A Survey for Ph.D. Qualifier Exam.

A Survey of Adaptive Middleware

S. M. Sadjadi

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

Abstract

Developing distributed applications is a difficult task due to three major problems: the complexity of programming interprocess communication, the need to support services across heterogeneous platforms, and the need to adapt to changing conditions. Traditional middleware (such as CORBA, DCOM, and Java RMI) addresses the first two problems to some extent through the use of a "black-box" approach, such as encapsulation in object-oriented programming. However, traditional middleware is limited in its ability to support adaptation. To address all the three problems, *adaptive* middleware has evolved from traditional middleware. In addition to the object-oriented programming paradigm, adaptive middleware employs several other key technologies including computational reflection, component-based design, aspect-oriented programming, and software design patterns. This survey paper proposes a three-dimensional taxonomy that categorizes different adaptive middleware approaches. Examples of each category are described and compared in detail. Suggestions for future research are also provided.

Keywords: adaptive middleware, taxonomy, computational reflection, component-based design, aspect-oriented programming, software design patterns, static adaptation, dynamic adaptation, quality of service, dependable systems, embedded systems.

Contents

1 Introduction

Developing distributed applications is a difficult task for several reasons. First, writing code for interprocess communications is tedious and error prone. Low level socket programming and marshalling and unmarshalling messages are examples of such code. Second, supporting multiple interacting platforms is difficult. Many heterogeneous hardware devices, computer networks, operating systems, and programming languages have emerged during the last two decades. Distributed applications are more likely than stand-alone applications to involve heterogeneous technologies. Third, adapting to dynamic changing conditions is hard to achieve without the right tools and techniques. Emerging distributed applications often involve multimedia communication, mobility, embedded computing, group communications, and high availability. Addressing these issues means that systems must adapt to changing conditions, such as unexpected security attacks, hardware failures, and dynamic network environments.

To tackle the first two problems, middleware was invented. Traditionally, middleware hides the underlying details of interprocess communication and heterogeneous technologies from the application developers using a "black-box" paradigm such as encapsulation in object-oriented programming. Although traditional middleware solves these problems to some extent, it is limited in its ability to support adaptation. *Adaptive middleware* has evolved from traditional middleware to solve all the three problems together. We identify two types of adaptation provided by adaptive middleware: static and dynamic. *Static adaptation* can occur during compile or startup time, and *dynamic adaptation* occurs only after startup time.

In addition to the object-oriented programming paradigm, adaptive middleware employs four key software technologies in order to support adaptation. *Computational reflection* [1] enables middleware to inspect, reason about, and adapt itself at run time. *Component-based design* [2] enables decomposition of middleware functionality, which makes it easier to manage and modify the structure of the middleware both statically and dynamically. *Aspect-oriented programming* [3] enables separation of middleware cross-cutting concerns (such as quality-of-service, energy consumption, security, and fault tolerance) at development time; later, at compile or run time, these concerns can be selectively woven into the application code. Finally, *software design patterns* [4] enable reuse of best adaptive designs, such as the virtual component pattern [5], in adaptive middleware.

This paper uses three orthogonal methods to classify adaptive middleware. The first method, proposed by Schmidt [6], categorizes middleware into four layers: host-infrastructure, distribution, common-services, and domain-services. The second method, introduced here, classifies middleware according to its adaptation type. We define and focus on four types of adaptation: configurable, customizable, tunable, and mutable. The third method, also introduced here, classifies adaptive middleware according to the application domain. We focus on three major domains: QoS-oriented systems, dependable systems, and embedded systems.

The remainder of this survey paper is divided into the following sections. Section 2 provides background on middleware and reviews a well-established taxonomy of traditional middleware. Section 3 discusses supporting paradigms and standards for adaptive middleware. Section 4 proposes a three-dimensional taxonomy of adaptive middleware. Section 5 introduces, compares, and categorizes several recent adaptive middleware projects according to the proposed taxonomy. Section 6 brings together all the projects discussed in Section 5 and shows where each project fits in the big picture. Finally, Section 7 concludes the paper and suggests possible future research directions.

2 Middleware Background

Middleware is connectivity software that encapsulates a set of services residing above the network operating system layer and below the user application layer. Middleware facilitates the communication and coordination of application components that are potentially distributed across several networked hosts. Moreover, middleware provides application developers with high-level programming abstractions, for example, use of remote objects instead of socket programming [7]. In this manner, middleware can hide interprocess communication, mask the heterogeneity of the underlying systems (hardware devices, operating systems, and network protocols), and facilitate the use of multiple programming languages at the application level. Middleware can also be considered as a "glue" that enables integration of legacy applications [8], effectively implementing the session and presentation layers (layers 5 and 6) of the ISO OSI reference model [9]. Next, we discuss a well-known taxonomy of traditional middleware [9]. Following that, object-oriented middleware (upon which most adaptive middleware projects introduced in this paper are based) is discussed at length.

2.1 A Taxonomy of Traditional Middleware

Emmerich [9] provides a frequently referenced taxonomy of middleware. His taxonomy is based on the type of programming-language abstraction that middleware provides for interaction among distributed software components: transactional, message-oriented, procedural, or object-oriented. The corresponding primitive communication techniques are distributed transactions, message passing, remote procedure calls, and remote object invocations, respectively. We note that Bakken [8] introduced a similar taxonomy that also includes four classifications: distributed tuples, message-oriented, remote procedure call, and distributed object. In this paper, however, we use the Emmerich's taxonomy to help lay a foundation for our later discussion of adaptive middleware.

Transactional middleware supports distributed transactions among processes running on distributed hosts. Originally, this type of middleware was targeted at interconnecting heterogeneous database systems. Goals include providing data integrity, high-performance, and availability using the two-phase commit protocol [10]. IBM CICS [11] and BEA Tuxedo [12] are two examples of this category.

Message-oriented middleware facilitates asynchronous message exchange between clients and servers using the message-queue programming abstraction [9], a generalization of the operating system mailbox. Messages do not block a client and are deposited into a queue with no specific receiver information. In addition, the message-queue abstraction decouples clients and servers, which enables interaction among otherwise incompatible systems. IBM MQSeries [13] and Sun Java Message Queue [14] are two examples of this category.

Procedural middleware extends the procedure call in procedural programming languages to include *remote procedure calls (RPC)*, where the body of the procedure resides on a remote host and can be called the same way as a local procedure. Birrell and Nelson [15] implemented the first full-fledged version of RPC. Sun Microsystems adopted RPC as part of its open network computing. Later, Open Group developed a standard for RPC called distributed computing environment (DCE) [16]. Most Unix and Windows operating systems now support RPC facilities.

Finally, *object-oriented middleware* is based on both the object-oriented programming paradigm and the RPC architecture. It provides the abstraction of a *remote object*, whose methods can be invoked as if the object were in the same address space as its client. Encapsulation, inheritance, and polymorphism are often supported by this type of middleware. CORBA [17], Java RMI [18], and DCOM [19] are three major object-oriented middleware approaches.

Among these four types, in this survey, our primary focus is on object-oriented middleware, which is the basis for most research in adaptive middleware. Therefore, the remainder of this section reviews the three major object-oriented middleware approaches.

2.2 Object-Oriented Middleware

Object-oriented middleware separates an object interface (a set of functionally related methods) from its implementation, provides a local representation of a remote object, and hides the inter-process communication between a remote object and its local representation.

The *Common Object Request Broker Architecture (CORBA)* [17] is a distributed object framework proposed by the Object Management Group (OMG) [20]. CORBA supports distributed object-oriented computing across heterogeneous hardware devices, operating systems, network protocols, and programming languages. Figure 1 illustrates the CORBA components described as follows. The *Object Request Broker (ORB)* [21], the core of CORBA, allows objects to interact transparently with other objects (located locally or remotely). A CORBA object is represented by its interface, is identified by its reference, and is realized in an object-oriented program as a local object called "servant." The client of a CORBA object acquires the object reference called interoperable object reference (IOR) and calls methods on this reference as if the object were located in the client address space. The *Interface Definition Language (IDL)* is a language for defining CORBA interfaces. An IDL compiler is used to automatically generate the code for stubs and skeletons. An *IDL stub* represents a servant locally in the client address space and an *IDL skeleton* represents a client locally in the servant address space. IDL stubs and skeletons marshal and unmarshal requests to enable transmission of the requests over a network.

The *dynamic invocation interface (DII)* enables clients to directly access the underlying request mechanisms at run time to generate dynamic requests to objects, whose type (interface) were not known at the client compile time. The *interface repository* provides the type information that a client needs to dynamically create a request. Similarly, the *dynamic skeleton interface (DSI)* enables an ORB to deliver requests to a servant that does not have compile-time knowledge of the type of the object it supports (*e.g.,* a gateway object may not know the type of the target objects to which it is forwarding requests). The *implementation repository* enables late deployment of CORBA objects. The implementation repository receives the first request targeted to a CORBA object, looks up the object meta information in its database, activates the object, and forwards the request "permanently" to the target object. Permanent forwarding, in contrast to transient forwarding, also causes automatic forwarding of all future requests from the same client and to the same target object directly from the client ORB. The *object adapter* activates servants and dispatches requests to them. The *ORB interface* provides access to standard ORB services, such as resolving the CORBA initial services like the naming service. The *general inter-ORB protocol (GIOP)* is a standard for inter-ORB communication that enables interoperability among different CORBA-compliant ORBs. The *Internet inter-ORB protocol (IIOP)* is a specific mapping of the GIOP specification that runs over TCP/IP connections.

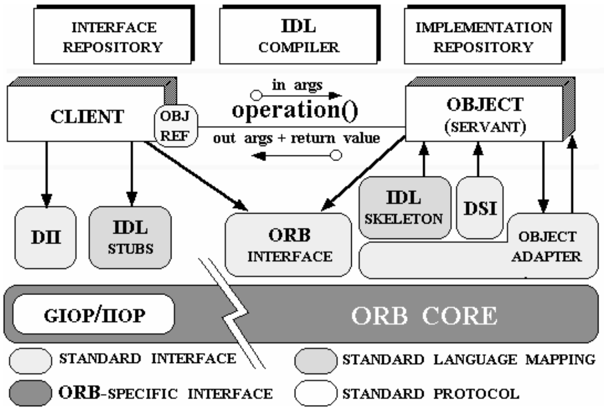


Figure 1: CORBA architecture [22].

The *Java remote method invocation (Java RMI)* [18] was proposed by JavaSoft to support the development of distributed Java-based applications. Unlike CORBA, Java RMI supports only the Java language, but similar to CORBA, Java RMI supports distributed computing across heterogeneous hardware devices and operating systems using the Java Virtual Machine (JVM). Instead of CORBA marshalling and unmarshalling, Java RMI uses object serialization, which preserves the type of the objects being serialized. Figure 2 depicts a typical Java RMI application. The `registry` in Java RMI is similar to a CORBA naming service, which resolves a symbolic name to an actual remote object reference. A server object registers itself with the registry, where a client object can look up the remote object address. Java RMI can dynamically load the class bytecode of an object that is passed between remote objects using Java reflection. As shown in Figure 2, Java RMI can use a web server to load class bytecodes. Similar to CORBA, Java RMI supports dynamic request invocation using Java reflection.

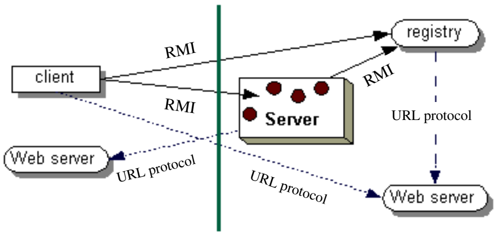


Figure 2: A typical Java RMI application [23].

