2003

An Architectural Pattern for Adaptable Middleware Infrastructure

Jason J. Mitchell
University of North Florida

AN ARCHITECTURAL PATTERN FOR ADAPTABLE MIDDLEWARE INFRASTRUCTURE

by

Jason J. Mitchell

A project submitted to the
Department of Computer and Information Sciences
in partial fulfillment of the requirement for the
degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES

April, 2003

APPROVAL BY THE PROJECT COMMITTEE

The project "An Architectural Pattern for Adaptable Middleware Infrastructure" submitted by Jason J. Mitchell in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the Project Committee: Date

Signature Deleted

4/30/03

Arturo Sanchez, Ph.D.
Project Director

Signature Deleted

4/25/03

Judith Solano, Ph.D.
Chairperson of the Department

Signature Deleted

4/30/2003

Charles Winton, Ph.D.
Graduate Director

ACKNOWLEDGEMENT

I wish to explicitly express gratitude towards my eternal companion for enabling me all the successes of my life.

CONTENTS

FIGURES

ABSTRACT

Middleware technologies change so rapidly that designers must adapt existing software architectures to incorporate new emerging ones. This project proposes an architectural pattern and guidelines to abstract the communication barrier whereby allowing the developer to concentrate on the application logic.

We demonstrate our approach and the feasibility of easily upgrading the middleware infrastructure by implementing a sample project and three case studies using three different middlewares on the .NET framework.

Chapter 1

THE ROLE OF MIDDLEWARE AND APPROACHES TO IT

1.1 Distributed Communication

Software applications need to be distributed for many reasons. Because of the increasing need to build these applications and the existence of so many communication protocols, certain types of middlewares have been developed to isolate developers from dealing with low-level details that are foreign to the core functionality of the application at hand. True distributed systems application should not be aware of such communication boundaries; ideally, it should be handled by the underlying run-time systems themselves. However the state of the art in distributed computing and large-scale enterprise development in general isn't quite there yet. An approach that was popularized by CORBA [OMG98] consists of introducing a software layer that abstracts out many of the subtleties associated with communication issues.

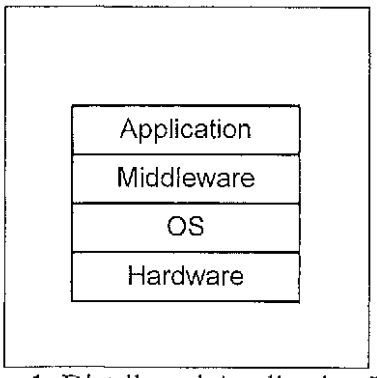

Figure 1: Distributed Application Layers

Figure 1 shows how a new layer of software called middleware now sits between applications and the operating system abstracting the network communication heterogeneity and simplifying distributed communication for the application developers. New demands are now being placed on the middleware layer and taken away from the application developer. Factors such as component integration, adaptive environments, and real-time interactions drive middlewares to new heights [Tripathi02]. Thus, the variety and complexity of middlewares is increasing.

1.2 Approaches to Middleware-Based Architectures

There is a plethora of middleware architectures, frameworks, and protocols. They try to tackle different problems and complexities. Each additional feature of a middleware has a cost associated with it; most of the time it's a performance hit or a new learning curve to tackle for the team.

New policy-driven middleware approaches like QuO handle many scenarios such as dynamic security requirements, ad hoc networking of devices, and context-aware computing [Tripath02].

Resource management becomes a key factor in the middleware arena. Resource awareness and dynamic reallocation of resources are important responsibilities of a resource management system. A way of adapting to the network via reflection techniques is a key approach one framework has attempted to accomplish [Duran00].

Many other examples of middleware architectures make use of some of the aspects discussed already. Some examples are Artic Bean developed at the University of Tromso [Anderson01], a composable reflective framework at the University of California, Irvin [Venkatasubramanian01], an open network platform protocol developed at Ericson [Jozic], and an Advanced Communication Toolkit (ACT) developed at Rutgers University [Francu99].

Most of these implementations are either built or based on commercial object-oriented middleware technologies such as OMG's CORBA, Sun's RMI, Microsoft's COM+, and IBM's MSQ. All of these commercial implementations offer great advantages when building a distributed system, and work well for certain scenarios.

It is even easy to choose which one will work best for the current implementation of the application given its domain. The unavoidable problem that arises is change: the domain, the complexity, the environment, or the application will change, and this may mean that the middleware infrastructure needs to be changed to adapt to the new requirements. What designers have to do is expect the inevitable and prepare for it.

1.3 Discussion

Let us suppose that we have chosen a middleware and have written a client/server system. This means that we have an application logic that interfaces with the middleware Application Programming Interface (API). This also means that we

have probably defined a messaging infrastructure whereby we have defined the messages being passed between certain components of the application. Most of the time this is done by means of some sort of interface definition language (IDL) so the messaging infrastructure knows how to marshal/un-marshal the complex types across the network, which calls for mappings between our application-specific complex types and the types defined as our messages. An application so designed is inherently prone to be tightly dependent on the middleware in question!

What does this mean for our application developers? They would potentially need to modify large segments of the logic that uses the middleware API when evolution imposes the use of a new middleware. This also means that they would need to write a new set of classes to map to the new set of interface definition types. This not only means more development but the applications themselves need to be recompiled, retested, and redeployed.

This is far too much overhead for something that could have been avoided from the beginning. This project will show how an approach to avoid these pitfalls, which could lead to a better utilization of resources such as time and money.

