Optimizing Layered Middleware

Ömer Erdem Demir[1], Eric Wohlstadter[2], Stefan Tai[3], and Prem Devanbu[1]

[1] UC Davis, Davis, California Republic, USA.
[2] University of British Columbia, Vancouver, BC, Canada
[3] IBM Research, Hawthorne, NY, USA

Abstract. Middleware is often built using a *layered* architectural style. Layered design provides good separation of the different concerns of middleware, such as communication, marshaling, request dispatching, thread management, etc. Layered architecture helps in the development and evolution of the middleware. It also provides tactical side-benefits: layers provide convenient protection boundaries for enforcing security policies. However, the benefits of this layered structure come at a cost. Layered designs can hinder performance-related optimizations, and actually make it more difficult to adapt systems to conveniently address late-bound requirements such as dependability, access control, virus protection, and so on. We present some examples of this issue, and outline a new approach, under investigation at UC Davis, which includes ideas in middleware, architectures, and programming models.

1 Introduction

Distributed systems, comprising components running within different organizational (and thus different administrative) boundaries will underlie future e-commerce, and e-government applications. Integration of data and function is critical for such systems. Middleware plays a key role in this integration, by providing useful model-based abstractions such as distributed objects.

Middleware platforms such as CORBA and Web Services are structured in a layered form, with lower layers providing communication, marshaling etc., and upper layers providing convenient programming abstractions such as remote method invocation or typed messaging services. Layered structure provides software engineering benefits such as separation of concerns, information hiding, portability and evolvability; layers also provide convenient protection boundaries where authentication and access control can be used to protect critical resources. However, for certain types of features, layered architectures become a hindrance. The layers become a barrier that precludes easy adaptation of the middleware.

In our research, we explore *systematic* approaches that allow the middleware to adapt to application needs and environmental considerations. We begin in section 2 with an example to illustrate the problem; in section 3 we describe our general approach and preliminary results; in section 4 we present related work, and conclude with a summary.

2 An Example

Distributed systems operate on public networks, and so are subject to adverse network conditions, random faults, adversaries, erratic human behavior, and viruses. They must handle a range of conditions, such as network outages, market demand spikes, unauthorized access attempts (hackery), security-related incidents (denial-of-service attacks, worms), and software failures. Requirements dealing with such conditions are typically referred to as "Quality of Service" (QoS) In earlier work on DADO [18] we introduced a model-based approach to implementing such features into a new type of component, called an *adaptlet.* An adaptlet is a crosscutting transform that is applied to a distributed system, to add QoS features. Adaptlets have many advantages: they are model-driven, late-bound, and support heterogeneous implementation and introduction. However, adaptlets provide adaptations *exclusively in the application-layer*; we present below an example that illustrates why this is not always a good thing.

We present an example in this section of an adaptation that can be conveniently implemented in the application layer. We also argue that a lower-layer implementation, although more difficult to program, provides much better performance.

2.1 Login Rejection

Consider a service protected by an authentication mechanism, using a password. Such a mechanism can be prone to a *dictionary attack* where someone repeatedly tries to guess the password. We can enhance the authentication mechanism with an adaptation to to resist such attacks: it tracks the number of failed login attempts, within a certain period, from a specific IP address. If a threshold is exceeded, a fixed number of subsequent requests from that IP address are summarily rejected. An application-layer adaptation, using DADO or similar mechanisms, can do this. However, a lower-layer (*e.g.* socket-layer) adaptation would improve performance, by rejecting a request before it is transmitted through the middleware marshaling and dispatching layers.

Unfortunately, without the convenience of abstractions, programming adaptations at lower levels is difficult. Application programmers need to know about the intricate details of the underlying platform. It is also potentially unsafe to make lower-level customizations, since a buggy implementation may compromise all the applications sharing the platform. Moreover, such platform-specific customizations are not portable: programmers may have to re-write adaptations when they are porting to a new platform. Still, lower layer adaptations provide far better performance, as we now describe.