The *Distributed Component Object Model (DCOM)* [19] was proposed by Microsoft as a distributed extension to the Component Object Model (COM) [24]. Similar to CORBA, DCOM supports heterogeneous programming languages, but unlike CORBA and Java RMI, DCOM supports only Windows-based platforms. Figure 3 illustrates the DCOM architecture. The *service control mangager (SCM)*, like the CORBA ORB core, is responsible to locate an object implementation. In DCOM terminology, the *object proxy* and *object stub* are the equivalent names as the CORBA stub and skeleton. Unlike CORBA and Java RMI, DCOM supports neither multiple inheritance nor exceptions at the IDL level. However, with regard to inheritance, DCOM supports multiple interfaces using a binary standard similar to the C++ vtable [19]. DCOM also supports dynamic invocation using the IDispatch interface [19]. For more detailed comparisons of CORBA, Java RMI, and DCOM, please refer to [7, 25, 26].

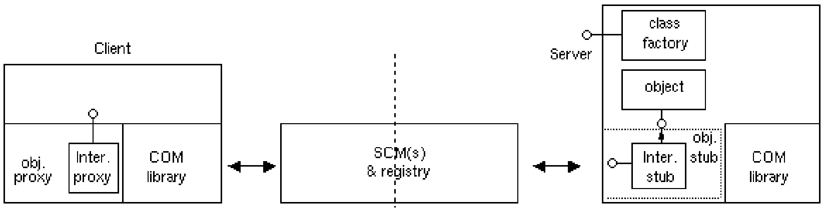


Figure 3: DCOM architecture [26].

3 Key Supporting Paradigms for Adaptation

In addition to the foundation provided by the design and use of traditional middleware platforms, numerous advances in programming paradigms [1–4, 27–34] have also contributed to the design of adaptive middleware. Although many important contributions have been made in this area [28–34], a review of the literature shows that four paradigms, in addition to object-oriented paradigm, play key roles in supporting adaptive middleware: computational reflection [1], component-based design [2], aspect-oriented programming [3], and software design patterns [4, 27]. Each is discussed in turn as follows.

3.1 Computational Reflection

Computational reflection [1,35] refers to the ability of a program to reason about, and possibly alter, its own behavior. *Reflection* enables a system to “open up” its implementation details for such analysis without compromising portability or revealing the unnecessary parts [36]. In other words, reflection exposes a system implementation at a level of abstraction that hides unnecessary details, but still enables changes to the system behavior [1, 35]. As depicted in Figure 4, a reflective system (represented as *base-level* objects) has a self representation (represented as *meta-level* objects) that is *causally connected* to the system, meaning that any modifications either to the system or to its representation are reflected in the other [37]. The *base-level* part of a system deals with the "normal" (functional) aspects of the system, whereas the *meta-level* part deals with the computation (implementation) aspects of the system. A *meta-object protocol (MOP)* is a meta-level interface that enables "systematic" (as opposed to ad hoc) inspection and modification of the base-level objects.

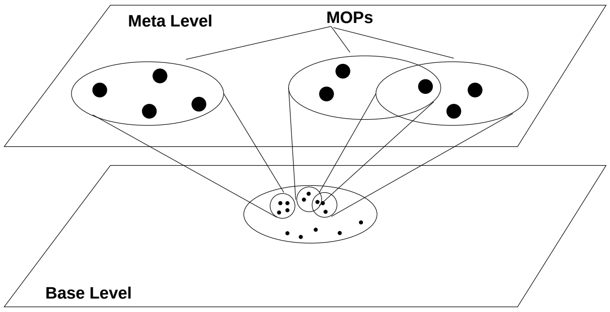


Figure 4: Relationship between meta-level objects and base-level objects.

Computational reflection has been studied for several years in the context of programming languages [1, 35, 38–41] and operating systems [42–44]. Recently, reflection has also been studied in middleware, where it enables adapting the behavior of a distributed application by modifying the middleware implementation. Reflective middleware is often concerned with adapting non-functional aspects of distributed applications including QoS, performance, security, fault tolerance, and energy management. Section 5 describes several examples of reflective middleware [45–53].

We also note that several reflective programming languages [54–57] have been proposed recently to support development of distributed systems and reflective middleware. *MetaJava* [54] extends Java reflection with behavioral reflection that enables modifying the behavior of the Java RMI package at run time (*e.g.,* encrypting requests before transmitting them over a network). *Program Control Logic (PCL)* [56] provides a programming framework that enables programmers to design, develop, and optimize the performance of adaptive distributed applications [56]. A source-to-source compiler is provided, which inputs meta code specified in a language very close to C++ and Java (PCLC and PCLJ respectively) and outputs a program source in C++ or Java that is then compiled and linked with the base program. *Adaptive Java* [55] is an extension to Java that introduces new language constructs to support behavioral reflection. In its behavioral reflective meta-model architecture, Adaptive Java separates monitoring the behavior (introspection) from changing the behavior (intercession), using "refractive" and "transmutative" meta methods, respectively. *Iguana/J* [57] extends the Java Virtual Machine to intercept method invocation, object creation, and field read and write at run time. Iguana/J can adapt the intercepted operations by loading new code dynamically. These and other reflective languages [1, 35, 38–41] are likely to facilitate the development of adaptive middleware and distributed applications.

3.2 Component-Based Design

Software components are software units that can be independently produced, deployed, and composed by third parties [2]. Components are self-contained: components clearly specify what they require and what they provide. Component-based design (CBD) supports the large scale reuse of software by enabling assembly of "commodity-off-the-shelf" (COTS) components from a variety of vendors [7]. The independent deployment of components enables *late composition* (also referred to as *late binding*), which is essential for adaptive systems. Late composition provides coupling of two compatible components at run time through a well-defined interface. A system developed using CBD is an amalgam of components that can be reorganized easily.

When applied to middleware, CBD provides a flexible and extensible system [47,49,50,53,58–60]. Specially, a middleware can be customized to specific application domains, through the integration of domain-specific components, and can evolve using third-party components. Moreover, component-based middleware can be dynamically adapted to its environment using late composition. Examples of major component-based middleware solutions are DCOM [19] (discussed earlier), EJB [61], and CCM [62]. *Enterprise Java Beans (EJB)* [61] is a middleware component model for Java proposed by Sun Microsystems that enables Java developers to use off-the-shelf Java components, or *beans*. Since EJB is built on top of Java technology, EJB components can only be implemented using the Java language, however. The EJB component model supports adaptation by automatically supporting services such as transactions and security for distributed applications. The *CORBA Component Model (CCM)* [62] is a distributed component model proposed by OMG that can be considered as a cross-platform, cross-language superset of EJB. CCM supports adaptation by enabling injection of adaptive code into component containers (*i.e.,* the component themselves remain intact).

3.3 Aspect-Oriented Programming

The third major software development paradigm used in adaptive middleware is aspect-oriented programming (AOP). Kiczales et al. [3] realized that complex programs are composed of different intervened *cross-cutting concerns* (properties or areas of interest such as QoS, energy consumption, fault tolerance, and security). While object-oriented programming abstracts out commonalities among classes in an inheritance tree, cross-cutting concerns are still scattered among different classes, complicating the development and maintenance of applications. AOP enables separation of cross-cutting concerns during development time. Later, during compile or run time, an *aspect weaver* can be used to weave different aspects of the program together to form a program with new behavior. AOP proponents argue that disentangling the cross-cutting concerns leads to simpler development, maintenance, and evolution of software.

Naturally, these benefits are important to adaptive middleware. Moreover, AOP enables factorization and separation of cross-cutting concerns from the middleware core [63], which promotes reuse of cross-cutting code and facilitates adaptation. Using AOP, customized versions of middleware can be generated for application-specific domains. Yang et al. [64] and David et al. [65] both provide a two-step approach to dynamic weaving of aspects, in the context of adaptive middleware, using a static AOP weaver during compile time and reflection during run time. Other aspect-oriented middleware projects [53, 66–69] are described in detail in Section 5.

3.4 Software Design Patterns

Software design patterns [4, 27] provide a way to reuse best software designs practiced successfully for several years. The goal of software design patterns is to create a common vocabulary for communicating insight and experience about recurring problems and their known "refined" solutions.

It is very costly, time consuming, and error-prone to independently rediscover and reinvent solutions to middleware challenges. Schmidt and colleagues [4] have identified a relatively concise set of patterns that enables developing adaptive middleware. For example, the virtual component pattern [5], used in TAO [70] and ZEN [60], enables adapting a distributed application to the memory constraints of embedded devices by providing a small middleware footprint including only a minimum core, and a set of "virtual" components, whose code can be dynamically loaded on demand. Numerous adaptive middleware projects [45, 49, 50, 53, 58, 60, 66, 67, 70–72] benefit from the use of adaptive design patterns, as discussed in Section 5.

The paradigms introduced in this section address only part of the adaptation techniques used by numerous recent and ongoing adaptive middleware projects. The next section describes the three-dimensional taxonomy we use to categorize these activities.

4 A Taxonomy of Adaptive Middleware

Adaptive middleware enables modifying the behavior of a distributed application after the application is developed in response to some changes in functional requirements or operating conditions. Adaptive operating systems [42–44] also enable adaptation after development time. Adaptive operating systems, however, typically provide low-level and non-portable adaptive services. Adaptive middleware, on the other hand, can exploit an underlying adaptive operating system, while providing a high-level abstraction of system resources that supports portable adaptive code. Adaptive operating systems are beyond the scope of this survey paper. In the remainder of this section, we describe a three-dimensional taxonomy for classifying adaptive middleware projects. The first dimension was introduced by Schmidt [6], while the second and third are proposed by the author.

4.1 Middleware Layers

Schmidt [6] decomposes middleware into four layers: host-infrastructure, distribution, common-services, and domain-services. Figure 5 illustrates these layers.

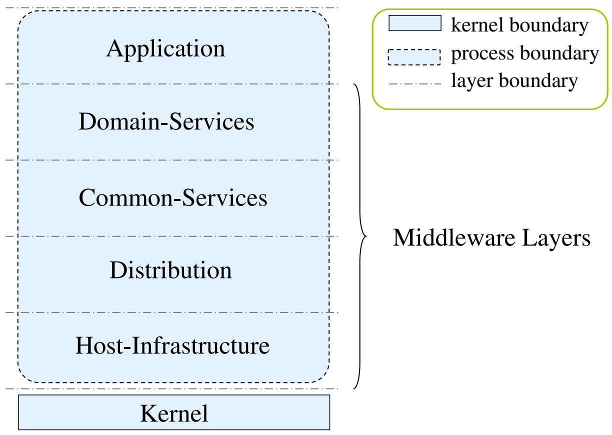


Figure 5: Middleware layers described by Schmidt [6].

The *host-infrastructure layer* resides directly atop the operating system kernel and provides a higher-level API than the operating system API that hides the heterogeneity of hardware platforms, operating systems and, to some extent, network protocols. Host-infrastructure middleware provides generic services to the upper middleware layers by encapsulating functionality that would otherwise require many tedious, error-prone, and non-portable code, such as socket programming and thread communication primitives. ACE [71], Rocks [73] and MetaSockets [53], described in Section 5, are examples of adaptive middleware in this layer.

The *distribution layer* resides atop the host-infrastructure layer and provides a high-level programming abstraction, such as remote method invocation, to application developers. Using the distribution layer, developers can write distributed applications similar to stand-alone applications. Moreover, this layer hides the heterogeneity of network protocols and, to some extent, the heterogeneity of operating systems and programming languages. CORBA [17], DCOM [19], and Java RMI [18], discussed earlier, are the main solutions to the distribution layer. In Section 5, we will provide some examples of adaptive middleware at this layer including TAO [70], DynamicTAO [45], and OpenORB [46].

The *common-services layer* resides atop the distribution layer and provides functionality such as fault tolerance, security, load balancing, event propagation, logging, persistence, real-time scheduling, and transactions. The high-level services provided in this layer can be reused in different applications. QuO [66], IRL [74], and FRIENDS [51], also described in Section 5, are example of adaptive middleware in this layer.

Finally, the *domain-services layer* resides atop the common-services layer and is tailored to a specific class of distributed applications. Unlike the common-services layer, the high-level services in this layer can be reused only for a specific domain. Boeing Bold Stroke architecture [75] is an example of adaptive middleware in this layer.