Chapter 2

AN ARCHITECTURAL PATTERN APPROACH

2.1 The Problem

Figure 2 depicts a component-based architecture with different forms of middleware used between them. The point here is that many different middlewares may and should be used to handle different scenarios in the context of a distributed enterprise system. The task for the architect is to design the system in such a way such that adapting to change is accomplished with minimal effort.

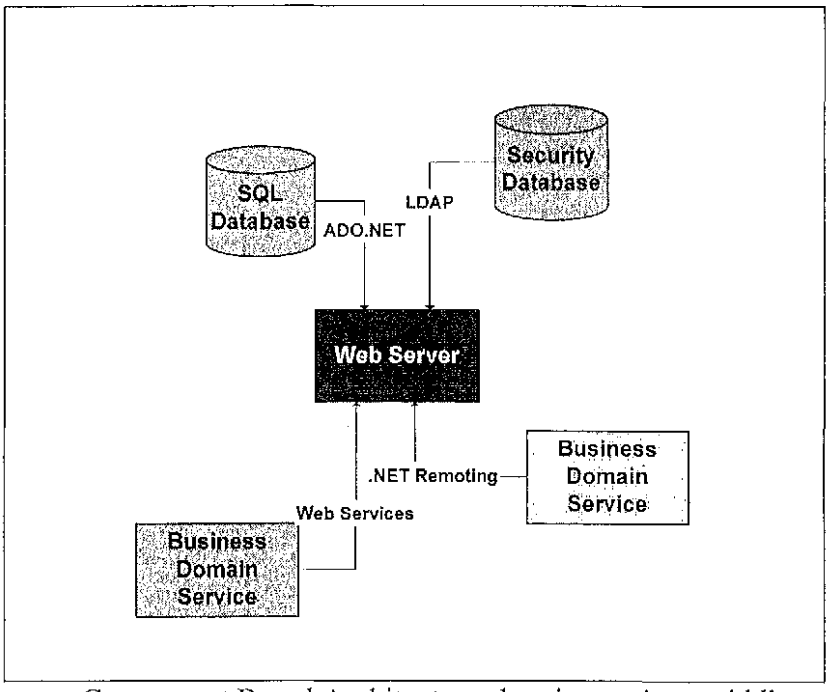

Figure 2: Component Based Architecture showing various middlewares

While only four different middlewares are mentioned, dozens more (some of which were mentioned in the previous chapter) could be used interchangeably depending on certain requirements of the system. For instance, if the web server and the business domain service interact within the local area network .NET remoting offers the best performance. If however our business domain service needs to be used with applications over the wide area network, then we might want to use web services because they are designed to go over the HTTP protocol and pass through firewalls.

This requires a layer of abstraction between the applications and the middleware.

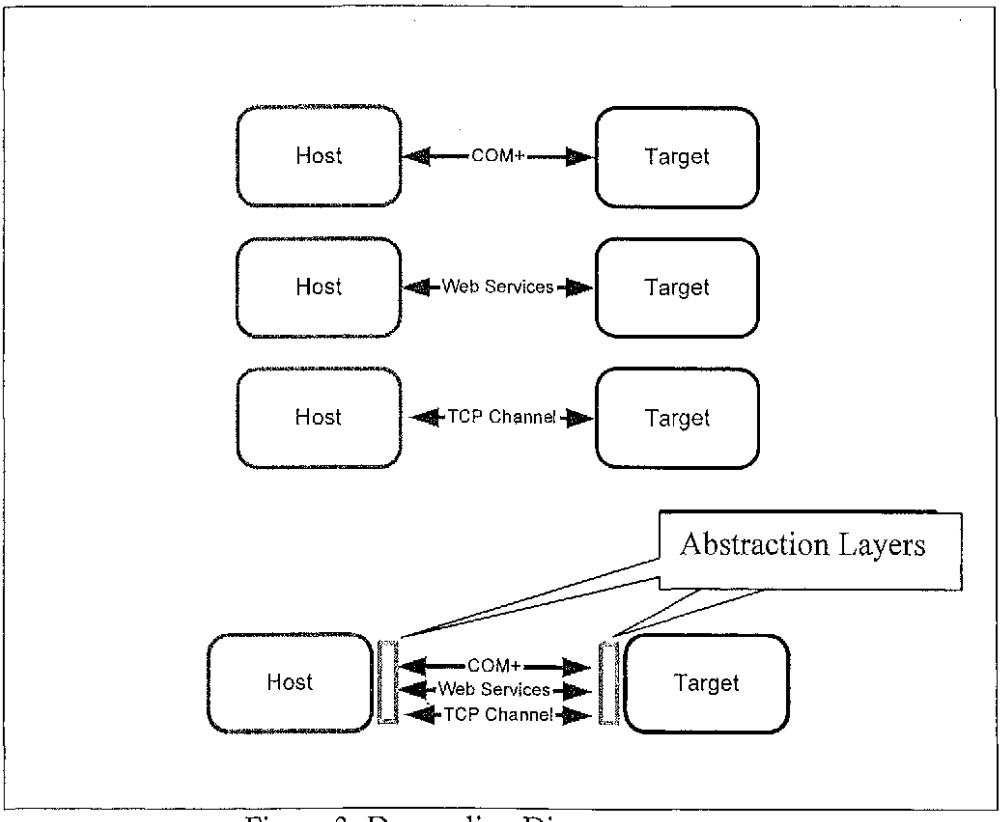

Figure 3: Decoupling Diagram

Figure 3 shows several two-component diagrams. The top picture shows that an application can be coupled to the three different middlewares that it may use. The diagram at the bottom illustrates an approach to de-couple the application from the middleware API by creating an abstraction layer that existing middleware infrastructure details can easily be bound to, which will allow new bindings containing different middleware infrastructure logic to be cleanly bound to the generic host that uses it.

