

Protocol Independent Programming Using Openwings Connector Services

by

Wade Wassenberg, Software Engineer, Motorola ISD

ABSTRACT - *The importance of distributed computing in today's computer systems continues to grow. Many advancements in Software Engineering have made software quicker to produce and easier to reuse. Unfortunately, these advancements have not been widely applied to inter-process communication. That is to say, until now. Protocol Independent Programming provides the means by which parallel and distributed software can be more quickly produced and much easier to reuse. This paper will explore Protocol Independent Programming and discuss one Protocol Independence Framework—Openwings Connector Services.*

KEY WORDS - *Protocol Independent Programming (PIP), Protocol Independence Framework (PIF), Protocol Independent Design (PID), Service-Oriented Programming (SOP), Openwings Connector Services, Distributed, Parallel, Inter-Process Communications Protocol (ICP), Reuse.*

INTRODUCTION

As time has passed and Software Engineering has matured, we have seen such paradigm shifts as adopting formal programming languages over assembly languages and abandoning functional languages for object-oriented languages. As a result of these advancements, we have been able to more quickly produce software that is much easier to reuse. Now, Software Engineering sits on the verge of yet another paradigm shift. This new advancement is known as Service-Oriented Programming (SOP) [1]. With SOP comes the demand for quicker and easier solutions for parallel and distributed programming.

Currently, most parallel and distributed software encapsulates some or all of the details of the inter-process communication protocol (ICP) that it uses. This forces the need for redesign, reprogramming, recompilation, and retesting if the ICP ever changes. At the same time, the non-ICP portions of the software cannot be reused or redeployed in other systems that do not use the dependency ICP. This is in direct opposition to the current reuse model provided by Object-Oriented

Programming. In addition, software that is dependent on an ICP is typically more complex and takes longer to write. Software Engineers have even gone as far as to alter their design in order for a particular ICP to work with their system.

These are some of the most important problems that exist in parallel and distributed software today. These problems can be solved by using Protocol Independent Programming (PIP). PIP is the process of writing code that is independent of any ICPs.

This paper will discuss PIP as the solution to the problems that currently exist in inter-process communication. It will also discuss Openwings Connector Services, which is one Protocol Independence Framework (PIF) that can be used to achieve the goals of PIP.

BENEFITS OF PIP

PIP offers a simple solution to many of the problems that exist in inter-process communication. Among the benefits of PIP are:

The Benefits of PIP

- Better designs
- Quicker production
- Greater reusability

Protocol Independent Design (PID) is the process of designing code to be independent of inter-process communications. Software Engineers that use a PID are less likely to alter their design in order for a particular ICP to work. In fact, there is no need to alter the design for a particular ICP because the design is protocol independent! Because no compromises have to be made, the design is completely focused on solving the particular problem. This, of course, yields a better design.

Learning or creating new protocols and incorporating them into a design, takes time. Taking an existing protocol and molding it to work with a specific problem

also takes time. PID allows the Software Engineer to ignore the details of the IPC and simply write the code that solves the problem. Because there is no time spent incorporating a protocol, production is quicker. Reusability is the most important benefit of PIP. Because PIP produces code that is independent of inter-process communication, the code can be reused or redeployed in any environment, using any ICP. Not only that, reuse or redeployment can be achieved without the costs of redesign, reprogramming, recompilation, or retesting. PIP extends the benefits of Object-Oriented Programming (OOP) to parallel and distributed software.

PROTOCOL INDEPENDENCE FRAMEWORKS

A Protocol Independence Framework (PIF) is a set of APIs that can be used to standardize and automate PIP. The APIs of a PIF contain interfaces and abstract classes that provide a protocol abstraction layer. In application source code, these interfaces and abstract classes are used in place of code specific to the ICP:



Figure 1. PIF Abstractions

The interfaces and abstract classes of the PIF are implemented and extended by protocol specific classes:



Figure 2. Protocol Specific PIF Implementations

These protocol specific classes are either implemented by the developer or obtained from a vendor. Typically, protocol vendors will provide implementations of the PIF API for their specific protocol products. In the instance that a vendor will not provide an implementation for a protocol, the developer may opt to implement the PIF APIs for that protocol. This should only have to be done once; after which the protocol can be reused by any application that is implemented to the PIF APIs:

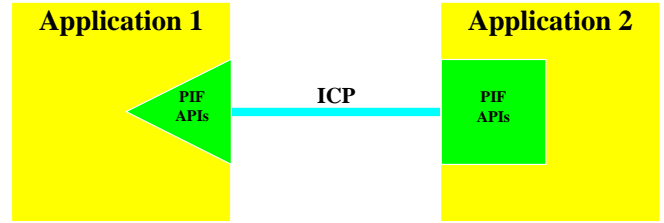


Figure 3. Applications Using PIF Implementations

The end result of using a PIF is protocol independent applications that lend themselves to better design, quicker production, and greater reusability.

