Scalable Evolution of Highly Available Systems[1]

Jason O. Hallstrom
The Ohio State University
Columbus, OH 43210
hallstro@cis.ohio-state.edu

William M. Leal
The Ohio State University
Columbus, OH 43210
leal@cis.ohio-state.edu

Anish Arora
The Ohio State University
Columbus, OH 43210
anish@cis.ohio-state.edu

Abstract
The demand for highly available software systems has increased dramatically over the past several years. Such systems must be developed using a discipline that supports unanticipated runtime evolution. We characterize the desiderata of a programming model that provides such support, and describe the design and implementation of an architecture satisfying these criteria. The Dynamic Reconfiguration Subsystem (DRSS) is an interceptor-based open container architecture that supports the development of highly available systems by enabling the scalable, dynamic deployment of cross-cutting software modifications. We have implemented a prototype of DRSS using Microsoft's .NET Framework.

Keywords
Assurance systems, highly available systems, dynamic reconfiguration, runtime evolution, online maintenance, mediation, interception, interceptor, wrapper

1. Introduction
As the global economy increasingly relies on software systems for business-critical applications, taking systems down for maintenance becomes increasingly less feasible. It seems inevitable that software vendors will be expected to follow the trend set by the hardware industry, providing a guarantee of 99.999% uptime. Given that software systems are subject to constant evolutionary pressure, systems hoping to compete in an environment with such demanding availability requirements must provide support for online maintenance. That is, systems must provide support for dynamic adaptation, enabling systems to accommodate evolving system requirements, environmental conditions, and fault scenarios, without disrupting the services they supply.

Providing support for online maintenance is possible if all future design concerns can be made manifest during requirements analysis. Under this optimistic assumption, dynamic reconfiguration can be viewed as a logical extension of design for change [1]. The portions of the design that are likely to change can be encapsulated behind modules that abstract over the possible variations. Implementations of these modules, one for each possible variation, can be *statically deployed* as part of the application assembly. The application can be designed to switch between these modules in response to user input, fault detection, evolving environmental conditions, etc.

Unfortunately, accurate prediction is not the common case. Given the extended lifetime of many software products, enumerating all possible changes, let alone programming for these changes, is an overwhelming task at best. In distributed environments, where requirements are heterogeneous and evolve independently, one might question whether accurate prediction is possible at all. Even if it were, the architect blessed with the powers of prediction is unlikely to be able to surmount the complexity of a system flexible enough to handle the potentially unlimited number of future variations. As a consequence, the system might never be developed at all: paralysis by analysis [2].

Given the seeming impracticality of the task, one alternative to an exhaustive enumeration of all possible evolutionary paths is to adopt a programming model that intrinsically supports unanticipated runtime evolution. That is, a programming model which enables the *dynamic deployment* of software modifications without the software having been designed to support specific evolution scenarios. We identify the following as key desiderata for a programming model enabling dynamic deployment:

- **Support for cross-cutting concerns.** The reconfiguration support provided must not be biased toward the system's original modularization. Future requirements typically dictate modularizations that are inconsistent with the original decomposition. The ability to adapt to these *cross-cutting concerns* must be afforded, as these concerns represent the common evolutionary case.
- **Scalability.** The model must accommodate the scale of modern application architectures, which are burgeoning in size by an order of magnitude every five to ten years [3, 4]. The reconfiguration support would have to enable the convenient deployment of modifications that affect a large number of component instances, without sacrificing the ability to perform more fine-grained alterations. That is, the programming model would have to provide for flexible scoping with respect to the effects of reconfiguration. Moreover, as the number of potential

[1] This work was supported in part by an unrestricted grant from Microsoft Research.

modifications is unbounded, the model would have to provide an intellectually manageable means by which to track the alterations that have been applied to a running system.

- **Support for compositional reuse.** Future requirements in one application are likely to appear as requirements in others; the programming model should extend the benefits of compositional reuse to the alterations applied in a running system. These recurring, unanticipated requirements are often non-functional, and typically arise as a result of changing environmental conditions. As an example, load tolerance is often designed after the initial application deployment. It would be useful here to have a generic load-balancing component that could be applied to any component instance, regardless of its type.
- **Accessibility to practitioners.** Software development is in part a social process. The industry has been slow in incorporating the results of academic research into standard programming practice [3]. Hence, the programming model should not require a radical departure from current programming practice; rather, it should impose as few design constraints as possible.

We have developed an application infrastructure using Microsoft's .NET Framework that provides a programming model consistent with the aforementioned desiderata. The Dynamic Reconfiguration Subsystem (DRSS, pronounced 'Doris') is an open container architecture (i.e. an extensible hosting environment) that enables runtime adaptation to unanticipated design concerns through interceptor-based mediation. The prototype implementation of DRSS, as well as several DRSS-based applications, is available in both binary and source form at www.cis.ohio-state.edu/siefast/msr-cont-self-maint.

Paper Organization

The remainder of this paper is organized as follows: Section 2 provides a brief introduction to software mediation, and characterizes the two fundamental techniques for its enablement. Section 3 presents the design and implementation of DRSS, followed by an example in Section 4 that illustrates several novel uses of the architecture. Section 5 provides an overview of related mediation-based approaches, with an emphasis on existing interceptor architectures. The final section summarizes the primary contributions of the research, and provides pointers to future work.

2. Software Mediation

Software mediation refers to the transparent interposition of code between the interface of a functional unit and the functional unit's implementation. In this context, a functional unit is any encapsulated element of behavior that provides its functionality through a well-defined interface. Procedures, functions, object methods, and software libraries are examples of functional units, all of which are potential mediation targets.

Mediation is unique in that it is bi-directionally transparent; neither the targets nor their clients are designed to explicitly invoke interposed code. The code is executed automatically in response to invocation requests on the original target units. As a result, mediation code can be developed independently, and used to extend or modify the behavior of target units without modifying either the targets or their clients. This makes the mediation approach particularly appealing for altering the functionality of commercial software, as source code is typically unavailable.

