fragmented objects. This provides a seamless framework for non-traditional object semantics such as per-client state or object replication. Similar to our approach the Quality Objects [14] project provides the abstractions of aspects and advice. However, since weaving is done at the stub layer, QuO implementations don't allow bypassing a large portion of the middleware stack. Our approach is compatible with these middleware adaptations and meta-programming facilities. For instance, DADO adaptations can be implemented to execute in the lower levels of the middleware, thus improving the performance.

Partial evaluation is program transformation via specialization [25]. The goal here is to precompute as much possible of the program given known inputs. Partial evaluation has been used to optimize the SunRPC [31]. By partially evaluating the SunRPC stack, given a distributed application, nearly three times speed-up is achieved. The same techniques are applied successfully to the BSD packet filter interpreter and Linux signal delivery mechanisms [30]. A general-purpose, automated partial evaluator, however, cannot determine which parts of an application can bypass the middleware. We let the developer determine this, and simply make it much easier for her to write the code that bypasses the unwanted middleware layers

Also related is the pluggable protocols work of O'Ryan *et al* [35] where different protocols can be substituted within the middleware runtime to improve performance. Zhang, Dapeng and Jacobsen [53] advocate "just-in-time" middleware. By analyzing applications IDLs, they eliminate portions of the middleware platform corresponding to unused features. This can reduce the footprint of the middleware. Our goal is complementary to both of the above: to move actual application layer functionality into the lower layers of the middleware to provide improved performance. We expect that, in the future, middleware will be much more finely tuned to specific application needs; tuning will be accomplished by using techniques such as ours and the ideas in the 2 papers cited above.

7. Conclusion

The layered architecture used in middleware effectively hides lower-level details and provides convenient abstractions; however at times application developers may not need all the functions provided by these layers. In such cases, the middleware layers are an unnecessary overhead. However, writing code that bypasses the layers involves writing cumbersome, low-level, platform-specific code. We present an aspect-oriented, IDL-driven, code-generation-based approach whereby developers can write applications which cross-cut the middleware layers, using bypass advice, without having to give up the convenience of middleware abstractions, and type checking. Furthermore, since the bypassing code is not platform-dependent, it can run on any platform that supports bypassing using our approach. We present our toolset, development process, runtime environment; we also present several case studies of bypassing applications using our approach. Our case studies show that bypassing can provide more efficient usage of a server's resources, thus offering better experience for all users, specially under conditions of high load, thus offering more usable "head-room" for servers. Our approach will continue to offer an advantage to server-side applications as hardware performance and network performance improves, providing them the ability to handle increasing demands from both legitimate users and more powerful attackers.

Limitations and future work: Sometimes higher-layer elements such as POAs and interceptors are used to add quality-of-service features, bypassing which may be undesirable. Currently, we don't address this issue, and must rely on the developer's and the deployer's awareness of QoS features that must not be bypassed. We will investigate constraint-based reasoning approaches to this problem. Currently, if many bypasses apply to the same application method, they are ordered in the order of appearance in the BDL. We are implementing support to allow explicit ordering. Our approach has been implemented for Orbacus and JacORB; we are working on getting precise data for JacORB; however, preliminary indications suggest similar benefits for both ORBs.

In addition, the model-driven approach also applies to service-oriented middleware, based on WSDL. We have done some preliminary experiments in this setting, with XSUL and Apache Axis, and have observed promising performance improvements. Our work only considers server-side bypassing. Client-side bypassing might also help, (e.g., to handle exceptions) if the client is under heavy load. In future work, we will consider this. We also want to repeat our work for the web services platform, which is based on the HTTP protocol. Given current implementations of the web services platform, we expect that our approach will provide even greater benefits, for example by executing bypassed application code before the web server processes HTTP requests.

Acknowledgments

We gratefully acknowledge support from NSF CISE Grant No. 0232348.

A. Syntax of BDL

The syntax for BDL is shown in figure 5. The top level non-terminal is *socket_adaptation*. After the `bypass` keyword, the bypass designer can specify whether the bypass is scoped for all instances, or for a particular instance (`static`); she can also specify if the bypass is applied at initialization time (`automatic`) or is activated manually at runtime. The adaptation body is a list adaptation declarations. Each declaration can be an adaptation pointcut, an adaptation operation declaration, or an advice declaration. The operation declaration defines the interface (parameter and return types) of an advice operation. The advice binding declaration specifies the binding of an operation to a pointcut. The binding declaration could specify a `before` or `after` binding to a joinpoint labeled by an identifier. The actual joinpoint identifier is defined by specifying an identifier, and a pointcut along with any variables that are bound at the joinpoint, just as in AspectJ. Variables that are not bound are specified by an escape "_" In this case, the only allowable joinpoints are individual operations that are specified in an IDL interface; these are the only operations that can subject to bypassing. Hence we don't support other pointcut constructors such as conjunctions, `within` or or `cflow`.

