Active Mediation to Enable Efficient Use of Multiple Web Services

David Liu[1], Jun Peng[2], Neal Sample[3], Kincho H. Law[4], and Gio Wiederhold[5]

Abstract

Fundamental to using computational services over the web is the ability to compose them, and transfer results from one service as input to the next service. This paper presents the use of active mediation in applications that employ multiple web services which are not fully compatible in terms of data formats and contents. Active mediation allows an application acting as a client of web services to modify the interfaces of the remote services. This approach increases the applicability of the services, reduces data communication among the services, and enables the application to control complex computations. A number of features are presented to support active mediation: (1) mobile classes are introduced to allow dynamic information processing; (2) active mediators are incorporated into web services to support the execution of the mobile classes; and (3) service access protocols are designed for invoking active mediation. The importance of active mediation is illustrated with various applications, such as relational operations, dynamic type conversion, and result extraction. Finally, it is shown that active mediation can greatly improve the performance of an application utilizing multiple web services.

Keywords

Web service; composed application; service composition; direct data transmission; active mediation; mobile class

---

[1] Ph.D. Student, Department of Electrical Engineering, Stanford University, Stanford, CA 94305. E-mail: davidliu@stanford.edu
[2] Research Associate, Department of Civil and Environmental Engineering, Stanford University, Stanford, CA 94305. E-mail: junpeng@stanford.edu
[3] Ph.D. Student, Department of Computer Science, Stanford University, Stanford, CA 94305. E-mail: nsample@stanford.edu
[4] Professor, Department of Civil and Environmental Engineering, Stanford University, Stanford, CA 94305. E-mail: law@stanford.edu
[5] Professor, Computer Science Department, Stanford University, Stanford, CA 94305. E-mail: gio@db.stanford.edu

1 Introduction

We experience a continuous increase in both the size and the performance of computer networks. As networks become pervasive and ubiquitous, all computing facilities can be accessed from any geographic location. This development enables the use of remote software services over the web. Rather than constructing software applications by writing and acquiring components at the application site, the application creator can use functionalities provided by existing remote services. Delegating the maintenance of software that has not been written by oneself is an important benefit [Wiederhold 2003]. The use of Commercial off-the-shelf (COTS) to compose large programs was foreseen already decades ago and become part of the object-oriented approach to software construction [Mcllroy 1969]. Consistency of interfaces and data semantics is crucial in that approach.

This vision of software composition was rescaled into the megaprogramming framework, where major software components are provided as autonomous services managed by independent service providers [Boehm and Scherlis 1992; Wiederhold et al. 1992]. However, autonomous services are typically heterogeneous and adhere to a variety of conventions for control and data. Lightweight services were hypothesized to deal with required data conversions and keep the megaprograms simple. Even when standards are promulgated, such as SQL [Ullman 1988], the precise meaning and scope of the output will not necessarily match the expectations of another service. A prime example of available autonomous services today are information providers, which expose their functionalities through XML, SQL, and report generators, but are not geared to interoperate with other services, as analytical services or predictive simulations [Wiederhold 2002]. Web services are a special type of autonomous services that are made available on the web [Roy and Ramanujan 2001]. Web services range from comprehensive services such as customer relationship management to more specific services such as travel reservation, book purchasing, weather forecasts, financial data summaries, and newsgathering. Other services include engineering, logistics, and business services [Wiederhold and Jiang 2000].

The composition of multiple web services into a composed application consists of three phases. First, existing services that provide composable functionalities are catalogued. Existing services may decide to participate and expose their interfaces. Missing services are constructed, or, rather, their construction by others is encouraged or contracted. Second, the composed application is specified so that it will employ the most suitable combination of web services. Many issues need to be considered when composing web services, including scalability of the services, robustness of the services, security of the service interaction, effective and convenient specification of the compositions, and performance of the composed applications. Third, the composed application is executed.

This paper introduces active mediation to enhance efficient execution of applications employing composed services. Active mediation allows code to be provided to remote services to resolve format and content incompatibilities [Liu et al. 2003b]. Without being able to delegate such a capability to the remote service such incompatibilities have to be resolved at the application site. All results from one web service now have to be shipped to the application site, handled there, and then shipped to the next web service. This inefficiency is implicit in all common composition protocols, such as CORBA [Vinoski 1997], DCOM [Eddon and Eddon 1998], Microsoft .NET [Kirtland 2000], and SOAP [Box et al. 2000]. The concept of active mediation will have a major impact on the semantic web [Berners-Lee et al. 2001]. In that motivating article the composed application is operated on a handheld device, but the cost of shipping intermediate data to and from the handheld is not addressed.

The remainder of this paper is organized as follows. Section 2 reviews the background and related work in service composition and mediation technology. Section 3 describes active mediation in more detail and presents our implementation for service composition. Section 4 illustrates three application scenarios of active mediation. Section 5 summarizes the benefits of active mediation and then examines the performance of active mediation. Section 6 provides a summary and reviews the implications of active mediation to service composition.

2 Background and Related Work

This section provides a brief review of service composition technology. We describe our reference service composition infrastructure which allows dataflow to be separated from control flow. Furthermore, we review the related work in mediation technology which is applied to resolve incompatibilities among autonomous sources and services. The combination of the two leads us into active mediation, where the mediation tasks are performed remotely to deal with the problem of efficiently handling composition of autonomous web services. These web services process substantial amount of data and/or are controlled by nodes that have limited bandwidth.