This abstraction layer must remain thin enough as to not to compromise the flexibility of the concrete middleware and not to introduce unnecessary dependencies in the application with respect to it

In the previous chapter we discussed two key areas where system designs can cause a lot of overhead when trying to switch between middlewares. The first was the Application Programming Interface (API) and the second was in the messaging infrastructure. We shall now explain these in more detail.

2.2 The Application Programming Interface (API) Perspective

Each application must be written to interact with the API of the middleware. This means that we must reference external libraries and couple some business logic to interoperate with the middleware. Switching to a new middleware therefore entails changing that code to now interface with the new middleware's API. Now that we have changed some of our business logic the whole system needs to be retested and the interaction code needs to be redeployed.

An example of this scenario that is often found is when the application logic is built so that it obtains a reference to the remote server to then pass it through the code to be called upon when necessary; which could potentially (tightly) couple the system to the middleware.

The solution to this is to have the host applications bind to an interface. Then, implement code to bind the logic between the interface and the middleware. This not only allows new bindings to be introduced but also saves us from having to recompile, and even re-test the application logic that uses it.

2.3 The Messaging Perspective

The other place where designers might not foresee the need for future changes is in the messaging between the components. Two applications communicate with each other through the transfer of complex data types.

Current implementations of middleware offer some sort of interface definition language to define the complex types so that they can be marshaled and un-marshaled to be sent across the network

This creates a problem when designers couple the application to this data representation. With each message being passed between applications we must define the types and instruct the middleware how to send types across the network. A substantial amount of work is required to map large data objects in any interface

definition language. When applied several times to different middleware, the headache of re-implementation surfaces quickly [Emmerich99].

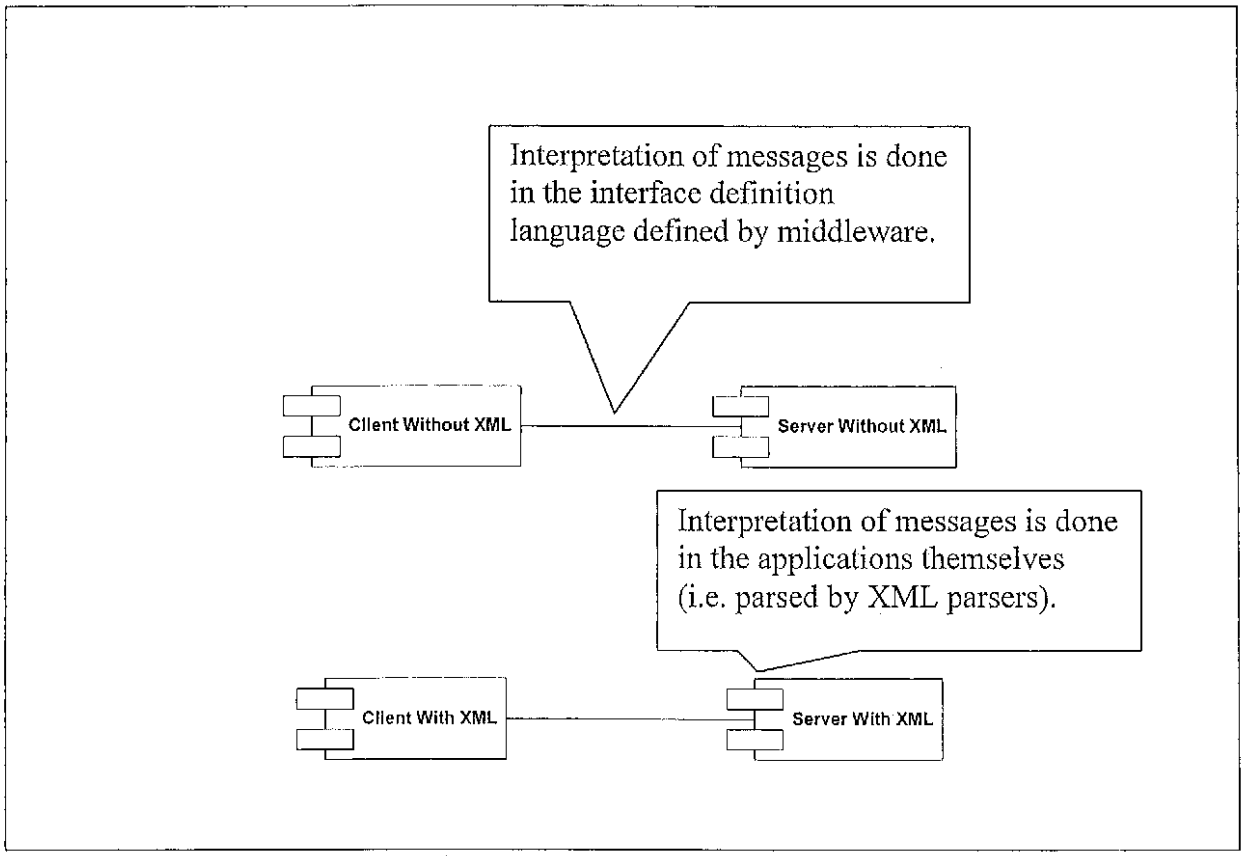

Figure 4: Approaches to Message Interpretation

Figure 4 shows two approaches. The first entails defining the data types within the middleware, which means that we must do this for each middleware. The second approach entails interpreting the data types in the applications themselves.