References

[1] S. V. Adve, A. F. Harris, C. J. Hughes, D. L. Jones, R. H. Kravets, K. Nahrstedt, D. G. Sachs, R. Sasanka, J. Srinivasan, and W. Yuan. The illinois grace project: Global resource adaptation through cooperation. In *Workshop on Self-Healing, Adaptive, and self-MANaged Systems (SHAMAN) (held in conjunction with the 16th Annual ACM International Conference on Supercomputing)*, 2002.

[2] F. J. Ballesteros, R. Jiménez-Peris, M. Patiño-Martínez, F. Kon, S. Arévalo, and R. H. Campbell. Using interpreted compositecalls to improve operating system services. *Softw., Pract. Exper.*, 30(6):589–615, 2000.

[3] T. Braun and C. Diot. Protocol implementation using integrated layer processing. *SIGCOMM Comput. Commun. Rev.*, 25(4):151–161, 1995.

[4] A. T. Campbell, H. G. D. Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela. A survey of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 29(2):7–23, 1999.

[5] D. Clark. Modularity and efficiency in protocol implementation, 1982. `http://www.faqs.org/rfcs/rfc817.html`.

[6] D. D. Clark. The structuring of systems using upcalls. In *SOSP '85: Proceedings of the tenth ACM symposium on Operating systems principles*, pages 171–180, New York, NY, USA, 1985. ACM Press.

[7] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM '90: Proceedings of the ACM symposium on Communications architectures & protocols*, pages 200–208, New York, NY, USA, 1990. ACM Press.

[8] D. D. Clark, J. Wroclawski, K. Sollins, and R. Braden. Tussle in cyberspace: Defining tomorrow's internet. In *SIGCOMM*, 2003.

```
socket_adaptation:
                    bypass (perinstance|static) (automatic|manual)
                          interface identifier {   adaptation_body }   ;

adaptation_body:
                    adaptation_decl_list

adaptation_decl_list:
                    (adaptation_decl)*

adaptation_decl:
                    adaptation_pointcut ; |
                    adaptation_op_dcl ; |
                    adaptation_advice_binding ;

adaptation_advice_dcl:
                    adaptation_before_binding |
                    adaptation_after_binding

adaptation_op_dcl: op_type_spec identifier parameter_dcls (raises_expr)?

adaptation_before_binding :
                    before identifier ( bind_list ) : identifier ( bind_list )

adaptation_after_binding :
                    after identifier ( bind_list ) returning ( identifier ) : identifier ( bind_list )

adaptation_pointcut:
                    pointcut identifier ( (param_dcls) ) : pointcut_body

pointcut_body :
                    joinpoint ( (|| joinpoint)*)

joinpoint:
                    exec ( (op_type_spec) ((scoped_name) .)? identifier ( bind_list ) )

bind_list:
                    (
                          bind_name (, bind_name)* (, ..)? |
                    (..)?
)

bind_name:
                    identifier | _
```

Figure 5. Syntax of the bypass definition language

[9] M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Proc. of the International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[10] M. Conti, E. Gregori, and G. Turi. A cross-layer optimization of gnutella for mobile ad hoc networks. In *MobiHoc '05: Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing*, pages 343–354, New York, NY, USA, 2005. ACM Press.

[11] D. Dean and A. Stubblefield. Using Client Puzzles to protect TLS. In *USENIX Security Symposium*, 2001.

[12] C. Demarey, G. Harbonnier, R. Rouvoy, and P. Merle. Benchmarking the Round-Trip Latency of Various Java-Based Middleware Platforms. *Studia Informatica Universalis Regular Issue*, 4(1):7–24, 2005.

[13] Ö. Demir, P. Devanbu, E. Wohlstadter, and S. Tai. Optimizing layered middleware. *Proceedings of the 5th international workshop on Software engineering and middleware*, pages 33–38, 2005.

[14] G. Duzan, J. Lorall, R. Schantz, R. Shapiro, and J. Zinky. Building adaptive distributed applications with middleware and aspects. In *Proc. of the International Conference on Aspect-Oriented Software Development*, 2004.

[15] D. R. Engler, M. F. Kaashoek, and J. J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266. ACM Press, 1995.

[16] M. E. Fiuczynski, R. P. Martin, T. Owa, and B. N. Bershad. Spine: a safe programmable and integrated network environment. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*. ACM Press, 1998.

[17] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux oskit: a substrate for kernel and language research. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 38–51. ACM Press, 1997.

[18] E. Gamma, R. Helm, R. Johnson, and J. Viissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley, Reading, MA, 1994.