2.1 Software Services and Service Composition

An autonomous software service is an independent process that provides computation or information on request, perhaps for pay. It will expect data input, perhaps only a simple request, and respond with results. The application that invokes such a service can simply present the results to its customer – the most common mode today – or direct the result to another service for further processing.

The services live on the web, primarily on their owner's sites. At those sites they have access to background material, are maintained, and service logs are kept. Services are located at diverse physical locations, and access method is via the networks. Figure 1 illustrates a typical architecture that consists of many services interconnected by a communication network to conceptually form a composed application. Each service has four layers:

- The “Host” layer represents the hardware platform the service runs on. This layer provides the hardware means for executing application instructions and routing data through the communication network.
- The “Operating System” layer provides software support for the system resource required by the service. It manages the processes of software applications that perform the service. It also provides protocol support for the network intercommunications among different hardware platforms. For instance, the TCP/IP [Comer 2000] protocol support belongs to this layer.
- The “Access Protocol” layer provides protocol support for accessing the data and the functionalities of the service. A service client running in one kind of operating system can communicate with a service in another operating system. The access protocol defines how to encode a request in order to invoke a service. It also specifies the manner in which the service responds to the request.

- The "Autonomous Service" layer is the application layer, which exports the functionalities of the service. Data mapping is conducted at this layer so that the service can exchange information with its clients in a mutually understandable fashion.

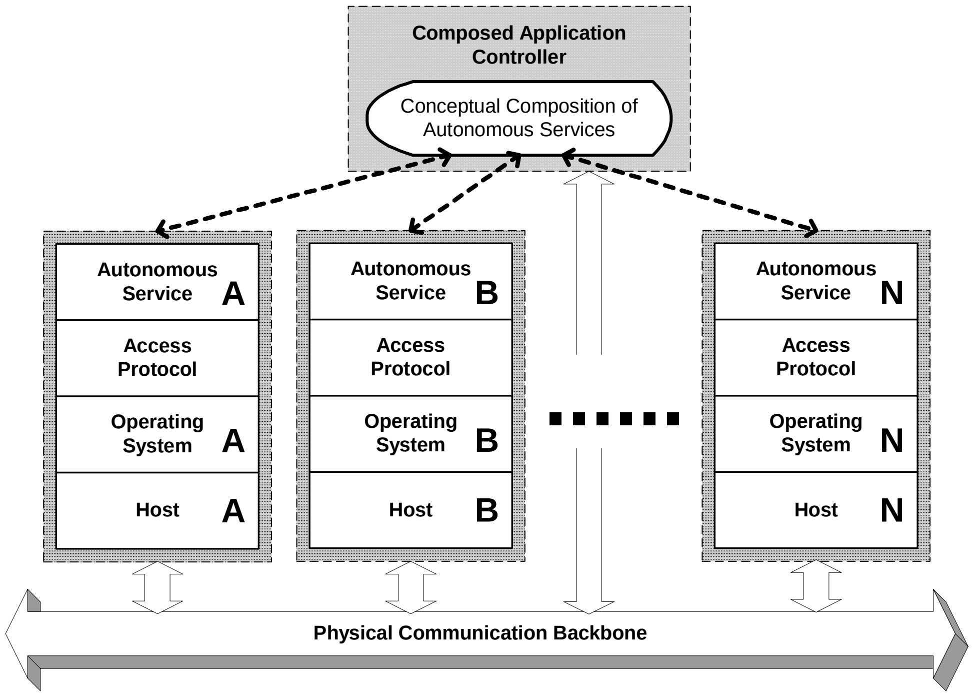

Figure 1: Hierarchical Model of Services

We use FICAS (Flow-based Infrastructure for Composing Autonomous Services), as described in more detail in earlier papers [Liu et al. 2003a; Liu et al. 2002], for the infrastructure of our service composition. Since there is an overhead for each remote invocation of a service, FICAS focuses on the composition of large and distributed services. The composed application execution model is similar to that of Idealized Worker Idealized Manager (IWIM) model [Arbab et al. 1993; Papadopoulos and Arbab 1997], where a composed program selects appropriate processes to perform a set of sub-tasks. The composed program is known as the manager, and the processes are called workers for that manager. In the case of FICAS, the composed application is the manager and services are workers.

A distinguishing characteristic of FICAS is its distributed data-flow model, which allows direct data-flow to occur among remote services. Control, i.e., the invocation of a remote service, remains centralized. In the common alternatives for remote service management, the site of the composed application is the central hub for all the control and all the data traffic, so that their model has both centralized control and centralized data-flow. The distributed data-flow model provides better performance and scalability than the centralized data-flow model. The distribution of data communications utilizes the network capacity among the services, and avoids bottlenecks at the composed application. Especially when the composed application resides on a mobile device, relying on centralized data-flow would severely stress its limited bandwidth. Figure 2 shows the structural model defined for a service. The service core encapsulates the computational software and provides the data processing functionalities. The data-flow and the control processing are distinct. The control-flow interface covers the event processing and the state management of the service core. The data-flow interface deals with the moving of data elements between the data containers and the processing of the data elements by the service core. While each component operates asynchronously, the service core ties its components into a coordinated entity.