There are essentially two techniques for enabling software mediation, each of which is discussed in the subsections that follow.

2.1. Wrapper-Based Mediation

The most common technique for achieving software mediation is based on the notion of a software wrapper [5]. The wrapper approach leverages the fact that clients access functional units through their public interface, rather than referencing their implementations directly. Clients need not be recompiled when a new implementation is substituted for an existing implementation of the same interface.

Wrapper-based mediation calls for the development of a new implementation of the interface provided by the target unit. The new implementation is a wrapper in that it maintains a reference to the original implementation; the wrapper provides its functionality by delegating calls on its interface to the referenced implementation. Code interposition occurs within the wrapper, which provides new behavior by adding additional processing logic before and after forwarding calls to the underlying target.

As an example, consider a target component `T` that implements interface `I`. During system testing it might be desirable to log invocations on `T`, recording the name of each method invoked, as well as the values of method arguments before and after each invocation. This can be achieved by developing a wrapper component `W` that implements interface `I`, the same interface implemented by `T`. Each method of `W` records the method name and argument values before and after invoking the corresponding method on an instance of `T`. Assuming clients access `T`'s functionality through interface `I`, `W` can be substituted for `T` transparently. Wrapper-based mediation is illustrated in Figure 1.

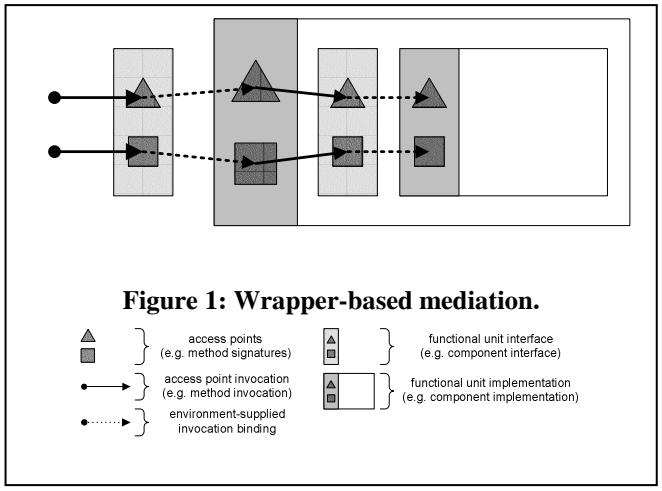


Figure 1: Wrapper-based mediation.

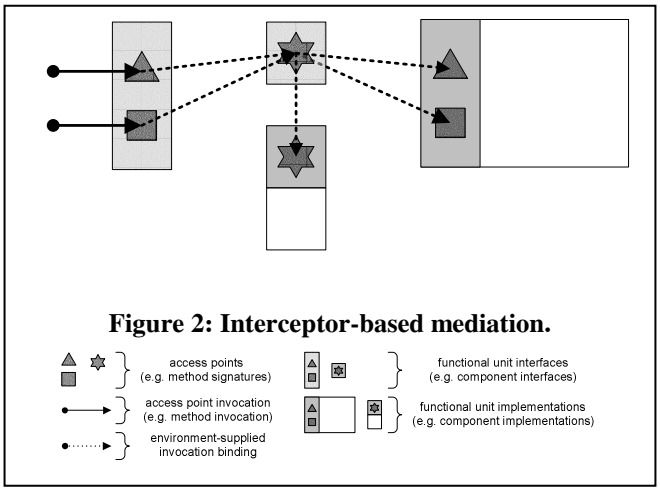


Figure 2: Interceptor-based mediation.

2.2. Interceptor-Based Mediation

The second broad class of mediation requires special runtime support, and relies on the notion of an *interceptor*. Like a wrapper, an interceptor is a functional unit that encapsulates the code interposed between an interface and an implementation. Unlike wrappers, however, every interceptor implements an identical interface. The interface of an interceptor accepts marshaled details, passed by the execution environment, about calls to targets and their corresponding completions. That is, interceptors capture invocation requests and responses.

When an interceptor is bound to a target, client calls through the target's interface are trapped by the execution environment. Details about the call, such as method name and argument values, are marshaled, and then passed to the appropriate interceptor(s). Similarly, details about the completion of the call, such as method name and return value, are likewise marshaled and passed to the appropriate interceptor(s). As interceptors operate on information about calls and returns, they can be viewed as meta-level functional units that modify the behavior of the functional-units to which they are applied.

Consider once again the invocation logging example. Since interceptors are passed detailed call information before invocations begin, and detailed result information after invocations complete, they offer an ideal solution. A logging interceptor can record invocation request and response details before and after each invocation, and can be transparently bound to arbitrary targets. Interceptor-based mediation is illustrated in Figure 2.

2.3. Approach Comparisons

Mediation offers the opportunity to transparently modify the behavior of existing systems. In deciding which class of mediation to use, we consider the advantages and disadvantages of each.

While it is possible for a wrapper to be applied to multiple targets, it can only be applied to targets with compatible interfaces. That is, a wrapper cannot be applied to a target that implements an interface which it itself does not implement. Moreover, wrappers do not cleanly encapsulate cross-cutting concerns. In the logging scenario, for example, each method in the wrapper implements nearly identical logging code. Wrappers do, however, have the advantage of providing type-safe access to invocation request and response information since they perform invocations on the target directly.

Interceptors do not afford the benefits of type-safety since they operate on a marshaled representation of invocation details. There is also a performance penalty due to the computation required to create these marshaled representations. They do, however, provide a number of advantages over wrappers. In particular, since interceptors do not make assumptions about the interfaces supplied by targets, they can be bound to a wide range of target instances, increasing their reusability potential. Additionally, since interceptors are meta-level functional units, they can conveniently encapsulate a wide range of cross-cutting concerns. In the logging scenario, for example, the code required to log method invocations is encapsulated within a single interceptor that can be bound to any target, regardless of the target's type. As interceptor-based mediation is capable of affecting cross-cutting modifications, and more closely meets the desiderata outlined in Section 1, DRSS relies on interceptors as the fundamental unit of change.

