

# Middleware Reliability Implementations and Connector Wrappers

J.H. Sowell and R.E.K. Stirewalt  
Department of Computer Science and Engineering  
Michigan State University  
East Lansing, Michigan 48824 USA  
Email: {sowellje,stire}@cse.msu.edu

## Abstract

Recently, Spitznagel and Garlan introduced connector wrappers, which cleanly specify reliability policies in a form that can be incorporated into existing architectural specifications. We recently implemented a middleware framework, called *Theseus*, that supports asynchronous communication among distributed objects under a variety of different reliability policies. Both connector wrappers and *Theseus* separate reliability concerns from developers of system components, and both allow different policy decisions to be woven into an existing architecture. This paper illustrates the synergistic relationship between these two complimentary approaches and discusses how we are using connector wrappers to identify new (as yet undiscovered) reliability policies to add to *Theseus*.

## 1 Introduction

Recently, Spitznagel and Garlan demonstrated the use of *connector wrappers* to cleanly separate common reliability policies, such as retry and failover, from an architectural specification. Connector wrappers build on a formal model of architectural connection, in which a connector specifies the coordination of actions among multiple architectural components, each of which is said to play a *role* in the interaction [1]. In this formalization, a *role specification* abstracts the behavior a component that can legally play the role, and a separate *glue specification* describes how the behaviors of two or more roles are coordinated. Building on these definitions, a connector wrapper introduces new behaviors by intercepting and potentially modifying the communication among the various roles. More precisely, a connector wrapper extends the original glue specification to produce a new glue specification that coordinates the existing roles and may even introduce new roles to support any new behavior introduced by the wrapper.

Middleware technologies, such as [8, 7, 14], are now being used as the basis for a large number of distributed architectures, and many of these technologies nicely reify the role–glue distinction. Thus, it seems natural to implement connector wrappers “in the middleware”, thereby allowing different wrappers to be added or removed without requiring changes to any of the components that are being coordinated by the connector. Our research is investigating how to design middleware frameworks that enable the transparent introduction of various reliability policy implementations. This work explores the synergies between policies that can be specified using connector wrappers and implementations that can be transparently incorporated into a suitable middleware framework.

Our work, *Theseus*, is a Java-based framework for constructing customized middleware configurations that support reliable, asynchronous communication among distributed objects. To support customization, we designed *Theseus* using ideas from role-based design [11] and feature composition in the GenVoca style [2]. Among other benefits, our design cleanly separates the implementation of coordination logic (i.e., the glue) from the implementation of the various proxies and services that represent the different roles. Consequently, it is easy to interpose new logic “in between” the roles and the glue.<sup>1</sup> In fact, we have been able to introduce a variety of reliability policies—such as retry, failover, and multiple failover—transparently, i.e., without affecting the implementations of the components that play the various roles.

To investigate the synergy between connector wrappers and middleware-based policy implementations, we first formalized the core functionality of *Theseus* services as a connector in the sense of [1]. This core functionality can be described by a connector that we call *request–reply*, which comprises three roles—a *client*, a *server*, and a *reply han-*

---

<sup>1</sup>This ability to interpose new logic is comparable to that provided by Schmidt et. al.’s *Interceptor* design pattern [12], and CORBA’s portable interceptors [9, 3].

abler. Components that play these roles are coordinated such that a client sends an asynchronous request for service to the server, which later notifies the reply handler once the service has been fulfilled.<sup>2</sup> This most basic functionality does not support reliability. Rather, reliability policies are implemented as reusable *collaborations*, which can be mixed in with the implementation of the Theseus core, thereby extending the core with support for reliable services. These reliability implementations resemble the connector wrappers described in [13], and may be composed in the same manner.