4.2 Adaptation Type

Adaptive middleware can be categorized with respect to the type of adaptation it provides. Figure 6 shows our proposed taxonomy of adaptive middleware with respect to adaptation type mapped to the application lifetime. If middleware enables adaptation during the application compile or startup time, we call it *static middleware* (*e.g.,* EmbeddedJava [76] minimizes the footprint of embedded applications during the application compile time). If middleware enables adaptation during the application run time, we call it *dynamic middleware* (*e.g.,* MetaSockets [53] load adaptive code during run time to adapt to wireless network loss rate changes). Note that, in Figure 6, dynamism increases from left to right.

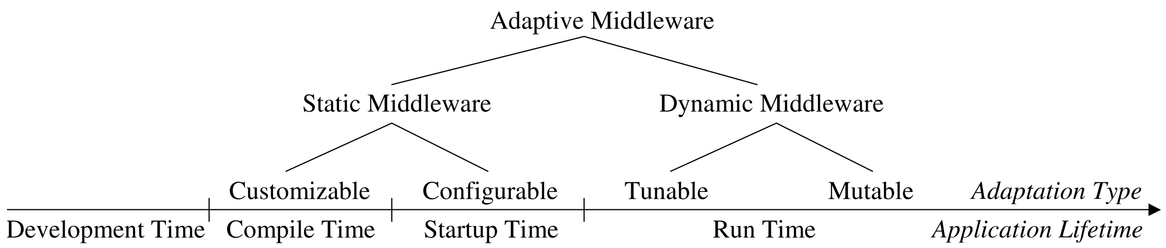


Figure 6: Taxonomy of adaptive middleware with respect to adaptation type.

Static middleware is divided further into customizable and configurable middleware. *Customizable middleware* enables adapting an application during the application compile (or link) time so that a developer can generate customized (adapted) versions of the application. Note that a customized version is generated in response to the functional and environmental changes realized after the application development time. Examples of adaptation mechanisms provided by customizable middleware are static weaving of aspects [3], compiler flags [60], and precompiler directives [60]. QuO [66] and EmbeddedJava [76], discussed in Section 5, are examples of customizable middleware.

Configurable middleware enables adapting an application during the application startup time, enabling an administrator to configure the middleware in response to the functional and environmental changes realized after the application compile time. Examples of adaptation mechanisms provided by configurable middleware include CORBA portable interceptors [77], optional command-line parameters, for example, to set socket buffer-size, and configuration files such as ORBacus configuration file [78]. In Section 5, we will describe several examples of cofigurable middleware including Eternal [79], IRL [74], and Rocks [73].

Dynamic middleware can be divided into tunable and mutable middleware. *Tunable middleware* enables adapting an application after the application startup time (but before the application is actually being used). Doing so enables an administrator to fine-tune the application in response to the functional and environmental changes that occur after the application is started. Examples of adaptation mechanisms provided by tunable middleware are "two-step" adaptation approaches (including static AOP during compile time and reflection during run time) employed by David et. al [65] and Yang et. al [64], the component configurator pattern [4] used in DynamicTAO [45], and the virtual component pattern [5] used in TAO [70] and ZEN [60]. We also define a variation of tunable middleware, repeatedly-tunable middleware, that enables repeated-tuning of applications during run time.

Mutable middleware is the most powerful type of adaptive middleware that enables adapting an application during run time. Hence, the middleware can be dynamically adapted while it is being used. The main difference between tunable middleware and mutable middleware is that in the former, the middleware core remains intact during the tuning process whereas in the latter, there is no concept of fixed middleware core. Therefore, mutable middleware are more likely to evolve to something completely different and unexpected. Examples of adaptive techniques provided by mutable middleware are reflection [46], late composition of components [60], and dynamic weaving of aspects [64,68]. OpenORB [46], also discussed in Section 5, is an example of mutable middleware.

4.3 Application Domain

The third dimension of our taxonomy categorizes adaptive middleware with respect to application domain. Our survey of the literature reveals that most adaptive middleware projects support one of these three main application domains: QoS-oriented systems, dependable systems, and embedded systems. Figure 7 illustrates our taxonomy of adaptive middleware based on application domains. *QoS-oriented middleware* supports real-time and multimedia applications, such as avionics systems, video conferencing and Internet telephony, that are required to meet deadlines and adhere to some QoS *contracts*, which define the acceptable levels of QoS. *Dependable middleware* supports critical distributed applications that are required to be correctly operational, such as military command and control and medical applications. *Embedded middleware* supports applications that are required to have small footprints to be able to run on very limited memory devices, including set-top boxes, smart phones, hand-held devices, industrial controllers, and scientific instruments. Each of the three domains are further refined in the next section, where we describe each of the refinements and use them to introduce specific adaptive middleware projects. We also discuss where each project fits with respect to the first two dimensions of our taxonomy.

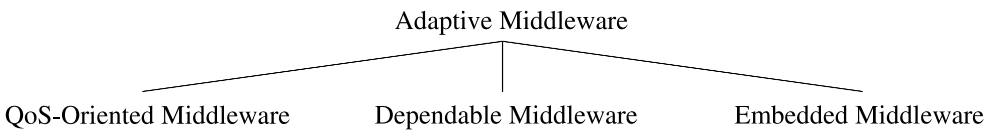


Figure 7: Taxonomy of adaptive middleware with respect to application domain.

5 Adaptive Middleware Projects

Before beginning our classification of adaptive middleware projects, we should emphasis that a given project may cross domains and may reside in more than one middleware layer. In such cases, we placed the project in the domain and layer that match its primary functionality. If a middleware project provides more than one adaptation type, we explicitly mention that the project provides hybrid adaptation. In addition, details of how supporting paradigms used in each project are discussed and compared to other related projects.

5.1 QoS-Oriented Middleware

QoS-oriented middleware supports distributed applications that require quality-of-service. We further refine QoS-oriented middleware into real-time, multimedia, reflection-oriented, and aspect-oriented middleware as shown in Figure 8. Examples of each category are described in turn as follows.

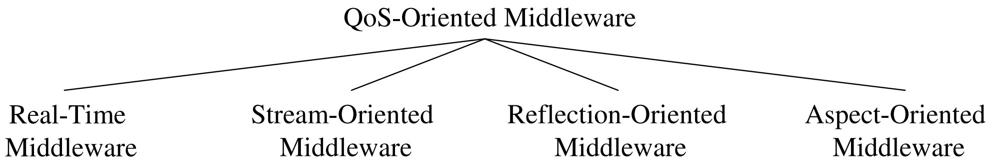


Figure 8: Taxonomy of QoS-Oriented middleware.

Real-Time Middleware. Real-time middleware is required to meet the deadlines defined by real-time applications. In a *hard* real-time middleware, a failure to meet a deadline may lead to loss of life or property. Life critical military and safety critical civilian distributed real-time systems are two examples of hard real-time applications. In a *soft* real-time middleware, however, a failure to meet a deadline is not as critical as in a hard real-time middleware.

One of the earliest middleware projects is Schmidt's *Adaptive Communication Environment (ACE)* [71, 80], a real-time object-oriented framework written in C++, that provides high-performance and real-time communication services. ACE employs software design patterns to support distributed applications with efficiency and predictability, including low latency for delay-sensitive applications, high performance for bandwidth-intensive applications, and predictability for real-time applications. Figure 9 illustrates the key components in the ACE framework. Note that the *OS Adaptation Layer* resides directly atop the native operating system APIs providing a platform-independent API. Hence, we place ACE in the host-infrastructure layer. ACE components can be dynamically updated using the service configurator pattern [4] and C++ dynamic binding feature. Therefore, we consider ACE as repeatedly-tunable middleware (but not mutable middleware) because the ACE core remains intact during the tuning process.

Schmidt et al. [70] extended their ACE work to create *the ACE ORB (TAO)*, a CORBA compliant real-time ORB built atop the ACE components, as shown in Figure 10. TAO enhances the standard CORBA event service to provide real-time event dispatching and scheduling required by real-time applications such as avionics, telecommunications and network management systems. Earlier versions of TAO employ the strategy design pattern [27] to encapsulate different aspects of the ORB internals, such as IIOP pluggable protocols, concurrency, request demultiplexing, scheduling, and connection management. A configuration file is used to specify the strategies used to implement these aspects during startup time. TAO parses the configuration file and loads the required strategies. Therefore, we consider TAO as configurable middleware. Recent versions of

TAO decomposes the C++ implementation of TAO into several core ORB components that can be dynamically loaded on demand using the virtual component pattern [5]. Therefore, we also consider TAO as repeatedly-tunable middleware. TAO naturally resides in the distribution layer because it is a CORBA compliant ORB.

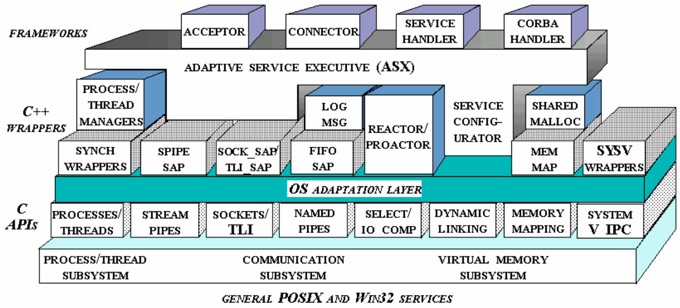


Figure 9: ACE architecture [81].

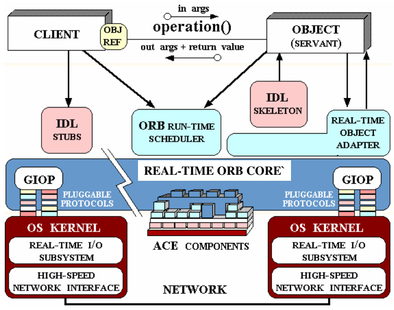


Figure 10: TAO architecture [82].

The *Component-Integrated ACE ORB (CIAO)* [58], also developed by Schmidt et al., is the TAO implementation of CORBA Component Model (CCM) [77], which also resides in the distribution layer. CIAO intended to provide component-based design to distributed real-time and embedded (DRE) system developers by abstracting systemic aspects, such as QoS requirements and composable meta-data units supported by the component framework. We consider the current CIAO implementation only as configurable middleware.

Researchers at the University of Illinois developed several adaptive middleware [45, 49, 50, 59]. Campbell et al. [45] adopted earlier version of TAO [70], which itself is considered only as configurable middleware, and built a dynamically adaptive version of TAO called DynamicTAO using computational reflection. To provide real-time services, DynamicTAO uses the Dynamic Soft Real-Time Scheduler (DSRT) [83] that provides QoS guarantees to applications with soft real-time requirements. Reflection in DynamicTAO does not use meta objects for reifying the ORB aspects. Reification in DynamicTAO is achieved using the service configurator pattern [4]. In other words, reflection is mainly used to implement the service configurator pattern. Figure 11 illustrates the DynamicTAO reified structure. The `DomainConfigurator`, `TAOConfigurator`, and `ServantConfigurator` are all realizations of service configurator pattern in DynamicTAO. A service configurator in DyanimcTAO exports the `DynamicConfigurator` interface, which is a CORBA IDL interface, defined also as the MOP for inspecting, adapting, loading, and unloading "component implementations" dynamically. Component implementations are organized in categories representing different aspects of the TAO ORB packaged as dynamically loadable libraries that can be linked to the ORB at run time. We consider DynamicTAO as repeatedly-tunable middleware.

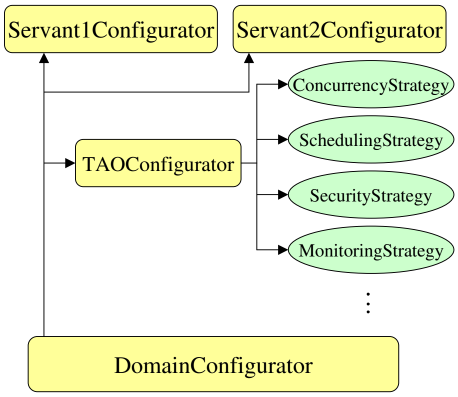


Figure 11: DynamicTAO reified structure [84].

