A Feature-Oriented Alternative to Implementing Reliability Connector Wrappers

J.H. Sowell and R.E.K. Stirewalt

Michigan State University, East Lansing, Michigan 48824 USA,
{sowellje,stire}cse.msu.edu

Abstract. Connectors and connector wrappers explicitly specify the protocol of interaction among components and afford the reusable application of extra-functional behaviors, such as reliability policies. Ideally, these specifications can be used for more than just modeling and analysis. We are investigating how to use them in the design and implementation of the middleware substrate of a distributed system. This paper reports our experience elaborating connectors and connector wrappers as instantiations of a feature-oriented middleware framework called Theseus, which supports the design of asynchronous distributed applications. The results of this case study indicate that the relationship between specification features and implementation-level features is not one-to-one and that some specification features have complex, often subtle, manifestations in Theseus' design. This work reports the lessons learned designing these strategies and suggests techniques for designing middleware frameworks and composition tools that more explicitly reify and expose the features specified by connectors and connector wrappers.

1 Introduction

Increasingly, distributed computing systems are deployed in volatile environments in which network connectivity is sporadic and unreliable. In response, a variety of *reliability policies* have been devised to shield users from the effects of this volatility. For example, *automatic retry* detects when a service request fails and automatically resends that request rather than propagating the exception to the client program. More sophisticated policies, such as *failover*, exploit redundant servers: When a request to one server fails that request is automatically forwarded to another rather than propagating the exception to the client program. In each case, an unreliable service is promoted to one that is reliable without altering the implementation of the unreliable service. Consequently, such policies tend to be incorporated into the middleware layer by *wrapping* an unreliable middleware implementation with code that intercepts service requests and performs the extra functionality.

To understand how to use wrappers to apply and compose reliability policies, Spitznagel and Garlan [1] developed a technique based on a formal behavioral model of architectural connection [2]. These *connector wrappers* impose policies

on an existing connector specification by extending and/or restricting its observable behaviors. Moreover, each connector wrapper has an implementation counterpart that can be applied to incorporate the policy into an existing middleware implementation. These *implementation wrappers* compose with the flexibility of their specification counterparts by treating the underlying middleware as a black box. Unfortunately, the resulting implementations may incur redundancies and inefficiencies that are unacceptable on the resource-constrained devices that are most often exposed to volatile environments.

This paper describes an alternative implementation of connector wrappers as *reusable refinements* rather than black-box wrappers. Reusable refinements appeal to an algebraic model of software composition called AHEAD [3] and are similar to ML functors [4] and mixin layers in C++ [5]. Refinements compose functionally, just like wrappers; however refinement composition is more fine grain and thus affords a tighter integration that enables the reuse and extension of existing abstractions. Designers may thus customize a base middleware with reliability strategies by applying refinements in a manner analogous to the application of connector wrappers, and the resulting configurations will not exhibit the redundancy and inefficiency introduced by implementation wrappers.

Our results exploit the fact that reliability strategies often employ the same design abstractions that are used to implement basic middleware services. For instance, many middleware systems use the *asynchronous completion token* pattern [6] to demultiplex asynchronous operation requests and responses. This pattern is also used to implement a strategy for *warm failover* [7] whereby clients copy outgoing requests to a redundant backup server, which silently serves each request in parallel with the primary. Were such a strategy implemented using a wrapper, the wrapper would require logic for demultiplexing requests and responses with the backup server even though such logic exists in the underlying middleware. Further, some reliability strategies may need to suppress behaviors, such as suppressing the responses sent by a server that is intended to play the role of a silent backup. Wrappers suppress behavior by *masking* the observable effects rather than forestalling the behavior itself. Using refinements, both duplication of functionality and masking of behaviors can be avoided.

We believe that implementing reliability strategies as refinements will enable system architects to easily construct efficient middleware solutions subject to a variety of reliability policies. To validate this claim, we implemented and applied a set of reliability refinements to Theseus, which is an asynchronous middleware implementation that we designed using the AHEAD model. In prior work, we described three reliability refinements and showed that these exhibit the compositional properties as their specification counterparts [8]. We have since implemented other refinements, including one for warm failover that is useful for comparing our approach with black-box wrappers. In the sequel, we describe the design of Theseus and how refinements compose to produce new middleware that incorporates various reliability policies. We then compare our refinement-based implementation of warm failover to the wrapper-based design of [9].

2 Background

By way of background, we first introduce the use of wrappers to enhance reliability in distributed systems and the connector-wrapper formalism, which models the behavior of these wrappers. Wrappers and connector wrappers exhibit a useful structural correspondence between implementation and specification; however wrappers also incur redundancies and inefficiencies that are unacceptable to resource-constrained devices. We contend that reliability enhancements can be implemented so as to exhibit the same correspondence with connector wrappers but without incurring the redundancies and inefficiencies of wrapper-based implementations. The key is to implement the enhancements as reusable refinements under the AHEAD model of composition, which supports the definition of modules whose implementation abstractions are left open for further refinement by other modules under composition. This section concludes with a brief introduction to the AHEAD model.

2.1 Reliability-enhancing Wrappers

Software architects often wish to augment communication among the components in a distributed system to incorporate extra functionality, such as logging, encryption, and even strategies for enhancing reliability in the face of network failures. Augmentation is accomplished by *intercepting* messages that cross the boundary between client components and the communication or middleware layer. Once intercepted, these messages are then either dropped, transformed and then forwarded to their original destination, or routed to another destination. Interception has proved useful for adding reliability to a variety of system calls (C.f., [10, 11]). The technique is now supported directly in modern middleware systems (C.f., CORBA's portable interceptor interface [12, 13]).[1]

This paper is concerned with reliability enhancements that are implemented using wrappers, which serve to both mediate client access to a service as well as augment that service with extra-functionality, as with interception. To preserve the original interface, these wrappers are implemented based on the proxy pattern [15][2]. As an example, consider the addition of logging and data encryption to messages sent by client components to a remote server component. Suppose the class `MiddlewareStub` represents the type of the client-side stub object, such as might be generated from an IDL specification of the server component. The additional functionality is implemented by a hierarchy of wrapper objects that conform to the class model in Figure 1. Commensurate with the wrapper pattern, each class implements a common interface, which in this example is called

[1] A powerful model of interception is captured by the composition-filters object model, which allows designers to deploy filters that intercept and drop, modify, or reroute messages among arbitrary objects in a system [14].

[2] Readers familiar with GoF design patterns will notice *wrapper* is a synonym for the adapter and not the proxy pattern; we use the term *wrapper* for consistency with Spitznagel's wrappers, which also implement the proxy pattern[9].

`MiddlewareStubIface`, and the two wrappers implement their methods by delegation. Software architects typically use wrappers to augment middleware with

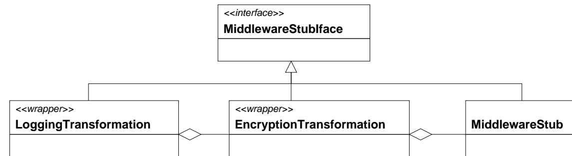

Fig. 1. Adding functionality using wrappers.

new functionality, to suppress existing behaviors, or to mask faults.

While easy to implement, wrappers (and interception techniques generally) suffer from two problems. First, components of the original system (that being wrapped) may be "orphaned" when their behaviors are suppressed in favor of behaviors introduced via interception and wrapping. Such orphans, even though no longer actively contributing to the behavior of the system, remain in place and continue to consume resources, both computational and spatial. Second, the extra functionality may incur redundancies and inefficiencies that are unacceptable on the resource-constrained devices that are most often exposed to volatile environments. In the case of wrappers, these redundancies owe to the treatment of the service being wrapped as a black box whose internal resources (i.e., those that might be reasonably reused by the extra functionality) are not accessible to the wrapper. Interception-based techniques suffer a similar problem.