3. DRSS Design

The DRSS architecture extends the benefits of dynamic reconfiguration to distributed, component-based software, and fulfills the desiderata outlined in Section 1. While the prototype implementation uses C# as the implementation language, and the associated Common Language Runtime as the execution environment, the design does not rely on language constructs or runtime features specific to .NET. The DRSS design can be realized in any language that treats interfaces[2] as first-class constructs, and provides proper support for reflection, dynamic class loading, and remote communication. In short, DRSS can be implemented in most modern object-oriented languages, including C#, Visual Basic .NET, SmallTalk, and Java, to name a few.

3.1. The DRSS Platform

One of the primary desiderata guiding the design of the architecture was to prevent imposing significant constraints on application developers. This is achieved by factoring the reconfiguration logic out of individual components, and providing the support through a *component container*. A component container, like those used to host Enterprise Java Beans [6], is a runtime environment designed to manage the execution of component instances, and provides a set of common services to the instances it hosts. Depending on the services offered, components are imbued with persistence, rollback recovery, distributed transaction support, etc., just by virtue of executing within the container.

The *DRSS platform* provides dynamic reconfiguration services to its hosted instances, without the components having been explicitly designed to support dynamic reconfiguration. The runtime reconfiguration services provided by the platform include the ability to dynamically add, remove, and replace component implementations, as well as the ability to dynamically deploy cross-cutting modifications, while managing the scope of their effect.

The platform is implemented as a standard component type, an instance of which must be declared and instantiated by any DRSS-based application. As part of its interface, the platform provides methods for registering component instances to be hosted. Hosted instances are accessible to remote processes, and become dynamically reconfigurable without additional programming effort. The only requirement on the components is that they separate their interface from their implementation. As this is considered best practice in every modern design discipline, the requirement will already be met by most modern component designs.

As part of its initialization process, the DRSS platform registers itself as one of the instances to be hosted. This self-referential registration serves two purposes. First, it exposes the interface of the DRSS platform to remote processes. This exposure enables the development of remote management applications that administer the reconfiguration of DRSS-based applications. Second, and more important, self-referential registration extends the reconfiguration services offered by the platform to the platform itself. The DRSS platform is *open* in that it is itself reconfigurable; the set of services it provides can be extended dynamically. Both the container, and the instances it hosts can dynamically adapt to future design concerns.

3.2. The DRSS Call Model

To enable interceptor-based mediation, the DRSS platform introduces additional levels of indirection between clients and targets, and provides the request and response marshaling required of any environment supporting this class of mediation[3]. The implementation is similar to that used by Java RMI [7], CORBA [8], and other commercial middleware implementations. Unlike existing middleware, however, the primary reason for introducing indirection and marshaling in DRSS is to enable interceptor-based mediation.

Indirection and marshaling are achieved by introducing additional components on either side of every component collaboration. These additional components, referred to as *proxies* and *skeletons*, are generated dynamically by the DRSS platform, and serve as intermediaries between clients and targets. Clients invoke target methods through proxies, which transmit the requests to corresponding skeletons. The receiving skeletons handle each request on their respective target instance, and transmit the results back to the sending proxy, which then notifies the initiating client, who is unaware of the additional levels of indirection.

A skeleton component is generated at the point of target registration, and maintains a reference to the newly registered target. The skeleton provides only a single method, which accepts a marshaled representation of a client call on the referenced target. When the skeleton receives this *invocation request*, the request is unmarshaled, and the skeleton performs the corresponding target invocation. The results of the invocation are then marshaled into an *invocation response*, which is then transmitted back to the proxy.

A proxy component is generated at the point a client requests a reference to a registered target, regardless of whether or not the target is hosted by a remote platform.

[2] Note that pure abstract base classes, like those offered by C++, are semantically equivalent to interfaces.

[3] This indirection and marshalling introduces call overhead that must be taken into account when designing reconfigurable applications. In practice this concern is easily dispatched, as the number of components that might be reconfigured is typically small.

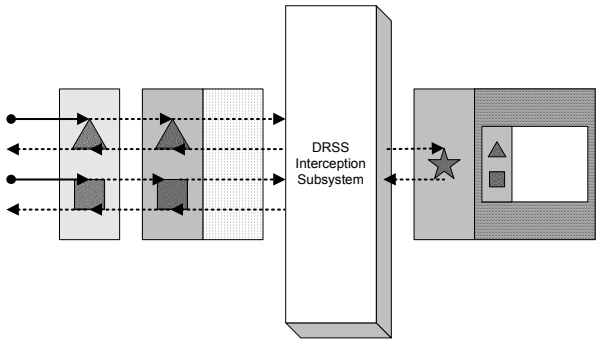
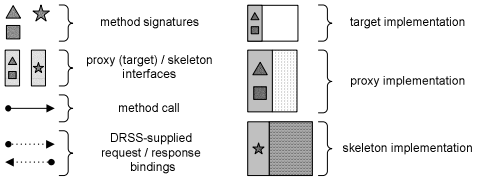


Figure 3: Proxies and Skeletons in DRSS.