Stream-Oriented Middleware. Stream-oriented middleware provides a continuous data streaming abstraction to the multimedia application developers. Video conferencing, Internet telephony, and digital libraries are some examples of multimedia applications.

Researchers at Lancaster University have conducted several projects in multimedia middleware [37, 46, 85, 86]. Blair et al. [37] have investigated the middleware implementation for mobile multimedia applications which can be dynamically adapted in response to the environmental changes in the context of Adapt project. In the OpenORB project [46], the successor of the Adapt project, Blair et al. continued their investigation studying the role of computational reflection in middleware. More recently, Blair et al. [85] designed OpenORB v2 that adds a component-based design framework to the OpenORB reflective framework. OpenCOM [86] is the implementation of OpenORB v2, designed for Microsoft COM systems. All above mentioned projects are greatly influenced by the ITU-T/ISO RM-ODP [87], a meta standard for multimedia applications. Unlike TAO [70] and DynamicTAO [45], none of Adapt, OpenORB, and OpenORB v2 projects are CORBA compliant.

OpenORB uses reflection to provide dynamic adaptation. The implementation of the OpenORB current reflective architecture is based on the reflection model illustrated in Figure 12. OpenORB categorizes reflection into structural and behavioral reflection [84], a distinction first introduced in [88]. *Structural reflection* is the ability of a system to inspect and modify its internal architecture, and *behavioral reflection* is the ability of a system to inspect and modify its computation. Structural reflection is modeled by the "architecture" and "interface" meta-models, and the behavioral reflection is modeled by the "interception" and "resources" meta-models. The *architecture meta-model* provides access to an object using its object graph. The *interface meta-model* provides access to the methods, associated attributes, and inheritance structure of each interface of an object. The *interception meta-model* provides interception hooks for each interface of an object including message arrival, dispatching, marshalling and unmarshalling interception hooks. The *resources meta-model* provides access to available resources per address space and enables resource reservation. Unlike DynamicTAO [45] that uses reflection mainly to implement the service configurator pattern, OpenORB provides an ORB wide reflection. Therefore, we consider OpenORB as mutable middleware.

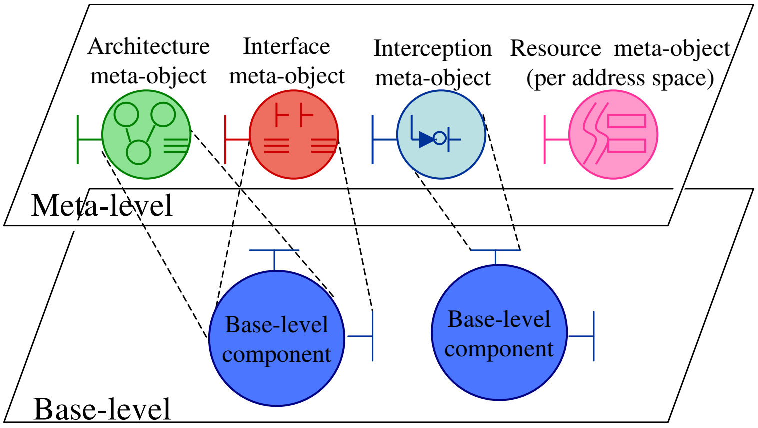


Figure 12: OpenORB reflection model [84].

OpenORB supports stream-oriented applications using "explicit binding," as opposed to the implicit binding provided in CORBA. In explicit binding, remote objects are bound explicitly by a programmer. Figure 13 illustrates the result of an explicit binding in a live video application, which represents the end-to-end communication path. Using the OpenORB reflection meta-model (in this case only architecture meta-model), an MPEG encoder can be replaced by an H.263 encoder that uses much lower bandwidth adapting the application to situation that network bandwidth available is decreasing at run time.

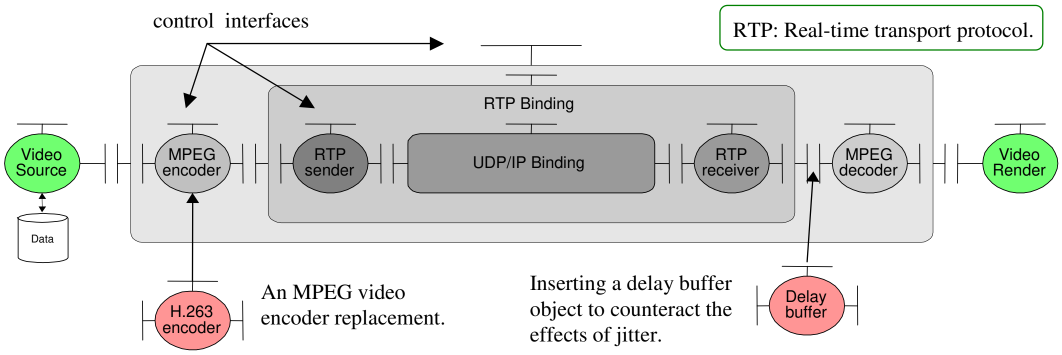


Figure 13: Open binding in OpenORB [89].

Squirrel [90,91] is QoS-oriented middleware specialized for distributed multimedia applications. Squirrel uses the Infopipes abstraction [92] to support streaming data. The designers argue that

CORBA stubs and skeletons generated from IDL interfaces follow a standard protocol (marshalling and unmarshalling) that is not suitable for multimedia applications with different QoS requirements. To solve this problem, Squirrel introduces *smart proxies* [91], which are service-specific stubs that include adaptive code. A smart proxy for a specific application can be developed and shipped to the client program statically (during compile time) or dynamically (during run time). Figure 14 illustrates dynamic smart proxy shipping in a live video application. We consider Squirrel at the distribution layer because, similar to CORBA stubs, Squirrel uses smart proxies to hide the interprocess communication details from application developers. We consider Squirrel as both tunable and mutable middleware because of its ability to statically and dynamically load smart proxies. However, Squirrel is not considered as repeatedly-tunable middleware because the tuning occurs just once at the remote object binding time.

MetaSockets [53], developed at Michigan State University, also address the issue of adaptable multimedia streams. MetaSockets are created from existing Java socket classes using Adaptive Java, a reflective extension to Java, whose structure and behavior can be adapted dynamically in response to external stimuli. As illustrated in Figure 15, MetaSockets provide a pipeline abstraction similar to that of Squirrel [90]. However, the adaptation supported in MetaSockets are finer-grained due to dynamic insertion and removal of *filters* instead of the whole pipeline as in Squirrel. A filter is a Java class that can be developed by third parties and can be inserted into the MetaSocket pipeline during run time to adapt the application behavior. Hence, we consider MetaSockets as repeatedly-tunable middleware. Unlike Squirrel, the MetaSocket pipeline does not hide the socket programming from application developers. As such, we place MetaSockets in the host-infrastructure layer.

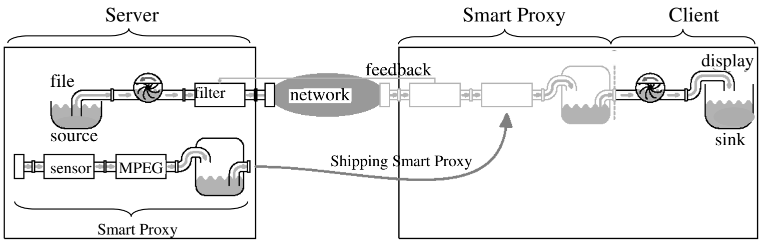


Figure 14: Squirrel: dynamic shipping of a smart proxy [91].

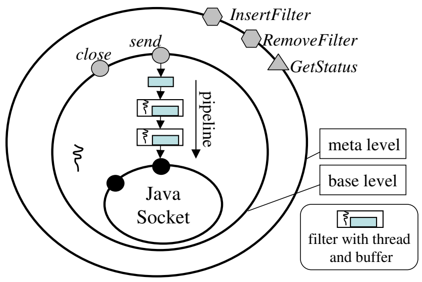


Figure 15: MetaSockets pipeline.

Reflection-Oriented Middleware. Reflection-oriented middleware supports QoS-oriented applications using computational reflection as its primary focus. No specific consideration for real-time and stream-oriented applications is provided by this type of middleware. Although DynamicTAO [45] and OpenORB [46] introduced before also employ computational reflection, but their primary focus are on real-time and stream-oriented applications, respectively. Therefore, we do not classify them in this category.

OpenCorba [48] is a CORBA compliant ORB that uses reflection to expose and modify some internal characteristics of CORBA. OpenCorba is implemented in NeoClasstalk, a reflective language based on Smalltalk [93]. OpenCorba reifies various properties of the ORB through explicit meta-classes. Figure 16 illustrates dynamic changing of the `StandardClass` class with the `BreakPoint+StandardClass` class at run time. Unlike DynamicTAO [45] and OpenORB [46], OpenCorba does not provide a global view of ORB, but similar to DynamicTAO preserves an intact ORB core during tuning process. Hence, we consider OpenCorba as repeatedly-tunable middleware. OpenCorba provides finer-grained adaptation than DynamicTAO and coarser-grained adaptation than OpenORB: DynamicTAO supports per-ORB adaptation, OpenCorba supports per-class adaptation, and OpenORB supports per-interface adaptation.

FlexiNet [47] is another CORBA compliant ORB implemented in Java that uses reflection to provide dynamic adaptation. FlexiNet is designed as a set of components, which can be dynamically assembled. Similar to DynamicTAO [45], FlexiNet provides coarse-grained ORB-wide adaptation. FlexiNet can dynamically modify the underlying communication's protocol stack through the replacement and insertion of layers. Similar to OpenORB [46], FlexiNet also provides fine-grained per-interface adaptation, as depicted in Figure 17. In FlexiNet, replaceable meta-objects can intercept requests in stubs and skeletons. These meta-objects realize channel configuration policies that are used to adapt stubs and skeletons. We also consider FlexiNet as repeatedly-tunable middleware.

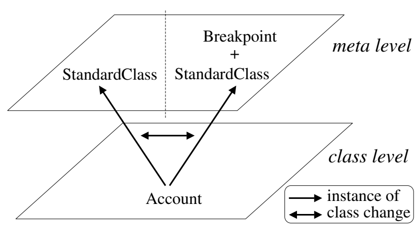


Figure 16: Reflection in OpenCorba [48].

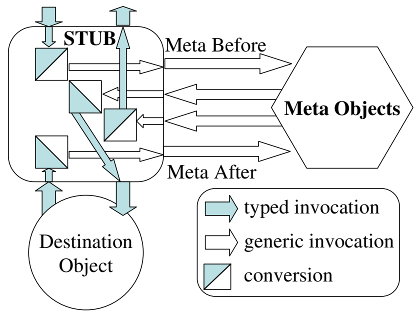


Figure 17: FlexiNet architecture [47].

Aspect-Oriented Middleware. Aspect-oriented middleware supports QoS-oriented applications using aspect-oriented programming paradigm. Similar to reflection-oriented middleware, aspect-oriented middleware provides no specific consideration for real-time and stream-oriented applications.

Researchers at BBN Technologies [66] have developed an adaptive framework for CORBA and Java RMI applications that supports QoS using aspect-oriented programming paradigm. QuO provides a high-level QoS abstraction at the common-services layer. Figure 18 illustrates QuO components residing between the application and distribution ORB. QuO wraps CORBA stubs and skeletons using functional delegates. As illustrated in Figure 18, the delegate intercepts outgoing requests and incoming replies. The delegate consults the "contract," using the `premethod` and `postmethod` methods. The contract is part of the QuO kernel that is aware of acceptable QoS regions and adapts the application behavior by modifying requests and replies according to the current system status monitored by system condition objects. QuO provides a quality description language (QDL) to write contracts that specifies QoS regions. The `quogen` utility can be used to translate these contracts to high-level languages such as C++ and Java. In addition, QuO provides an aspect-oriented structure description language (ASL) that enables developers to write generic or application-specific aspects. Later, the `quogen` utility can be used to generate delegates from CORBA object interfaces written in IDL, aspects written in ASL, and contracts written in QDL. We consider QuO as customizable middleware because QuO adapts an application during the application compile time using the `quogen` utility. The delegates in QuO are similar to the statically shipped smart proxies in Squirrel [90]. However, delegates can also wrap skeletons on the server side whereas smart proxies are only at the client side.

