Separating Introspection and Intercession to Support Metamorphic Distributed Systems [*]

E. P. Kasten, P. K. McKinley, S. M. Sadjadi, and R. E. K. Stirewalt

Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824
{kasten,mckinley,sadjadis,stire}@cse.msu.edu

Abstract

Many middleware platforms use computational reflection to support adaptive functionality. Most approaches intertwine the activity of observing behavior (introspection) with the activity of changing behavior (intercession). This paper explores the use of language constructs to separate these parts of reflective functionality. This separation and "packaging" of reflective primitives is intended to facilitate the design of correct and consistent adaptive middleware. A prototype language, called Adaptive Java*, is described in which this functionality is realized through extensions to the Java programming language. A case study is described in which "metamorphic" socket components are created from regular socket classes and used to realize adaptive behavior on wireless network connections.*

Keywords: adaptive middleware, reflection, component design, mobile computing, wireless networks, forward error correction.

1 Introduction

Increasingly, distributed applications are required to adapt to their environment during execution. This need arises partly from the emergence of a dynamic and heterogeneous mobile computing infrastructure, and partly from users who expect the Internet to provide the same, or higher, quality of service as found in telephone and cable television networks. To meet (or even approach) these expectations, distributed software must adapt to its environment in several dimensions. For example, communication software must accommodate wireless networks that are far less reliable and stable than their wired counterparts. In addition, user interfaces must conform to devices with widely varying display characteristics and capabilities, from conventional workstations to palmtop devices. Moreover, many applications must provide their own fault tolerance capabilities, given the reliance on commodity operating systems and hardware resources, both inside the network and at its edge. Finally, applications must confront the vulnerability of a connectionless packet infrastructure by protecting themselves against intrusions and other security threats.

Adaptability can be implemented in different parts of the system. One approach introduces a layer of adaptive *middleware* between applications and underlying transport services [3, 6, 9, 17]. An appropriate middleware platform can help to insulate application components from platform variations and changes in network conditions and can simplify the implementation of fault tolerance and security services. Many approaches to the design of adaptive middleware involve computational reflection [10, 16], which refers to the ability of a computational process to reason about (and possibly alter) its own behavior. Typically, the *base-level* functionality of the program is augmented with one or more *meta* levels, each of which observes and manipulates the base level. In object-oriented environments, the entities at a meta level are called meta-objects, and the collection of interfaces provided by a set of meta-objects is called a meta-object protocol, or MOP.

In this paper, we propose a model for adaptive components that is designed to facilitate the construction and evolution of MOPs for different cross-cutting concerns: communication quality-of-service, fault tolerance, security, and so on. The model is based on the concept of providing separate component interfaces for observing behavior (*intro-*

[*] This work was supported in part by the U.S. Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, and in part by National Science Foundation grants CDA-9617310, NCR-9706285, CCR-9912407, EIA-0000433, and EIA-0130724.

spection) and for changing behavior (*intercession*). Therefore, a component's meta level contains two types of primitive operations: *refractions*, which provide a (limited) view of the underlying base-level component, and *transmutations*, which modify the functionality of the base-level component. This separation is intended to simplify the development of adaptive functionality by restricting the ways in which components can be manipulated, thereby helping to ensure correctness and consistency among different MOPs. We consider *metamorphic systems* as those that allow principled run-time intercession.

The remainder of the paper is organized as follows. In Section 2, we discuss the origins of reflection and its application to middleware. Section 3 describes the basic concept of separating reflection in order to support the development of metamorphic software. In Section 4, we describe a prototype implementation whereby we added several new constructs to the Java programming language; we refer to the prototype as *Adaptive Java*. Section 5 illustrates an example in which a normal Java socket class is transformed into a "metamorphic" socket that supports a variety of refractions and transmutations. Section 6 discusses related work in the adaptive middleware community, and Section 7 presents our conclusions and discusses future directions.