2.2 A Case Study

We have implemented the *Login Rejection* adaptation at the CORBA application (interceptor [17]) level and at the socket level. We compared the performance

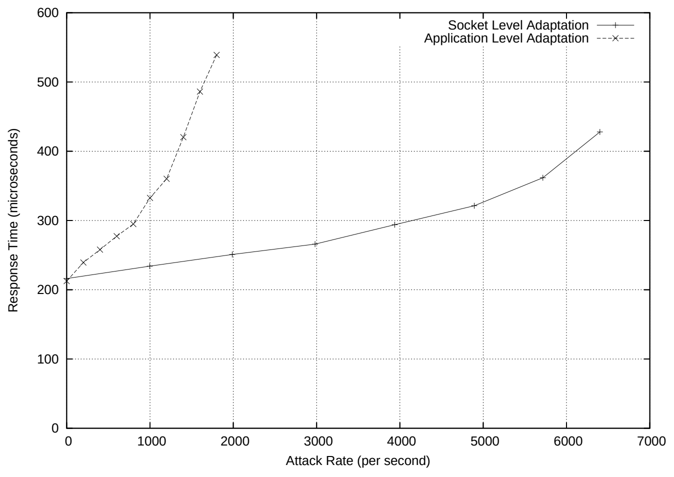

Fig. 1. *Login rejection*: Performance while under attack, with defenses implemented at application layer or socket layer (X-axis: attack rate (per second), Y-axis: response time (μsec). to "good" clients) Socket-layer defense tolerates four times the attack rate, whence our attacking machine saturates.

impact of adaptations at the application layer and an adaptation in the socket layer.

We implemented the adaptations using AspectJ [8] for the open source CORBA middleware JacORB [3]. The performance study was done for a single server and multiple clients. The server machine is an 800MHz Intel Pentium machine with 512MB memory running Microsoft Windows XP. The server is JacORB 2.2 with either application-layer adaptations or socket-layer adaptations. We used an asymmetrical configuration where client machines are more powerful than the server machine. This was done to simulate a large-scale attack on the server machine. One Intel Xeon 2GHz dual-CPU with 1GB memory machine is used for sending attacks to the server. Another machine, Intel Pentium M 1.6 MHz with 1GB memory, is used to send the legitimate requests to the server. Server and client machines are all connected through a 100 Mbit local area network.

The goal of this study is to measure the response rate of the server to legitimate clients, while it simultaneously defends itself from intensive attempts to guess the password. We present the result of this performance evaluation in Figure 1. The x-axis represents the attack rate of the malicious clients ($events/sec$). The y-axis represents the response time perceived by the legitimate clients in microseconds. For both the socket- and the application-level adaptations, as the attack rate increases, response time increases as well. The response time of the application-level adaptation degrades rapidly as the attack rate increases. However, the socket-layer implementation shows much better performance, tolerating several times the attack rate. While the application layer adaptation saturates at around 1800 attack events per second, the socket layer adaptation is not yet in saturation at around 6400 attacks per second; in fact, we were unable to to drive the socket-layer version into saturation with our server and client machine configuration.

2.3 The Difficulty

Implementing changes at the lower levels of the middleware involves intimate details of the data representations and API calls. The lower layers of the system are designed for performance instead of ease of programming. It is risky to make changes in the lower layer because it is shared by many applications. For instance, processing requests at the lower layer involves low-level code on the byte-array representation of the requests such as

```
int request_id = (m[15] & 0xFF ) |
                 ((m[14] & 0xFF ) ≪ 8) |
                 ((m[13] & 0xFF ) ≪ 16) |
                 ((m[12] & 0xFF ) ≪ 24).
```

This type of code is difficult to write. Any errors made in this level of the middleware, which is shared by many applications, may render the transformed adaptation useless or even affect the whole system adversely. Moreover, low-level adaptations developed for a certain middleware may not be portable, since some parts of them will be specific to the target platform.

2.4 Other Examples

We briefly list a few more examples of cases where application-layer functionality can be migrated to lower layers.

Malware Filtering If there is a virus that is spreading using a specific application-layer vulnerability, we can filter requests using application-layer semantics, but implement it at the socket layer.

Request Forwarding If requests arriving at a server are to be redirected (*e.g.,* for load balancing, or fault-tolerance) using application-level information, these can be migrated to the lower layers.