To overcome this obstacle requires the ability to reconfigure a base system when augmenting it with extra functionality. The earliest work in this regard uses static recomposition based on object-oriented frameworks, the most notable of these being Schmidt's ACE framework [16]. More recent work has investigated the development of software modules that, when composed with a base system, are able to reconfigure that system. Such compositions must be able to refine existing components, including those hidden behind an opaque API, to support extra-functional behaviors or replace them with components that do support the desired behaviors. More recent work (C.f., [17, 18]) applies dynamic recomposition to introduce reliability enhancements. In these cases, reflection and/or meta-object protocols are used to reconfigure the original application to use the components that best fit the environment at hand. Further still, such techniques have also been combined with class loading technologies, in particular the JBoss extensible middleware platform, to completely add and/or remove components from a running system [19].

2.2 Connector Wrappers

The term *connector* refers to a mechanism for composing architectural components [20]. Connectors abstract communication mechanisms and protocols, such

as procedure calls, remote procedure calls, pipes in a pipe-and-filter system, and many of the communication services that are generally referred to as "middleware". More abstractly, a connector represents a pattern of *interaction* among a set of components, called the *roles* of the connector. Allen and Garlan developed a formal model of connectors as stylized CSP specifications, thereby enabling an architect to rigorously specify an architecture as the composition of components and connectors and use formal analysis tools to reason about the resulting behavior [2].

To facilitate reasoning in the presence of wrappers, Spitznagel and Garlan developed a theory of *connector wrappers*, which are stylized CSP specifications designed to compose with a connector specification to yield a new connector, whose behaviors are an extension or restriction of the original [1]. Connector wrappers are useful for specifying different strategies for implementing reliability policies, such as retry and failover, in isolation without regard to the details of a particular connector. The basic idea is to introduce (via parallel composition) additional processes that synchronize with messages at the component–connector interface to precisely model the behavior of the aforementioned wrapper modules. Because connector wrappers faithfully model the structure and behavior of wrappers, an architect may specify a new connector as the composition of one or more connector wrappers with some connector and then semi-automatically generate a conforming implementation. Spitznagel's system provides generation tools that implement this capability.

Unfortunately, connector wrappers cannot yet utilize reconfiguration-based mechanisms, which generally do not adhere to an algebraic model of composition. The next section describes a model of composition that is algebraic in nature and that supports fine-grained recomposition of the base system, allowing for both reuse of abstractions and the avoidance of orphaned components.

2.3 AHEAD

AHEAD is an algebraic model of software composition in which complex, feature-rich programs are synthesized from base programs by applying reusable refinements [21]. Here, a base program is a collection of classes, and a refinement is a collection of classes and/or *class fragments*, which can be applied to extend an existing program with new functionality by using and/or refining the classes defined by that program. AHEAD treats base programs and refinements as *layers*. A base program is a stand-alone layer or *constant* (i.e., the layer contains no class fragments) and a refinement is a parameterized layer (i.e., layer that must "plug in" to another, subordinate layer).

The composition of AHEAD refinements and simple programs is depicted visually using diagrams such as Figure 2(a). This figure illustrates the refinement of a base program (const). Here, the inner-most boxes are classes or class fragments. The solid rectangles demarcate refinements. The dotted lines from one class to another indicate the refinement of a class with the code and data in a class fragment; for example, $\mathsf{b}_{\mathsf{const}}$ is refined by b_{f1}. Because const is the bottom-most layer, it contains only classes and not class fragments.

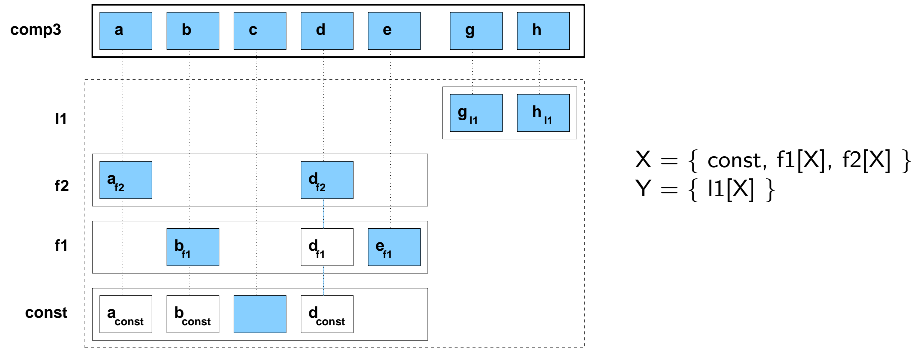

Fig. 2. Layered refinement in AHEAD.

Moving up one level, f1 is a refinement whose constituents refine two classes, and that adds a new class e which uses classes from the subordinate layer. Composing this refinement with const effectively synthesizes a new collaboration that implements that of const augmented by the feature implemented by f1 and results in a new, composite layer we will refer to as comp1. This composition is specified textually via the *type equation* comp1 = f1⟨const⟩. Moving up another level, f2 comprises two class refinements, which collaborate to implement another new feature. In this example, f2 refines comp1, thus creating a new layer that implements the functionality of const augmented with the features of both f1 and f2. The resultant type equation is comp2 = f2⟨f1⟨const⟩⟩.

Figure 2 applies one final refinement, l1, which contains complete classes and no fragments. While l1 may appear to be a constant, it is a refinement in the sense that it adds new abstractions (g_{l1} and h_{l1}) that use classes in the subordinate layer. For this type of refinement, we will will often simply say layer l1 *uses* comp2. The uppermost layer (comp3, in bold) illustrates the final composition that implements this collaboration. Notice that the classes in this uppermost layer are the most refined of each subordinate layer, as indicated by the grey boxes.

AHEAD also employs a type system for reasoning about, classifying, and codifying the relationships between layers. In this type system, layers that share a common interface are elements of a *realm*, and that common interface can be thought of as the *realm type*. For example, the layers of Figure 2(a) and their relationships are expressed in Figure 2(b). In this figure, const is a constant of realm X whose classes implement the interfaces that comprise the type of this realm, namely the class interfaces a, b, c, and d. As we saw earlier, f1 and f2 refine the layers below them; here, this is formalized by the presence of a realm parameter that conveys that these layers augment layers of type X.

Based on this small set of layers, many different compositions may be instantiated, e.g., f1⟨const⟩, f2⟨f1⟨const⟩⟩, and l1⟨f2⟨const⟩⟩. Each of these instantiations synthesizes a set of classes, whose instances collaborate to implement all the features of the base program and each of the refinements. We call such a collection

of collaborating objects a *configuration*. Notice that some type equations denote new refinements rather than whole programs. For example, the type equation $\mathsf{cf1} = \mathsf{f1}\langle\mathsf{f2}\rangle$ denotes a valid composition, but because the class refinements of $\mathsf{f2}$ depend on classes provided by its realm parameter, $\mathsf{cf1}$ is simply a composite refinement; it cannot be instantiated as specified to produce a configuration of collaborating objects. When a composition may not denote a complete program, we may opt to use the more general functional composition operator ($\circ$). For example, we could rewrite the definition of $\mathsf{cf1}$ as $\mathsf{cf1} = \mathsf{f1} \circ \mathsf{f2}$.

As noted in the Section 1, reliability strategies do not always map to a single layer; rather, they are often implemented by a collection of layer refinements that collaborate to implement a complete reliability strategy. For instance, consider a reliability strategy that is implemented by the collaboration of $\mathsf{l1}$ and $\mathsf{f1}$. In AHEAD, such collaborations may also be represented by $\{\mathsf{l1}, \mathsf{f1}\}$, which is a *collective* (set of layers) that represents the collaboration implemented by this composite refinement. Under AHEAD, a *model* is a set of constants and refinements (each of which may themselves be collectives) whose constituents are the building blocks of a product line[21].