2 Background

Our interest in reflection arises from our work on the RAPIDware project [13], which addresses the design and use of adaptive middleware to protect critical infrastructures, such as power grids, financial systems, and command and control networks. Such systems require run-time adaptation in order to survive hardware failures, network outages, and security attacks. For many systems, an event as simple as increased packet loss on a wireless channel can trigger different responses, depending on the situation: increasing redundancy on a communication channel (quality of service); establishing communication on an alternative network interface (fault tolerance); invoking services for disconnected operation (mobile computing) and responding to possible malicious channel jamming (security). Reflective middleware offers a principled (as opposed to ad hoc) means to observe and modify base-level behavior [5], thereby facilitating the coordination of such responses.

A key issue that arises in the application of reflection to middleware platforms is the degree to which the system should be able to change its own behavior. A completely open implementation implies that an application can be recomposed entirely at run-time. In the extreme, all the default components of the system can be destroyed and new ones instantiated, such that the goal of the base-level computation is changed. (A spreadsheet can be recomposed as a video player!) On the other hand, limiting adaptability also limits the ability of the system to survive adverse situations.

We begin our investigation of this problem by focusing on the reflective interfaces exhibited by components. Rather than considering MOPs as orthogonal portals into base-level functionality [5], we consider an alternative model in which MOPs are constructed from a set of primitive operations, or *atoms*, that provide access to component behavior. As shown in Figure 1, while different MOPs are defined for different aspects of adaptive behavior (e.g., fault tolerance, security, quality-of-service, power consumption), they likely will overlap in their use of these atoms. This design appears to exhibit several desirable features. First, explicitly defining intersections in MOP functionality may facilitate coordinated adaptation to events. Second, additional MOPs can be constructed to address issues that did not arise in the original design. Third, limiting interaction with the base level may improve the ability of the system to check, at run-time, the consistency of modifications with the specified behavior of the component.

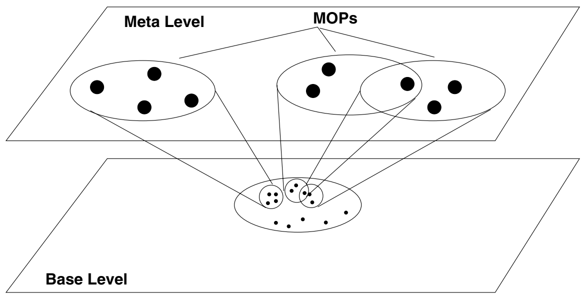

Figure 1. MOPs implemented with primitive operations.

In this paper, we propose an approach to defining and constructing such primitive meta-operations, based on whether the operation involves introspection or intercession. We point out that in her landmark paper on computational reflection [10], Maes introduces the following definition: *Computational reflection is the activity performed by a computational system when doing computation about (and by that possibly affecting) its own computation.*

Interestingly, many works on reflection reinforce the parenthetical implication in Maes' definition and combine introspection and intercession. By contrast, as shown in Figure 2, we view introspection and intercession as orthogonal operations, which are furthermore orthogonal to the underlying computation itself. The goal of the computation dimension is to fulfill the principle goal imbued by the designer. The introspection dimension enables the application to observe itself, while the intercession dimension enables the application to modify its own behavior and structure. In the remainder of the paper, we describe a component model based on this concept, as well as a prototype language that implements the model.

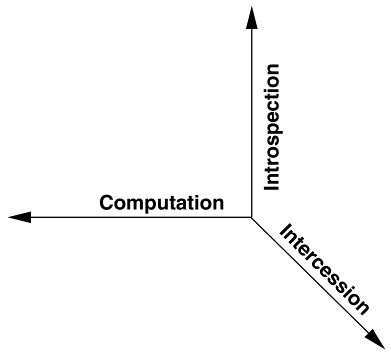

Figure 2. Dimensions of component behavior.