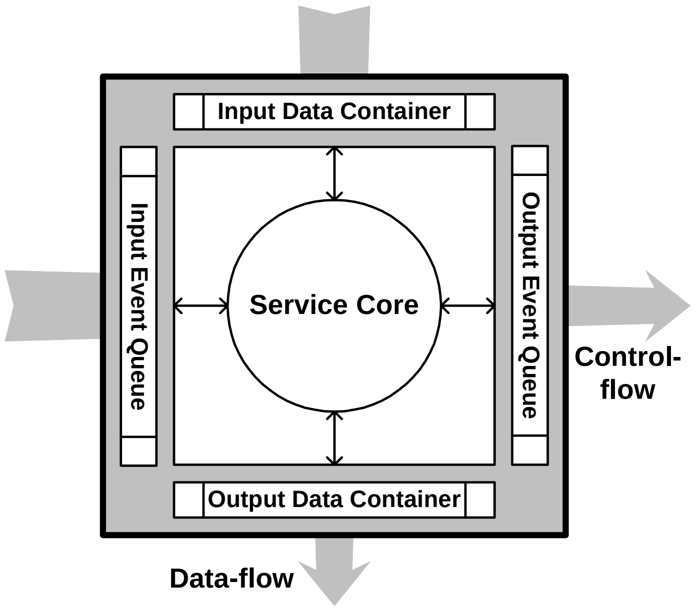

Figure 2: Structural Model of Autonomous Service

Services export the functionalities of their encapsulated software. Although their capabilities differ, the protocol used to export the functionalities is similar. The services share many common components, such as the event queues and the data containers. In addition, the interactions among the components are largely identical. Hence, the construction of a service is significantly simplified by assembling the common components into a standard module, which we call the service wrapper. The wrapper facilitates the encapsulation of computations as services, and provides the support for accessing the services through an event-based access protocol, called Autonomous Service Access Protocol (ASAP) [Liu et al. 2002].

While FICAS is used as the reference infrastructure in our research, the derived results can be applicable to other compositional frameworks, such as Globus [Foster and Kesselman 1997] or Ninja Paths [Gribble et al. 2001]. Although each framework has its own features (e.g., brokering, security, etc.), they are all based on the concept of services as network-enabled entities that provide some functionalities through the exchange of messages.

2.2 Mediation

Services are usually built by leveraging existing software capabilities and information resources. These resources have had incompatibilities similar to those now seen in web services. Mediators are intelligent middleware that sit between the information sources and the clients [Wiederhold 1992; Wiederhold and Genesereth 1997]. Mediators reduce the complexity of information integration and minimize the cost of system maintenance. They provide integrated information, without the need to integrate the actual information sources. Specifically, mediators perform functions such as accessing and integrating domain-specific data from heterogeneous sources, restructuring the results into objects, and extracting appropriate information.

Figure 3(a) illustrates the mediation architecture, which conceptually consists of three layers. The information source provides raw data through its source access interface. The mediation layer resides between the information source and the information client, performing value-added processing by applying domain-specific knowledge processing. The information client accesses the integrated information via the client access interface. The architecture of the service can be mapped to the mediation architecture, as shown in Figure 3(b). The application software resides in the information source layer, the service wrapper resides in the mediation layer, and the composed application resides in the information consumer layer. The software is accessed through the application specific interface. The service wrapper obtains the information from the services and exposes its capabilities through the access protocol.

In traditional mediators, code is written to handle information processing tasks at the time the mediators are constructed. Such mediators are static, and only modified when the sources change interfaces or behavior. Static mediators are appropriate when resource behavior is known at construction time. Mediators remain distinct from specific services, but enable integrated information from multiple services to be supplied to their clients. In contrast, the active mediators introduced in this paper allow clients to adapt the services, in particular their interfaces.

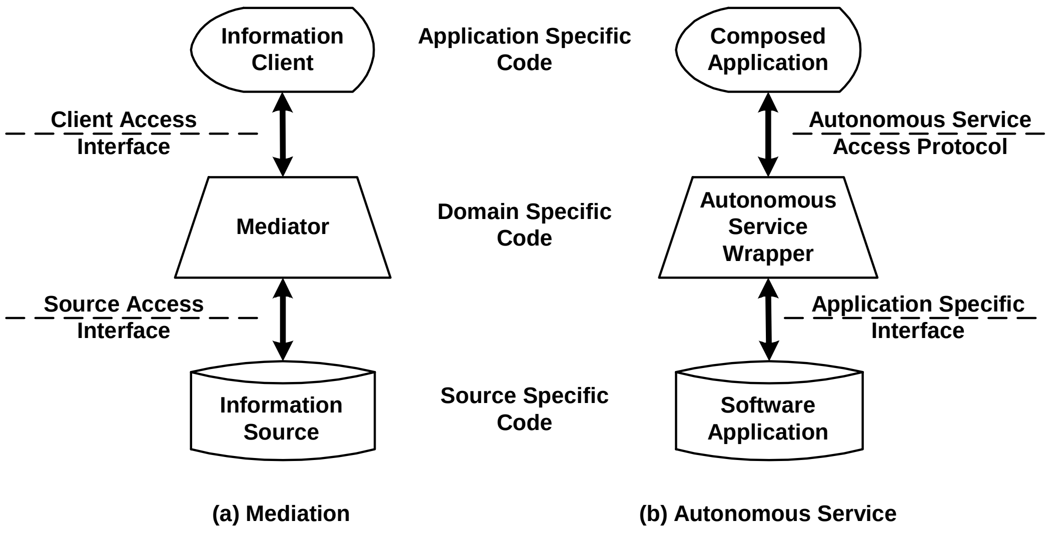

Figure 3: Conceptual Layers in Mediation and Service

3 Active Mediation

While the FICAS approach gains from direct data communication among the services, it does not have the capability provided in mediator nodes to map incompatible sources or to integrate information from diverse sources. But in the semantic web setting, where we expect a large collection of autonomous and diverse providers, we cannot expect that each service will deliver results that will be fully compatible and useful to further services that the composed application will need to invoke.