In our example, such a product line would comprise configurations represented by, e.g., const, $\mathsf{f1}\langle\mathsf{const}\rangle$, $\mathsf{l1}\langle\mathsf{f2}\langle\mathsf{const}\rangle$, and so on. A model of this product line is

$$\mathsf{M} = \{\{\mathsf{const}\}, \{\mathsf{f1}\}, \{\mathsf{l1}, \mathsf{f2}\}, \{\mathsf{l1}, \mathsf{f1}\} \ldots\} \tag{1}$$

Here, M comprises a constant ($\{\mathsf{const}\}$) and a set of refinements (the remaining collectives). A member of this product line is instantiated by

$$\begin{aligned} \mathsf{rs} &= \{\mathsf{l1}, \mathsf{f1}\} \circ \{\mathsf{const}\} && (2)\\ &= \{\mathsf{l1}, \mathsf{f1} \circ \mathsf{const}\} && (3)\\ &= \mathsf{l1} \circ \mathsf{f1} \circ \mathsf{const} && (4)\\ &= \mathsf{l1}\langle\mathsf{f1}\langle\mathsf{const}\rangle\rangle && (5) \end{aligned}$$

Using collectives, we can represent a reliability strategy as a single unit that can be applied to a base program even when that unit comprises multiple refinements. We use such a model and its constituent collectives to mirror the application of connector wrappers, each of which corresponds to a collective that implements a reliability strategy, to connectors, i.e., base middleware implementations. This model of reliable middleware is presented in Section 4.

3 Theseus

One of the goals of our approach is to augment middleware services with reliability by refining the abstractions used in the implementation of these services rather than by treating the services as a black box and wrapping them with extra functionality. To accomplish this goal requires a middleware framework whose design exposes these major abstractions and makes them available for further

refinement. We developed a middleware framework called Theseus that is organized according to an AHEAD model. This section describes this model and shows how it is used to synthesize custom asynchronous middleware services that support a variety of different reliability policies. The Theseus model comprises two realms: MSGSVC, whose layers implement a variety of different message services, and ACTOBJ, whose layers implement variants of the distributed active object pattern. Most of the layers in these realms implement different reliability strategies (or low-level services that support different reliability strategies). Using Theseus, an architect can easily customize middleware services to support specific reliability policies without the duplication and efficiency burdens inherent in the use of wrappers.

3.1 Message Service

In Theseus, the message service provides queue-like communication abstractions that implement a simple, reliable[3] message-oriented middleware. A client of the message service sends data by enqueuing a message in a peer's inbox (which typically resides in a separate address space on a remote machine) and receives data by retrieving messages from its inbox. The sending end of the message service is a peer messenger whose interface is specified by `PeerMessengerIface`; the receiving end is a message inbox whose interface is specified by `MessageInboxIface` (Figure 3). An inbox is bound to a universal resource identifier (URI) and listens

```
public interface MessageInboxIface {
  public Serializable retrieveNextMessage ();
  public LinkedList retrieveAllMessages ();
  public boolean hasMessages ();
  public int numQueued ();
}

public interface PeerMessengerIface {
  public void sendMessage ( Serializable msg );
  public void setURI ( String URI );
  public ResourceIdentifierIface getURI ();
  public void connect ();
}
```

Fig. 3. Interfaces in the message service.

for, receives, and queues messages sent to that URI. These details are hidden by `MessageInboxIface`, through which the inbox client treats the network like a queue, receiving messages with calls such as `retrieveAllMessages`. A peer

[3] in the sense that it is built atop a connection-oriented transport such as TCP.

messenger connects to an inbox, given its URI, and sends messages (in our case, any serializable object) by invoking `sendMessage`.

Figure 4 depicts the MSGSVC realm, whose layers define and refine these abstractions. The constant in this realm is rmi[4], which comprises classes that

MSGSVC = { rmi, idemFail[MSGSVC], bndRetry[MSGSVC],
indefRetry[MSGSVC], cmr[MSGSVC], dupReq[MSGSVC] }

Fig. 4. Message service realm layers.

implement the most basic form of these abstractions. The remaining layers are reliability-enhancing refinements, which we will describe as the need arises. One such refinement is bndRetry (bounded retry), which augments an existing `PeerMessenger`[5] to, in the event of a communication failure, suppress the communication exception(s) and retry some number of times ($maxRetries > 0$) before giving up and throwing the exception.

Figure 5 shows the layered representation of an assembly that applies bndRetry to the basic message service rmi. In Figure 5, bndRetry refines classes of the

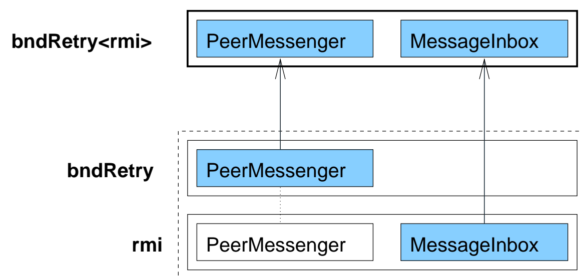

Fig. 5. Visual stratification of bndRetry⟨rmi⟩.

rmi layer, namely, `PeerMessenger`. In the remainder of our diagrams, the grey classes are the most refined, and the layer in bold represents the client's view of the assembly. Clients always use the most refined implementation of an interface (indicated by the arrows); in the case of the `PeerMessengerIface`, the most refined implementation is that of bndRetry. Because the bndRetry layer did not refine `MessageInbox`, the rmi implementation remains the most refined implementation of `MessageInboxIface`.

[4] For convenience, we built our message service atop RMI; the message service abstractions are general and may also be implemented atop object streams, TCP, or any other connection-oriented transport.

[5] An implementation of `PeerMessengerIface`; our classes follow the convention that interfaces are suffixed by "`Iface`" and the corresponding implementation is not.

3.2 Active Objects

Theseus' second realm is called ACTOBJ because its layers define classes and class refinements that implement different variations of the distributed active object pattern [6]. An *active object* is an object that has its own thread of control (the *execution thread*), listening for operation requests and executing the corresponding operations when the requests arrive. A complete operation execution in this model has three phases: invocation and queueing, dispatching and execution, and returning results. In the first phase, a *proxy* (in the sense of [15]) marshals the invocation into an operation request (referred to simply as a *request*) and queues it on an *activation list*. The execution phase is initiated by the *scheduler*, which is a loop (running in the execution thread) that dequeues requests from the activation list to be executed. In the simplest case, the scheduler dequeues these in FIFO order. Once dequeued, requests are passed to the *dispatcher* to be invoked on the *servant*, which is an object that actually implements the behavior modeled by the active object [6]. When the servant completes this invocation, the results are returned to the client.

The distributed active object pattern follows the same basic architecture, except that the operations invoked by a client are executed in an object that lives in a separate address space. Middleware stubs and skeletons insulate the client from the details of communication with the remote active object [6]. The stub behaves like the proxy, except rather than queuing the requests on an activation list, they are sent via some form of inter-process communication (IPC) to the skeleton, which resides in the same address space as the servant. This skeleton comprises a scheduler that schedules requests to be executed in the execution thread. Once a request has been dequeued, unmarshaled, and executed, the results are then sent back to the client via IPC.

The ACTOBJ realm type comprises interfaces, such as `SchedulerIface` and `DispatcherIface`, whose instances collaborate to implement distributed active objects. The layers that implement and refine these are shown in Figure 6. Notice this realm contains no constants. The core layer is parameterized by

ACTOBJ = { core[MSGSVC], respCache[ACTOBJ], eeh[ACTOBJ], ackResp[ACTOBJ] }

Fig. 6. Active object realm layers.

the MSGSVC realm. Among others, core contains two classes, `StaticDispatcher` and `FIFOScheduler`, which implement the `DispatcherIface` and `SchedulerIface` interfaces respectively, and which are designed to use subordinate services that are defined in the MSGSVC realm type. Nothing in the implementation of class `FIFOScheduler` (respectively `StaticDispatcher`) depends on the particular implementation of the `MessageInboxIface` (respectively `PeerMessengerIface`) interface. Thus, core is "parameterized by" the MSGSVC realm.