To allow for extensibility an "open binding" approach [Fitzpatrick98] allows for self-description or meta-data information and late binding. This means that the applications will be responsible for parsing and interpreting the messages being

passed across. The only thing the middleware knows about is that a character string is being passed across. This exchanging of strings is where the flexibility and decoupling of data and messaging definition come into play.

The current primary choice for this is the extensible markup language (XML), which offers a generic loosely coupled integration environment. The messaging infrastructure is overall more extensible and adaptable and lays the messaging infrastructure foundation for a heterogeneous and diverse market of middleware communications [Nusser01].

With the introduction of an adapter-like pattern abstracting the API and an extensible messaging infrastructure the groundwork will be laid for our architectural pattern, which when instantiated appropriately leads to highly flexible and adaptive distributed systems. This will become more and more important as many new middlewares will be introduced in the next years to come.

Appendix A describes our pattern in a format similar to the one used by Stephen Stelting [Stelting02], and includes a recipe for instantiating this pattern. In the next chapter we will demonstrate case studies that use our pattern.

Chapter 3

A CASE STUDY

3.1 Problem Domain

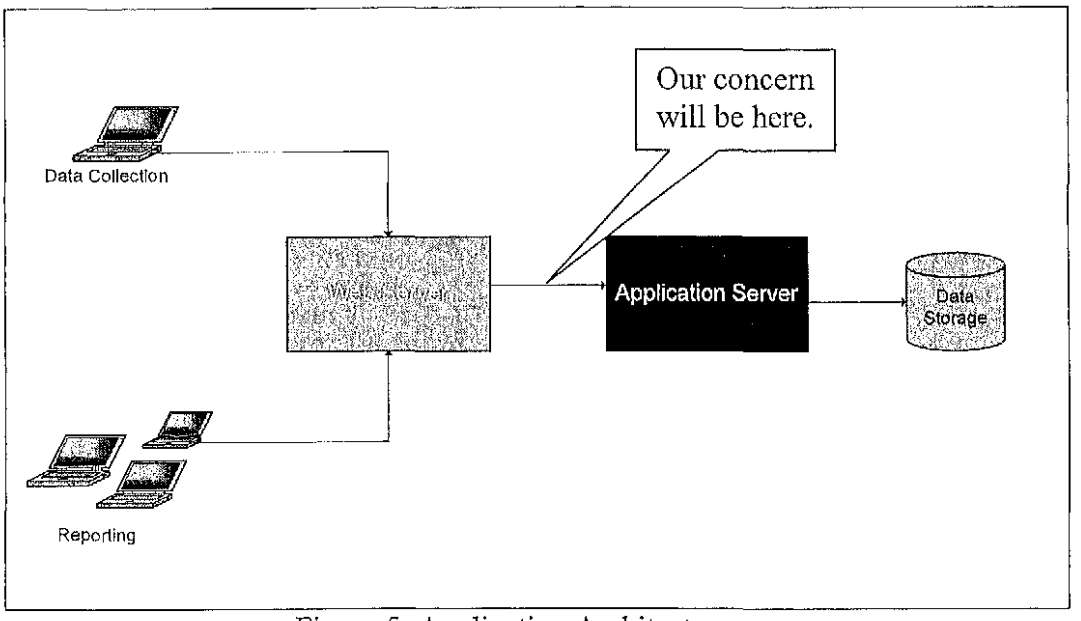

Figure 5: Application Architecture

The application that spawns our case studies is a sports statistic application used to report real time statistics of athletic events. Figure 5 shows the architectural layout of our example application.

The figure also highlights the communication link our case study focuses on. We will use the terms "web server" and "application server" to distinguish between the two servers.

Depending on factors such as network layout, performance requirements, and flexibility we would choose among many different middlewares to best fulfill the

requirements of the system. For our case study we will use three different types of middleware all provided by the .NET framework. We chose the .NET framework because of the inherent XML tools it provided. We could have just as easily used any other platform.

3.2 Design

For our design we will now instantiate our architectural pattern (c.f. Appendix A) and develop a solution that will enable the swapping of a new middleware easily.

The second step in the pattern "Implementation" section is about allowing our applications to interpret the messaging infrastructure. We also mention that the best way to do this is by using XML as the format for passing such messages.

The .NET framework offers some great tools when it comes to serializing classes into character streams using XML. For our case studies we have decided to use these tools.

We just have to create our complex types that we would like to use and then auto-generate the XML marshaling of the complex type to a character string, which implies that we will not have to describe our types to the middleware; we only express one type of message going across the middleware, namely a simple character string.