Active mediation applies the notion of mobile code [Fuggetta et al. 1998] to provide unforeseen remote information processing. Specifically, matching, reformatting, rearranging, and mapping of data being sent or received among services can be embodied in mobile code, and shipped by the composed application to the remote service as needed. Remote services that can accept active mediation now have the ability to adapt their behavior to the client requests. For instance, an information client can forward a compression routine to a service so that queried information is compressed before returned. A service that computes monthly taxes due state-by-state can have chronological transaction data rearranged and summarized as it needs. In general, with active mediation to provide client-specific functionalities, services can be viewed as if they were intended for the specific needs of the client.

We will now provide some specifics about our implementation of mobile classes and their roles in the active mediation of composed services.

3.1 Mobile Classes

A mobile class is an information-processing module that can be dynamically loaded and executed. Conceptually, a mobile class is a function that takes some input data elements, performs certain operations, and then outputs a new data element. Mobile classes can be implemented in many general-purpose programming languages. In our work, Java is chosen as the specification language for mobile classes. First, Java is suitable for specifying computational intensive tasks. There are many available standard libraries that provide a wide range of computational functionalities. Second, Java has extensive support for portability. Java programs can be executed on any platform that incorporates a Java virtual machine. Third, Java supports dynamic linking and loading. Java class files are object files rather than executables in the traditional sense. Linking is performed when the Java class files are loaded onto the Java virtual machine. Compiled into a Java class, the mobile class can be dynamically loaded at runtime.

Figure 4 presents the *MobileClass* interface, which contains a single function that represents the functionality of a mobile class. The *execute()* function takes a vector of data elements as the input and generates a data element as the output. The *execute()* function is overloaded by a mobile class to provide specific processing functionality. Figure 5 shows the definition of the *DataElement* class representing a data element, which is used for data exchange among services in FICAS. Since services and mobile classes use the same representation for data, data can be exchanged among the services and the mobile classes without any conversion. Internally, a data element is represented in XML. There are two constructors for *DataElement*, one for creating an empty data element, the other for creating a data element based on its XML representation. The class provides functions to query the type and the size of the data element. In the case that the data element is of a primitive type (i.e., boolean, integer, real, or string), functions are provided to set, fetch and compare values for the data element. Otherwise, the content of the data element can be fetched as a byte array.

Figure 6 shows a simple mobile class that converts data from integer to float. The *int2float* class implements the *MobileClass* interface. The *execute()* function takes the first argument for the mobile class as the input data, converts the data from an integer number into a floating point number, and returns the floating point number as the output data element.

```
public interface MobileClass {
  public DataElement execute(Vector params);
}
```

Figure 4: Definition of the MobileClass Interface

```
public class DataElement {
  public DataElement();
  public DataElement(Document doc);

  Document doc();             // Return XML document representation
  byte[] getByteArray();      // Return byte array representation
  String toString();          // Return a string in XML printout form

  int getSize()               // Return the size of the element
  int getType()               // Return the type of the element

  DataElement setValue(boolean value);
  DataElement setValue(double value);
  DataElement setValue(int value);
  DataElement setValue(java.lang.String value);
  DataElement setValue(byte[] arr);

  boolean getBooleanValue();  // Return boolean value
  int getIntValue();          // Return integer value
  double getRealValue();      // Return double value
  String getStringValue();    // Return string value

  int compare(DataElement e);
  boolean eq(DataElement e);  // Equal to the argument
  boolean ge(DataElement e);  // Greater than or equal to
  boolean gt(DataElement e);  // Greater than
  boolean le(DataElement e);  // Less than or equal to
  boolean lt(DataElement e);  // Less than
  boolean ne(DataElement e);  // Not equal to
}
```

Figure 5: Definition of the DataElement Class

```
public class int2float implements MobileClass
{
  public DataElement execute(Vector params) {
    DataElement arg = (DataElement) params.firstElement();
    int val = arg.getIntValue();
    double result = new Double(val).doubleValue();
    return new DataElement().setValue(result);
  }
}
```

Figure 6: Example Mobile Class that Converts Data from Integer to Float

Once coded, the mobile class is compiled into a Java class and put into a repository with a known URL. The Java class will be looked up later when the mobile class is invoked by a composed application. The following statement can be used to invoke the mobile class from a composed application:

Variable = MCLASS (mclassname, param1, param2, …)

The argument *mclassname* refers to the name of the mobile class, followed by the input parameters for the mobile class. The parameters for the invocation of the mobile class can be either literals or variables. Literals represent constant value, and variables represent the data elements. When the statement is executed, the composed application first locates the mobile class. The name of the mobile class, along with the URL of the repository, determines where the Java byte codes for the mobile class can be located. The location to find the Java byte codes is determined in the following rules:

- If the name of the mobile class starts with "*http://*", then the URL for loading the Java byte codes can be obtained by appending "*.class*" to the name of the mobile class. For example, if the mobile class name is "*http://mobile.class.repository/int2float*", then the Java class file for the mobile class can be found at "*http://mobile.class.repository/int2float.class*".
- If the name of the mobile class is a normal string, then the URL for loading the Java byte codes can be obtained by prefixing the URL of the mobile class repository and appending "*.class*" to the name of the mobile class. For example, if the URL for the mobile class repository is "*http://mobile.class.repository*", then the Java class file for the mobile class *int2float* can be located at "*http://mobile.class.repository/int2float.class*".