3.3 Synthesizing Middleware Services

To create a set of middleware services, we instantiate objects from the classes defined in the assembly core⟨rmi⟩, which is our most basic middleware assembly and the one that is refined to create all of the other variants. Here, the message service is refined to include core's abstractions for building active objects, as is depicted in Figure 7. Notice that none of the classes in core refine any of those

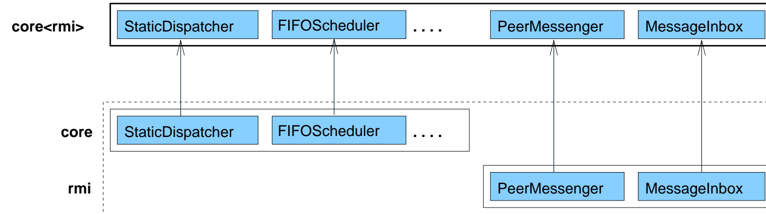

Fig. 7. Layers of a simple middleware.

in the rmi layer. Rather, core uses rmi's concrete classes in the same sense that `FIFOScheduler` uses `MessageInboxIface`. Notice also that the rmi classes are still visible in the assembly and are thus available for further refinement, as are the classes in core. That the classes defined in a subordinate layer remain visible to superior layers allows the functionality defined in these superior layers to tap into and reuse the basic abstractions used to implement the subordinate functionality.

To illustrate how this visibility accommodates refinement, consider a bounded retry policy that prescribes (1) suppressing errors, (2) performing a bounded number of retries, and (3) throwing an exception if these retries fail. Our basic middleware, specified by core⟨rmi⟩, comprises classes that implement the minimum functionality necessary to implement active objects; accounting for any type of exceptional conditions is not part of that minimal functionality. Now consider how this basic middleware assembly must be modified to implement a bounded retry strategy: The first two requirements are met by the bounded retry augmentation of the message service. To meet the third requirement, we must augment the stub, which does not account for exceptions, to properly transform internal exceptions thrown by the message service into those declared to be thrown by the active-object interface. To this end, we refine the pertinent classes in the active object layer.

The stub is implemented by an instance of the active object's interface that performs the first phase of invocation marshaling. To create such an instance of an arbitrary active object interface, we generate these instances using Java's Dynamic Proxy Framework [22]. Such an instance is referred to as a dynamic proxy and is generated by a static factory method that is parameterized by a metaobject representation of the interface [6] and an instance of the `InvocationHandler`

[6] i.e., an instance of class `Class` in Java used to generate the dynamic proxy itself

interface, which will be used by the dynamic proxy to process operations invoked on the proxy. The dynamic proxy itself is an object that marshals invocations of its operations into two objects: an instance of class `Method` that represents the operation invoked and an array of `Objects` that are the parameters of this invocation, which it immediately passes to its instance of `InvocationHandler` for processing.

The core layer defines a class `TheseusInvocationHandler` that is responsible for completing invocation marshaling. This class implements the `InvocationHandler` interface; thus its instances can be passed to the static factory method that generates the dynamic proxy. At run-time, these instances use an instance of class `PeerMessengerIface` to send the resultant request to the skeleton of the active object. In the minimal assembly, i.e., core⟨rmi⟩, the invocation handler does not account for exceptions. In the more realistic case, the underlying network may fail or the server on which the active object resides may crash; in either event, the peer messenger that is used by the invocation handler will throw an `IPCException`[7]. To accommodate this possibility, we refine the `TheseusInvocationHandler` to transform these exceptions into the exceptions that the active object's interface declares in its throws clause.

The refinement that performs this transformation is eeh (exposed exception handler) and is depicted in Figure 6. A minimal middleware augmented by eeh is expressed by eeh⟨core⟨rmi⟩⟩. Adding bounded retry to the message service completes the functionality specified by the bounded retry policy; the configuration then becomes eeh⟨core⟨bndRetry⟨rmi⟩⟩⟩. The layers that implement this configuration are shown in Figure 8.

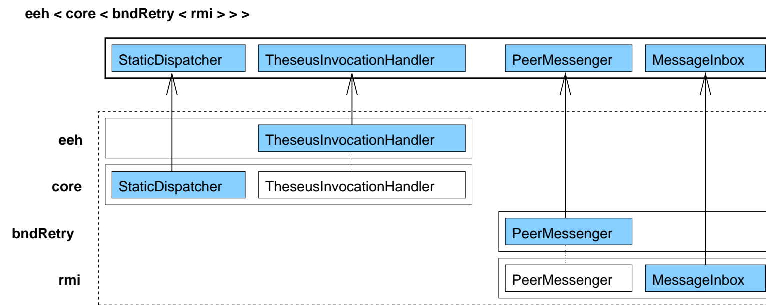

Fig. 8. Layered implementation of the bounded retry strategy.

[7] The astute reader will notice that `PeerMessengerIface` does not declare any exceptions; `IPCException` is an unchecked runtime exception that need not be declared in a throws clause. To avoid polluting the interfaces of realm types with throws declarations for checked exceptions they may or may not have to handle, we encapsulate all checked exceptions in runtime exceptions, placing the responsibility for managing such exceptions on the developer.

3.4 Efficiency Improvements in Bounded Retry

To see how our design forestalls the duplication that arises when implementing reliability strategies via wrappers, consider a wrapper-based implementation of the bounded retry policy. The wrapper would have to be applied to the server stub, i.e., the object returned by RMI's `Naming.lookup` call. Upon communication failure, a remote exception is propagated from the underlying transport up to the wrapper, where it is caught and responded to by invoking the operation on the base stub again. Notice that in this scenario, each retry subsequent to the initial failure must perform the entire client side invocation process, including the re-marshaling of the same invocation.

In Theseus, by contrast, the class of the object that is used to send the marshaled invocation, i.e., the class that implements the `PeerMessengerIface` interface, is available for further refinement. And indeed, the bndRetry layer refines this class with the retry logic, thereby placing the retry logic "beneath" the marshaling logic. Consequently, this implementation avoids the cost of re-marshaling for each retry. The composite (bold) layer in Figure 8 depicts the synthetic collaboration whose participants include the eeh-augmented invocation handler and the bndRetry-augmented peer messenger. Instances of these refined classes collaborate to implement our bounded retry strategy. In the next section, we describe a model that facilitates applying such collaborating refinements to a middleware as a single unit.

4 An AHEAD Model of Reliable Middleware

In the previous section we implemented a strategy for bounded retry in two phases. The first phase refined the message service to suppress exceptions and retry some bounded number of times before giving up. The second phase refined the active object layer to transform internal exceptions into those declared by the active object interface, i.e. those a client of the stub would expect based on the active object's interface. Consequently, the functionality associated with the bounded-retry connector wrapper manifests not as a single layer, but rather as a collective that comprises two layers. In fact, most of the connector wrappers specified in Spitznagel's thesis cannot be implemented as a single layer without some degree of duplication. However, all of them can be implemented using collectives that comprise multiple layers.

We now show how to represent this product line as an AHEAD model whose elements are collectives. The resulting model contains one constant, the most-basic assembly core⟨rmi⟩, and one collective for each reliability strategy. As we describe our model and how it is used, we also focus on how we use our AHEAD model to group the structural changes affected by refinements such that they correspond to reliability connector wrappers. As we will see, a base middleware, such as core⟨rmi⟩, corresponds to a middleware connector specification and collectives that implement a reliability strategy correspond to reliability connector wrappers. As we describe our model, we will make these correlations explicit.