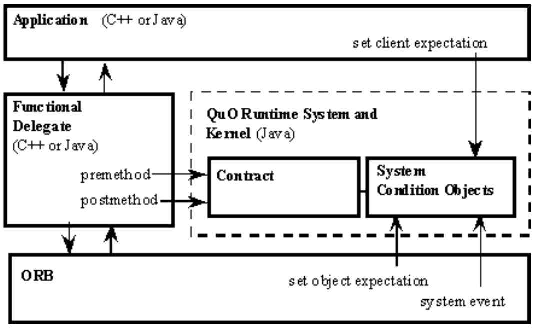
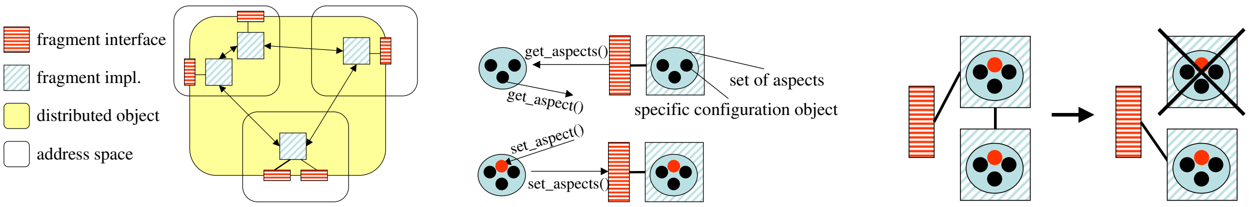


Figure 18: QuO architecture [94].

AspectIX [68] is another aspect-oriented middleware that is based on the fragmented (distributed) object model [95]. Figure 19(a) illustrates a distributed object that has four fragments distributed over three programs over a network. A fragment is divided into a fixed interface and a flexible implementation. A fragment implementation can be as simple as a CORBA stub or as complicated as a smart fragment that can cache previous replies locally or change its behavior using dynamically inserted aspects. Figure 19(b) shows how an AspectIX-aware application can dynamically inspect and adapt the set of aspects residing inside a fragment implementation. Figure 19(c) shows how a fragment implementation can be dynamically exchanged, transparent to the application. Unlike QuO that weaves aspects into the application statically, AspectIX enables "dynamic" weaving of aspects at run time. Hence, we consider AspectIX as repeatedly-tunable middleware. Smart proxies in Squirrel [90] are similar to smart fragments in AspectIX. Similar to MetaSockets [53], AspectIX can be repeatedly tuned in a finer-grained fashion than Squirrel. Squirrel and MetaSockets, however, do not provide the *aspect abstraction* that AspectIX provides, which is broader than the pipeline abstraction in Squirrel and filter abstraction in MetaSockets. We place AspectIX at the distribution layer because of its distributed object model that hides the interprocess communication from the developer.



(a) Distributed object model. (b) Aspect configuration. (c) Fragment exchange.

Figure 19: AspectIX: an aspect-oriented middleware based on the distributed object model [68].

5.2 Dependable Middleware

Dependable middleware supports critical distributed applications, such as military command and control and medical applications, that are required to be correctly operational. We further refine dependable middleware into three categories: reliable communcation, fault-tolerant, and load-balancer. Examples of each category are described in turn as follows.

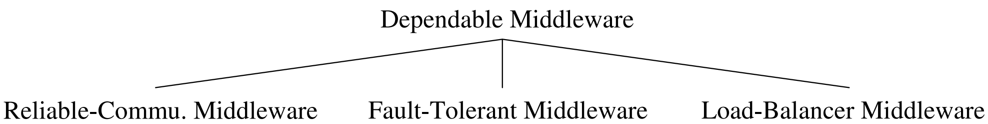


Figure 20: Taxonomy of Dependable middleware.

Reliable Communication Middleware. Reliable communication middleware provides reliable communication services, residing at the host-infrastructure layer and directly atop the network transport layer such as TCP, to support emerging application domains such as mobile computing, highly available, and fault-tolerant systems.

Miller et al. [73] at University of Wisconsin-Madison developed the *reliable sockets (Rocks)* that protect socket-based applications from network failures. Specifically, Rocks address connection failures in mobile computing using a hearbeat, reconfiguration, and reply mechanism. Examples of connection failures addressed by Rocks include unexpected modem disconnections and IP address change as a result of mobile device movements or DHCP lease expiration. Rocks resumes sessions automatically after recovering from a period of disconnestion. As depicted in Figure 21, Rocks employ an interception-based approach, using the "preloading" feature of the Linux loader [73], that interposes the Rocks library between the application code and the kernel TCP socket. The Rocks library exports the socket API, which is the same as the kernel socket API to be used transparently by the application, and the Rocks enhanced API (`RE-API`) to be used by Rocks-aware applications. Rocks monitor the TCP socket send and receive buffers and keep a copy of in-flight packets to prevent data loss in the presence of connection failure. After reconnection, Rocks first resends the packets in the in-flight buffers and then resumes the TCP socket to continue its normal operation. By definition, Rocks reside at the host-infrastructure layer. We consider Rocks as configurable middleware because the interposition of the Rocks library occurs during the application startup time.

The *reliable packets (Racks)* [73], also developed by Miller et al., are alternative solution to Rocks that solves the following problems introduced by Linux preloading feature. First, the preloading feature depends on the dynamic linker while, for security reasons, dynamic linker is disabled on "setuid" binaries (a binary that runs with extra privileges, but do not compromise security). Second, system libraries may not correctly support preloading because they might have used static calls that cannot be trapped using preloading. Finally, it is possible that other interposed libraries coexist with the Rocks library. The ordering of the interposed libraries affects the correct functioning of the Rocks library. As depicted in Figure 22, Racks are developed as a separate process (`Rackd`), as opposed to Rocks that use an in-process approach. Instead of intercepting the socket calls using the Rocks interposed library, Racks intercept and manipulates packets using a "packet filter" approach [96], which is a kernel mechanism that enables user processes to select and intercept outgoing and incoming packets. By definition, Racks reside at the common-services layer. We consider Racks as configurable middleware because of the need to register a packet filter with the kernel during the application startup time.

We noticed that all the fault-tolerant middleware projects that we studied from the literature (discussed next) provide fault-tolerance using service replication. All these projects provide consis-

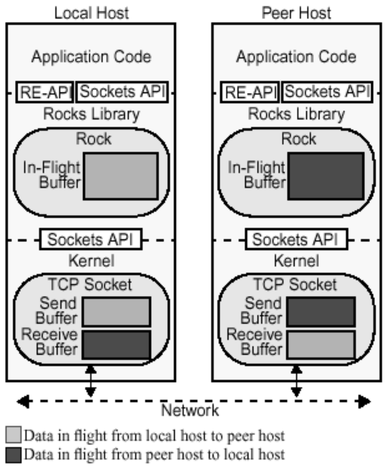


Figure 21: Rocks architecture [73].

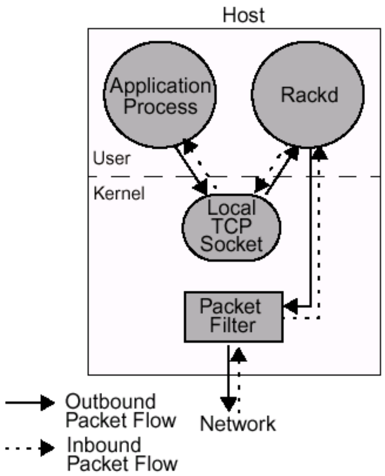


Figure 22: Racks architecture [73].

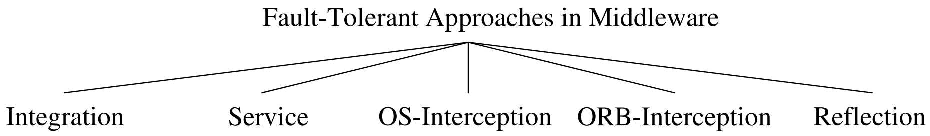


Figure 23: Different approaches to fault-tolerance in middleware.

tency among the service replicas using a reliable group communications service. Isis [97], Horus [98], Ensemble [99], Totem [100], and Coyote [101] provide such reliable group communications services. For similarity, we only describe Ensemble here. The Ensemble framework [99] from Cornell University supports protocol graphs constructed from fine-grained components. The framework supports QoS monitoring by inserting detectors in the protocol graph. These detectors can trigger dynamic adaptation by distributing a new protocol-graph specification to all involved participants using a reconfiguration protocol. We consider Ensemble as repeatedly-tunable middleware. Ensemble naturally resides at the host-infrastructure layer.

Fault-Tolerant Middleware. Fault-tolerant middleware enables applications to continue operating in the presence of faults using service replication. To keep the replicas consistent, fault tolerance middleware benefits from the reliable group communications services described before. As shown in Figure 23, we identified five different approaches to fault-tolerance in middleware: integration, service, interception, service+interception, and reflection. The first three categories were identified before by Narasimhan et. al [102] and the last two are introduced by the author. Examples of each approach are described in turn as follows.

Electra [103] from Softwired and *Orbix+Isis* [104] from IONA Technology and Isis Distributed Systems are two CORBA compliant ORBs that provide fault-tolerance by integrating object replication mechanism inside their ORBs. As depicted in Figure 24, adapter objects are used

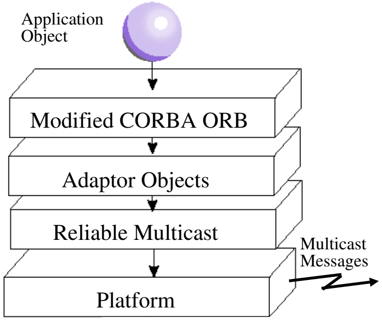


Figure 24: The integration approach [102].

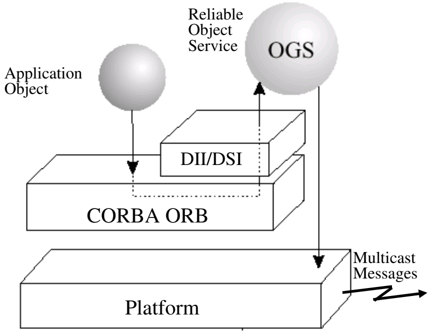


Figure 25: The service approach [102].

to enable the modified ORB to use the services provided by the reliable multicast. Orbix+Isis uses Isis [97] and Electra can use either Horus [98] or Isis [97] as their reliable multicast service. The integration approach is transparent to the application code, but requires both sides of application to use the same modified ORB. Naturally, Electra and Orbix+Isis reside at the distribution layer. We consider Electra and Orbix+Isis as configurable middleware because they can be configured during the application startup time.

Object Group Services (OGS) [105], developed by Pascal Felber, provides a CORBA object service [77] that supports fault tolerance for CORBA applications. As depicted in Figure 25, the OGS object uses CORBA DSI and DII interfaces [77] to receive requests from the client object. The OGS object then uses its fault-tolerant protocol to communicate with the service replicas. It then returns the reply to the application object. This approach is transparent to the ORB but is not transparent to the application objects. By definition, OGS resides at the common-services layer. We consider OGS as configurable middleware because of the need to associate the OGS objects to application objects during the application startup time.