We envision two major benefits from a more thorough understanding of the synergy between connector wrappers and middleware policy implementations. First, if general connector wrappers can be implemented in the middleware, then system architects could adorn architectural specifications with connector wrappers, which can then be transparently folded into the underlying middleware platform. A second benefit is the insight into the Theseus architecture afforded by formalization. Specifically, after formalizing Theseus' core functionality and reliability policies in terms of formal connectors and connector-wrappers, we discovered that none of our existing reliability policies exploited the reply handler role. This observation led us to question whether there might also exist more powerful reliability policies that use all three roles.

To begin to validate this observed synergy, we implemented all the dependability policies described in [13]. We then synthesized different middleware implementations that support various compositions of these policy implementations. For example, the composition described in Section 2.3 comprises both bounded retry and failover; upon error the resultant composition first attempts  $n$  retries before failover to a known backup. In the next section we present elided implementation details of our reliability policy implementations and the correlations with their architectural counterparts. Section 3 discusses the initial benefits of this analysis and ongoing work.

## 2 Wrapper Implementation

To effectively discuss our policy implementations, we first introduce some basic Theseus concepts. We then outline our implementations of the reliability policies described in [13] and an example composition.

---

<sup>2</sup>In some architectures, the client and reply handler roles might be played by the same component.

## 2.1 Theseus

### 2.1.1 Request–Reply

Theseus' asynchronous communication is based on a *request–reply* protocol, similar to that used in network systems [15]. This protocol coordinates three different kinds of components—a *client*, a *server*, and a *reply handler*. A server is a component that provides some set of services that can be requested by one or more service clients. Service requests are asynchronous, which means the client does not wait for the server to fulfill service requests. Rather, when the server completes the requested service, it notifies the reply handler, possibly passing return data in the notification.

For each service in the interface of a server, the last parameter is a reference to a reply handler, whose operations, hereafter called *service replies*, represent notifications of completion of service invocations. The parameters of a service reply carry any data that may need to be returned by the service. By convention, a reply handler exists in the client's address space, but the reply handler may be distinct from the client.

Returning to our parallels between implementation and architecture, the request–reply protocol can be modeled formally by the glue of a Theseus-based connector because it dictates how clients, servers, and reply handlers interact. Supplementing this glue to support additional behaviors, namely reliability policies, requires intercepting messages among the objects that collaborate to implement the request–reply protocol. We now describe how objects in Theseus are wrapped to intercept (and then attempt to recover from) failure messages.

### 2.1.2 Theseus Objects

As in many middleware systems, a remote object, in our case the server, is a distributed active object that comprises a local (relative to the client) stub and a remote skeleton [12]. The *stub* handles the client side of an asynchronous service invocation, which entails marshaling operation invocations into requests and initiating request transport. The *skeleton* receives these marshaled requests and unmarshals them into invocations of services on the server. To request a service from a server, the client stub maintains a *servant*, which is a proxy<sup>3</sup> for the (remote) skeleton. A client requests a remote service by invoking an operation on the stub, which then constructs a service request and dispatches it to the servant. The servant is responsible for transporting service requests to the (remote) skeleton.

In terms of role and glue specifications, dispatching a request to the servant can be modeled as an event that synchronizes two actions, one from the client role (where the

---

<sup>3</sup>All proxies referenced here are in the sense of the proxy pattern as defined in [6].

client role is played by the stub) and one by the glue (where the servant implements one of the actions in the glue specification). To implement our reliability policies, we decouple these actions so that they correspond to distinct events, between which we can interpose additional logic to recover from failures. Our implementation uses a servant wrapper to: (1) intercept requests as they are dispatched to the servant, and (2) intercept exceptions that are raised by the servant so as to mask communication failures from the client component. We now describe how this form of interception allows us to implement different reliability policies.