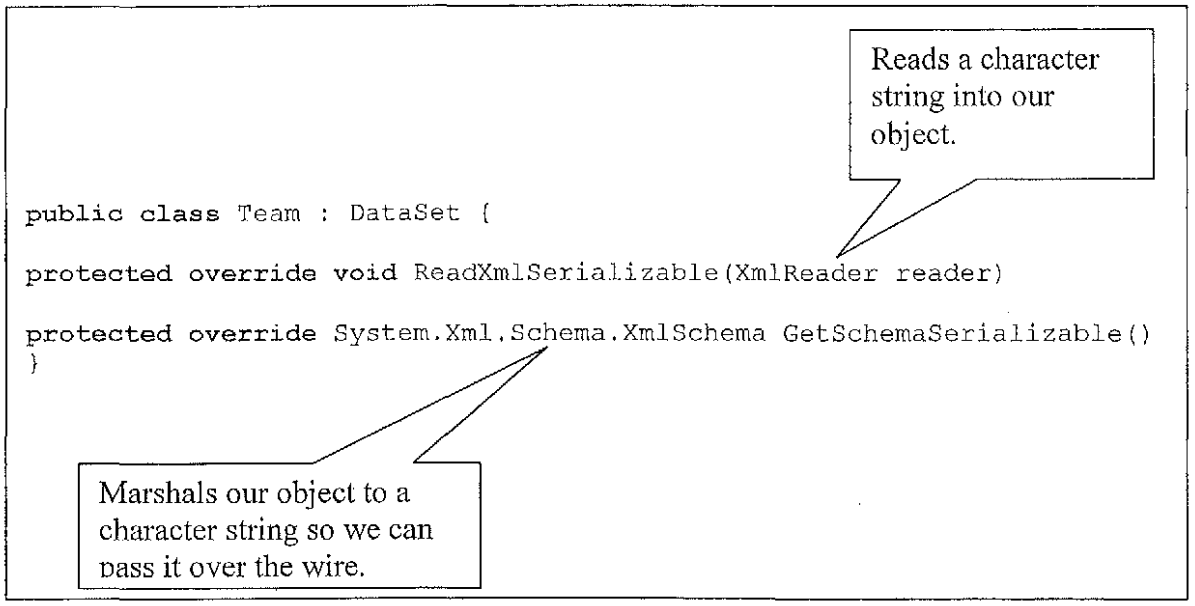

Figure 6: NET Message

Therefore, the messaging infrastructure will be the same for any middleware we decide to use. We will not mention the messaging in the three different case studies because they are all the same. Our applications will do the interpretation (parsing) of the messages independent of the middleware. This de-coupling allows the middlewares to change and we will never have to describe to the middleware how to marshal our messaging infrastructure.

Figure 6 above shows the portion of a class that was auto-generated by the .NET framework. It shows that the tool will create the methods to marshal any complex data type into a character string and read from a character string back into our object. This class has all of our typed parameters being passed across for ease of use within our other code.

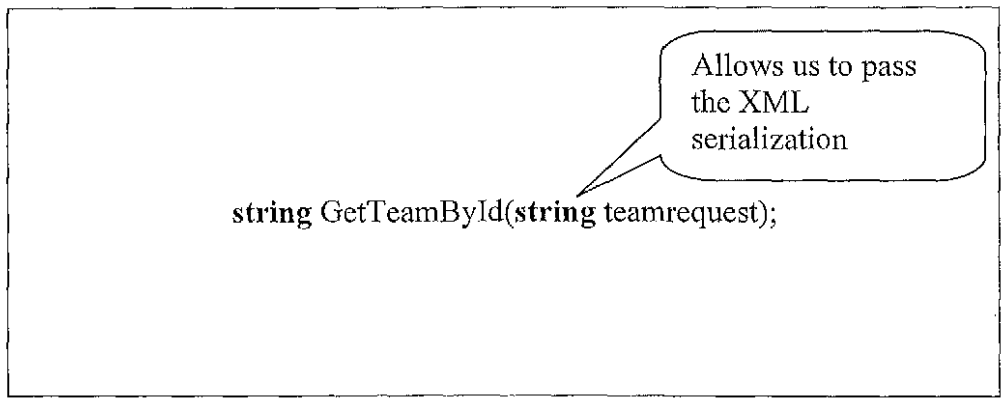

Figure 7: Interface Definition

This allows our application logic to handle any changes in our communication messages without ever changing the middleware. This also means that if we change the middleware we don't have to map our objects or define our complex types to the middleware.

The first step of our pattern describes a way to de-couple the middleware API from the host application. To do this we will create an interface that the host will bind to. Then we will implement the interface with a class that will act as a bridge to the target that will service the request.

The code above in Figure 7 is the interface that the client code would bind to. The implementation of this object will be dynamically loaded. As long as the interface doesn't change, the host application logic would not have to change or be re-compiled or be re-tested.

3.3 Case 1: COM+

3.3.1 Client side:

Using Microsoft's distributed communication protocol COM+, Figure 8 shows an example in C# of how to obtain a reference to the remote server and invoke the service layer to retrieve the team.

This class would implement the Item Service interface and act as a proxy to the remote server. There are references and configuration setup that would be coupled with this class. This implementation of the client side API to COM+ retains all syntax referring to COM+. The (XML) messaging is returned to the host and its code knows nothing about the interactions with COM+.

```
public class COMTeamClient : ITeamService
{
 public string GetTeamById(string teamrequest)
 {
        try
        {
              TeamMgr mgr = new TeamMgr();
              return mgr.GetTeamById(teamrequest);
        }
        catch(Exception ex)
        {
         throw ex;
   }
 }
}
```

Figure 8: COM+ Client

```
[Transaction(TransactionOption.Required)]
[Guid("822A6BC5-1C84-4052-838E-FA47E6EDADC3")]
public class TeamComponentService : ServicedComponent, ITeamService
{
        public string GetTeamById(string teamId)
        {
                return new TeamService().GetTeamById(teamId);
        }
}
```

This is how we forward the request.

Figure 9: COM+ Server

The class in Figure 8 implements the ITeamService interface. This is done so the web server application can bind to this interface and won't need to be changed if we add a new implementation.