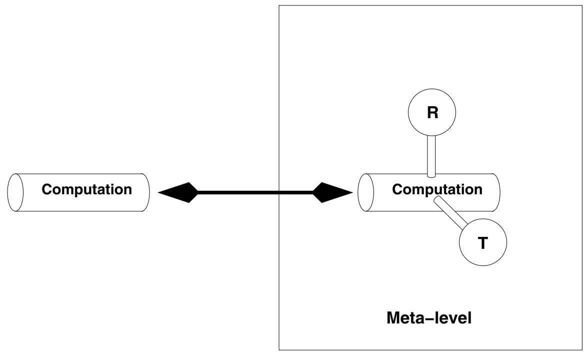

Figure 3. Basic Metamodel

3 Model of Adaptive Components

The basic building blocks used in our adaptive system are *components*. A component can be accessed through three interfaces corresponding to the three dimensions discussed above. Operations in the computation dimension are referred to as *invocations*. Operations in the introspection dimension are called *refractions*, since they offer only a partial view of internal structure and behavior. Moreover, refractions are not allowed to change the state or behavior of the component. Operations in the intercession dimension are called *transmutations*; they are used to modify the computational behavior of the component.

Refractive and transmutative interfaces are implemented by meta-components to support introspection and intercession. Figure 3 illustrates the structure implied by our understanding of these component interfaces. In this figure, the meta-level reflects the base-level computation, but also provides a set of refractions (R) and transmutations (T) that can be used to inspect and modify the base-level computation. For example, the base-level computation of a network socket includes methods to send and receive data packets. A refractive interface for an adaptive version of a socket might include operations to observe packet characteristics (size, frequency), while the transmutative interface might enable custom filtering or modification of arriving packets before delivery to the application.

The Role of Encapsulation. Most object-oriented languages are based on a static binding of inheritance between subclasses and superclasses. This structure prohibits dynamic restructuring of a program at run-time. For this reason, we adopt encapsulation as the principle mechanism for the composition in our system. Encapsulation provides a means by which the functionality of a component can be extended or limited by dynamically encapsulating it within another. Moreover, the addition, deletion and exchange of encapsulated components can be carried out dynamically at run-time. The composition of a system can be viewed as the parameterization of one component with another. We represent these relationships using a notation from GenVoca [2]. Specifically, S = G[F] states that system S is composed of component G parameterized by F, or that F is encapsulated by G. Multiple components can be encapsulated within another component, as represented by S = G[E,F].

Absorption. Components are constructed from classes defined in an object-oriented language (OOL). We used Java in our study. The process of constructing a component from an existing class is referred to as *absorption*. An object can be considered as a component instance that lacks refractive and transmutative capacity. That is, an object is essentially a black box that does not facilitate reflection.

Figure 4 illustrates the absorption of a class and the metafication of the resulting “base-level” component to support refractions and transmutations. As part of the absorption procedure, mutable methods called *invocations* are created on the base-level component to expose the functionality of the absorbed class. Invocations are mutable in the sense that they can be added and removed from existing components at run-time using meta-level transmutations.

The relationship between invocations on the base-level component and methods on the base-level class need not be one-to-one. Indeed, when a component is added to an adaptive system it may be necessary to modify the component’s interface such that it fits properly into the system structure. Since component interfaces are mutable and composed of primitive operations, augmentation of an existing interface is possible. However, some of the base-level methods may be occluded or even combined under a single invocation as the system’s form is modified. For example, we might create a base-level socket by absorbing a socket class. However, the base-level socket may provide a customized interface for use in a particular application domain.