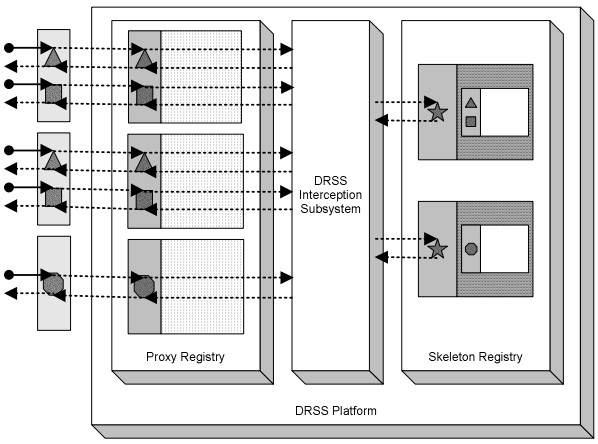
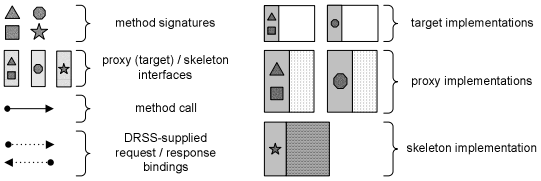


Figure 4: The DRSS Platform.



Rather than returning a reference to the target directly, which would of course not be possible for remote targets, the DRSS platform returns a reference to the newly-generated proxy. From the client's perspective the proxy is indistinguishable from the actual target, and serves as a surrogate that generates a representation of each client call, transmitting the resulting invocation requests to the appropriate skeleton. After transmitting each request, the proxy waits to receive the corresponding invocation response before notifying the waiting client.

Although it is convenient to think of proxies and skeletons as communicating directly, invocation requests and responses are actually transmitted by means of the *DRSS interception subsystem*, which serves as a conduit between proxies and skeletons. Before a request or a response is passed to the subsystem for transmission, it is placed within an *envelope* that contains the globally unique addresses of the envelope's source and destination, each of which correspond to an active proxy or skeleton. The interception subsystem uses this information to route requests and responses to their intended destinations. The DRSS call model, as implemented by proxies and skeletons, is illustrated in Figure 3.

3.3. The Proxy and Skeleton Registries

In the general case, DRSS hosts a number of target instances, and additionally provides management services to the associated proxies and skeletons. DRSS maintains these platform-generated components in the *proxy registry* and *skeleton registry*, respectively. The membership of these registries is handled by the platform, which automatically registers each proxy and skeleton at the point of creation, and removes them when they are no longer required. More specifically, each proxy component is removed when the number of clients using the proxy reaches zero. Each skeleton component is removed when its corresponding target instance is unregistered by invoking the appropriate platform method.

In addition to managing their membership, the platform provides methods for clients to query the proxy and skeleton registries. Clients can use these methods to enumerate the registrants, determine the target associated with a particular proxy or skeleton, or conversely, to determine the proxies and skeleton associated with a particular target. A management component might, for example, use these methods to determine whether components of a particular type are registered with a remote platform. Once particular target instances have been identified, the manager might determine the clients using those instances by querying the associated proxies. A high-level view of the proxy registry, the skeleton registry, and their relationship to the DRSS platform are illustrated in Figure 4.

3.4. Basic Reconfiguration Services

While the primary motivation for introducing proxies and skeletons between component collaborations is to enable interceptor-based mediation, the indirection they provide offers further benefits. Since clients do not reference component implementations directly, new target implementations can be substituted dynamically for existing target implementations without disrupting the services supplied to component clients.

Module substitution is achieved by temporarily blocking invocation requests to an existing target's skeleton, and setting the skeleton's target reference to a new (perhaps dynamically-loaded) target instance. If the component being replaced is stateful, the abstract state of the old target may need to be transferred to the new target. This is achieved by allowing *hot-swappable* components to optionally provide a method for serializing their abstract state, and a corresponding method for de-serializing their abstract state. If present, these methods are used by the platform to perform the appropriate state transfer before unblocking invocation requests to the corresponding skeleton. The design of the state transfer facility leverages work presented in [9] and [10].

The methods provided by the platform for dynamically adding, removing, and replacing component instances enable DRSS-based applications to adapt to unanticipated design concerns without bringing the applications down for maintenance. These component-level approaches are limited, however, in that they can only handle future concerns that respect existing component boundaries. To handle unanticipated cross-cutting concerns, DRSS relies on fine-grained interceptor-based mediation.

3.5. Interception in DRSS

Interceptors in DRSS operate on the envelopes transmitted between proxies and skeletons by the interception subsystem. Each of these *envelope interceptors* provides a method that accepts an envelope as argument, and provides additional behavior in the path of one or more component collaborations. The interceptors are associated with the platform-generated proxies and skeletons, providing per-client and per-target scoping. That is, interceptors associated with a particular skeleton modify the behavior of the target methods, regardless of which client performs the invocations. Interceptors associated with a proxy, on the other hand, modify the behavior of the target methods only when the methods are invoked through that particular proxy. This enables components to evolve uniformly in response to global requirements, or to evolve individually in response to client-specific requirements.

To provide additional control over the code interposed between component collaborations, DRSS differentiates envelope interceptors as being either *send-type* or *receive-type*. Send-type interceptors are invoked immediately after an envelope is passed from a proxy or a skeleton to the interception subsystem for delivery. Analogously, receive-type interceptors are invoked immediately before an envelope is delivered by the interception subsystem to a proxy or a skeleton. Subdividing interceptors into these two classes decouples the code interposed before an invocation from the code interposed after an invocation, enabling independent control over pre and post-processing logic.

Envelope interceptors are also components, and are composable to enable code and reasoning reuse at the interceptor level. As every interceptor implements a common interface, this is easily achieved by composing interceptors as chains. An *interceptor chain* is an ordered sequence of interceptors that implements the standard interceptor interface. That is, each chain is treated as a single interceptor. When an interceptor chain receives an envelope, it sequentially invokes each interceptor in the chain, providing the composite behavior of the interceptors it contains.

Analogous to the classification of individual interceptors, interceptor chains are classified as either send-type or receive-type, depending on the type of interceptors they contain. The platform binds one chain of each type to every proxy and skeleton that it creates, based on binding requests provided by the platform client. The initial chain bindings for each skeleton are specified at the point of target registration, and the initial bindings for each proxy are specified each time a reference to the corresponding target is requested. Default bindings are supplied if specific binding requests are not provided. The basic chain model is illustrated in Figure 5.