Once a Java class file is located, it is loaded onto a service for execution. The *execute()* function of the mobile class is invoked at that time.

3.2 Enabling Active Mediation

To enable active mediation in FICAS, a composed application needs to be able to invoke mobile classes on a service, and the service needs to support the execution of the mobile classes.

To allow a composed application to coordinate the invocation of mobile classes on services, two events, MCLASS and MCLASSREPLY, are added to the event-based service access protocol. As shown in Table 1, the MCLASS event is sent from a composed application to a service to invoke a mobile class, and the MCLASSREPLY is used by the service to acknowledge the composed application.

Table 1: Mobile Class Events in the ASAP Protocol

Event Type	**Event Syntax**

MCLASS	`<EVENT>`
	  `<NAME>` **`MCLASS`** `</NAME>`
	  `<ASID>` *source-service* `</ASID>`
	  `<ASID>` *destination-service* `</ASID>`
	  `<FID>` *flow-id* `</FID>`
	  `<CLASS>` *mclass-name* `</CLASS>`
	`</EVENT>`

MCLASSREPLY	`<EVENT>`
	  `<NAME>` **`MCLASSREPLY`** `</NAME>`
	  `<ASID>` *source-service* `</ASID>`
	  `<ASID>` *destination-service* `</ASID>`
	  `<FID>` *flow-id* `</FID>`
	  `<REPLY>` *reply* `</REPLY>`
	`</EVENT>`

The MCLASS event initiates the invocation of the mobile class. The *source-service* field specifies the composed application that initiates the request. The *destination-service* field specifies the target service that executes the mobile class. The *flow-id* field identifies the request. It is also used to tag all the input and output parameters for the mobile class. The *mclass-name* specifies the name of the mobile class. After the execution of the mobile class, the target service generates a MCLASSREPLY event to inform the composed application.

A service supports the execution of mobile classes through the incorporation of an active mediator. Figure 7 illustrates the architecture of the active mediator:

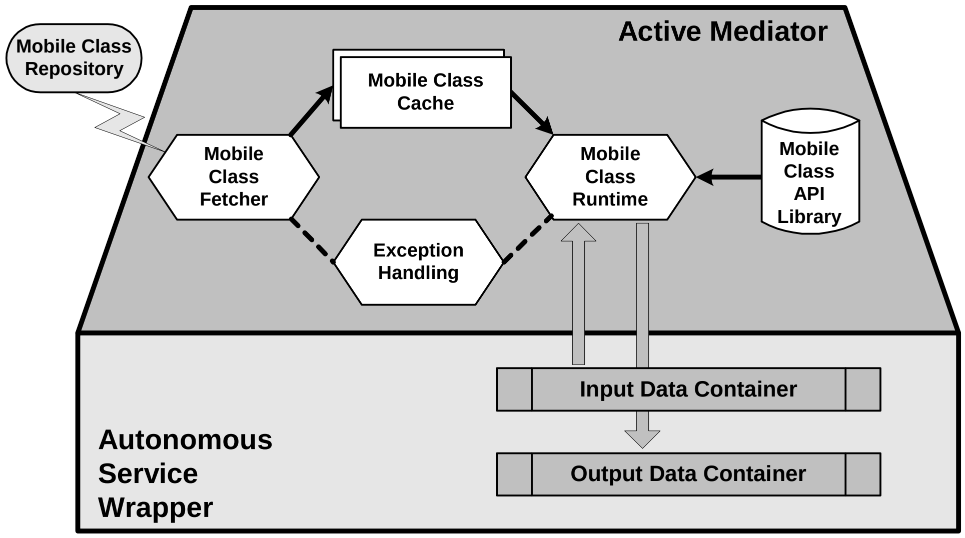

Figure 7: Architecture of the Active Mediator

- The Mobile Class Fetcher is responsible for loading the mobile class. The source location of the Java class is specified by the *mclass-name* in the MCLASS event. The loaded Java class is stored into the Mobile Class Cache.
- The Mobile Class Cache is a temporary storage for mobile classes. The Mobile Class Cache is used to avoid the duplicate loading of a mobile class. It is looked up every time before any mobile class is loaded. Only when the cache miss occurs, the Mobile Class Fetcher is used to load the Java byte code.
- The Mobile Class API Library stores the utility classes that make the construction of mobile classes more convenient. For instance, the Java Development Kit library [Arnold et al. 2000] is provided as part of the Mobile Class API Library.
- The Mobile Class Runtime is the execution engine for the mobile classes. To execute a mobile class, the Mobile Class Runtime loads the Java class from the Mobile Class Cache and invokes the *execute()* function. The input parameters of the *execute()* function are

looked up from the Input Data Container using the *flow-id* contained in the MCLASS event. The result of the *execute()* function is put into the Output Data Container.

- The Exception Handling module provides error handling for the loading and the execution of the mobile class.

Upon receiving the MCLASS event, a service directs the Mobile Class Fetcher to load the mobile class into the Mobile Class Cache. The *execute()* function of the mobile class is then invoked to process data local to the service. Since the service wrapper handles the interchange of the data among services, the active mediator is only concerned with the data processing local to the service.