OPENWINGS [2] CONNECTOR SERVICES: ONE PROTOCOL INDEPENDENCE FRAMEWORK

Openwings Connector Services is one example of a PIF. Connector Services is an interface based API that provides a solid foundation for building protocol independent applications. As mentioned earlier, a PIF must provide some form of protocol abstraction layer. In Connector Services the protocol abstraction consists of three parts: a set of user-defined protocol independent interfaces, a set of API-defined protocol independent interfaces, and protocol specific implementations of these two sets of interfaces. The user-defined interfaces are known as Contracts:

Pattern Name: Contracts[1]
Problem: How can behaviors be defined independent from implementations?
Context: Anywhere it is desirable to hide implementation details.
Solution: The concept of an interface construct was added to Java to describe a behavior both in syntax and semantics. The methods, method types, method parameter types, and field types prescribe the interface syntax. The comments, method names, and field names describe the semantics of the interface. The “contract” is defined by the combination of these syntax and semantics. In this model objects that implement the interface agree to the “contract” prescribed by that interface. These interfaces can use inheritance, including multiple inheritance.

In Connector Services, the Contract is used to hide inter-process communication details. Also, because Contracts are user-defined, they provide the Software Engineer with the flexibility to give the methods of the Contract interface meaningful names. A PIF does not have to provide this Contract mechanism; rather, this was a design decision chosen for Connector Services.

Because Contracts are user-defined, there still needs to be some sort of standardized piece to the framework. In Connector Services, this piece of the framework is the API. The Connector Services API defines several standardized protocol independent interfaces that are used in conjunction with Contracts to achieve meaningful but standardized abstractions from the inter-process communication:



Figure 6. Connector Diagram

In fact, this is where Connector Services gets its name.



Figure 4. Contract and API Abstractions

The interfaces of the Contract and the API are used to implement an object for a specific protocol. This implementation is known as a Proxy:

Pattern Name: Proxy[1]
Problem: How can a contract and a programmatic interface be delivered in a mobile fashion?
Context: Proxies can be used behind any interface to add functionality to an object.
Solution: Proxies can provide an object that implements a contract or take an object that implements a contract. Proxies are the primitives used to create connectors.

Pattern Name: Connector[1]
Problem: How can one protocol independent application communicate with another protocol independent application?
Context: Anywhere protocol independence is used.
Solution: Connectors provide an abstraction for protocol independence. Connectors are composed of two proxies: a Sender Proxy and a Receiver Proxy. The Sender Proxy provides an object that implements a contract, and the Receiver Proxy takes an object that implements the same contract. Connectors can naturally be chained.

One of the important things to remember about Connectors is that they are interchangeable. For instance, a Connector written for Contract A and Protocol X can be interchanged with a Connector written for Contract A and Protocol Y. This is because both Connectors implement Contract A and the Connector Services API:

When using Connector Services, there are two types of Proxies: a Sender Proxy and a Receiver Proxy. A Sender Proxy must implement both the Contract and the SenderProxy interface defined by the Connector Services API. The implementations of the Contract methods in the Sender Proxy translate those method calls into protocol specific transmissions. A Receiver Proxy must implement the ReceiverProxy interface defined by the Connector Services API, but it may or may not implement the Contract interface. The Receiver Proxy receives the protocol specific transmissions from the Sender Proxy and translates them back into Contract-specific method calls:

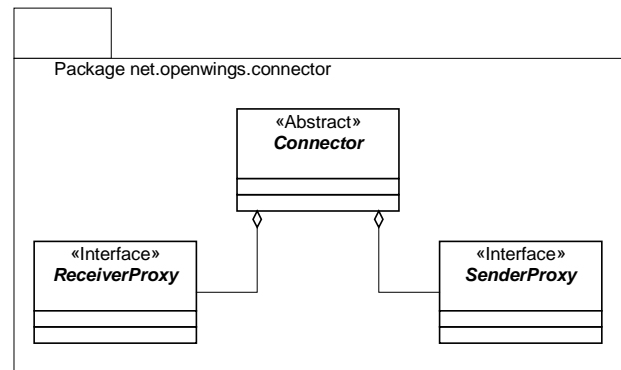


Figure 7. Connector Class Diagram

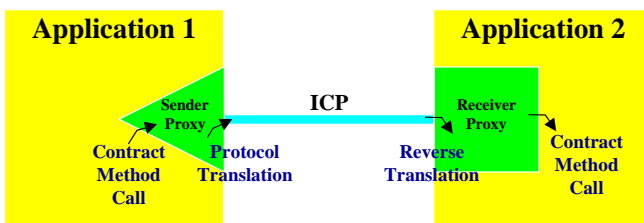


Figure 5. Protocol Transmission Sequence

When combined, the Sender Proxy and Receiver Proxy are known as a Connector:

FLAVORS OF CONNECTORS

The Connector Services API provides interfaces for the two flavors of connectors: Synchronous and Asynchronous. The Synchronous Connector API is used to implement Connectors for synchronous protocols such as RMI[4].

CREATING AND BUILDING CONNECTORS

Connector Services provides the framework to build protocol independent applications. However, a Connector still must be written for every Contract/Protocol combination the application wishes to use. How does this save time? Connector Services not only provides a Connector API, but it also provides a Connector Generation API. There are currently three ways to create a connector:

Methods For Creating Connectors

- Manual Generation
- Assisted Generation
- Full Generation

The Generation API of Connector Services provides interfaces and abstract classes that can be used to create tools that do assisted generation and/or full generation. Typically, a protocol vendor will provide a full generator for its specific protocol products. A full generator will generate a complete usable Connector for a given Contract interface. This is typically done at a command-line and requires no additional developer programming.

In some cases, full generation is infeasible. This can happen for many different reasons. In these cases, the developer is forced to use assisted generation or manual generation. In some cases, it is possible for a generator to generate the shells for the connector, and the developer is then forced to complete the shell implementations. This may be the case when a protocol requires many inputs to define its behavior. In other cases, it is possible for a generator to generate one proxy and not the other:

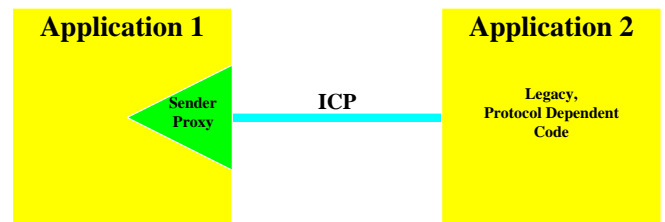


Figure 10. Assisted Generation for Legacy Applications

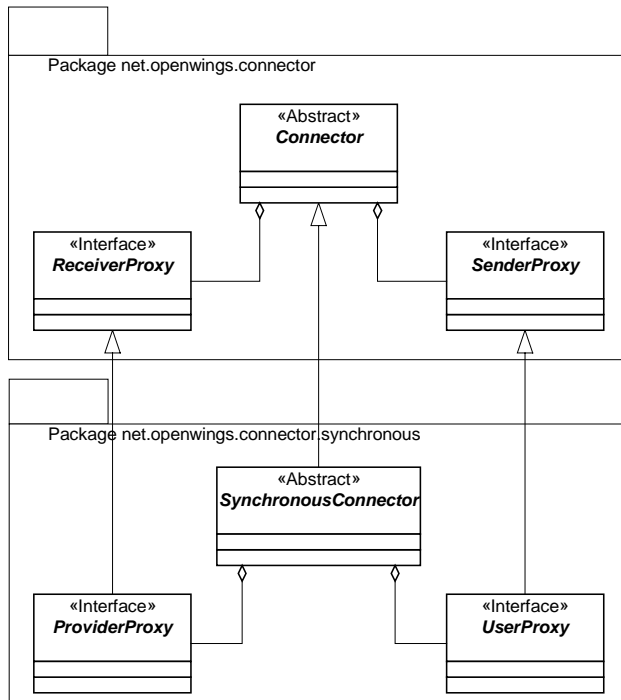


Figure 8. Synchronous Class Diagram

The Synchronous Connector API is used to implement Connectors for synchronous protocols such as RMI[4].