[19] J. Gao, P. Steenkiste, E. Takahashi, and A. Fisher. A programmable router architecture supporting control plane extensibility. *IEEE Journal of Communications*, 38(3), Mar. 2000.

[20] R. Grimm and B. N. Bershad. Separating access control policy, enforcement, and functionality in extensible systems. *ACM Trans. Comput. Syst.*, 19(1):36–70, 2001.

[21] W. Halfond and A. Orso. Amnesia: Analysis and monitoring for neutralizing sql-injection attacks. 2005.

[22] F. J. Hauck, R. Kapitza, H. P. Reiser, and A. I. Schmied. A flexible and extensible object middleware: Corba and beyond. In *Proc. of the Fifth Int. Workshop on Software Engineering and Middleware*, 2005.

[23] D. Hoffman and D. Weiss, editors. *Software Fundamentals: Collected Papers by David L. Parnas*. Addison-Wesley, 2001.

[24] E. Hu, P. Joubert, R. King, J. LaVoie, and J. Tracey. Adaptive fast path architecture. *IBM Journal of Research and Development*, April 2001.

[25] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., 1993.

[26] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072, 2001.

[27] F. Kon, F. Costa, G. Blair, and R. H. Campbell. Adaptive middleware: The case for reflective middleware. *CACM*, July 2002.

[28] F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Commun. ACM*, 45(6):33–38, 2002.

[29] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, C. M. a, and R. H. Campbell. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In *IFIP/ACM International Conference on Distributed Systems Platforms*, pages 121–143. Springer-Verlag New York, Inc., 2000.

[30] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems*, 19(2):217–251, 2001.

[31] G. Muller, E.-N. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 116–126, 1997.

[32] K. Nahrstedt, S. H. Shah, and K. Chen. *Resource Management in Wireless Networking*, chapter Cross-Layer Architectures for Bandwidth Management in Wireless Networks. Kluwer Academic Publishers, 2004.

[33] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Using interceptors to enhance CORBA. *IEEE Computer*, 32(7):62–68, 1999.

[34] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Eternal - A Component-based Framework for Transparent Fault-Tolerant CORBA . *Software Practice and Experience*, July 2002.

[35] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons. The design and performance of a pluggable protocols framework for real-time distributed object computing middleware. In *Middleware '00: IFIP/ACM International Conference on Distributed systems platforms*, pages 372–395, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[36] D. Parnas. The criteria to be used in decomposing systems into modules. *Communications of the ACM*, 14(1):221–227, 1972.

[37] C. Poellabauer, H. Abbasi, and K. Schwan. Cooperative run-time management of adaptive applications and distributed resources. In *MULTIMEDIA'02: Proceedings of the tenth ACM international conference on Multimedia*, pages 402–411, New York, NY, USA, 2002. ACM Press.

[38] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: streamlining a commercial operating system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 314–321, New York, NY, USA, 1995. ACM Press.

[39] I. Pyarali, C. O'Ryan, D. Schmidt, N. Wang, W. Kachroo, and A. Gokhale. Applying optimization principle patterns to desgin real-time ORBs. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*. USENIX, 1999.

[40] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting. Cassyopia: Compiler assisted system optimization. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.

[41] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proc. of the 4th Operating Systems Design and Implementation (OSDI)*, pages 347–360, October 2000.

[42] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[43] D. C. Schmidt. Gperf: A perfect hash function generator. In *Proceedings of the nd C++ Conference, San Francisco, California*, 1990.

[44] D. C. Schmidt and C. Cleeland. Applying patterns to develop extensible and maintainable ORB middleware. In *IEEE Communications Magazine*, volume 37, pages 54–63. IEEE CS Press, 1999.

[45] J. A. Stankovic. Good System Structure Features: Their Complexity and Execution Time Cost. *IEEE Transactions on Software Engineering*, SE-8(4):306–318, July 1982.

[46] J. Stockenberg and A. van Dam. Vertical Migration for Performance Enhancement in Layered Hardware/Firmware/Software Systems. *Computer*, 11(5):35–50, 1978.

[47] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–382, New York, NY, USA, 2006. ACM Press.

[48] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications*, 35(1):80–86, Jan. 1997.

[49] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 193–204. ACM Press, 2004.

[50] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.

[51] E. Wohlstadter, S. Jackson, and P. Devanbu. DADO: Enhancing middleware to support cross-cutting features in distributed, heterogeneous systems. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 174–186, 2003.

[52] E. Wohlstadter, S. Jackson, and P. Devanbu. Dado: Enhancing middleware to support crosscutting services. In *Proceedings ICSE*, 2003.

[53] C. Zhang, D. Gao, and H.-A. Jacobsen. Towards just-in-time middleware architectures. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 63–74, New York, NY, USA, 2005. ACM Press.