3.3.2 Server side:

Now on the server side, the class that accepts the COM+ request would then forward the request onto the service layer as show in Figure 9. The service layer would retrieve the respective team and return the XML payload string to this method to be passed back over COM+.

Just like with the client's side, all API references are kept within this abstraction layer so that they are not coupled with the applications that are using them. Furthermore, these classes would be kept in a separately linked library so that none of the application logic using this abstraction layer would have to be re-compiled after the initial release.

3.4 Case 2: .NET Remoting

Let's suppose that COM+ did not suffice as a middleware between the applications. Now we have to change all of the code that references the COM+ API and change it so it will then reference .NET remoting syntax.

3.4.1 Client Side:

Figure 10 shows the implementation of the same interface mentioned before but now this implementation obtains a reference using a different set of API libraries. Notice we will not have to change any of the code that uses this implementation. As long as we dynamically load this class we won't have to compile, test, or re-deploy and host application code.

```
public classs RemotingTeamClient : ITeamService
{
 public string GetTeamById(string teamrequest)
                                        {
        try
        {
                string url = "http://localhost/TeamService/Team.rem";
                TeamMgr mgr = (ITeamService) Activator.GetType(typeof
(ITeamService),url);
                return mgr.GetTeamById(teamrequest);
        }
        catch(Exception ex)
        {
         throw ex;
   }
 }
}
```

Obtains a reference to a .NET remoting server.

Figure 10: .NET Remoting Client

3.4.2 Server Side:

```
public class TeamRemotingService : MarshalByRefObject, ITeamService
        {
                public TeamRemotingService()
                {
                }

                public Team GetTeamById(int teamId)
                {
                        return new TeamService().GetTeamById(teamId);
                }
}
```

This is how you set an object up to use .NET remoting.

Figure 11: .NET Remoting Server

Figure 11 shows how to setup a server object to be obtainable by .NET remoting, using middleware-specific syntax, and there is some configuration necessary to set this up as well. The minimal configuration changes and the new implementation of this class is all that was needed to swap out one middleware infrastructure to another one from the server end. Once again, we didn't have to make any changes to the TeamMgr class and anything it uses. This saves us from having to test it.

3.5 Case 3: Web Services

As a third example we will now communicate with the remote server using web services. Under the .NET framework we would need to change some configuration information, such as add a reference to the web service, and compile the web service proxy. Each toolkit used to create a web client or server would be different.

Once the proxy is built you just refer to it like any other object. The .NET framework has done a lot to make the integration with web services very seamless. There are many more protocols such as CORBA that make it more difficult to integrate with.

The server side portion is not so straightforward. Not only one needs to extend a web class, but also to mark each method as one published by this web service. Figure 12 shows an example of this.

3.5.1 Server Side:

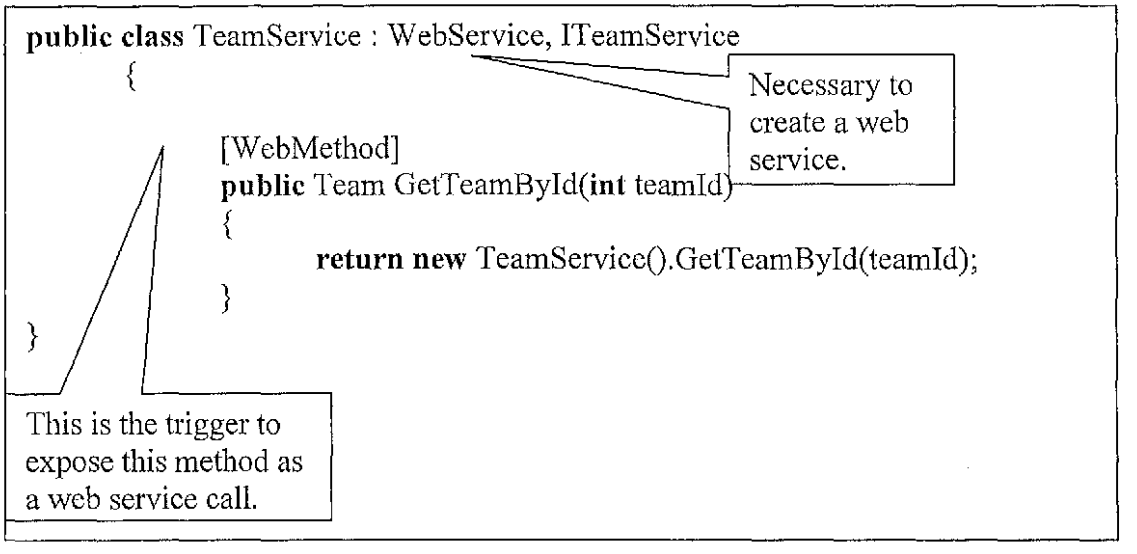

Figure 12: Web Service Server

Above is an example of how to listen to web services under the .NET framework. The web service method GetTeamById will listen for a request. Once a request is accepted this service will then passed the request onto the web service independent service layer that will service this request.

3.6 Summary of Case Studies

As shown in all three cases, the decoupling of the middleware infrastructure from our business applications can be achieved by applied our architectural pattern, which calls for the separation of the application from API-specific functionality, and the introduction of an extensible messaging framework. We demonstrated this strategy with three different middlewares but this could just as easily been done with any middleware on the market.