Simple Application Functionality Application functionality that is simple, static in scope (true for all instances of a server), and/or idempotent (or nearly so, such as with cached responses) would be candidates for lower-layer implementations.

Our goal is to actually *generate* highly optimized lower-layer code (such as the example shown above) while providing developers the convenience of programming at the application layer.

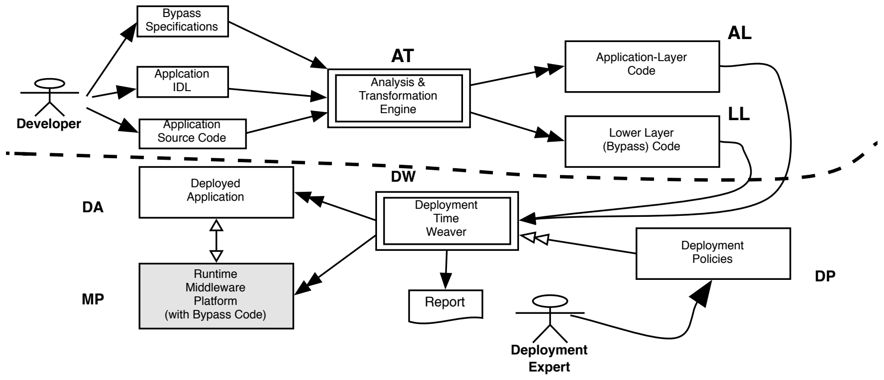

Fig. 2. Cross-layer implementation of application functionality. Middleware-specific tools analyze application code and derive potential lower-layer adaptations. Weavers driven by deployment-time policies determine the actual configuration of middleware and application

3 General Approach

Our general approach is based on the use of analysis and optimization in the deployment of adaptations. An outline of our approach is presented in Figure 2.

There are two parts to our approach: development time (above the dashed line) and deployment time (below). It includes tooling (double-border boxes) and some middleware-specific elements (gray boxes). The development time part centers around an analysis and transformation engine (AT) (top double-outline box) which is driven (white double-head arrow) by a set of middleware-specific analysis and transformation rules (MTR). These rules identify which application-layer API calls and operations can be migrated to the socket layer in the middleware. The developer implements the middleware-based application in the usual way. In addition, she provides annotations on the IDL that identify specific methods and applicable exceptions that are candidates for migration. The AT engine uses this information to analyze the source code and generate (black double-headed arrow) application-layer (AL) code and lower-layer code. The lower-layer code (LL) is a candidate adaptation (in our current prototype, it is generated as an AspectJ aspect, using Meta-AspectJ [21]) that could be woven into the socket layer of the middleware. The generated AL and LL code are used at deployment time by a weaver (DW), which weaves the lower layer adaptation into the middleware proper, and also generates the requisite application code. The weaving of the (transformed) application (DA) layer code into the lower layers of the middleware platform (MP) is governed by deployment policies (DP) specified by the deployment expert.

We now discuss these elements in more detail.

3.1 Analysis & Transformation

The AT engine is based on program slicing, and is driven by knowledge of the middleware platform. We rely on information provided the developer, in the form of annotations on the application IDL, to identify portions of specific application method implementations that can be transformed into the lower layers. Consider, for example an application method like

```
float getDataResidue (in String name, in String key) throws Redirect
```

The programmer-provided IDL annotations may indicate, for example that this method can be moved to the lower layer, in case the `Redirect` exception is thrown. This could be indicated using a notation such as the following:

```
(throw Redirect) & (cflow (call (getDataResidue)))
```

This indicates a particular pattern in the control flow of the application, beginning with the call to the `getDataResidue` method, and ending with the `Redirect` exception being thrown. In the spirit of AspectJ pointcuts, we would allow other event patterns to be used in these control-flow patterns, such as wildcards on methods, exceptions, and on `return`s from methods. In the case a `return` is used in the pattern, the programmer would have to identify the specific `return` that is to processed at the socket layer. Using this information, the AT engine identifies candidate locations in the code where the `Redirect` exception is thrown, and constructs a backward slice starting with each of those locations. Currently our AT engine selects only those slices which:

1. Access only method invocation parameters
2. Contain no accesses to non-static data and non-static method calls, and
3. Make no calls to other remote objects or remote methods.
4. Use no middleware service API calls (such as the POA API, the interface repository, etc).