Metafication. Metafication enables the creation of refractions and transmutations that operate on the base component, as shown in Figure 4. Refractions and transmutations

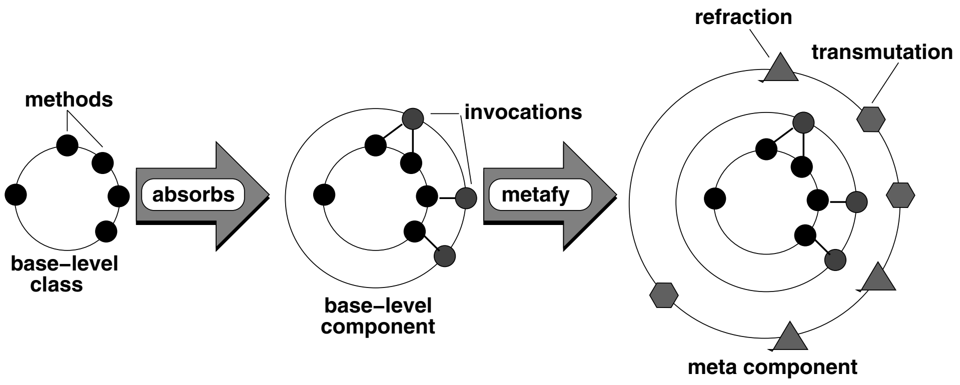

Figure 4. Component absorption and metafication.

embody limited adaptive logic and are intended for defining *how* the base level can be inspected and changed. The logic defining *why and when* these operations should be used is provided at other meta levels or by other components entirely. That is, a component may refract and transmute itself, or may be refracted and transmuted by another.

Implementing a metamorphic system using components creates a uniform self-representation in two different ways. First, except for the need of boot strapping from the underlying OOL classes, the system can be built with the uniform use of components. Second, both the base- and meta-levels are constructed using components. Thus, the meta-level can also be given a meta-level. This meta-meta-level can be used to refract and transmute the meta-level. In theory, this reification of meta-levels for meta-levels could continue infinitely [10]. Returning to our socket example, a meta-meta-level might include a transmutation that replaces a transmutation at the meta-level, in order to change the *way* in which filters are added to the base-level component.

4 A Prototype Language: Adaptive Java

In order to gain a better understanding of how the addition of refractive and transmutative elements to a language might affect its use and structure, we defined a prototype language, Adaptive Java, as an extension to Java. In this initial study, we simply used CUP [7], a parser generator for Java, to implement Adaptive Java. CUP takes our grammar productions for the Adaptive Java extensions and generates an LALR parser, called ajc, which converts Adaptive Java code into Java. Semantic routines were added to this parser such that the generated Java code could then be compiled using a standard Java compiler.

We emphasize that Adaptive Java is a prototype whose purpose is to improve our understanding of which language constructs and mechanisms are desirable in dynamic and adaptive languages. Eventually, we intend to reimplement the Adaptive Java grammar as a compiled language and extend the JVM to support needed dynamic constructs such as dynamic casting and immutable invocations and variables. In this section, we provide examples of the language constructs used to code refractive and transmutative software. Due to space limitations, details of the grammar and implementation are deferred to a technical report [8].

Basic Component Structure. The definition of a typical Adaptive Java component is similar to that of a Java class, as shown in Figure 5. Component objects are instantiated using the `new` operator. Constructors are essentially identical to Java constructors and are immutable, but provide flexibility in the initial instantiation of a component. Standard Java methods are replaced by *invocations* and standard immutable variable declarations are supplemented with mutable variable declarations. Mutable variables can be added to and removed from components, via transmutations at the meta-level, in much the same way as invocations.

```
/* A simple component */
component BasicComponent {
  /* Constructor */
  public BasicComponent() { ... }

  /* Invocation  */
  public invocation void
      method1(String arg) { ... }
            .
            .
            .
}
```

Figure 5. Adaptive Java component structure.

Optionally, a component can be declared to extend another component. Extending a component encapsulates the extended component within the newly declared component. The extended component is called the *inner component* whereas the extending component is called the *outer component*. Inheritance is simulated by examining the outer component's invocations for the desired invocation. If the invocation is not found, then a recursive search is performed of encapsulated components. For instance, let S = A[B[C]] be a system composed by extending component C with B and then extending B with A. If the execution of invocation `A.exec()` is requested, first component A, then B, and finally C will be searched for `exec()`. The first instance of `exec()` that is found will be executed, thus allowing inner invocations to be overridden by those found at more outer encapsulation levels. Simulating inheritance in this way allows the inheritance chain to be decomposed and recomposed with different components.