3.6. Chain Sharing

While every proxy and skeleton is associated with a pair of chains, the chains need not be created on a per-proxy or per-target basis. Associating unique interceptor chains with every proxy and skeleton sets a narrow scope over the effects of interception; modifying one interceptor chain directly modifies the behavior of only a single client or target. While a narrow scope can be desirable, the approach makes it difficult to uniformly evolve a group of clients and targets, and hinders the deployment of modifications that cross-cut component instances. To accommodate a wide range of evolution scenarios, the DRSS platform provides flexible scoping over the effects of interception by allowing interceptor chains to be shared among proxies and skeletons.

The instances that share a particular interceptor chain form an *evolution group*. As send-type and receive-type chains are shared independently, every proxy and skeleton is a member of exactly two evolution groups. Any modifications to a shared chain directly affect all members

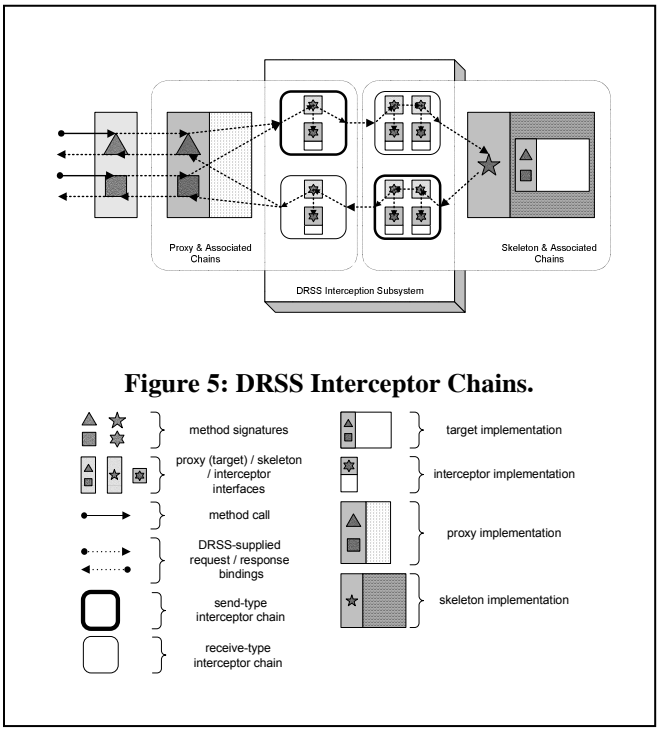


Figure 5: DRSS Interceptor Chains.

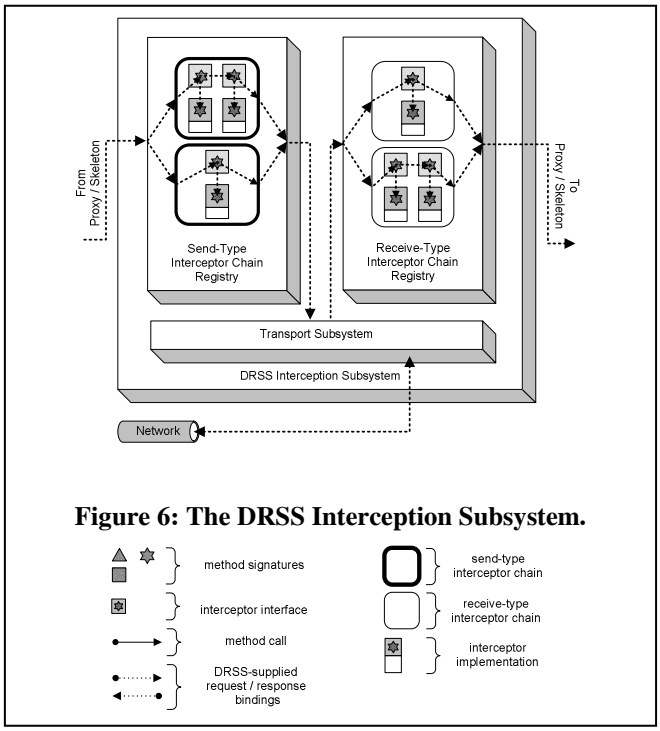


Figure 6: The DRSS Interception Subsystem.

of the associated group, typically providing uniform evolution across all instances. Once again considering interceptors as meta-level components, evolution groups are similar to meta-classes in ABCL/R [11], meta-groups in ACT/R [12], and meta-spaces in Apertos [13]. Similar group-based scoping mechanisms are found in other reflective languages and architectures.

3.7. The Interception Subsystem

Interceptor chains are managed by the interception subsystem, which maintains the chains in two separate registries depending on their type. The subsystem provides methods, exposed to clients through the interface of the platform, for administering these chains, as well as their bindings to proxies and skeletons. With respect to chain management, the subsystem provides methods for enumerating the managed chains, creating new chains, removing unbound chains, and managing the individual interceptors they contain. Clients can use the latter methods to enumerate the interceptors contained in each chain, and to insert new (perhaps dynamically-loaded) interceptors into, as well as remove interceptors from, individual interceptor chains. Binding management methods allow clients to enumerate the chain bindings, as well as to atomically unbind and rebind chains to individual proxies and skeletons, allowing the instances to migrate between evolution groups. Similar to the synchronization required during module substitution, the interception subsystem temporarily blocks access to the appropriate chains while the chains or their bindings are being updated.

When the interception subsystem receives an envelope for delivery, either from a proxy or a skeleton, it first invokes the appropriate send-type interceptor chain. If the delivery is not canceled by any interceptor in the chain[4], the envelope is eventually passed to the *transport subsystem*. As the name suggests, the subsystem provides a routing and transport service, routing envelopes between remote platforms. When the interception subsystem receives an envelope from the transport subsystem, it invokes the appropriate receive-type interceptor chain before delivering the processed envelope to its intended destination[5]. The interception subsystem is illustrated in Figure 6.