Slices that meet this requirements can be transformed into lower-layer code, using middleware-specific transformation rules. This code can act to short-circuit middleware up-calls that are unnecessary, thus improving efficiency. The remaining residual code of each method (i.e. not in the slice) remains at the application layer, to handle requests that could not be short circuited.

We are currently implementing the analysis part using the SOOT [4] framework. Specifically, we are extending the Indus [2] tool from the Bandera [1] group at Kansas State University. We now discuss the general structure of such generated code.

3.2 Generated Code

The generated code will have the structure of the slice that was identified in the application layer code, but the actual implementation details will be based on lower-layer operations.

For example, a predicate on application-level data might be transformed into some set of operations on the raw (marshaled) data from the socket. As another example, throwing a particular exception back to the client might involve constructing a specific message and sending it on a socket connection. For efficiency reasons, the bulk of the message could be reused, changing only the data specific to a particular `throw` operation. Static data would be accessible in both lower and upper layers, and could be modified or accessed from either layer.

We expect that the general structure of a lower-layer adaptation would be as follows:

```
/* Code to access request parameters in incoming message */
/* Code (optional) to modify/access static state */
/* Code to construct reply message */
/* Reply message transmission */
```

Currently, we are generating this code as an aspect in AspectJ [8], using the meta-AspectJ toolset [21]. Since our initial prototype is based on JacORB [3], the aspect is intended to be woven into the JacORB IIOP packet-handling layer.

3.3 Deployment Considerations

There are complications in inserting application-layer functionality directly into the lower layers of the middleware. Because of this, deployment-time considerations must influence how lower-layer adaptations are actually engaged into the middleware.

Here are some factors that may influence how lower-layer adaptations are deployed:

1. Processing a request at a lower layer might vitiate application-layer scheduling or priority considerations. Thus a stateless, but lower-priority request might be processed entirely at the socket layer, ahead of stateful, but higher priority requests that need application-layer processing. If there are realtime deadlines, or if there are priorities assigned to requests, then care must be taken to avoid priority inversions.
2. Similar issues apply to security or access control, which is typically performed by application-layer interceptors (See, *e.g.* Narasimhan *et al* [13]).
3. Concurency issues can be deployment dependent. In CORBA [14], the portable object adapter allows a choice between single thread or multiple thread processing. If the lower-layer adaptation is stateful, care must be taken to synchronize data between the lower and upper layers; indeed, in such cases lower-layer adaptations may not be practical.

The ideal way to deal with such issues is for the developer and deployer to independently specify, in a high-level formal notation, the actual dependencies between application events (such as returns and exception throws) and policy classes, such as security, concurrency, real-time scheduling etc. A decision procedure would then recommend adaptations that can be safely moved into the middleware in a specific deployment context. Our long-term is to construct a declarative, reasoning framework similar to GlueQoS [19] to support this style of development and deployment. For now, we simply expect the deployment expert to directly specify whether a specific lower-layer adaptation is to be inserted into the middleware, or not.

One issue that naturally arises here is the *dynamic adaptation* of the lower layers of the middleware in response to application-specific conditions that arise at run-time. In some cases, the relative performance of application-layer and socket layer may actually depend on run-time conditions: the socket layer adaptation, if it imposes slight additional load on every request, may only show better performance under very heavy load conditions. In such settings, it would desirable to "engage" the socket layer adaptation only under heavy load. In order to perform this type of adaptation, one could rely on the use of dynamic aspects [6, 15]. Alternatively, the socket layer could be pre-instrumented with a "hook" that checks a flag to see if any adaptations need to be run. Although our current design uses static aspects, it is natural and complementary to add such a capability, and we plan to do so.

3.4 Summary

Our goal is to allow the application programmer to have it both ways: abstract programming, and efficient implementation. So in effect, each architecture must come with a set of adaptation transforms, which encapsulate knowledge on how to transform adaptations in that architecture. Thus, it is very clear that one must provide a convenient programming model for implementation of such transforms.