Absorbing Existing Classes. The `absorbs` keyword is used to construct a component from a regular Java class. Figure 6 shows the Adaptive Java code for absorbing a Java socket class into a socket component that can be used only to receive packets. Invocations are created to expose selected functionality of the absorbed class. In this example, only the `receive()` and `close()` methods are exposed. The absorbed class is accessed by the absorbing component through the `base` keyword. The other methods of the base class are hidden at this level.

```
/* receive-only socket component */
public component RecvSocket
  absorbs Socket
{
  /* constructor */
  public RecvSocket(int port,
    String group, byte ttl)
    throws UnknownHostException,
           IOException
  {
    setBase(new Recv(port, group, ttl));
  }

  public invocation void
    receive(DatagramPacket p)
    throws IOException
  {
    base.receive(p);
  }

  public invocation void close()
    throws IOException
  {
    base.close();
  }
}
```

Figure 6. Absorbing a class into a component.

Reifying a Meta-Level. Meta-components encapsulate other components and support only reflective functionality. The encapsulated component is called the base level. Meta components are declared using the `metafy` keyword. Figure 7 shows an example where we metafy the RecvSocket component defined in Figure 6. A transmutation is defined that changes the degree of data compression implemented by the socket. A simple refraction is defined to return the total number of bytes that have traversed the socket.

```
/* Meta receive-only socket component */
public component MetaRecvSocket
  metafy RecvSocket
{
  /* Constructor */
  public MetaRecvComponent(int port,
    String group, byte ttl)
    throws UnknownHostException,
           IOException
  {
    setBase(new RecvComponent(port,
            group,ttl));
  }

  /* Transmutation that sets the data
     stream compression level. */
  public transmutation void
    SetCompression(int level)
  {
            .
            .
            .
  }

  /* Refraction that returns the
     observed bytes transferred by
     the RecvSocket component.  */
  public refraction long GetBytesXmit()
  {
            .
            .
            .
     return bytes_transferred;
  }
}
```

Figure 7. Metafying a component.

Implementation Issues. Two implementation issues warrant further comment. First, all invocations (refractions, transmutations, and even computational invocations) are called using the `invoke` keyword. Shown below is an example for a RecvSocket component called rSock. This action causes the retrieval of a matching invocation object from a HashMap that is then cast to the appropriate invocation type. Although the use of the `invoke` keyword may seem somewhat complicated, this design enables an indirect binding for invocations, which in turn supports adaptation of component interfaces.

```
invoke rSock.receive(pckt);
```

A related issue is the immutability of refractions. Refractions, by definition, provide only viewing portals into the processing of the base level program. From the perspective of a refraction, the base level is immutable. Immutability comes in two basic flavors. An object that is *shallow* immutable disallows any *set* operations (e.g. assigning to a variable) or calls to base-level methods. Thus, only *get* operations, for example, reading a variable, are available to refractions. *Deep* immutability is a superset of shallow immutability that disallows calls to base-level methods that issue set operations or that call other methods that modify base-level behavior or structure. As of this writing, Adaptive Java supports only shallow immutability in refractions. However, we recognize that deriving algorithms and tools that can verify base-level methods as usable by refractions would help extend meta-level functionality. These types of algorithms could also be used for the automatic categorization of meta-level invocations as either refractions or transmutations, obviating the necessity of their explicit declaration. We are addressing these issues in our ongoing work.