3.8. Interceptor Communication

Send-type and receive-type interceptors often collaborate to provide their services. In these cases, send-type interceptors may have to transmit additional information to collaborating receive-type interceptors. In a distributed environment, for example, a send-type interceptor might need to transmit a caller's local clock value, or perhaps a public key for authentication purposes. DRSS

[4] An interceptor cancels the delivery of an envelope by invoking the appropriate method on the envelope component. Once an envelope has been canceled, it will not be forwarded to further interceptors in the chain.

[5] When the interception subsystem invokes the appropriate interceptor chains, it additionally passes references to the local platform and transport layer. These components are directly accessible to interceptor implementations.

accommodates this requirement by providing a stack of content within each envelope. In the default case, the topmost entry on this stack is an invocation request or an invocation response. Send-type interceptors can, however, piggy-back content by pushing additional entries on the content stack. Corresponding receive-type interceptors, which expect particular content on top of the stack, pop the piggy-backed entries off before allowing control to pass to the next interceptor in the chain.

DRSS provides basic protection against unexpected content mismatches due to incompatible send-type and receive-type chains. Send-type interceptors are required to declare the format of the content (if any) that they add to the stack. Similarly, receive-type interceptors are required to declare the format of the content (if any) that they remove from the stack. The *content signature* of a chain is the concatenation of the content declarations of its constituent interceptors. When an envelope is processed by a send-type chain, the interception subsystem marks the envelope with the corresponding content signature. Before the interception subsystem passes an envelope to a receive-type chain, it verifies that the content format expected by the chain matches the content contained in the envelope[6]. In the event of a mismatch, the interception subsystem bypasses the chain, and places the envelope in an associated *mismatch queue*. Whenever that chain is reconfigured, the subsystem checks to see whether any of the stalled envelopes can be delivered under the new configuration.

By adopting a discipline of reconfiguring send-type chains before reconfiguring receive-type chains, temporary type mismatches can be accommodated. Envelopes sent under new send-chain configurations will be stalled until the corresponding receive-type chains have been properly configured. This approach yields logical atomicity when updating a pair of distributed chains.

4. DRSS Applications

Mediation has been used in a number of contexts to non-invasively extend the functionality of existing software systems. The approach has been used to transparently provide the benefits of distribution at the operating system [14] and file system levels [15], as well as to provide transparent file versioning control [16]. With respect to software assurance, mediation has been used in container architectures to provide invariant-based state monitoring [17], and in middleware to provide fault tolerance through component replication [18]. DRSS is designed to handle all these mediation tasks, and to do them dynamically.

[6] The content signatures need not match exactly. If the content signature contained in the envelope is a proper suffix of the receive-type chain's signature, the interception subsystem will bypass the appropriate interceptors in the chain.

We have a particular interest in fault tolerance, and have begun to use DRSS for that purpose. By way of illustration, we describe our use of DRSS for fault tolerance in a home networking project.

4.1. The Aladdin Project

The Aladdin project [19] aims to provide dependable and extensible support to heterogeneous, unreliable devices in unreliable home networks. At the heart of the system is a lookup service that maintains information about the various device and controller objects currently in the network, and responds to queries about the availability of these objects. The information is maintained using *soft state* – state with a limited lifetime that disappears unless that lifetime is refreshed.

Providers are objects that periodically announce themselves to the lookup service. Each announcement refreshes their state at the lookup service and extends the lifetime of that state. Subscribers are objects that query the lookup service for providers and, when found, access the providers directly.

Because the system is unreliable, the lookup service can fail. To demonstrate the applicability of DRSS for adding fault tolerance to a system, the lookup server was replicated, and interceptors were inserted to provide stabilizing fault tolerance [20]. When faults occur, the system may exhibit incorrect behavior temporarily, but is guaranteed to eventually operate correctly.

In a good state, a single lookup server has the status of being leader, and is accessible to the other objects. All the lookup servers accept refresh messages from providers, but only the leader responds to subscribers; the non-leaders are *hot spares*, ready to be put into service when needed. The faults tolerated are the failure of a lookup server, network partition, restart and repair of a lookup server, and inconsistent states in which there are multiple leaders or no leaders at all.

The use of interception proves to be a natural solution. The code for the underlying system does not depend on the presence of DRSS. In particular, no lookup server replica or other object is aware of which replica is the leader. Interceptors elect a leader from among the available lookup servers, multicast provider refresh messages to all replicas, and direct subscriber inquiries to the current leader. Although we had access to the source code of the underlying system, we did not need it. We were able to make the system fault tolerant using only interceptors.

We could have handled this problem using wrappers instead of interceptors. However, doing so would have restricted future adaptability since a wrapper only works with a specific component. Since Aladdin is designed to be extensible, we would like to reuse the multicast and leader

election interceptors with other components. The inherently generic nature of interceptors makes this possible.

4.2. Reusable Interceptor Library

One of the aims of Aladdin is extensibility, so in the course of the project we developed a library of reusable interceptors.

- **Multicast interceptor.** This interceptor transmits a single message to a designated set of components.
- **Stabilizing leader election interceptors.** These interceptors elect a leader from among those components with which they are associated.
- **Message loss fault interceptor.** This interceptor periodically causes a message to be discarded, simulating the loss of a message in a network.
- **Message reordering fault interceptor.** This interceptor periodically buffers some number of messages and transmits them in a random order.
- **Message argument reordering fault interceptor.** This interceptor periodically inspects a message and checks for two or more arguments of the same type. If found, they are reordered.
- **Distributed recording interceptors.** This consists of a send interceptor and a receive interceptor. The objective is to record logically sequential events using vector clocks [21]. The log shows what happened in the system and in what order, and can be used for debugging or replay. The interceptors maintain a vector clock at each intercepted component. When the send interceptor receives a message (to be sent), it updates the local logical clock, adds the vector to the message, and logs the vector clock and message. When the receive interceptor receives a message (to be delivered), it strips off the vector clock, updates the local logical clock, and logs the local logical clock and message.