4 Related Work

In this Section we discuss related work focused on adaptation and customization of the middleware.

There are specially constructed ORBs [9, 10, 5] that allow easy modification of the internal structure for adapting to the unforeseen changes in the operation environment. DynamicTAO [10] is a reflective CORBA ORB designed on top of TAO [16]. 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 [13]. The DADO system [18] 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. Our approach is compatible with these middleware adaptations and meta-programming facilities. For instance, DADO adaptations can be transformed to execute in the lower levels of the middleware, thus improving the performance. Our approach allows middleware architectures to embody transformations that can allow the optimization of the adaptations.

Partial evaluation is program transformation via specialization [7]. Programs can be specialized using techniques such as symbolic computation, unfolding functions and program point specialization. The goal of partial evaluation is to precompute as much possible of the program given known inputs to optimize the program. Partial evaluation has been used to optimize the SunRPC [12]. 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 [11]. Unlike partial evaluation, our approach analyzes the application code to optimize the system for the application by weaving the parts of the application into the middleware. The partial evolution work does not fuse application functionality with the lower layers of the system. Our approach is not general and relies on the knowledge of the architecture of the middleware. However, unlike partial evaluatio, we expect that our approach will be easier to use and not require manual tuning.

Perhaps the most closely related work to ours is the "Just-in-time Middleware" work by Zhang, Dapeng and Jacobsen [20]. By analyzing applications IDLs, they eliminate portions of the middleware platform corresponding to unused features. They aim to reduce the footprint of the middleware if some supported features are not needed. Our goal is complementary: to move actual application layer functionality into the middleware to provide improved performance. In general, we enthusiastically embrace Zhang, Dapeng and Jacobsen's notion of "Just in Time Middleware"; 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 of Zhang, Dapeng and Jacobsen.

5 Current State and Conclusion

This project is in its early stages. Our preliminary results show that application-specific lower-layer adaptations are more efficient than the higher-layer adaptations. Lower layers are designed for performance instead of ease of programming. It is risky to make changes in the lower layer because it is shared by many applications. Since our approach facilitates construction and integration of lower-layer adaptations, application developers can enjoy more convenient programming, and users can enjoy better performance.

References

1. The Bandera Project web page. http://indus.projects.cis.ksu.edu.
2. The Indus static analyzer. http://indus.projects.cis.ksu.edu.
3. JacORB: Java implementation of the OMG Corba standard. http://www.jacorb.org.

4. Soot: a Java optimization framework. http://www.sable.mcgill.ca/soot/.
5. G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski. The design and implementation of Open ORB 2. *IEEE Distributed Systems Online*, 2(6), 2001.
6. J. Bon&#233;r. What are the key issues for commercial aop use: how does aspectwerkz address them? In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 5–6, New York, NY, USA, 2004. ACM Press.
7. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., 1993.
8. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
9. F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Commun. ACM*, 45(6):33–38, 2002.
10. 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.
11. 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.
12. 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.
13. P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Using interceptors to enhance CORBA. *IEEE Computer*, 32(7):62–68, 1999.
14. Object Management Group. The Common Object Request Broker Architecture. http://www.corba.org.
15. A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for java. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 100–109, New York, NY, USA, 2003. ACM Press.
16. 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.
17. N. Wang, D. Schmidt, O. Othman, and K. Parameswaran. Evaluating meta-programming mechanisms for orb middleware. *IEEE Communication Magazine, special issue on Evolving Communications Software: Techniques and Technologies*, 2001.
18. 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.
19. E. Wohlstadter, S. Tai, T. Mikalsen, I. Rouvellou, and P. Devanbu. Glueqos: Middleware to sweeten quality-of-service policy interactions. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 189–199, Washington, DC, USA, 2004. IEEE Computer Society.

20. C. Zhang, G. Dapeng, and H.-A. Jacobsen. Towards just-in-time middleware. In *To appear in AOSD'05*.
21. D. Zook, S. S. Huang, and Y. Smaragdakis. Generating aspectj programs with meta-aspectj. In G. Karsai and E. Visser, editors, *GPCE*, volume 3286 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2004.