4.1 A Reliable Middleware Model

Our model $\mathsf{THESEUS}$, is

$$\mathsf{THESEUS} = \{\mathsf{BM}, \mathsf{RS_0}, \mathsf{RS_1}, \ldots, \mathsf{RS_n}\} \tag{6}$$

where BM is a collective that represents the base middleware and each $\mathsf{RS_i}$ $(0 \leq \mathsf{i} \leq \mathsf{n})$ is a collective that represents some reliability strategy. Our base middleware (BM) is $\{\mathsf{core_{ao}} \circ \mathsf{rmi_{ms}}\}$, which is equivalent to $\{\mathsf{core_{ao}}, \mathsf{rmi_{ms}}\}$[8] where the subscripts ao and ms indicate layers in the active object and message service realms, respectively. Similarly, each strategy $\mathsf{RS_i}$ is a collective of the form $\{\mathsf{refinement_i_{ao}}, \mathsf{refinement_i_{ms}}\}$, where $\mathsf{refinement_i_{ao}}$ applies to (refines) an active-object layer and $\mathsf{refinement_i_{ms}}$ applies to a messages-service layer.

Elements of this product line are synthesized by choosing a set of reliability strategies and then applying these in sequence to the base-middleware assembly. An example application of the first two strategies, $\mathsf{RS_0}$ and $\mathsf{RS_1}$, is

$$\begin{aligned}
\mathsf{rm} &= \mathsf{RS_1} \circ \mathsf{RS_0} \circ \mathsf{BM} && (7)\\
&= \{\mathsf{ref_1_{ao}}, \mathsf{ref_1_{ms}}\} \circ \{\mathsf{ref_0_{ao}}, \mathsf{ref_0_{ms}}\} \circ \{\mathsf{core_{ao}}, \mathsf{rmi_{ms}}\} && (8)\\
&= \{\mathsf{ref_1_{ao}}, \mathsf{ref_1_{ms}}\} \circ \{\mathsf{ref_0_{ao}} \circ \mathsf{core_{ao}}, \mathsf{ref_0_{ms}} \circ \mathsf{rmi_{ms}}\} && (9)\\
&= \{\mathsf{ref_1_{ao}} \circ \mathsf{ref_0_{ao}} \circ \mathsf{core_{ao}},\ \ \mathsf{ref_1_{ms}} \circ \mathsf{ref_0_{ms}} \circ \mathsf{rmi_{ms}}\} && (10)
\end{aligned}$$

There are three important properties of this composition. First, refinements naturally apply to layers in the realm that they refine. In Equation 9, the active-object refinement $\mathsf{ref_0_{ao}}$ composes with $\mathsf{core_{ao}}$, and the message-service refinement $\mathsf{ref_0_{ms}}$ composes with $\mathsf{rmi_{ms}}$. Second, the order of refinement is preserved. Namely, Equation 7 indicates the refinements should be applied right to left: $\mathsf{RS_0}$, then $\mathsf{RS_1}$. In Equation 10 this ordering has been preserved in the refinement of both the active-object and message-service layers.

The third property is this structural representation's high-level mapping to connectors and connector wrappers. Here, BM implements the base middleware and corresponds to a connector that specifies the behavior of communication among components that use this base middleware. The collectives $\mathsf{RS_0}$ and $\mathsf{RS_1}$ correspond to reliability connector wrappers that augment BM. The collectives that implement reliability strategies decompose further into reusable refinements, much like Spitznagel's connector wrappers decompose into connector transforms, but do not exhibit a strict one-to-one correspondence.

4.2 Reliable Middleware Examples

We now illustrate three applications of synthesis using the $\mathsf{THESEUS}$ model.

[8] Recall uses relationship described in Section 2.3; $\mathsf{core_{ao}}$ uses $\mathsf{rmi_{ms}}$, as illustrated in Figure 7.

Bounded Retry As noted earlier, our bounded retry strategy is implemented by the bounded retry ($\mathsf{bndRetry_{ms}}$) refinement of the message-service realm and the exposed exception handler ($\mathsf{eeh_{ao}}$) refinement of active-object realm. Under our model, this strategy is implemented as the collective

$$\mathsf{BR} = \{\mathsf{eeh_{ao}}, \mathsf{bndRetry_{ms}}\} \tag{11}$$

A model of a bounded retry augmented middleware bri is thus

$$\begin{aligned} \mathsf{bri} &= \mathsf{BR} \circ \mathsf{BM} & (12)\\ &= \{\mathsf{eeh_{ao}}, \mathsf{bndRetry_{ms}}\} \circ \{\mathsf{core_{ao}}, \mathsf{rmi_{ms}}\} & (13)\\ &= \{\mathsf{eeh_{ao}} \circ \mathsf{core_{ao}}, \mathsf{bndRetry_{ms}} \circ \mathsf{rmi_{ms}}\} & (14) \end{aligned}$$

The visual stratification of the layers, as specified by Equations 12 and 13 is shown in Figure 9. In this figure, the bottom two layers implement the middle-

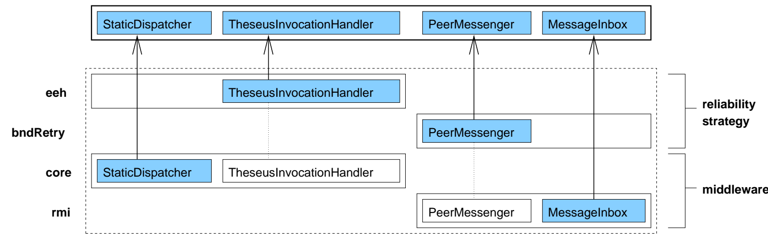

Fig. 9. Grouping bounded-retry layers into a collective.

ware and the top two are the refinements that implement the bounded retry strategy. This visual stratification is expressed by Equation 14, whose layers are the same as in Figure 8. Because each refinement in this model is local to a specific realm (either message service or active object) the refinements may be applied in arbitrary order; however refienements are not, in general, commutative.

Figures 8 and 9 visually depict the tighter binding of AHEAD refinements that is shown equationally in Equations 12-14. Figure 9, and the corresponding Equation 12, appear much as we would expect under connector wrapper composition, applying BR, which implements the bounded retry reliability connector wrapper, to BM, which implements an existing connector.

Idempotent Failover The idempotent failover policy specifies that, in the event of a communication failure, the client should connect to a known backup. In the simple version of failover, operations are assumed to be idempotent and therefore the original (primary) server and backup need not be synchronized with one another. Upon failure of the primary, a failover refinement will suppress

the exception and silently switch over to the backup. Further, this policy assumes a perfect backup that never fails; thus, once failover occurs, no additional communication exceptions will arise.

Our implementation of this strategy is very similar to that of the bounded retry refinement of the message service. In this case, instead of initiating a retry loop on a communication exception, the class refinement simply resets the URI of the peer messenger (via `setURI`, Figure 3) to that of the backup, connects (via `connect`, Figure 3) to the corresponding inbox, and proceeds as normal. Under our model, the strategy is

$$\mathsf{FO} = \{\mathsf{idemFail}_{\mathsf{ms}}\} \tag{15}$$

and an application is

$$\begin{aligned} \mathsf{foi} &= \mathsf{FO} \circ \mathsf{BM} && (16)\\ &= \{\mathsf{idemFail}_{\mathsf{ms}}\} \circ \{\mathsf{core}_{\mathsf{ao}}, \mathsf{rmi}_{\mathsf{ms}}\} && (17)\\ &= \{\mathsf{core}_{\mathsf{ao}}, \mathsf{idemFail}_{\mathsf{ms}} \circ \mathsf{rmi}_{\mathsf{ms}}\} && (18) \end{aligned}$$

In this example, FO comprises only a refinement of the message service. Because failover is “perfect”, no exceptions propagate up to the client. As such, there is no need to refine the active object with exception transformation logic such as exposed exception handler.