4 Application of Active Mediation

In terms of functionality, a mobile class is similar to a lightweight service. Both can provide modularized computational functionalities to a composed application. However, a mobile class is not independent. Whereas even a lightweight service is managed independent of a composed application, a mobile class is created and maintained within the client application. It is shipped to and executed in a computational service when needed. Whereas any service is a process, a mobile class is only a piece of code, dynamically loaded and executed at runtime with a service. More importantly, mobile classes and services serve different purposes. A few application scenarios are discussed to help identify when the mobile class is useful in facilitating service composition.

4.1 Mobile Class for Data Processing

To demonstrate the capability of mobile classes, we examine how mobile classes are used to support data processing. Specifically, we look at examples using relational data operations. Table 2 lists the relational operators, their relational algebra representations, and the corresponding mobile class interfaces. The relational operators conduct processing on one or more input relations, and generate a new relation as the output. A mobile class is constructed for each relational operator. The input relations of the relational operator are the input parameters to the mobile class, and the value returned by the mobile class corresponds to the output relation of the relational operator. For the mobile classes, the input and the output relations are encapsulated as data elements. The encoding of the relations into data elements is predefined and understood by the mobile classes. Various schemes may be used. For instance, Peng et al. [2003] discussed using XML as the data representation standard for encoding scientific data, including relational tables.

The following presents a brief discussion of the relational operators and their mobile class implementations:

- Unary Operators (σ, π): The select operator σ selects tuples that satisfy a given predicate condition. The mobile class implementation of a select operator takes a relation as the input data element, checks the condition on every tuple within the relation, and generates a result data element that contains all the satisfying tuples. The project operator π reduces the number of columns in a relation with only the desired attributes left. The mobile class implementation of a project operator takes a relation as the input data element, truncates all the undesired attributes, and returns the resulting relation as the output.
- Set Operators ($\bigcup$, $\bigcap$, $-$): The union operator $\bigcup$ returns the tuples that appear in either or both of the relations. The intersection operator $\bigcap$ returns only the tuples that appear in both of the relations. The difference operator – returns the tuples that appear in the first relation but are not in the second relation. The mobile class implementations of the set operators take two relations as the input data elements, perform the set operation on the relations, and return the resulting relation as the output.
- Combination Operators ($\times$, $\bowtie$): The Cartesian product operator $\times$ associates every tuple of the first relation with every tuple of the second relation. The theta join operator $\bowtie$ combines a selection with Cartesian product, forcing the resulting tuples to satisfy a specific predicate condition. The mobile class implementations of the combination operators take two relations as the input data elements, perform the combination operations on the relations, and return the resulting relation as the output.

Table 2: Relational Operators and Their Corresponding Mobile Classes

Operator	**Relational Representation**	**Mobile Class**
Select	$O = \sigma_{cond} (A)$	O = MCLASS(“select”, A)
Project	$O = \pi_{attr} (A)$	O = MCLASS(“project”, A)
Union	$O = A \bigcup B$	O = MCLASS(“union”, A, B)
Intersect	$O = A \bigcap B$	O = MCLASS(“intersect”, A, B)
Difference	$O = A - B$	O = MCLASS(“difference”, A, B)
Cartesian product	$O = A \times B$	O = MCLASS(“cartesian”, A, B)
Theta join	$O = A \bowtie_{cond} B$	O = MCLASS(“join”, A, B)

```
public class Select implements MobileClass
{
  public DataElement execute(Vector params) {
    DataElement arg = (DataElement)params.elementAt(0);
    Element source =
      (Element)arg.doc().getElementsByTagName("TABLE").item(0);

    NodeList rows = source.getElementsByTagName("ROW");

    Document result = new DocumentImpl();
    Element root = result.createElement("TABLE");

    for (int i=0; i<rows.getLength(); i++) {
      Element row = (Element) rows.item(i);
      if (condition_is_met(row))
        root.appendChild(InsertRow(result, row));
    }

    result.appendChild(root);

    return new DataElement(result);
  }

  ...
}
```

Figure 8: Example Mobile Class that Implements the Select Operator

Figure 8 shows the mobile class that implements the select operator σ. The input data element contains the input relation encoded in XML. The mobile class first decodes the input data element, then applies the selection condition on individual rows, and finally inserts the selected rows into the result relation. The result relation is encoded into a data element before it is returned as the output of the mobile class.

4.2 Mobile Class for Type Mediation

Data generated by a service can be directly used by other services if they share the same data types, structure, formats, and granularities, etc. However, such homogeneity cannot be assumed within a large-scale service composition infrastructure. Data exist in various types and will continue to appear in different types that suit different applications. Many potential services never took the need for composition into account, assuming their results were what the consumer desired. Data type conversion is inevitable in supporting service composition.

Traditionally, a service serving as type broker or a distributed network of type brokers is used to mediate the difference among data in various formats [Ockerbloom 1998]. The type brokers can convert data from one format to another acceptable format for the information client. The type brokers serve as proxies connecting client requests with appropriate source services. A type graph is used to figure out a chain of necessary conversions. An example of automating this process can be seen in [Chandrasekaran et al. 2000]. There are two issues associated with using type brokers: efficiency and availability.