Chapter 4

CONCLUSIONS

The case studies presented in our project demonstrate that swapping among three different middlewares can be accomplished by a small amount of configuration changes and only a few systematic modifications to the source code. The real key is that none of the actual business logic on the client and server side needed to be recompiled or tested. Only the code that depended on the specific middleware infrastructure dependent had to be altered.

Since changes associated with the middleware are inevitable for some application domains, developers should prepare in advance to face them. In this project we have presented an architectural pattern that enables the interchanging of middlewares with minimal effort and overhead for the development team.

REFERENCES

[Anderson01]
Anderson, Anders. "Artic Beans: Configurable and Reconfigurable Enterprise Component Architectures". IEEE Distributed Systems Online, 2001, Vol 2, Number 7. See http://dsonline.computer.org/0107/features/and0107.htm

[Duran00]
Duran, Hector. "A Resource Management Framework for Adaptive Middleware". IEEE, March 2000 pp 206-209.

[Emmerich99]
Emmerich, Wolfgang. Schwarz, Walter. Finkelstein, Anthony. "Markup Meets Middleware". ACM. Proceedings of the Seventh IEEE Workshop on Future Trends of Distributed Computing Systems. December 20 1999, Tunisia, South Africa, p 261.

[Fitzpatrick98]
Fitzpatrick, Tom. Blair, G. Coulson, G. Davies, N. Robin, P. "Supporting Adaptive Multimedia Applications through Open Bindings". Proceedings of the International Conference on Configurable Distributed Systems, March 04-06, 1998. Annaolis, Maryland.

[Francu99]
Franco, Cristian. Marsic, Ivan. "An Advanced Communication Toolkit for Implementing the Broker Pattern".. Proccedings of the 19th IEEE International Conference on Distributed Computing Systems. May 31 – June 4 1999. Austin, Texas, p 458.

[Geihs01]
Geihs, Kurt. "Middleware Challenges Ahead". IEEE-Computer, Jan-June 2001, Volume 34, pp 24-30.

[Charles97]
Thompson, Charles. "A Scout's Guide to Three-Tier Architecture". Database Programming and Design, August 1997.

[Gold-Berstein98]
Gold-Berstein, Beth. "Race to the Middle", Database Programming & Design: Volume 11, February 1998, pp 28-33.

[Jozic]
Jozic, Danijel. Osmanlic, T. Huljenic, D. Sinkovic, V. "Open Network Platform for Multiprotocol Communication". Proceedings of the 25th Annual IEEE Conference on Local Computer Networks. November 09-10, 2000, Tampa, Florida.

[Mullender02]
Mullender, Maarten. "Some Architectural Patterns for the Enterprise", Webcast, Microsoft 2002. See http://www.microsoft.com/usa/webcasts/ondemand/960.asp

[Nusser01]
Nusser, Gerd. Schimkat, Dieter. "Rapid Application Development of Middleware Components by Using XML", Proceedings of the 12th International Workshop on Rapid System Prototyping. June 25-27, 2001, Monterey, California.

[OMG98]
OMG. "The Common Object Request Broker: Architecture and Specification Revision 2.2". 492 Old Connecticut Path, Framingham, MA 01701, USA, Febrary 1998.

[Schaeffer99]
Schaeffer, Jonathan. Sztipanovits, Janos. Karsai, Gabor. Moore, Michael. Ledeczi, Akos. Long, Earl. The Enterprise Model for Developing Distributed Applications. Proceedings of the IEEE Conference and Workshop on Engineering of Computer-Based Systems. March 07-12, 1999, Nashville, Tennessee, pp 225.

[Stelting02]
Stelting, Stephen. Maassen, Olav. Applied Java Patterns. Published by Sun Microsystems Press A Prentice Hall Title. 2002.

[Tripathi02]
Tripathi, Anand. "Challenges Designing Next-Generation Middleware Systems". Communications of the ACM, June 2002, Volume 25, No. 6, pp 39-42.

[Venkatasubramanian01]
Venkatasubramanian, Nalini. Deshpande, Mayur. Mohapatra, Shivjit. Sebastian, Gutierrez-Nolasco, Wickramasuriya, Jehan. "Design and Implementation of a Composable Reflective Middleware Framework". Proceedings of the 21st International Conference on Distributed Computing Systems, April 16-19, 2001, Mesa, Arizona, pp 644.

[Venkatasubramanian02]
Venkatasubramanian, Nalini. "Safe Composability of Middleware Services". Communications of the ACM, June2002, Vol 45, No. 6, pp 49-52.

.NET Remoting, Tutorial, Microsoft 2002.
See http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/hawkremoting.asp

.NET Remoting, Tutorial, 2002.
See http://www.dotnetremoting.cc/

APPENDIX A

Adaptable Middleware Pattern

Pattern Properties

Type: Behavioral

Level: Component/Architectural

Purpose

To introduce an abstraction layer that decouples the application from the middleware being used.

Introduction

Let's assume we have a distributed system. We would then probably decide to go with some form of middleware between applications. We might then later decide to switch middlewares. We want to limit the changes necessary to switch between them. We would also like to limit any other efforts such as testing, compiling, and deploying already completed systems.

Applicability

This pattern is very useful when distributed systems are using some sort of middleware. It is also applicable when the two communicating applications are built under different platforms.

Description

This pattern is broken up into two parts. It involves separating the application logic from the Application Programming Interface (API) and separating the data interpretation from the middleware.

To separate the API we define an interface between the target and host application. On both sides we build the business logic to bind to these interfaces. This implies that once the logic is built and tested as long as the interfaces don't change this existing logic also doesn't need to be changed either.