5 Example: MetaSockets

In order to evaluate the design of the Adaptive Java language constructs, we used Adaptive Java to develop a component called a "metamorphic" socket, or simply, metasocket. In response to external events, an application or middleware platform can use refractions and transmutations on this component to observe and modify socket functionality. In this study, we used metasockets to enhance the quality of wireless audio channels at run time. Figure 8 shows the physical configuration of our experiments, where live audio is streamed from a workstation to multiple iPAQ handheld computers running Windows CE. The audio stream is transmitted on a 100 Mbps Ethernet LAN to a wireless access point, where it is multicast at 11 Mbps on an 802.11b wireless LAN.

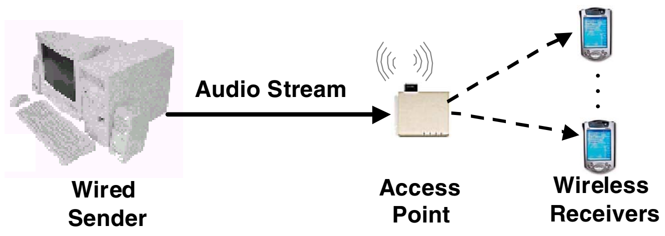

Figure 8. Physical experimental configuration.

The audio streaming application comprises two main parts. The Recorder uses the Java Sound API to read audio data from a workstation's microphone and multicast it on the network. The Player receives the audio data and plays it using the Java Sound API. Both applications were written in Adaptive Java and converted into pure Java using the ajc parser. They communicate using MetaSockets instead of regular Java sockets.

Error Control on Wireless Networks. The characteristics of wireless LANs are very different from those of their wired counterparts. Factors such as signal strength, interference, and antennae alignment produce dynamic and location-dependent packet loss [12]. These problems affect multicast connections more than unicast, since the 802.11b MAC layer does not provide link-level acknowledgements for multicast frames. Forward error correction (FEC) can be used to improve reliability by introducing redundancy into the data channel. For example, an (n, k) *block erasure code* converts k source packets into n encoded packets, such that any k of the n encoded packets can be used to reconstruct the k source packets [11]. In multicast scenarios, a single parity packet can be used to correct independent single-packet losses among different receivers [14].

In an earlier study [13], our group implemented several FEC "filters," based on block erasure and other FEC codes, and studied their performance when integrated into a composable proxy framework for wireless nodes. That approach used detachable Java I/O streams, which enable filters to be inserted, deleted, and reordered on a running data stream. While supporting adaptability, the design of that framework was ad hoc. In the current study, we explore how to realize similar adaptive functionality using the principled approach of refractive and transmutative components. Specifically, we move this functionality into the socket component itself and modify the behavior of the component at run time.

MetaSocket Design. Figure 9 depicts the structure of a `MetaSocket` component. The base component, called `SendSocket`, was created by absorbing the existing Java `Socket` class. Certain public members and methods are made accessible through invocations on `SendSocket`. Since this component is intended to be used only for sending data, the invocations available to other components are `send()` and `close()`. Hence, the application code using the computational interface of a metamorphic socket looks similar to code that uses a regular socket. In addition, three invocations (`SetBuffer`, `GetFilter`, `GetLastFilter`) are intended for use by the meta-level. The `SendSocket` was metafied to create a meta-level component called `MetaSocket`. `GetStatus()` is a refraction that is used to obtain the current configuration of filters. `InsertFilter()` and `RemoveFilter()` are transmutations that are used to modify the filter pipeline.