The *Eternal system* [106] from UCSB and Eternal Systems, illustrated in Figure 26, provides fault-tolerance using an interception approach. Eternal intercepts system requests originated from unmodified CORBA ORBs (or Java Virtual Machines) targeted for the kernel TCP/IP protocol stack using the operating system user-level extensions [107]. Eternal captures the system calls using the "`/proc/pid`" file in Unix systems. For the reliable multicast service, Eternal uses Totem [100]. This approach is transparent to both the application and ORB. We place Eternal at the host-infrastructure layer, atop Totem. We consider Eternal as configurable middleware because of the configuration that occurs during the application startup time to set up the monitoring and capturing of system calls.

Interoperable Replication Logic (IRL) [74, 108], developed by Baldoni et al., and *fault-tolerant* *service (FTS)* [109], developed by Friedman et al., are two middleware examples that provide fault tolerance using both the service and interception techniques. IRL and FTS both use CORBA request portable interceptors [77] to intercept requests (requests, replies, and exceptions). Figure 27 illustrates the IRL basic architecture. The client request portable interceptor forwards request to the local proxy that provides a fault-tolerant service again using object replication. By definition, IRL and FTS reside at the common-services layer. We consider IRL and FTS as configurable middleware because they both require to introduce an `ORBInitializer` class to the application ORB during the application startup time (to register their specific request portable interceptors with the application ORB).

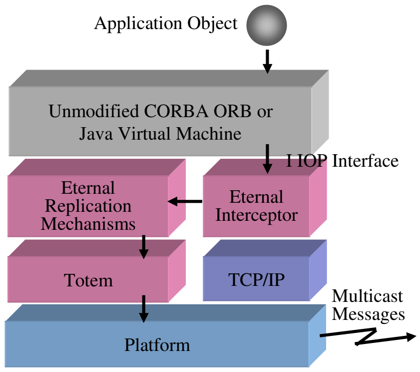


Figure 26: The Eternal architecture [79].

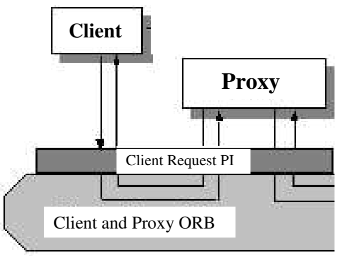


Figure 27: The IRL basic architecture [108].

Finally, the *FRIENDS system* [51], developed by Fabre et al., provides a meta-level architecture including libraries of meta objects for fault-tolerance, secure-communication, and group-based distributed applications. FRIENDS enables non-functional mechanisms to be implemented at the meta level. Reflection is recursively used to address various non-functional requirements: fault tolerance using several replication strategies, security using ciphering and authentication protocols, and communication using atomic multicast protocols. Figure 28 shows the multi-level implementation of the client-server protocol using fault-tolerant, security, and communication meta objects. A specialized MOP is developed in FRIENDS using open compilers [51], which enables intercepting the interactions between CORBA objects. Using this MOP, FRIENDS provides fault-tolerance replicas as CORBA objects (or meta objects). By definition, FRIENDS resides at the common-services layer. We consider FRIENDS as repeatedly-tunable middleware.

Load-Balancer Middleware. Load-balancer middleware enables distributed applications to continue operation even in the presence of high load. *TAO load balancing (TAO-LB)* [72], developed by Schmidt et al., adds load balancing to TAO [70]. TAO-LB employs an *adaptive on-demand architecture*, which works as follows. First, a client receives a handle to the load balancer instead of

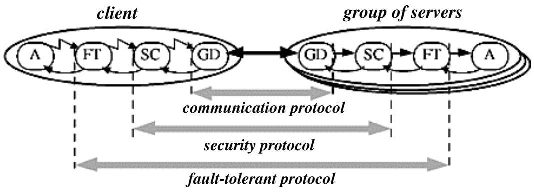


Figure 28: FRIENDS architecture [51].

the target object. Next, using CORBA standard LOCATION_FORWARD mechanism [77], the load balancer redirects the initial client request to the appropriate target object replica. The CORBA client continues using the new object reference (obtained as part of the LOCATION_FORWARD message) to communicate with this replica directly until it is either done or redirected again. An adaptive load balancer that forwards requests on demand can monitor the load on each replica continuously. Using this load information and the policies specified by the application, the load balancer can determine whether the load is distributed equally. When the load becomes unbalanced, the load balancer can communicate with the replica ORBs and ask them to redirect their clients back to the load balancer. The load balancer can then redirect the clients to less loaded replicas. Similar to IRL [74] and FTS [109], TAO-LB uses CORBA portable interceptors [77] to provide transparent load balancing to both applications and ORBs. By definition, TAO-LB resides at the common-services layer. We consider TAO-LB as configurable middleware because of the need to register its interceptors during the application startup time.

5.3 Embedded Middleware

Recent advances in portable computing devices have given rise to the need for embedded middleware, which supports applications that require small footprints and are limited in their available resources, especially memory. We group embedded middleware into two classes: minimum and swappable. Examples of each category are described in turn as follows.

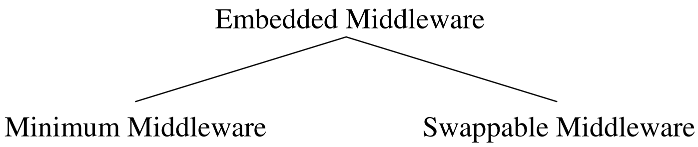


Figure 29: Taxonomy of Embedded middleware.

Minimum Middleware. Minimum middleware provides a minimum footprint middleware that can be used either by a specific domain of applications or just by one specific application. The former has a minimum core with a fixed API that enables adding optional feature to the minimum core. The latter, however, does not have a minimum core, hence, only the required features constitutes the middleware.

PersonalJava [110] and EmbeddedJava [76] constitute a minimum Java that reimplements the full set of Java APIs in order to fit into smaller devices with limited memory. PersonalJava is designed for "web-connected" devices such as set-top boxes, smart phones, and hand-held devices like PDAs. As depicted in Figure 30, a minimum standard core is required on every PersonalJava-enabled device to enable the web functionality. Unlike PersonalJava, EmbeddedJava enables automatic creation of customized APIs relative to the requirements of one application, as opposed to one application domain, which results in smaller footprints. As such, EmbeddedJava enables Java applications on very limited memory embedded devices, including industrial controllers, process controllers, and scientific instruments. Every EmbeddedJava application may include different set of classes since there is no required core functionality for all embedded devices as depicted in Figure 31. We place both PersonalJava and EmbeddedJava at the host-infrastructure layer because they support heterogeneous platforms. We consider both PersonalJava and EmbeddedJava as customizable middleware because they produce minimum applications during the application compile time.

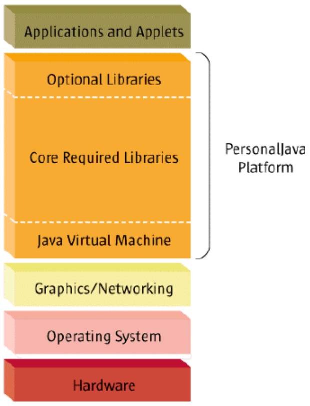


Figure 30: PersonalJava architecture [110].

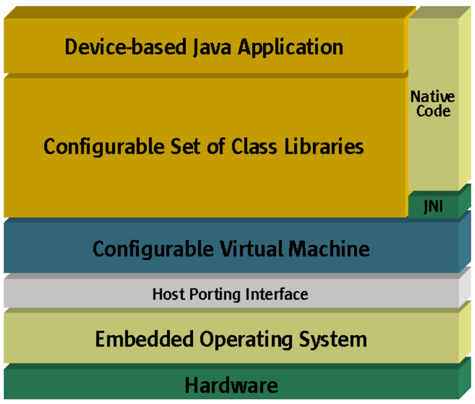


Figure 31: EmbeddedJava architecture [76].

Orbix/E [111] from IONA TECHNOLOGY is a light-weight and high-performance CORBA ORB designed for embedded devices. The size of an Orbix/E can be as small as 100KB for client and 150KB for server programs. We consider Orbix/E as customizable middleware because it allows a developer to generate customized versions of Orbix/E. We also consider Orbix/E as configurable middleware because of its ability to parse configuration files during the application startup time, for example, to load optional pluggable protocols. Similar to PersonalJava (and unlike Emebedded-Java), Orbix/E requires a minimum fixed core functionality. Orbix/E resides at the distribution layer.

Swappable Middleware. Swappable middleware enables optional portions of middleware to swap in and out conserving the amount of memory used for the middleware. *Universal Interoperable Core (UIC)* [50] is the successor of LegORB [49] both developed at UIUC. UIC, in addition to the small footprint provided in LegORB, can adopt one or more *personalities* such as CORBA, Java RMI, and DCOM for interoperability purposes. Figure 32 illustrates the interaction between the UIC core and its personalities. UIC personalities can be either customized statically during the application compile time, or tuned dynamically using late composition of components during run time. UIC minimum ORB core runs uninterruptedly while ORB strategies and servants are dynamically updated. We consider UIC as both customizable and repeatedly-tunable middleware. A UIC client-side ORB for PalmOS can be as small as 16KB. UIC exploits customizable adaptation for the rare and expensive changes during compile time, and exploits repeatedly-tunable adaptation for the frequent and inexpensive changes during run time. Using UIC, the same server objects can interoperate with different personalities without modifying their implementations. UIC naturally resides at the distribution layer.

ZEN [60], developed by Schmidt et al., is a TAO successor implemented in Java and Real-Time Java [112] that provides a micro-ORB architecture, as illustrated in Figure 33. ZEN identifies several major ORB services, such as object adapters and transport protocols, that can be moved out of the micro-ORB kernel. The virtual component pattern [5] is employed to make each service dynamically pluggable. Each ORB service itself is decomposed into smaller pluggable components that can be loaded into the ORB at run time only when required. Because of this feature, we consider ZEN as repeatedly-tunable middleware. ZEN also employs profiling and reflection techniques to monitor and inspect the optimized configuration found during the application tuning phase. The optimized configuration is written into a configuration file that can be used for next runs of the application. ZEN parses the configuration file during the application startup time and cofigures the middleware accordingly. Therefore, ZEN can also be considered as configurable middleware. ZEN naturally resides at the distribution layer.

6 The Big Picture

In this section, the three-dimensional taxonomy, introduced in Section 4, is mapped into three two-dimensional tables that provide a higher-level view of the detailed discussions provided in Section 5

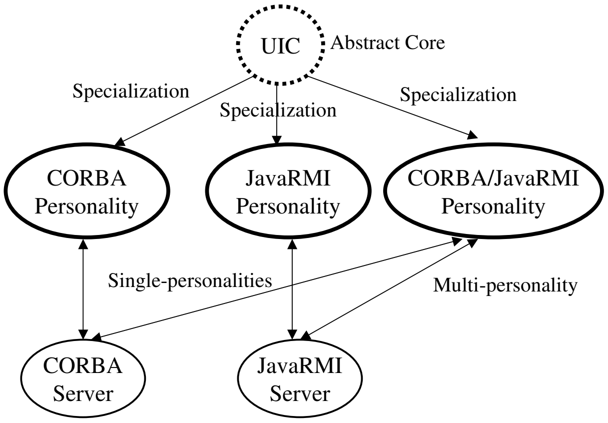


Figure 32: The UIC personalities [50].

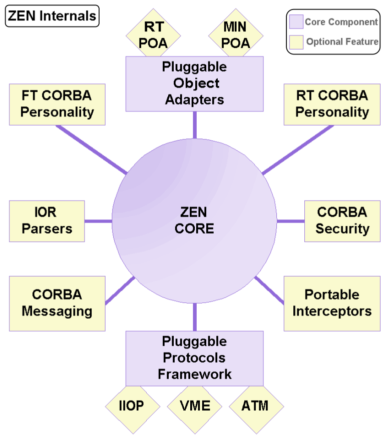


Figure 33: The ZEN architecture [60].

to better understand the adaptive middleware research status.