Since the fault interceptors must be parameterized, we developed a fault injection framework to customize the interceptors before they are added to a running system. Although this library and framework were developed in support of our fault tolerance objectives, similar libraries and frameworks can of course be developed for other purposes.

5. Related Work

Software mediation has proven useful in a number of contexts. As we have presented the design and implementation of a general purpose mediation-based architecture, we focus the remainder of the discussion on tools offering similar functionality. We first present an overview of the more important static approaches, and then consider tools offering dynamic support.

The additional indirection and marshaling behavior required to realize interceptor-based mediation comes standard as part of most distributed middleware implementations. CORBA, for example, includes support for *portable interceptors* as part of its core specification. The allowable interceptor behaviors are constrained, however, in that they cannot alter argument values, nor can they directly respond to invocation requests [22, 8]. Interceptor composition is also under-specified, so the results of composition can be unpredictable [23]. Moreover, dynamic insertion and removal of interceptors is not supported, precluding dynamic adaptation to unanticipated evolution scenarios.

The Java RMI specification does not directly support interception, although implementation mechanisms have been proposed [24, 25]. These proposals have been focused on implementation mechanics, however, and do not consider interceptor composition, or dynamic deployment. Microsoft's .NET Remoting [26] directly supports interceptors and interceptor chains[7]. Interceptors can be statically composed, but they cannot be deployed dynamically.

The Unix operating system and its variants provide mechanisms for capturing system calls and their completions. A number of tools have been developed which leverage this facility to mediate user level calls to the operating system [27, 28]. Similar tools have been developed based on binary-level modification of target images, primarily aimed at Win32 operating systems [29, 30]. Both sets of tools offer compositional reuse of interposed code. Additionally, because they operate at the system call and binary level, respectively, they can be used to mediate calls to arbitrary applications, regardless of how those applications were developed. Dynamic deployment is also possible, though the approaches do not offer mechanisms for blocking access to targets during reconfiguration: applications using the mediated targets must not access the targets during reconfiguration. Because they are variants of the wrapper-based approach, the approaches cannot handle cross-cutting concerns, nor do they provide services for scalable deployment and management.

The work presented in [31] describes an interception architecture for E``Speak middleware designed to support runtime adaptation to evolving quality of service (QoS) requirements. Since the architecture is designed to support runtime evolution, it is similar to the DRSS model. The QoS architecture supports compositional reuse of interceptors, dynamic interceptor insertion and removal, and provides remote management facilities. The architecture does not, however, support flexible scoping: all

[7] The .NET Remoting documentation refers to interceptors and interceptor chains as sinks and sink chains, respectively.

clients share a single interceptor chain, as do the target components. In addition, the underlying middleware, E``Speak, is not generally available, so the implementation is of limited practical use.

The work presented in [32] describes a meta-level architecture for building highly-available systems, and shares a number of similarities with DRSS. The architecture supports compositional reuse of meta-level dependability objects, dynamic meta-object insertion and removal, and provides remote meta-level management facilities. The work does not, however, appear to support flexible scoping. Additionally, the approach relies on an academic computation model, language, and execution environment, making the implementation of limited practical use.

6. Concluding Remarks

In designing our dynamic reconfiguration facilities, we identified the following key desiderata: accommodation of unanticipated cross-cutting concerns, scalability, support for compositional reuse, and accessibility to practitioners. An evaluation of alternatives indicated that an approach based on dynamic interception meets these desiderata. A dynamic approach is required since static approaches impose an enormous (if not impossible) burden on the developer of a complex, long-running system. Interceptor-based mediation is required since wrapper-based solutions preclude accommodation of cross-cutting concerns and tend not to be scalable. Our design for DRSS is dynamic and interceptor-based, satisfying these desiderata.

Future work aims to enhance DRSS in a number of ways. First and foremost, we plan to develop a proper security model. At present there are no restrictions on which components can access reconfiguration methods. Hence a malicious component could reconfigure a running system to change behavior or reveal confidential information. We plan to enforce security policies that restrict the invocation of reconfiguration methods to trusted components. Second, we plan to improve the efficiency of remote communication. At present, DRSS uses an ad-hoc and inefficient middleware implementation; we plan to replace this implementation with .NET Remoting. Finally, we plan to develop a robust front-end for administering DRSS-based applications.

We also plan to build several tools using DRSS, continuing our interest in fault tolerance.

- We plan to develop libraries and frameworks for fault detection and correction, and to extend our library and framework for fault injection. This will let us quickly configure detectors and correctors for a particular system, and to inject faults to test them.
- We plan to use DRSS to develop a framework for continuous testing of running systems. The framework will provide facilities to divert spare resources for testing, quickly releasing those resources back into service should system demand increase. Faults will be injected into the test partition of the system, allowing us to evaluate tolerance components in a live environment. In connection with this project, we also plan to develop a library of support interceptors. These include visualization interceptors to give a graphical view of system operation, advanced logging interceptors, etc.
- We plan to develop a general-purpose management system that can be used to monitor and tune a running system. This system will monitor faults, adjust the rate of detection according to the frequency of faults, and remove from service those components believed to have permanent faults. It will also be responsible for managing the continuous testing mentioned above.

In conclusion, DRSS is a flexible and scalable approach to maintenance of complex, long-running systems whose requirements and environments cannot be fully specified in advance.

Acknowledgements

Throughout this project we have received valuable feedback and development support from Anish Arora's Dependable Distributed and Networked Systems group. We would especially like to thank Mariana Barca for the design and implementation of the DRSS Aladdin project, Vishvesh Sahasrabudhe for the design and implementation of the fault injection framework, and Michael Gibas for the design and implementation of the distributed recording service. Thanks also to the anonymous reviewers for their useful feedback in improving the overall presentation.