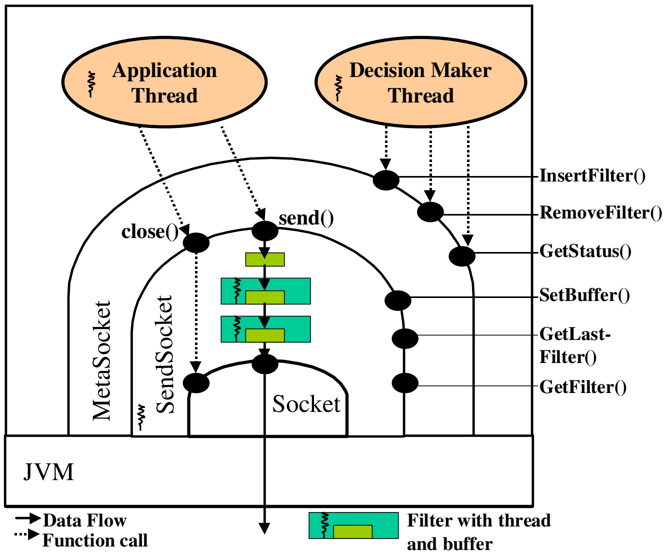

Figure 9. Structure of a MetaSocket.

Operation. A separate thread, called a Decision Maker, resides within each application and controls the behavior of its respective MetaSocket based on current observable status. The Decision Maker monitors various conditions (observed packet loss, signal strength) and decides how to adapt the MetaSocket; we are currently evaluating its effectiveness in managing FEC parameters for connections across our wireless LAN [15]. For purposes of testing the MetaSocket interfaces, however, we also developed an interactive administration utility that enables us to manipulate MetaSockets directly. Figure 10 shows an example trace where we streamed audio from a desktop to an iPAQ across an 802.11 wireless LAN. We used the transmutative interface to insert an (8,4) FEC filter dynamically, significantly reducing packet loss, until we removed the filter.

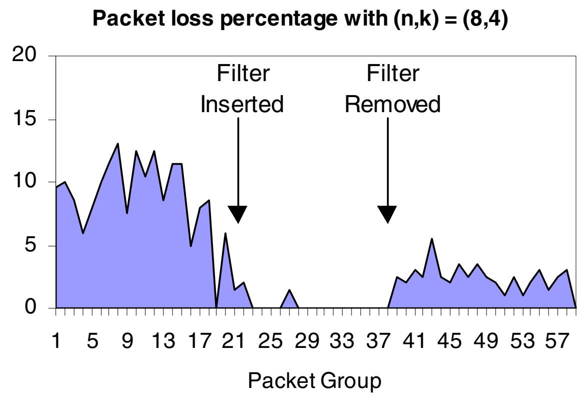

Figure 10. Sample results of dynamically changing MetaSocket configuration.

Of course, MetaSockets can be adapted in many ways besides the insertion and deletion of FEC filters. In addition to other properties related to quality-of-service (packet size, congestion control), a MetaSocket could be used to report socket usage patterns to an intrusion detection system or to maintain synchronization between sockets on duplicate connections for fault tolerance. In some cases, the same meta-level operation may be used by multiple entities. For example, information on traffic patterns may be of interest to both an intrusion detection system and a performance management system. We are presently investigating these topics. Additional details on the design and operation of metasockets are described in a technical report [15].

6 Related Work

In recent years, numerous research groups have addressed the issue of adaptive middleware frameworks that can accommodate dynamic, heterogeneous infrastructures. These projects have greatly improved the understanding of how middleware can accommodate device heterogeneity and dynamic network conditions, particularly in the area of adaptive communication protocols and services. A comprehensive treatment is impractical in this paper. Instead, we focus on those contributions that are most related to the work presented here.

Several projects involve adaptive extensions to CORBA [3, 6, 9, 17]. For example, in the Adapt project at Lancaster [6], CORBA is extended to support open bindings, which enable manipulation and reconfiguration of communication paths through the use of object graphs. This mechanism could be used directly to implement dynamically composable services for FEC and other QoS-related functions. In contrast to a CORBA-based design, however, our focus in this study is on programming language constructs to support adaptive interfaces to arbitrary components.

The PCL project being conducted by Adve [1] also focuses on language support for adaptability. PCL is intended for use directly by applications. Our concept of "wrapping" classes with base components is similar to the use of *Adaptors* used in PCL. However, modification of the base class in PCL appears to be limited to changing variable values, whereas Adaptive Java transmutations can modify arbitrary structures or subcomponents. Moreover, by combining encapsulation with metafication, Adaptive Java can be used to realize adaptations in multiple meta-levels.