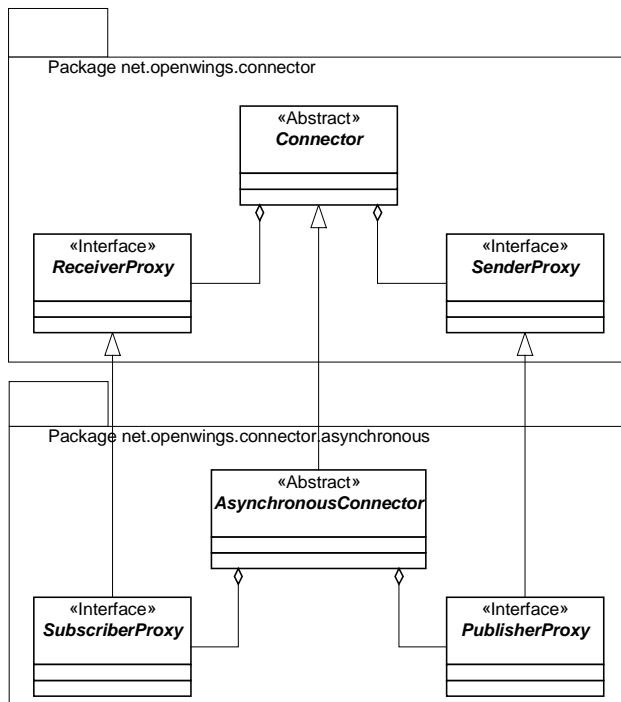


Figure 9. Asynchronous Class Diagram

OTHER ADVANTAGES OF CONNECTORS

The interchangeability of connectors provides several other advantages. These advantages include:

The Advantages Of Interchangeability

- Protocol upgradeability
- Security interchangeability
- Compression interchangeability
- Hardware interchangeability
- Protocol-to-Protocol translation

In order to stay competitive, protocol vendors continue to upgrade their products. At the same time, developers would like to use those upgrades without having to rewrite their applications. Well, because the details of the protocol are hidden inside the Connector, a new Connector, that contains the upgrades, can be swapped with the old one. No changes to the application are necessary.

Some ICPs provide better security than others. Some are encrypted while others are wide open. In some cases applications are deployed in environments where security is not important, while in other cases security is important. Here, a secure Connector can be deployed with the application in the secure environment, and a non-secure Connector can be deployed in the non-secure environment.

Sometimes it is desirable, for bandwidth purposes, to compress transmitted data. Some protocols may provide compression where others may not. Again, the best Connector for the job can be deployed with the application without modification to the application.

Sometimes applications wish to communicate on non-standard hardware. Some examples include UART Serial ports, USB, and Fire Wire. In these cases, a Connector can be written to hide the details of these hardware protocols, and the application never knows!

In some rare instances, it may be necessary for one application to use one protocol and another application to use a different protocol. This is sometimes the case when an application on a PC wishes to communicate with an application on a handheld device. Here, a Connector that knows how to translate between two protocols could be deployed.

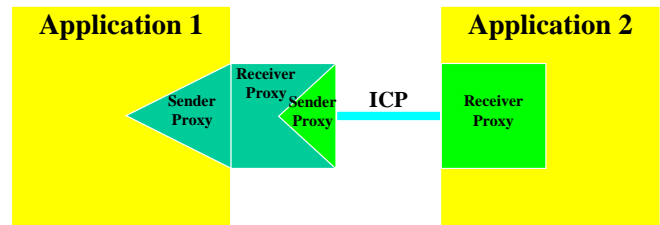


Figure 11. Protocol Translation

These are just a few of the instances where Connector interchangeability is advantageous.

SUMMARY

Protocol Independent Programming yields applications that are designed better, quicker to produce, and much easier to reuse. If we, as an industry, adopt a Protocol Independence Framework as a standard, our applications will be able to communicate with each other much more freely. As a Protocol Independence Framework, Connector Services provides a good foundation for building protocol independent applications. Part of the abstraction provided by Connector Services is user-defined. This gives the developer the flexibility to write meaningful Contract interfaces that increase code readability. Connectors can typically be generated for a given Contract interface requiring no additional programming from the developer. Also, Connectors are interchangeable, and this adds several other advantages to using Connector Services.

Protocol Independence is the Object-Oriented equivalent to inter-process communication. It saves time and increases reusability. These are the two factors that have influenced the path that Software Engineering has taken throughout its existence. Now these factors are being applied to other, still infantile areas of Software Engineering. Protocol Independent Programming is just one of those applications that hopes to revolutionize the way we engineer software.

REFERENCES

- [1] Introduction to Service-Oriented Programming, <http://www.openwings.org/download/specs/ServiceOrientedIntroduction.pdf>
- [2] Openwings, <http://www.openwings.org>
- [3] Architecture Description Language (ADL), http://www.cs.cmu.edu/~acme/acme_documentation.html
- [4] Remote Method Invocation (RMI), <http://java.sun.com/products/jdk/rmi/index.html>