Bounded Retry and Idempotent Failover As a final illustration of the model, consider the application of both the bounded retry and idempotent failover strategies. The idea behind this composite reliability strategy is to retry the primary some finite number of times before failing over to the backup. As such, we apply bounded retry first, then failover; the application of this composite strategy is

$$\begin{aligned} \mathsf{fobri} &= \mathsf{FO} \circ \mathsf{BR} \circ \mathsf{BM} && (19)\\ &= \{\mathsf{idemFail}_{\mathsf{ms}}\} \circ \{\mathsf{eeh}_{\mathsf{ao}}, \mathsf{bndRetry}_{\mathsf{ms}}\} \circ \{\mathsf{core}_{\mathsf{ao}}, \mathsf{rmi}_{\mathsf{ms}}\} && (20)\\ &= \{\mathsf{idemFail}_{\mathsf{ms}}\} \circ \{\mathsf{eeh}_{\mathsf{ao}} \circ \mathsf{core}_{\mathsf{ao}}, \mathsf{bndRetry}_{\mathsf{ms}} \circ \mathsf{rmi}_{\mathsf{ms}}\} && (21)\\ &= \{\mathsf{eeh}_{\mathsf{ao}} \circ \mathsf{core}_{\mathsf{ao}}, \mathsf{idemFail}_{\mathsf{ms}} \circ \mathsf{bndRetry}_{\mathsf{ms}} \circ \mathsf{rmi}_{\mathsf{ms}}\} && (22) \end{aligned}$$

Attending to the refinements of the message service, bounded retry is applied first, then failover, as intended. Under these refinements,

1. A communication exception thrown by $\mathsf{rmi}_{\mathsf{ms}}$ will be suppressed by $\mathsf{bndRetry}_{\mathsf{ms}}$, which will attempt to reconnect and resend the marshaled request some bounded number of times.
2. If the $\mathsf{bndRetry}$ does not successfully reconnect, it will throw the communication exception.
3. $\mathsf{idemFail}$ will suppress this exception, connect to the backup, and resend the marshaled request.

In Spitznagel’s connector-wrapper specification of each strategy, exceptions are modeled by the action error [1]. In these specifications, the error action is intercepted and triggers recovery; in this case, first a bounded retry, then failover.

Here, we see AHEAD collectives also compose, both structurally and behaviorally, in the same manner as connector wrappers.

Now consider if the order were changed to

$$\mathsf{fobri} = \mathsf{BR} \circ \mathsf{FO} \circ \mathsf{BM} \tag{23}$$

idemFail would immediately switch over to the backup on failure, occluding any communication exception from reaching bndRetry and would be functionally equivalent to Equation 16. This is also the case in the corresponding connector specification; the juxtaposition finds the error action immediately triggering failover behavior, just as the exception does in our implementation.

This also illustrates how a semantic conflict, namely the overlapping of the recovery strategies used, may cause one refinement to occlude another. Because a failover augmented middleware will never throw a communication exception, the $\mathsf{eeh}_{\mathsf{ao}}$ is not needed and adds unnecessary processing. Under AHEAD, this is a problem of composition optimization. While it is possible to inspect such an equation and remove exposed exception handler, this optimization is not "automatic" and requires some form of higher reasoning about the semantics of composite refinements.

5 Contrasting Implementation Strategies

We believe that our AHEAD-style implementation of reliability strategies offers a useful alternative to wrappers without sacrificing the flexibility of composition and reasoning provided by the connector-wrapper formalism. In previous sections we briefly illustrated a simple efficiency improvement in a message service implementation of bounded retry (Section 3.4). We now describe the implementation of a more complex reliability policy, warm failover, and identify where our approach eliminates both redundancy and the need for additional logic for suppressing behavior.

5.1 Warm Failover

Warm failover is a reliability policy that uses a backup server providing reliability via redundancy in a client-server architecture. This policy is a variation of process pairs and takeover in transaction systems[23]. The original server is referred to as the primary. The backup is "warm" in the sense that it is kept in sync with the primary via some strategy dependent mechanism. Under this policy, if the primary fails, the client uses the backup to recover lost responses. The client then promotes the backup to the role of primary, which means the client sends requests to and expects responses from the backup and the backup, correspondingly, accepts requests from and sends responses to the client. This policy assumes a "perfect" backup that will not fail, and, as such, does not account for the failure of the backup.

The strategy we employ to implement warm failover is referred to as silent backup. Under this strategy, the client sends each request to both the primary

and the backup. The primary processes these requests and sends the responses to the client. The backup also processes requests (and is thus in sync with the primary) but, rather than send the responses to the client, the backup caches these in an outstanding response cache, which is keyed on the response's unique id (an asynchronous completion token). This cache is intended to store only the responses that the client has yet to receive from the primary. To maintain the cache as such, the client is obligated to send acknowledgements (that comprise a response id) to the backup when it receives a response from the primary, indicating that response may be removed from the cache. Upon failure of the primary, the client sends a control message to the backup indicating this failure and promotes the backup to the role of primary. When the backup receives such a message, it sends any outstanding responses to the client and, henceforth, upon processing a request, sends the response to the client rather than caching it.

5.2 Theseus Refinements

To implement silent backup, we must augment the client and create a backup that fulfills the responsibilities outlined above. The primary remains unchanged.

Client Refinements To implement the client's responsibilities, we send each request to both the primary and the backup, activate the backup in case the primary fails, and send acknowledgements to the backup as the primary's responses arrive such that the backup may purge these from the response cache. The following describes the refinements that fulfill these responsibilities.

Duplicate Request (dupReq*)* refines `PeerMessenger` to connect to and send requests to both the primary and the backup. In the event that the primary fails, the peer messenger sends a special activate message to the backup, which indicates the backup should assume the role of the primary. Once the activate message has been sent, the peer messenger sends requests only to the backup.

Acknowledge Response ($\mathsf{ackResp}$*)* refines the active object layer to send acknowledgements indicating a response has been received. In Theseus, a variant of the dispatcher (`DynamicDispatcher`) is used to dispatch responses to threads dedicated to processing responses and making them available to the client. Here, this type of dispatcher is refined to send acknowledgements to the backup as it dispatches these responses.

The client side of silent backup is implemented by the collective SBC.

$$\mathsf{SBC} = \{\mathsf{ackResp_{ao}}, \mathsf{dupReq_{ms}}\} \tag{24}$$

The warm failover client wfc is instantiated by

$$\begin{aligned} \mathsf{wfc} &= \mathsf{SBC} \circ \mathsf{BM} && (25)\\ &= \{\mathsf{ackResp_{ao}}, \mathsf{dupReq_{ms}}\} \circ \{\mathsf{core_{ao}}, \mathsf{rmi_{ms}}\} && (26)\\ &= \{\mathsf{ackResp_{ao}} \circ \mathsf{core_{ao}}, \mathsf{dupReq_{ms}} \circ \mathsf{rmi_{ms}}\} && (27)\\ & && (28) \end{aligned}$$

The visual depiction of these layers is shown in Figure 10.

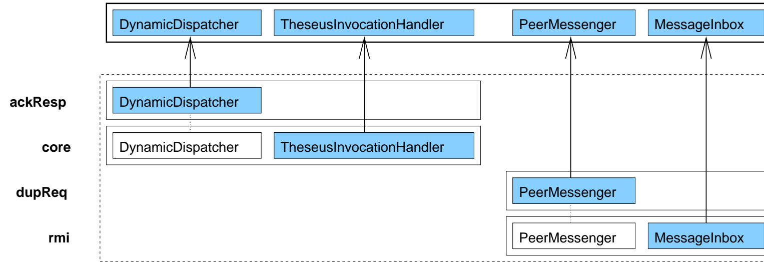

Fig. 10. Silent backup client configuration.