Also related is the concept of composition filters [4], which provide a mechanism for disentangling the cross-cutting concerns of a software system. This system declares filters that intercept messages received and sent by objects. As such, messages can be massaged and checked before they are delivered to an object, separating aspects, such as security authentication or bounds checking, from the objects that send and receive these messages. Adaptive Java's approach to composition using encapsulation could be used to instantiate a message filtering design where components are extended and invocations added such that a call to an invocation would be filtered through subsequent encapsulation layers. However, such a design would not have the declartive power provided by the composition filter declarative specification language.

7 Conclusions and Future Directions

In this work, we studied a possible approach, based on separation of introspection and intercession, for designing reflective primitives. We developed a prototype language, Adaptive Java, and showed how it can be used to construct adaptive components from existing classes. We demonstrated its use in building MetaSockets to support mobile computing. We intend these low-level mechanisms to provide a foundation for the construction and maintenance of meta-object protocols for cross-cutting concerns (communication quality, fault tolerance, security, power consumption). Our ongoing work addresses two key issues: the use of refractive and transmutative operations to support adaptability through run-time tailoring of component interfaces, and the application of rigorous software engineering techniques to ensure consistency of adapted software.

Further Information. A number of related papers and technical reports of the Software Engineering and Network Systems Laboratory can be found at the following URL: `http://www.cse.msu.edu/sens`.

References

[1] V. Adve, V. V. Lam, and B. Ensink. Language and compiler support for adaptive distributed applications. In *Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001)*, Snowbird, Utah, June 2001.

[2] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[3] C. Becker and K. Geihs. Quality of service and object-oriented middleware – multiple concerns and their separation. In *Proceedings of the International Workshop on Distributed Dynamic Multiservice Architectures*, Phoenix, Arizona, April 2001.

[4] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, October 2001.

[5] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England, September 1998.

[6] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin. A software architecture for adaptive distributed multimedia applications. *IEE Proceedings - Software*, 145(5):163–171, 1998.

[7] S. E. Hudson, editor. *CUP User's Manual*. Usability Center, Georgia Institute of Technology, July 1999.

[8] E. P. Kasten and P. K. McKinley. Adaptive Java: Refractive and transmutative support for adaptive software. Technical Report MSU-CSE-01-30, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, December 2001.

[9] F. Kuhns, C. O'Ryan, D. C. Schmidt, O. Othman, and J. Parsons. The design and performance of a pluggable protocols framework for object request broker middleware. In *Proceedings of the IFIP Sixth International Workshop on Protocols For High-Speed Networks (PfHSN '99)*, Salem, Massachusetts, August 1998.

[10] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Languages (OOPSLA)*, December 1987.

[11] A. J. McAuley. Reliable broadband communications using burst erasure correcting code. In *Proceedings of ACM SIGCOMM*, pages 287–306, September 1990.

[12] P. K. McKinley and A. P. Mani. An experimental study of adaptive forward error correction for wireless collaborative computing. In *Proceedings of the IEEE 2001 Symposium on Applications and the Internet (SAINT-01)*, San Diego-Mission Valley, California, January 2001.

[13] P. K. McKinley, U. I. Padmanabhan, and N. Ancha. Experiments in composing proxy audio services for mobile users. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, pages 99–120, Heidelberg, Germany, November 2001.

[14] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review*, April 1997.

[15] M. S. Sadjadi and P. K. McKinley. Design, implementation, and evaluation of metamorphic sockets, 2002. in preparation.

[16] B. C. Smith. Reflection and semantics in Lisp. In *Proceedings of 11th ACM Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[17] R. Vanegas, J. A. Zinky, J. P. Loyall, D. A. Karr, R. E. Schantz, and D. E. Bakken. QuO's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England, September 1998.