This work was supported in part by an unrestricted grant from Microsoft Research, for which the authors are grateful. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors, and do not necessarily reflect the views of Microsoft Research.

REFERENCES

[1] Parnas, D. L., *On the Criteria to be used in Decomposing Systems into Modules*, Communications of the ACM (December 1972), 1053—1058

[2] Shalloway, A., Trott, J., *Design Patterns Explained: A NewPerspective on Object-Oriented Design*, Addison-Wesley, 2001

[3] Gibbs, W., *Software's Chronic Crisis*, Scientific American, 86-95, September 1994

[4] Humphrey, Watts S., *The Future of Software Engineering: I*, news@sei interactive, 1st Q. 2001, at http://interactive.sei.cmu.edu/news@sei/columns/watts_new/2001/1q01/watts-new-1q01.htm

[5] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable OO Software*, Addison-Wesley, 1995

[6] Sun Microsystems, *J2EE 1.3 Specification*, July 2001, available at http://java.sun.com/j2ee/download.html

[7] Sun Microsystems, *Java Remote Method Invocation Specification 1.3*, December 1999, available at http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html

[8] Object Management Group, *Common Object Request Broker Architecture: Core Specification 3.0.2*, December 2002, available at http://www.omg.org/technology/documents/formal/-corba_iiop.htm

[9] Rodrigues, R., Castro, M., Liskov, B., *BASE: Using Abstraction to Improve Fault Tolerance*, Proceedings of the 18th Symposium on Operating Systems Principles (SOSP), October 2001

[10] Sridhar, N., Pike, Scott M., Weide, Bruce W., *Dynamic Module Replacement in Distributed Protocols*, International Conference on Distributed Computing Systems (ICDCS), 2003

[11] Watanabe, T., Yonezawa, A., *Reflection in an Object-Oriented Concurrent Language*, Proceedings of OOPSLA'88, Vol. 23 of ACM SIGPLAN Notices, 306-315, ACM Press, 1988

[12] Watanabe, T., Yonezawa, A., *An Actor-Based Meta-Level Architecture for Group-Wide Reflection*, Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages, de Bakker, J.W., de Roever, W.P., Rozenberg, G., editors, LNCS 489, 405-425, Springer-Verlag, 1990

[13] Yokote, Y., *The Apertos Reflective Operating System: The Concept and Implementation*, Proceedings of OOPSLA'92, Vol. 28 of ACM SIGPLAN Notices, 414-434, ACM Press, 1992

[14] Alexandrov, A., Ibel, M., Schauser, K., Scheiman, C., *Extending the Operating System at the User Level: the Ufo Global File System*. Proceedings of the USENIX Annual Technical Conference, 1997

[15] Dasgupta, P., Karamcheti, V., Kedem, Z.M., *Transparent Distribution Middleware for General Purpose Computations*, International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), 1999

[16] Korn, D.G., Krell, E., *A New Dimension for the UNIX File System*, Software Practice and Experience, 20(S1):19-34, June 1990

[17] Vecellio, G.J., Thomas, M.M., Sanders, R.M., *Container Services for High Confidence Software*, Workshop on Component-Oriented Programming (WCOP), 2002

[18] Narasimhan, P., Moser, L. E., Melliar-Smith, P.M., *Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance*, Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems, 81-90, 1997

[19] Wang, Y.M., Russell, W., Arora, A., Xu, J., Jagannathan, R., *Towards Dependable Home Networking: An Experience Report*, International Conference on Dependable Systems and Networks (ICDSN), 2000

[20] Dijkstra, E.W., *Self-Stabilizing Systems in Spite of Distributed Control*, Communications of the ACM, 17(11), 1974

[21] Ricart, G., Agrawala, A., *An Optimal Algorithm for Mutual Exclusion in Computer Networks*, Communications of the ACM, 24(1):9-17, 1991

[22] Baldoni, R., Marchetti, C., Verde, L., *CORBA Request Portable Interceptors: Analysis and Applications*, Concurrency and Computation: Practice & Experience, 2002

[23] Wegdam, M., van Halteren, A., *Experiences with CORBA Interceptors*, Workshop on Reflective Middleware, co-located with Middleware 2000, 2000

[24] Narasimhan, N., Moser, L.E., Melliar-Smith, P.M., *Interceptors for Java Remote Method Invocation*, Concurrency and Computation: Practice and Experience, 13:755-774, August 2001

[25] Santos, N., Marques, P., Silva, L, *A Framework for Smart Proxies and Interceptors in RMI*, Proceedings of The 15th International Conference on Parallel and Distributed Computing Systems (PDCS), Louisville, 2002

[26] Microsoft Corporation, *.NET Remoting Overview*, 2002, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconnetremotingoverview.asp

[27] Jones, M. B., *Interposition Agents: Transparently Interposing User Code at the System Interface*, 14th ACM Symposium on Operating Systems Principles (SOSP), 1993

[28] Krell, E., Krishnamurthy, B., *COLA: Customized Overlaying*, Proceedings of Winter USENIX, 3-8, 1992

[29] Balzer, R., Goldman, N., *Mediating Connectors*, Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS) Workshop, 73-77, 1999

[30] Hunt, G., Brubacher, D., *Detours: Binary Interception of Win32 Functions*, Proceedings of the 3rd USENIX Windows NT Symposium, 1999

[31] Pruyne, J., *Enabling QoS via Interception in Middleware*, HP Laboratories Technical Report (HPL-2000-29), February 2000

[32] Agha, G., Sturman, D.C., *A Methodology for Adapting to Patterns of Faults*, Koob, G.M., Lau, C.G., editors, Foundations of Dependable Computing: Models and Frameworks for Dependable Systems, chapter 1.2, Kluwer Academic Publishers, 1994