Backup Server Refinements The backup server should have all the functionality of the original server, modulo features for handling control messages, caching responses, and switching from the role of silent backup to that of primary. To this end, we refine our minimal middleware (core⟨rmi⟩) with the following refinements:

Control Message Router ($\mathsf{cmr_{ms}}$) is a refinement of the message service that accommodates specially formed control messages (acknowledgement and activate messages) that have the same expedited properties as TCP's out-of-band data [24] using existing operations of the `PeerMessengerIface` and `MessageInboxIface`. Control messages are serializable objects that implement `ControlMessageIface`, which provides getters for retrieving both the command type (such as "ACK" and "ACTIVATE") and the data payload of the message (such as the id of the response being acknowledged). When such an object is passed to `PeerMessenger`'s `sendMessage` operation, it delivers that object to the corresponding inbox as normal. The control message router layer refines the inbox to filter control messages so they are handled immediately (expedited) and not mistakenly passed along as service requests. On the inbox side of communication, listeners implement a `ControlMessageListenerIface` and register themselves as listeners, indicating which command type they are interested in being notified of. When a command of that type arrives, the inbox invokes the `postControlMessage` operation of the interested listeners.

Response Cache ($\mathsf{respCache_{ao}}$) augments the active object layer to cache responses. In a Theseus skeleton, the stub logic that marshals requests (Section 3.3) is used to marshal responses. We refine the invocation handler that participates in marshaling responses to store these in the cache rather than send them to the client. Further, the refined invocation handler implements `ControlMessageListenerIface` and is registered with the control message router

to listen for both acknowledgement and activate messages. Upon acknowledgement of a response, the invocation handler removes that response from the cache. Upon activate, the backup starts delegating requests to a live invocation handler (one that sends responses to the client rather than storing them), effectively switching to a configuration that is equivalent to that of the primary.

Our implementation of the server half of the silent backup strategy, SBS, is

$$\mathsf{SBS} = \{\mathsf{respCache_{ao}}, \mathsf{cmr_{ms}}\} \tag{29}$$

When instantiated, the corresponding configuration, sb, is

$$\mathsf{sb} = \mathsf{SBS} \circ \mathsf{BM} \tag{30}$$
$$\mathsf{sb} = \{\mathsf{respCache_{ao}}, \mathsf{cmr_{ms}}\} \circ \{\mathsf{core_{ao}}, \mathsf{rmi_{ms}}\} \tag{31}$$
$$\mathsf{sb} = \{\mathsf{respCache_{ao}} \circ \mathsf{core_{ao}}, \mathsf{cmr_{ms}}, \mathsf{rmi_{ms}}\} \tag{32}$$

The visual depiction of these layers is shown in Figure 11.

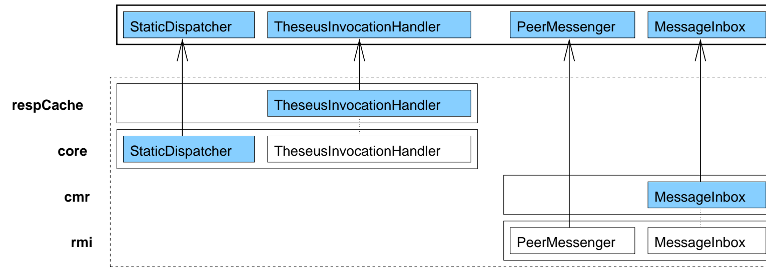

Fig. 11. Backup server configuration.

5.3 Implementing Silent Failover with Middleware Wrappers

We now contrast our implementation of silent failover with a wrapper implementation constructed using Spitznagel's wrapper transformations.

Duplicating Requests To duplicate requests sent by the client, the add observer wrapper is applied. This wrapper creates a duplicate middleware stub for communicating with the backup server. Each time an operation is invoked, the corresponding request is sent to both the primary and the backup. As such, the marshaling due to the second invocation is both functionally and structurally equivalent to the first, introducing redundant processing in redundant components. Because our approach refines the peer messenger to send the marshaled request to both the primary and the backup in the message service, we avoid the processing redundancy inherent in marshaling the same invocation twice.

Managing the Response Cache Acknowledging responses and maintaining a response cache requires the introduction of unique identifiers (asynchronous completion tokens) to both the request and the response and the introduction of an out-of-band message service comparable to our control message service. The easier of the two is the application of a data translation wrapper that introduces unique identifiers such that they are available for use by middleware wrappers. Upon client invocation, a data-translation wrapper cannot modify the marshaled request, but it can add a unique identifier to the invocation parameters. On the backup, a dual data translation wrapper wraps the servant and removes this identifier. Also on the backup, this wrapper must apply the unique identifier to the return data (the response) and store that response in a response cache. While these wrappers work, the introduction of unique identifiers is redundant with the corresponding middleware identifiers used to coordinate requests and responses (such as CORBA's object id[25] and RMI's UID[26]). In Theseus, refinements such as ackResp and respCache have access to the existing identifier marshaled into a request. As such, they non-destructively re-use these identifiers to maintain the response cache.

An auxiliary concern that arises when treating the middleware as a black box is how to "silence" the backup server. Silencing the backup requires somehow, non-invasively orphaning the components that send responses. Under black box wrapping, on the server side, wrappers are applied to the servant that is registered with the middleware. When a request is received by that middleware, it invokes the corresponding invocation on the servant and expects that servant to return some data that is to be sent back to the client. As such, one must affect suppression of the reply, caching it instead, when the servant returns. In some middleware systems, interceptor techniques (such as CORBA's interceptors[12, 13]) can be used to suppress replies by intercepting, caching, and then discarding each without sending them. However, not all systems support such facilities and would have to send some form of response to the client. As such, the client must suppress this behavior by discarding responses sent by the backup. In the latter case, the backup can not be made silent and will create additional traffic that silent backup was intended to avoid. In contrast, we silence the backup by applying a refinement (respCache) that replaces the invocation wrapper that sends the responses with one that caches them, effectively removing this component rather than orphaning it.

The more difficult part of managing the response cache is sending the expedited control messages needed to acknowledge responses and activate the backup server. Because conventional middleware, by its nature, hides the underlying communication primitives, expedited control messages and the corresponding out-of-band data channel must be implemented completely independently of the stub and skeleton infrastructure. To this end, client wrappers must contain hooks for communication with objects that instantiate and maintain an additional communication channel between the client and the backup for such expedited messages. Correspondingly, the wrapper on the backup must maintain similar hooks to objects that implement a server that listens for such messages and han-

dles the connections between itself and its clients. This solution introduces both complexity and a duplicate communication channel, further increasing system resource usage. Using refinements, the developer of silent backup may refine the existing message service to filter out control messages and post them to their listeners immediately, preserving the expedited nature of the out-of-band messages. Moreover, this refinement re-uses the existing channel and avoids the need for an auxiliary message service to manage out-of-band channels.

Recovery from Failure As per silent backup, in the event of an error, the client activates the backup, the backup sends outstanding responses to the client, and then the backup assumes the role of the primary. Implementing these with wrappers requires adding fairly extensive recovery logic that uses the out-of-band data channel to resend responses. This logic is added to both the client and backup server. On the backup, when the activate message is received, the responses are sent over the out-of-band data channel (because the middleware occludes access to the underlying communication channel), to the client. After sending the activate message, the client waits to receive these responses and delivers the corresponding results to the client via hooks into the stub wrappers. Depending on the type of middleware being wrapped, the delivery mechanisms will differ. For instance, if the wrapper augments a message-oriented middleware, the wrapper simply needs to add the response to a queue and invoke a notification method that indicates a new message has arrived. In the case of a synchronous middleware stub, the client is blocked on a synchronous call to the stub wrapper and is awaiting its return; delivery logic must use a setter method that allows it to set the return value of that wrapper and then notify it such that it will return that value to the client.