Now to separate the data from the communication medium we define a way to have our applications actually interpret the data independently of the transport. Providing meta-data information within our data messages does this. We will only allow one type of message to be passed across the middleware and that is a character string. This interface will adapt to any middleware of choice.

Implementation

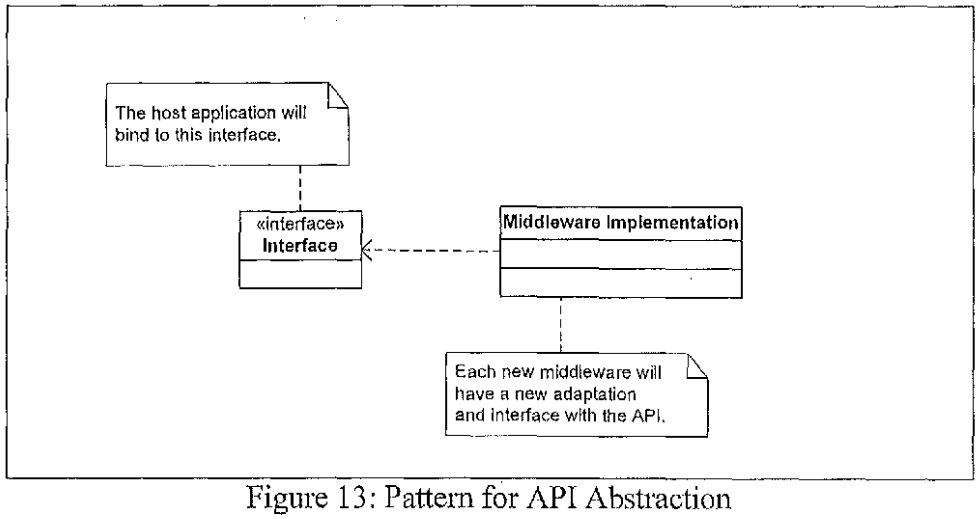

Figure 13: Pattern for API Abstraction

As shown in Figure 13, each service will have an interface defined, and each version of middleware will implement the interface, providing the middleware integration now decoupled from the application.

Secondly, the messaging infrastructure will be defined by passing a character string as the in parameter and returning a character string as the output. This way we can pass XML messages and the interpretation of the messages will be done with our application independent of the middleware.

Figure 14 shows where the interpretation of messages can take place. If the interpretation is done independent of the middleware then there is no need to re-do any mapping or defining of the types with the new middleware.

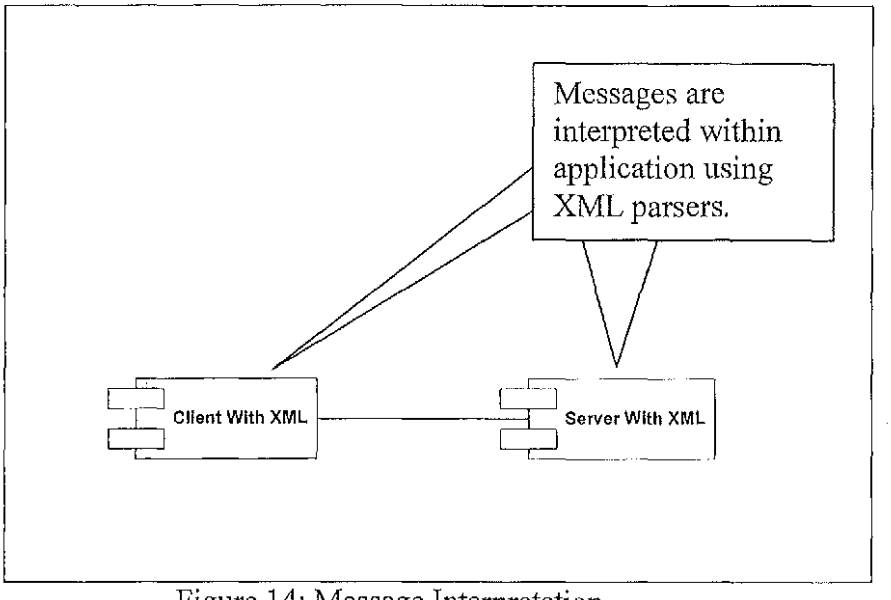

Figure 14: Message Interpretation

Benefits and Drawbacks

This will significantly reduce the overhead of switching to a new middleware infrastructure. This will also provide a way in which the application logic doesn't have to be retested and deployed. Only the code integrated the new API will have to be written and tested. Thirdly, this messaging infrastructure provides for a more extensible framework. The only drawback might be a loss of flexibility with respect to the services that specific middlewares might provide.

APPENDIX B

Source Code

Attached to this document is the entire source code of this demonstration application on a CD. It is a .NET solution with multiple projects containing all C# code.

VITA

Jason Mitchell has a Bachelor of Science degree from University of North Florida in Computer Science, 2000. Jason expects to receive his Master of Science in Computer and Information Sciences from the University of North Florida, May 2003. Dr. Arturo Sanchez of the University of North Florida is serving as Jason's project advisor. Jason is currently employed as a systems engineer at TNT Logistics and has been with the company for one year. Prior to that, Jason worked for ECI Telecom as a software engineer for two years.

Jason has interests in software engineering, project management, and distributed systems. Jason has extensive experience in the J2EE and .NET platform frameworks in both the presentation and business tiers. Jason has also done extensive relational data modeling. Jason has been married for 9 months.