### 2.1.3 Interception and Strategy Implementation

To mask servant errors from the client, we introduce a *servant wrapper* that can intercept servant failures and delegate them to special reliability objects that are capable of handling them. As its name suggests, the servant wrapper is a proxy for the servant. Thus, the connection between the stub and the servant must be reconfigured to dispatch requests to the wrapper rather than the servant itself, thereby allowing the wrapper to intercept failure messages and delegate them appropriately. The servant wrapper is parameterized by two objects—the original servant and a recovery agent. The *recovery agent* implements logic for handling and correcting servant failures. Recovery entails reconnection to the original server or a suitable backup such as replay of requests that may have been lost during the failure. The recovery agent interface is:

```
public interface RecoveryAgent
{ public boolean recover (); }
```

The constructors of implementations of `RecoveryAgent`, in turn, are parameterized with the information necessary for servant recovery (abstracted as servant information) and a *failure policy*, which implements actions to be taken if the existing servant cannot be recovered. Our failure policy implementations are only concerned with out-of-process failures such as network failure and server crashes. The failure policy interface is:

```
public interface FailurePolicy
{ public boolean handleFailure (); }
```

Thus, when the servant wrapper detects a servant failure, it invokes its `RecoveryAgent`'s implementation of `recover`, indicating the servant has failed and recovery should be attempted. If recovery succeeds, `recover` returns true. If recovery fails, the recovery agent consults its instance of `FailurePolicy` `f` by invoking `f.handleFailure()`. If the failure policy can handle the failure, it returns true, which is propagated on by the recovery agent, allowing the servant wrapper to continue fielding requests as they are dispatched to it. If the failure policy cannot handle the failure, it returns false, which is propagated on to the servant wrapper by the recovery agent,

ultimately resulting in a failure propagated all the way up to the client. Notice that the use of implementation-level wrappers enables the interception that is necessary to introduce reliability policy implementations into the middleware (rather than in client-component code).

## 2.2 Policy Implementations

**Indefinite Retry** Indefinite retry is the least complex of the reliability policies. The connector wrapper for indefinite retry repeatedly attempts an invocation until the call is successful. As such, failures are never propagated beyond the connector wrapper and recovery may continue forever. Indefinite retry assumes either that the server will be available again some time in the future or recovery will be terminated by an outside entity at some time in the future.

In Theseus, indefinite retry is implemented by a recovery agent called `IndefiniteRetry`, which is parameterized with only the information necessary to recover the servant. As recovery is attempted indefinitely, no failure policy is necessary. Pseudocode for `IndefiniteRetry`'s implementation of `recover` follows

```
public boolean recover () {
    while ( true ) {
        if ( servant recovered )
            return true;
    } }
```

As such, recovery is an infinite loop that only returns upon successful servant recovery.

**Bounded Retry** Bounded retry is only slightly more complex than indefinite retry. Rather than attempting to recover indefinitely, the connector wrapper specifying bounded retry prescribes attempting an invocation a finite number of times (`maxRetries`) before giving up and reporting the failure.

The Theseus implementation, `BoundedRetry` is an implementation of `RecoveryAgent` and is parameterized by three things: `ServantInfo` `si`, `maxRetries`, `period`, and a `FailurePolicy`. The pseudocode for `BoundedRetry`'s recovery follows, where `f` is an instance of `FailurePolicy`

```
public boolean recover () { }
    for ( int i=0; i<maxRetries; i++ ) {
        if ( servant recovered )
            return true;
        sleep ( period ); }
    return f.handleFailure (); }
```

`BoundedRetry` attempts recovery `maxRetries` times, sleeping `period` milliseconds between each attempt. If recovery is not successful, `handleFailure` is invoked. If `handleFailure` is successful, it returns true, which is

propagated up to the recovery handler, and on to the servant wrapper, which may then continue processing requests. If `handleFailure` also fails, failure is propagated up to the recovery agent, and from there on to the stub.

**Immediate Failover** Unlike the previous two policies, immediate failover does not attempt any retries. Instead, upon detecting a failure, communication is forwarded to a known, functionally equivalent backup.