First, the use of type brokers for type mediation can be inefficient. Large amount of data are forwarded to and from the brokers. The problem is exacerbated when a chain of conversions is involved. Figure 9(a) presents an example of data-flows in the type-broker architecture. Data from the source service are represented in the type T1, and the destination service consumes data in the type T3. Two type brokers are employed to convert source data from the type T1 to the type T3. Large amount of data are passed among the type brokers.

Second, the necessary type brokers may not exist for the desired data type. Since there are a large number of data formats, it is impractical to prepare a comprehensive set of type brokers covering all existing and future data types. New type brokers need to be created and maintained to conduct the type conversions needed by an application. However, type brokers, acting as services, are expected to be maintained independently of the composed applications.

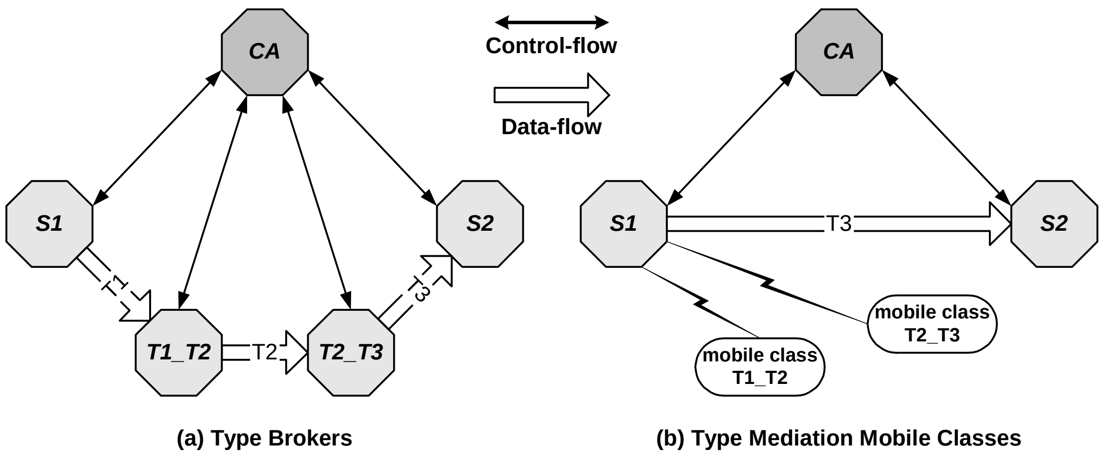

Figure 9: Type Mediation Using Type Brokers and Mobile Classes

In our approach, mobile classes are used in place of type brokers to handle data conversion and integration of multiple input streams. The mobile classes are created by the application programmers as part of the specification for the composed applications. Rather than forwarding data among the type brokers, the composed application loads the mobile classes on the services to provide the type mediation functions. Similar to the chain of type brokers, multiple mobile classes for type mediation can be utilized together. As shown in Figure 9(b), two mobile classes are used to convert data from type T1 to type T3. The type mediation is conducted at the source service, where the source data of type T1 is converted to type T3. Data in the consumable format T3 is directly sent to the destination service. Since the mobile classes are invoked on the source service, the multiple interim data transfers are eliminated and the data traffic is limited to essential transmissions. The application of the mobile classes successfully addresses the efficiency and availability issues associated with type mediation.

4.3 Mobile Class for Result Extraction

Services can produce a wide variety of data suitable for extraction and reporting [Sample et al. 2002]. A composed application has upstream services generating data that is consumed by downstream services. Selective result extraction is required when the output model of an upstream service is incompatible with the input model of a downstream service. For instance, an upstream service delivers an SQL cursor, but the downstream service expects a relation. In this example, an upstream service produces data progressively, while the downstream service consumes the data as a whole. Such a mismatch of how data is produced and consumed will stymie the downstream recipient. This is when mobile classes become valuable. A mobile class can be constructed to scroll the SQL cursor to fill a complete relation, and return the relation as the output. The mobile class is loaded onto the upstream service to mediate the output data for the downstream service. As the result, both services can collaborate despite the difference in how one generates data and how the other consumes the data.

Services are invariably built with the expected clients in mind. However, if their services are valuable, they will acquire more clients and be composed into more complex scenarios. Changing their interfaces would frustrate their original, intended clients. But ignoring new opportunities would cause the services to fall into disuse and be replaced by newer services with similar functionalities [Wiederhold 2003].

The difference in how an upstream service produces data and how a downstream service consumes data can present a seemingly insurmountable block to effective composition of services, particularly when the number of collaborating services and their applications grow. The use of active mediators, implemented through mobile classes, resolves the problems stemming from such incompatibilities.

5 Benefits and Performance of Active Mediation

Active mediation allows services to conduct client-specific information processing. We first list the benefits in general terms and then provide some algorithms for optimizing the performance. A number of application scenarios are abstracted in this section to demonstrate how active mediation addresses many important issues in conducting effective and efficient service composition.

5.1 Benefits of Active Mediation

With active mediation, web services can separate service-specific functionalities from client-specific functionalities, hence providing services as if they were constructed on a per-client basis. Particularly, active mediation provides benefits in the following areas:

- Increasing flexibility of services: Web services are built and maintained by service providers who are under their own administrative control. It is difficult, if not impossible, for service providers to anticipate all current and future clients, and to provide them with information in a ready-to-use form. Furthermore, there are inevitable delays in modifying the functionalities and interfaces of web services to satisfy the specific requirements from clients. Modifying an existing service for one class of clients will have unexpected effects on other clients. As the number of clients of a service increases, the service provider becomes more reluctant to make significant changes to the service. Active mediation allows service clients to dynamically expand the functionalities of a service, hence increases the customizability and flexibility of the service.