In our refinement-based implementation, by virtue of our ability to re-use existing abstractions, recovery is drastically simplified. Because the refined invocation handler caches the responses as it receives them, the recovery initiated by the activate message may simply iterate through these responses, replaying them to a live invocation handler (one whose configuration is identical to that of the primary's invocation handler) that will send them to the client via a peer messenger. From the perspective of the client, because these are sent by an invocation handler identical (in configuration) to that of the primary, these responses are sent directly to the client's inbox, where they will be retrieved and delivered exactly as if they had been sent by the primary.

Promoting the Backup to Primary To complete the transition from backup to primary requires transitioning the backup from silent to active. This returns us to the concern of how to silence the backup. If this was possible via a mechanism such as portable interceptors, the logic for processing activate message ought to reverse this augmentation such that the backup sends responses to the client. If the backup is already sending responses, the client's promotion of the backup to the primary should cause the client to begin accepting responses from the client rather than discarding them as before.

Using refinements, the implementation of this transition is also drastically simplified. Recall that the backup was made silent by replacing the live invocation handler with one that caches responses. Reversing the process is just as simple: the caching invocation handler is replaced with a live invocation handler, cached responses are replayed, and subsequent responses are sent to the client by the now live invocation handler.

5.4 Discussion

Both wrappers and AHEAD collectives an be used to augment middleware with reliability. Wrappers are more reusable in that they do not require the underlying middleware to have been designed according to an AHEAD model. Consequently, legacy systems may benefit from wrapper based reliability, transparent to both the original communication system and the applications that use these. However, with this reusability comes the potential for redundancy and inefficiency. Moreover, additional complexity may be introduced if existing components must be orphaned by an augmentation.

By contrast, we sacrifice reusability across multiple middleware implementations for a mechanism that accommodates fine-grain composition and refinement. As such, we avoid the complexity of suppressing, bypassing, or accommodating behaviors of a former middleware incarnation in favor of those behaviors expressed by newer strategies. Further, our type expressions make the structural composition of these systems clear.

The skeptical reader may question the redundancy and efficiency gains afforded by the seemingly minor improvements, such as removing duplicate stubs. These "minor" inefficiencies may snowball in a system in which thousands, or even millions, of stubs and skeletons are managing the sessions of an equal number of client-server interactions. At this scale, the cumulative gain is substantial. These improvements are especially important in systems with tight resource constraints, such as small devices, real-time systems, and high-availability systems. In such cases, computational and storage resources are at a premium.

6 Conclusions and Future Work

In her work implementing connector wrappers, Spitznagel identified a covering set of transforms that can be used to implement a variety of reliability strategies and may be applied to many different middleware implementations. Because these wrappers treat middleware as a black box, this portability comes at the cost of redundancy and increased resource consumption, neither of which are acceptable for small, mobile devices that are most in want of reliability.

This paper descibes an alternative implementation that capitalizes on the compositional properties of systems built under the AHEAD-model. Under this model, a system is enhanced by augmenting the base middleware with refinements that are relevant to the reliability policy at hand and recomposing the system. These refinements may then reuse resources that implement common

middleware abstractions, allowing the refinement itself to implement only the essence of the policy at hand.

To validate our alternative, we illustrate how a minimal middleware may be augmented by refinements to implement various reliability policies. Moreover, we provide a model that groups the refinements that collaborate to implement reliability strategies into collectives that may be applied as a single, composite refinement, analogous to how connector wrappers are applied to connectors. Finally, our evaluation of silent backup illustrates that our refinements do indeed avoid redundancy when compared with a wrapper-based approach.

Our future work intends to extend Theseus with the ability to incorporate reliability enhancements at run-time, using dynamic-reconfiguration techniques, such as [27, 28]. We expect this work to leverage dynamic recomposition representations, such as Dynamic WRIGHT[29], to support a design tool that allows developers to design multiple configurations and then evaluate the possible transitions between them.

Acknowledgements. Support for this work was provided by the Office of Naval Research grant N00014-01-1-0744 and by NSF grants EIA-0000433 and CCR-9984726.

References

1. Spitznagel, B., Garlan, D.: A Compositional Formalization of Connector Wrappers. In: Proceedings of the 2003 International Conference on Software Engineering, Portland, Oregon, USA (2003)
2. Allen, R., Garlan, D.: A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering Methodology **6** (1997) 213–249
3. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. In: Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society (2003) 187–197
4. Milner, R., et al.: The Definition of Standard ML - Revised. The MIT Press (1997)
5. Smaragdakis, Y., Batory, D.: Implementing layered designs with mixin layers. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 1445 (1998) 550–570
6. Schmidt, D., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. Volume 2. John-Wiley & Sons (2000)
7. Avizienis, A., Laprie, J.C., Randell, B.: Fundamental concepts of dependability. Technical Report 010028, UCLA (1984)
8. Sowell, J.H., Stirewalt, R.E.K.: Middleware Reliability Implementations and Connector Wrappers. In: Proceedings of ICSE 2004 Workshop on Architecting Dependable Systems, Edinburgh, Scotland, UK (2004)
9. Spitznagel, B.: Compositional Transformation of Software Connectors. PhD dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA (2004)
10. Becker, T.: Application Transparent Fault Tolerance in Distributed Systems. In: Proceedings of 2nd International Workshop on Configurable Distributed Systems, Pittsburgh, Pennsylvania, USA (2004)

11. Zandy, V.C., Miller, B.P.: Reliable network connections. In: MobiCom '02: Proceedings of the 8th annual international conference on Mobile computing and networking, ACM Press (2002) 95–106
12. : Portable Interceptors. In: Common Object Request Broker Architecture: Core Specification . Object Management Group (2002) 21–1–21–64 http://www.omg.org/technology/documents/formal/corba_2.htm.
13. C.Marchetti, L.Verde, R.Baldoni: CORBA Request Portable Interceptors: A Performance Analysis. In: Proceedings of the 3nd International Symposium on Distributed Objects and Applications (DOA 2001). (2001) 208–217
14. Bergmans, L., Aksits, M.: Composing crosscutting concerns using composition filters. Commun. ACM **44** (2001) 51–57
15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
16. Schmidt, D.: The ADAPTIVE Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications. In: Proceedings of the 12th Sun Users Group Conference. (1994)
17. Fabre, J.C., Perennou, T.: A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach. IEEE Transactions on Computers, Special Issue on Dependability of Computing Systems **41** (1998) 78–95
18. Parlavantzas, N., Coulson, G., Blair, G.: An Extensible Binding Framework for Component-Based Middleware. In: Proceedings of EDOC 2003, Brisbane, Australia (2003)
19. Fleury, M., Reverbel, F.: The jboss extensible server. In: Middleware. (2003) 344–373
20. Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., Zelesnik, G.: Abstractions for software architecture and tools to support them. IEEE Trans. Softw. Eng. **21** (1995) 314–335
21. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE Trans. Softw. Eng. **30** (2004) 355–371
22. : Package java.lang.reflect. (http://java.sun.com/j2se/1.4.2/docs/api/java/lang/reflect/package-summ%ary.html)
23. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers (1993)
24. Stevens, W.R.: UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI. Prentice Hall (1998)
25. : Common Object Request Broker Architecture: Core Specification. Object Management Group (2002)
26. : Class java.rmi.server.uid. (http://java.sun.com/j2se/1.4.2/docs/api/java/rmi/server/UID.html)
27. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Trans. Softw. Eng. **16** (1990) 1293–1306
28. Hillman, J., Warren, I.: An open framework for dynamic reconfiguration. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society (2004) 594–603
29. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. In: Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98), Lisbon, Portugal (1998)