In Theseus, this is implemented by replacing the servant wrapper’s original target (the current servant) with a servant that corresponds to a backup server. To this end, the recovery agent is parameterized by one object, an implementation of `FailurePolicy`, `FailoverPolicy`. `FailoverPolicy` is parameterized by `ServantInfo bsi`, the information necessary to construct a servant that represents the backup server, and the servant wrapper `ServantWrapper sw`, which implements `replaceServant ( Servant s )` for replacing the currently wrapped servant with the new servant `s`. The corresponding pseudocode is

```
public boolean recover ()
{ f.handleFailure (); }
public boolean handleFailure () {
  if ( backup construction successful )
  { sw.replaceServant ( backupServant );
    return true;
  } else
  { return false; } }
```

This implementation of `recover` immediately delegates failure handling to `FailoverPolicy f`. `FailoverPolicy`’s implementation of `handleFailure` first attempts to construct the backup servant `backupServant`. If successful, the existing servant is replaced in the servant wrapper by `backupServant` via `replaceServant`. Because service requests are dispatched to the servant wrapper, this replacement achieves the specified forwarding of messages to the backup. If construction of the backup servant fails, `handleFailure` returns false and the failure is propagated as before.

### 2.3 Example Composition: Bounded Retry and Failover

Given these policy implementations, our next step is to validate their composability. The example presented here is a bounded retry that resorts to failover if the existing servant can not be recovered. At the architectural level, this is achieved by applying the bounded retry connector wrapper and the failover wrapper. In Theseus, this is implemented by constructing a `RecoveryAgent` that delegates failures to an implementation of `FailoverPolicy`, as illustrated below

```
RecoveryAgent ra =
```

```
new BoundedRetry
( ServantInfo si,
  int maxRetries,
  int period,
  new FailoverPolicy
  ( ServantWrapper sw,
    ServantInfo bsi ) );
```

Above, we see the declaration of a `RecoveryAgent`, implemented as a `BoundedRetry`. `BoundedRetry` is parameterized by the servant info of the existing servant (`ServantInfo si`), integers `maxRetries` and `period`, and an instance of `FailurePolicy` implemented by `FailoverPolicy`. The new instance of `FailoverPolicy` is parameterized by the servant wrapper (`ServantWrapper sw`) and `ServantInfo bsi`, from which a backup servant may be constructed.

The resultant composition thus fulfills the specification by attempting `maxRetries` in `period` millisecond intervals and, if servant recovery is not possible, the composition replaces the existing servant with a backup, failing only if the backup is not available. This composition also illustrates how parameterization and delegation are used in Theseus’ implementations of these policies to introduce new reliability functionality.

## 3 Conclusions and Ongoing Work

Our initial results suggest a useful synergy between the policies described in [13] and our policy implementations. Moreover, formalizing Theseus as a connector provided unforeseen insights into additional connector wrappers that may lead to the development of new reliability policies. Specifically, we recognized additional interactions between the server and the reply handler, for which we had yet to develop reliability policies.

When considering the interaction between the server and the reply handler, we discovered several potential failure scenarios and recovery policies. For example, in the event of network failure, a connector wrapper may be applied to maintain a cache of replies that have been sent to the reply handler. Using this cache, recovery may entail simply re-sending cached replies rather than more expensive operations such as rollback or re-processing of requests. In terms of architectural specifications, this new connector wrapper would wrap the glue that coordinates the interaction between the server and the reply handler, adding behaviors such as negotiating the replies that need be resent. We are currently implementing this wrapper as a new policy implementation in Theseus.