Tables 1 shows the adaptive middleware projects categorized with respect to the middleware layers and application domain. As this table shows, we were not able to find research projects in the domain-services layer (except the Boeing Bold Stroke [75], for which we could not find detailed documentation). This table also shows that in the embedded middleware research, the trend is toward all-in-one solutions (as opposed to layered approaches) because of the memory footprint limitation.

Table 1: Adaptive middleware examples categorized by middleware layers and application domain.

Middleware Layers	Application Domain		
	QoS-Enabled	Dependable	Embedded
Domain-Services			
Common-Services	QuO	OGS, IRL, FTS, TAO-LB, Racks	
Distribution	TAO, CIAO, Orbix, OpenORB, Squirrel, DynamicTAO, FlexiNET, OpenCorba, AspectIX	FRIENDS, Electra, Orbix+Isis	UIC, ZEN, Orbix/E
Host-Infra.	ACE, MetaSockets	Eternal, Isis, Horus, Ensemble, Totem, Rocks	PersonalJava EmbeddedJava

Table 2 categorizes the adaptive middleware projects using the middleware layers and adaptation type. This table shows that adaptive middleware research has exploited both static and dynamic adaptations. Mutable middleware has received the least attention, probably because there is still need for safe adaptation research such as [113, 114] to mature. Safe adaptation can harness the dangerous power of dynamic adaptation provided by mutable middleware. Table 2 also shows that the trend is toward hybrid solutions to adaptation.

Table 2: Adaptive middleware categorized by middleware layers and adaptation type.

Middleware Layers	Adaptation Type			
	Static		Dynamic	
	Customizable	Configurable	Tunable	Mutable
Domain-Services				
Common-Services	QuO	OGS, IRL, FTS, TAO-LB, Racks	FRIENDS	
Distribution	Squirrel*, UIC*, Orbix/E*	TAO*, CIAO, Orbix, Electra, Orbix+Isis, Orbix/E*, ZEN*	TAO*, DynamicTAO, UIC*, Squirrel*, OpenCorba, FlexiNet, AspectIX, ZEN*	OpenORB
Host-Infra.	PersonalJava EmbeddedJava	Eternal, Horus, Isis, Totem, Rocks	ACE, MetaSockets, Ensemble	

*: hybrid adaptation.

Table 3 shows the supporting paradigms employed by each adaptive middleware project. This table shows that computational reflection and software design patterns have been relatively more studied in the adaptive middleware research than aspect-oriented programming and component-based design. This table also shows that there are relatively less academic research projects on DCOM than on CORBA and Java RMI.

7 Conclusion

In this survey paper, a well-established taxonomy of traditional middleware was reintroduced. Among transactional, message-oriented, procedural, and object-oriented middleware categories, we focused on the object-oriented middleware, which is the basis for most adaptive middleware approaches discussed in this paper. Object-oriented paradigm provides limited support for adaptation in middleware. We identified four major paradigms, in addition to the object-oriented paradigm, that are critical to adaptive middleware research: computational reflection, component-based design, aspect-oriented programming, and software design patterns. We proposed a three-dimensional taxonomy of adaptive middleware to categorize different efforts for supporting adaptation in distributed applications using adaptive middleware. Finally, all the adaptive middleware projects discussed in this paper were put together in the big picture to provide a better understanding of the adaptive middleware research status.

Table 3: Paradigms used in each adaptive middleware project.

Adaptive Middleware			Ref	CBD	AOP	SDP	Compliance
QoS-Enabled Middleware	Real-Time	ACE				√	
		TAO				√	C
		DynamicTAO	√			√	C
		CIAO		√		√	C
		Orbix	√			√	C
	Stream-Oriented	OpenORB	√	√			
		OpenORB v2	√	√			D
		Squirrel		√		√	C
		MetaSockets	√	√	√	√	
	Reflection-Oriented	FlexiNET	√	√			C
		OpenCorba	√				C
	Aspect-Oriented	QuO			√	√	C, R
		AspectIX			√		C, R
Dependable Middleware	Reliable Communication	Isis					
		Horus					
		Ensemble					
		Totem					
		Rocks					
		Racks					
	Fault-Tolerant	Electra					C
		Orbix+Isis	√			√	C
		OGS				√	C
		Eternal				√	C, R
		IRL	√			√	C
		FTS	√			√	C
		FRIENDS	√				C
	Load-Balancer	TAO-LB	√			√	C
Embedded Middleware	Minimum	Orbix/E	√			√	C
		PersonalJava	√				R
		EmbeddedJava	√				R
	Swappable	UIC	√	√		√	C, R, D
		ZEN	√	√		√	C

Ref: computational reflection. CBD: component-based design. AOP: aspect-oriented programming. SDP: software design patterns. C: CORBA compliant. R: Java RMI compliant. D: DCOM compliant.

Adaptive middleware is still an ongoing research that requires more work in the following areas. First, domain-specific middleware services requires serious attention. Several projects have successfully provided common-services in middleware. To enable reuse and separation of concern in each specific application-domain, however, domain-specific middleware services should also be widely available. Second, mutable middleware provides a powerful and at the same time dangerous dynamic adaptation that are more likely than other types of adaptive middleware to turn an application into something totally different and unexpected. To benefit from mutable middleware, we should harness its power by techniques such as safe adaptation. Third, applying overlapping adaptations to a distributed application may cause inconsistency in the application. This is the same problem as feature interaction problem in pattern recognition that needs to be addressed in adaptive middleware also. Finally, we realized that there is no one adaptive middleware solution that can adapt all distributed applications. Finding an optimized adaptive middleware solution using current state-of-the-practice adaptive middleware approaches is not an easy task. A developer needs to know all available middleware approaches and should spend a lot of time and money to find the optimized solution. Developing tools, techniques and high-level paradigms that assist a developer in this tedious process is a useful research area that promotes development of adaptive 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`.

Acknowledgements: Dr. Philip K. McKinley supervised this work and led me through this research area. I am very grateful and would like to thank him for his invaluable advice. Dr. Betty H.C. Cheng and Dr. R. E. Kurt Stirewalt were always available for all my questions. I would like to thank them for their help and contribution in this work. I would also like to thank the many researchers whose work are cited in this survey paper.

References

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

[2] C. Szyperski, *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1999.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag LNCS 1241, June 1997.

[4] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture*, vol. 2. John Wiley, 2001.

[5] A. Corsaro, D. Schmidt, R. Klefstad, and C. O'Ryan, "Virtual component a design pattern for memory constrained embedded applications," in *The 9th Conference on Pattern Language of Programs (PLoP 2002)*, 2002.

[6] D. C. Schmidt, "Middleware for real-time and embedded systems," *Communications of the ACM*, vol. 45, June 2002.

[7] A. T. Campbell, G. Coulson, and M. E. Kounavis, "Managing complexity: Middleware explained," *IT Professional, IEEE Computer Society*, pp. 22–28, September/October 1999.

[8] D. E. Bakken, *Middleware*. Kluwer Academic Press, 2001.

[9] W. Emmerich, "Software engineering and middleware: a roadmap," in *Proceedings of the Conference on The future of Software engineering*, pp. 117–129, 2000.

[10] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[11] E. S. Hudders, *CICS: A Guide to Internal Structure*. Wiley, 1994.

[12] C. L. Hall, *Building Client/Server Applications Using TUXEDO*. Wiley, 1996.

[13] L. Gilman and R. Schreiber, *Distributed Computing with IBM MQSeries*. Wiley, 1996.

[14] M. Hapner, R. Burridge, and R. Sharma, "Java message service specification," tech. rep., Sun Microsystems, Nov. 1999.

[15] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," in *ACM Transactions on Computer Systems 2(1)*, pp. 39–59, March 1984.

[16] Open Group, *DCE 1.1: Remote Procedure Calls*, 1997.

[17] Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701, USA, *The Common Object Request Broker: Architecture and Specification Revision 2.2*, Feb. 1998.

[18] Java Soft, *Java Remote Method Invocation Specification, revision 1.5, JDK 1.2 edition*, Oct. 1998.

[19] Microsoft Corporation, *Microsoft COM Technologies - DCOM*, 2000. http://www.microsoft.com/-com/dcom.asp.

[20] Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701, USA, *The Common Object Request Broker: Architecture and Specification, Revision 2.0*, July 1995.

[21] S. Vinoski, "CORBA: integrating diverse applications within distributed heterogeneous environments," *IEEE Communications Magazine*, vol. 14, no. 2, 1997.

[22] http://www.cs.wustl.edu/∼schmidt/corba-overview.html.

[23] http://java.sun.com/docs/books/tutorial/rmi/overview.html.

[24] Microsolft Corporation, *COM: Delivering on the Promises of Component Technology*, 2000. http://www.microsoft.com/com/default.asp.

[25] G. S. Raj, "A detailed comparison of CORBA, DCOM, and Java/RMI (with detailed code examples)," *Object Management Group (OMG) whitepaper*, Sept. 1998.

[26] E. Chung, Y. Huang, S. Yajnik, D. Liang, J. C. Shih, C. Wang, and Y. Wang, "DCOM and CORBA side by side," tech. rep., Microsoft DCOM Technical White Paper, 1997.

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, New York, NY: Addison-Wesley Publishing Company, 1995.

[28] K. Lieberherr, *Adaptive Object-Oriented Software The Demeter Method*. PWS, 1996.

[29] R. Rao, "Implementational reflection in Silica," in *Proceeding of ECOOP'91*, pp. 251–267, Springer-Verlag, 1991.

[30] G. Kiczales, "Beyond the black box: Open implementation," *IEEE Software*, vol. 13, Jan. 1996.

[31] L. Bergmans and M. Aksit, "Composing crosscutting concerns using composition filters," *Communications of ACM*, pp. 51–57, October 2001.

[32] N. Wang, D. C. Schmidt, O. Othman, and K. Parameswaran, "Evaluating meta-programming mechanisms for ORB middleware," *IEEE Communications Magazine, Special Issue on Evolving Communications Software: Techniques and Technologies*, October 2000.

[33] C. Simonyi, "The death of computer languages, the birth of intentional programming," 1995.

[34] W. Harrison and H. Ossher, "Subject-oriented programming (a critique of pure objects)," in *OOPSLA'93*, 1993.

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

[36] G. Kiczales, J. d. Rivieres, and D. G. Bobrow, *The Art of Metaobject Protocols.* MIT Press, 1991.

[37] G. Blair, G. Coulson, and N. Davies, "Adaptive middleware for mobile multimedia applications," in *Proceedings of the 8th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pp. 259–273, 1997.

[38] S. Chiba, "A study on a compile-time metaobject protocol," 1996.

[39] J. McAffer, "Meta-level architectue support for distributed obejcts," in *Proceeding of Reflection'96*, (San Francisco, CA, USA), pp. 39–62, 1996.

[40] H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa, "Object-oriented concurrent reflective languages can be implemented efficiently," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (A. Paepcke, ed.), vol. 27, (New York, NY), pp. 127–144, ACM Press, 1992.

[41] H. Okamura, Y. Ishikawa, and M. Tokoro, "AL-1/D: A distributed programming system with multimodel reflection framework," in *Proceedings of the Workshop on New Models for Software Architecture*, Nov. 1992.

[42] Y. Yokote, "Kernel structuring for object-oriented operating systems: The Apertos approach," in *Object Technologies for Advanced Software, First JSSST International Symposium*, vol. 742, pp. 145–162, Springer-Verlag, 1993.

[43] P. W. Madany, N. Islam, P. Kougiouris, and R. H. Campbell, "Reification and reflection in C++: an operating systems perspective," Tech. Rep. UIUCDCS–R–92–1736, University of Illinois at Urbana-Champaign, Department of Computer Science, Urbana-Champaign, 1992.

[44] D. B. Orr, "Applications of meta-protocols to improve OS services," in *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pp. 101–105, May 1995.

[45] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhaes, and R. H. Campbell, "Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000)*, (New York), April 2000.

[46] 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.

[47] R. Hayton, *FlexiNet Open ORB Framework.* APM Ltd., Oct. 1997.