In addition to highlighting additional connector wrappers, formalization afforded insight into how to implement an interesting semantic model of active-object concurrency, called *wait-by-necessity* [4]. In this model, active objects

asynchronously invoke the methods of other active objects and continue processing while these methods execute. Because methods may return values, the caller must somehow synchronize with the callee so that return values are not used until they are available. Wait-by-necessity semantics dictate that caller threads automatically block when accessing an object that is returned by an asynchronous operation that has yet to complete. In this model, return values are placed in so-called *future objects*, which will block any attempts to access any of their values before they are defined (i.e., before they are loaded with the value(s) returned by an asynchronous method invocation). Upon completion of the asynchronous operation, data is placed in the future and all threads blocked on that future are notified. All subsequent future accesses proceed as with any other object access. Futures are easily implemented in Theseus as reply handlers. However, because futures block client threads, the model is inherently sensitive to server-reply handler communication failures. We have also successfully applied the same reliability policy implementations we used for request-reply to our implementation of futures to provide reliable wait-by-necessity.

Two closely related pieces of work are Parlavantza et al.'s work on extensible binding types [10] and the FRIENDS [5] approach. One of the major differences between Theseus and FRIENDS is the use of reflection to introduce dependability and security. Theseus' design identifies message interception points and allows logic to be introduced by inserting wrappers to accommodate new behaviors. One point of comparison is whether the interception points in FRIENDS and Theseus allow for the same behaviors to be introduced. To determine this we identified the types of interception possible using the FRIENDS' `MetaObj` interface. Theseus provides a subset of these interception points and we believe the remaining interception points can be introduced as necessary.

Parlavantza's work employs component-based composition to construct a variety of middleware binding types. They describe binding types as different forms of interactions between distributed objects, such as remote invocation, media streams, group communication, shared data spaces, etc. In this respect, a binding type is similar to a connector. While they address constructing new binding types via composition, they do not directly address whether one could introduce reliability policies into their framework, but we expect they could employ wrapping and interception techniques similar to Theseus.

Ongoing work intends to further investigate connector wrapper implementations geared towards both reliability and other middleware functionalities. Our next step is to implement a generator to automatically construct Theseus middleware implementations based on a set of policies and policy compositions.

**Acknowledgements** Partial support for both authors provided by the Office of Naval Research grant N00014-01-1-0744. Partial support for the second author provided by NSF grants EIA-0000433 and CCR-9901017.

## References

- [1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering Methodology*, 6(3):213–249, 1997.
- [2] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering Methodology*, 1(4):355–398, 1992.
- [3] C. Marchetti, L. Verde, and R. Baldoni. CORBA Request Portable Interceptors: A Performance Analysis. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA 2001)*, pages 208–217, September 2001.
- [4] F. Baude and D. Caromel and D. Sagnol. Distributed Objects for Parallel Numerical Application. In *Mathematical Modelling and Numerical Analysis Modélisation 36(5)*, 2002.
- [5] J.-C. Fabre and T. Perennou. A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach. *IEEE Transactions on Computers, Special Issue on Dependability of Computing Systems*, 41(1):78–95, 1998.
- [6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [7] Microsoft® .net. <http://www.microsoft.com/net/>.
- [8] *Common Object Request Broker Architecture: Core Specification*. Object Management Group, December 2002.
- [9] *Common Object Request Broker Architecture: Core Specification*, chapter Portable Interceptors, pages 21–1–21–64. Object Management Group, 2002. <http://www.omg.org/technology/documents/formal/corba\2.htm>.
- [10] N. Parlavantzas, G. Coulson, and G. Blair. An Extensible Binding Framework for Component-Based Middleware. In *Proceedings of EDOC 2003*, Brisbane, Australia, September 2003.
- [11] T. Reenskaug, P. Wold, and O. Lehne. *Working with Objects: The OORAM Software Engineering Method*. Manning/Prentice Hall, 1996.
- [12] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. John-Wiley & Sons, 2000.
- [13] B. Spitznagel and D. Garlan. A Compositional Formalization of Connector Wrappers. In *Proceedings of the 2003 International Conference on Software Engineering*, Portland, Oregon, USA, May 2003.
- [14] Package java.rmi. <http://java.sun.com/j2se/1.4.2/docs/api/java/rmi/package-summary.html>.
- [15] A. S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, 1996.