[48] T. Ledoux, "OpenCorba: A reflective open broker," *Lecture Notes in Computer Science*, vol. 1616, 1999.

[49] M. Roman, M. Mickunas, F. Kon, and R. H. Campbell, "LegORB and ubiquitous CORBA," in *Proc. IFIP/ACM Middleware'2000 Workshop on Reflective Middleware (RM2000)*, (New York, USA), April 2000.

[50] M. Roman, F. Kon, and R. H. Campbell, "Reflective middleware: From your desk to your hand." IEEE Distributed Systems Online Journal 2(5), 2001.

[51] J. C. Fabre and T. Perennou, "A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach," *IEEE Transactions on Computers*, vol. 47, no. 1, pp. 78–95, 1998.

[52] W. Cazzola and M. Ancona, "mChaRM: a reflective middleware for communications-based reflection," Tech. Rep. DISI-TR-00-09, Universita degli Studi di Milano, May 2000. http://www.disi.unige.it/person/CassolaW/references.html.

[53] S. M. Sadjadi, P. K. McKinley, and E. P. Kasten, "Architecture and operation of an adaptable communication substrate," in *The 9th International Workshop on Future Trends of Distributed Computing Systems (FTDCS '03)*, May 2003.

[54] M. Golm, "Design and implementation of a meta architecture for Java," Master's thesis, Friedrich-Alexander-University, Erlangen-Nurenburg, Jan. 1997.

[55] E. P. Kasten, P. K. McKinley, S. M. Sadjadi, and R. Stirewalt, "Separating introspection and intercession in metamorphic distributed systems," in *Proceedings of the IEEE Workshop on Aspect-Oriented Programming for Distributed Computing (with ICDCS'02)*, (Vienna, Austria), July 2002.

[56] 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.

[57] B. Redmond and V. Cahill, "Supporting unanticipated dynamic adaptation of application behaviour," in *16th European Conference on Object-Oriented Programming*, vol. 2374, (Malaga, Spain), June 2002.

[58] N. Wang, D. C. Schmidt, and M. Kircher, "Towards an adaptive and reflective middleware framework for QoS-enabled CORBA component model applications," *the Distributed System online special issue on Reflective Middleware*, 2003.

[59] A. Singhai, A. Sane, and R. H. Campbell, "Quarterware for middleware," in *Proc. 18th International Conference on Distributed Computing Systems (ICDCS'98)*, (Amesterdam, The Netherlands), pp. 192–201, May 1998.

[60] R. Klefstad, D. C. Schmidt, and C. O'Ryan, "Towards highly configurable real-time object request brokers," in *Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, April - May 2002.

[61] Sun Microsystems, http://java.sun.com/products/ejb/, *Enterprise JavaBeans Technology*, 2001.

[62] Object Management Group, http://www.omg.org/cgi-bin/doc?ptc/99-10-05, *CORBA Components Model - FTF drafts for MOF cahpter.*

[63] G. T. Sullivan, "Aspect-oriented programming using reflection and metaobject protocols," *Communication of the ACM*, October 2001.

[64] Z. Yang, B. Cheng, R. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley, "An aspect-oriented approach to dynamic adaptation," in *Proceedings of Workshop On Self-healing Software*, Nov. 2002.

[65] P. C. David, T. Ledoux, and N. M. N. Bouraqadi-Saadani, "Two-step weaving with reflection using AspectJ," in *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, (Tampa, USA), October 2001.

[66] J. A. Zinky, D. E. Bakken, and R. E. Schantz, "Architectural support for quality of service for CORBA objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, 1997.

[67] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sander, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E. Schantz, "AQuA: An adaptive architecture that provides dependable distributed objects," in *The 17th IEEE Symposium on Reliable Distributed Systems*, Oct. 1998.

[68] M. Geier, M. Steckermeier, U. Becker, F. J. Hauck, E. Meier, and U. Rastofer, "Support for mobility and replication in the AspectIX architecture," Tech. Rep. TR-I4-98-05, Univ. of Erlangen-Nuernberg, IMMD IV, 1998.

[69] C. R. Becker and K. Geihs, "MAQS: management for adaptive QoS-enabled services," in *Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, 1997.

[70] D. C. Schmidt, D. L. Levine, and S. Mungee, "The design of the TAO real-time object request broker," *Computer Communications*, vol. 21, pp. 294–324, April 1998.

[71] D. C. Schmidt, "The ADAPTIVE Communication Environment: An object-oriented network programming toolkit for developing communication software," *Concurrency: Practice and Experience*, vol. 5, no. 4, pp. 269–286, 1993.

[72] O. Othman, "The design, optimization, and performance of an adaptive middleware load balancing service," Master's thesis, University of California, Irvine, 2002.

[73] V. C. Zandy and B. P. Miller, "Reliable network connections," in *ACM MobiCom 2002*, (Atlanta), September 2002.

[74] R. Baldoni, C. Marchetti, A. Termini, "Active Software Replication through a Three-tier Approach," in *Proceedings of the 22th IEEE International Symposium on Reliable Distributed Systems (SRDS02)*, (Osaka, Japan), pp. 109–118, October 2002.

[75] D. Sharp, "Reducing avionics software cost through component-based product line development," in *The Software Technology Conference*, (Salt Lake City, UT), April 1998.

[76] Sun Microsystems, *EmbeddedJava Application Environment*. http://java.sun.com/products/-embeddedjava/.

[77] Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701, USA, *The Common Object Request Broker: Architecture and Specification Version 3.0*, July 2003. Available at http://doc.ece.uci.edu/CORBA/formal/02-06-33.pdf.

[78] OOC, "ORBacus trader. ORBacus for C++ and Java," tech. rep., OOC, Object Oriented Concepts, Inc., 2000. http://www.iona.com/products/orbacus_home.htm.

[79] L. Moser, P. Melliar-Smith, P. Narasimhan, L. Tewksbury, and V. Kalogeraki, "The eternal system: an architecture for enterprise applications," in *the 3rd International Enterprise Distributed Object Computing Conference (EDOC'99)*, July 1999.

[80] D. C. Schmidt and S. D. Huston, *C++ Network Programming: Mastering Complexity Using ACE and Patterns*. Addison-Wesley Longman, 2002.

[81] http://www.cs.wustl.edu/∼schmidt/ACE-overview.html.

[82] http://www.cs.wustl.edu/∼schmidt/TAO.html.

[83] K. Nahrstedt, H. hua Chu, and S. Narayan, "QoS-aware resource management for distributed multimedia applications," *Special issue on multimedia networking, J. High Speed Network*, Dec. 1998.

[84] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The case for reflective middleware," *Communications of the ACM*, vol. 45, pp. 33–38, June 2002.

[85] G. S. Blair, G. Coulson, A. Andersen, M. Clarke, F. M. Costa, H. A. Duran, R. Moreira, N. Paralavantzas, and K. B. Saikoski, "The design and implementation of open ORB version 2." IEEE Distributed Systems Online Journal 2(6), 2001.

[86] M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas, "An efficient component model for the construction of adaptive middleware," *Lecture Notes in Computer Science*, vol. 2218, 2001.

[87] ITU-T/ISO, *Reference Model for Open Distributed Processing, Parts 1,2,3.*, 1995. ITU-T X.901-X.904 — ISO/IEC IS 10746-(1,3,3).

[88] T. Watanabe and A. Yonezawa, "Reflection in an object-oriented concurrent language," in *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applicaitons (OPSLA'88)*, (San Diego, CA, USA), pp. 306–315, Sept. 1988.

[89] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin, "Supporting adaptive multimedia applications through open bindings," in *Proceedings of International Conference on Congurable Distributed Systems (ICCDS'98)*, May 1998.

[90] R. Koster, A. P. Black, J. Huang, J. Walpole, and C. Pu, "Thread transparency in information flow middleware," in *Proceedings of Middleware'01 (IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing)*, Springer Verlag, Nov. 2001.

[91] R. Koster, *A Middleware Platform for Information Flows*. PhD thesis, Department of Computer Science, University of Kaiserslautern, Germany, July 2002.

[92] R. Koster, A. P. Black, J. Huang, J. Walpole, and C. Pu, "Infopipes for composing distributed information flows," in *Proceedings of the International Workshop on Multimedia Middleware*, pp. 44–47, ACM, Oct. 2001.

[93] A. Goldberg and D. Robson, *Samlltalk-80*. Addison-Wesley, 1989.

[94] R. Schantz, J. Loyall, M. Atighetchi, and P. Pal, "Packaging quality of service control behaviors for reuse," in *ISORC 2002, The 5th IEEE International Symposium on Object-Oriented Real-time distributed Computing*, (Washington, DC), April 29 - May 1 2002.

[95] M. van Steen, P. Homburg, and A. S. Tanenbaum, "The architectural design of Globe: A wide-area distributed system," Tech. Rep. 422, vrije Universiteit - Faculty of Mathematics and Computer Science, Amsterdam - Netherlands, March 1997.

[96] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *USENIX Winter*, pp. 259–270, 1993.

[97] K. P. Birman and R. van Renesse, "Reliable distributed computing with the Isis toolkit," *IEEE Computer Society Press*, 1994.

[98] R. V. Renesse, K. P. Birman, B. B. Glade, K. Guo, M. Hayden, T. Hickey, D. Malki, A. Vaysburd, and W. Vogels, "Horus: A flexible group communications system," Tech. Rep. TR95-1500, 23, 1995.

[99] R. van Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. Karr, "Building adaptive systems using ensemble," *Software Practice and Experience*, vol. 28, p. 963979, August 1998.

[100] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulis, and T. P. Archambault, "The Totem system," in *FTCS-25: 25th International Symposium on Fault Tolerant Computing Digest of Papers*, (Pasadena, California), pp. 61–66, 1995.

[101] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu, "Coyote: a system for constructing fine-grain configurable communication services," *ACM Transactions on Computer Systems*, vol. 16, no. 4, pp. 321–366, 1998.

[102] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "The interception approach to reliable distributed CORBA objects," in *Proceedings of the 3rd USENIC Conference on Object-Oriented Technologies and Systems (COOTS)*, USENIX, 1997.

[103] S. Maffeis, "Adding group communication and fault-tolerance to CORBA," in *Proceedings of the Conference on Object-Oriented Technologies*, pp. 135–146, 1995.

[104] "Orbix+Isis programmer's guide," Tech. Rep. D071-00, Isis Distributed Systems, Inc., IONA Technologies, Ltd., 1995.

[105] P. Felber, B. Garbinato, and R. Guerraoui, "Towards reliable CORBA: Integration vs. service approach," in *Special Issues in Object-Oriented Programming* (M. Mühlhäuser, ed.), pp. 199–205, dpunkt-Verlag, 1997.

[106] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "Transparent fault tolerance for enterprise applications," in *International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*, (L'Aquila, Italy), July-August 2000.

[107] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman, "Extending the operating system at the user-level: the Ufo global file system," pp. 77–90, 1997.

[108] C. Marchetti, L. Verde, and R. Baldoni, "CORBA request portable interceptors: A performance analysis," in *the 3nd International Symposium on Distributed Objects and Applications (DOA 2001)*, (Rome, Italy), Sept. 2001.

[109] R. Friedman and E. Hadad, "Client side enhancements using portable interceptors," in *the 6th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS01)*, January 2001.

[110] Sun Microsystems, *PersonalJava Application Environment*. http://java.sun.com/products/-personaljava/.

[111] IONA Technology, *Orbix/E*. http://www.iona.com/products/orbix-e.htm.

[112] K. Nilsen, "Issues in the design and implementation of real-time java," *Java Developers Journal*, 1996.

[113] B. H. Cheng, Z. Yang, and J. Zhang, "Enabling safe dynamic adaptation," Tech. Rep. MSU-CSE-03-11, Department of Computer Science, Michigan State University, East Lansing, Michigan, May 2003.

[114] N. Venkatasubramanian, "Safe composability of middleware services," *Communications of the ACM*, vol. 45, June 2002.