- Reducing communication traffic among services: Processing routines require and produce data, which has to be transmitted. In current composition approaches, all of this traffic passes through the site where the controlling application resides. In particular, an invocation of a service is carried out in a fashion similar to that of remote procedure calls [Birrell and Nelson 1984]. Parameters are sent to the services and the results are returned back to the composed application, which then processes and dispatches the data to other services. With active mediation intermediate data, which is often voluminous, bypasses the application node, reducing both overall traffic and demands on the controlling node. The controlling node will transmit code segments to the services sending or receiving data as appropriate for execution. The effect is yet a substantial reduction of data communication among the services and the composed application.
- Facilitating specification of computations: It is important to separate the compositional and computational specifications of a composed application [Perrochon et al. 1997]. This may be achieved by completely removing computational primitives from a compositional language, as in the case of CLAM [Sample et al. 1999]. The drawback of this approach is the difficulty in specifying application logic. Separate services need to be constructed even for handling simple arithmetic operations. While separating the compositional and computational specifications, the composed applications can enable complex computations by specifying and dynamically executing processing routines on services that are enabled with active mediation.

5.2 Placement of Mobile Classes

The placement of a mobile class has significant impact on the performance of the composed application. An example composed application, as shown in Figure 10, is used to demonstrate such impact. The composed application involves two services and one mobile class. Service *S1* randomly generates and returns a string whose size is specified by the input parameter. Service *S2* takes a string as input and immediately returns without doing anything. The mobile class *FILTER* takes a large string as input, filters through the content, and returns a string that consists of every tenth character of the input string. Effectively, the mobile class compresses the content by ten fold. Since the mobile class can be executed on any one of the services involved in the composed application, there are three potential placement strategies, as shown in Figure 11:

- <u>Strategy 1</u>: By placing the mobile class *FILTER* at the service that hosts the composed application controller, we can construct the execution plan as shown in Figure 11(a). *S1* generates the data element *A* and passes it to the composed application. The mobile class processes the data element *A* at the composed application, and the result *B* is then sent to *S2* for further processing.
- <u>Strategy 2</u>: By placing the mobile class *FILTER* at *S1*, we can construct the execution plan as shown in Figure 11(b). *S1* generates the data element *A* and processes it locally using the mobile class. The result *B* is sent from *S1* to *S2* for further processing.
- <u>Strategy 3</u>: By placing the mobile class *FILTER* at *S2*, we can construct the execution plan as shown in Figure 11(c). *S1* generates the data element *A* and passes it to *S2*. *S2* processes the data locally using the mobile class and then uses the result *B* for further processing.

To compare the strategies, we assume that the performance of loading and executing the mobile class is the same on all services. Strategy 1 requires both the input data element *A* and the output data element *B* to be transmitted among the composed application and the services. Thus Strategy 1 incurs the most communication traffic compared to the other two strategies and has the worst performance. Strategy 2 and Strategy 3 differ in the data content sent between the services. For Strategy 2, the data element *B* is sent from *S1* to *S2*. For Strategy 3, the data element *A* is sent from *S1* to *S2*. Since the data element *B* is one tenth in size compared to the data element *A*, Strategy 2 incurs the least amount of communication traffic. Therefore, Strategy 2 is the placement strategy that has the best performance.

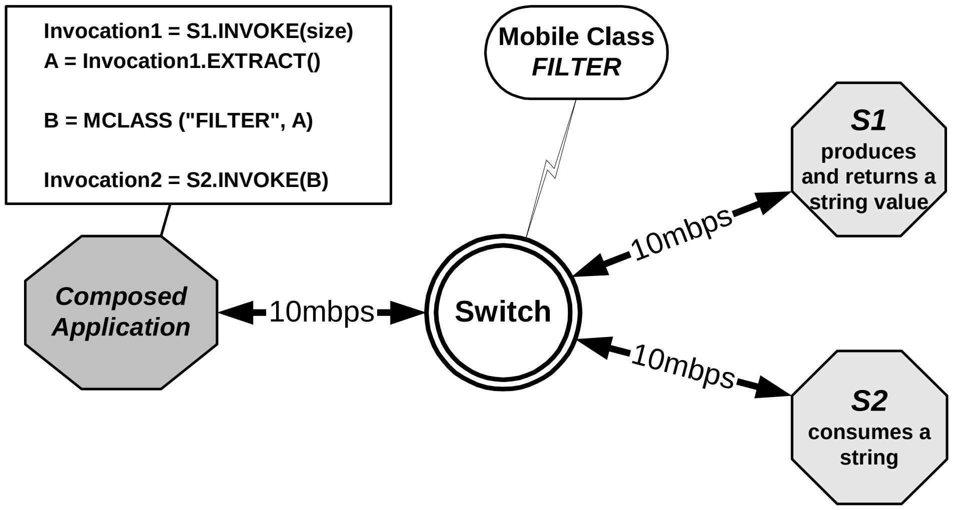
Invocation1 = S1.INVOKE(size)
A = Invocation1.EXTRACT()
B = MCLASS ("FILTER", A)
Invocation2 = S2.INVOKE(B)
Mobile Class FILTER
S1 produces and returns a string value
10mbps
Composed Application
10mbps
Switch
10mbps
S2 consumes a string

Figure 10: Example Composed Application that Utilizes the Mobile Class FILTER

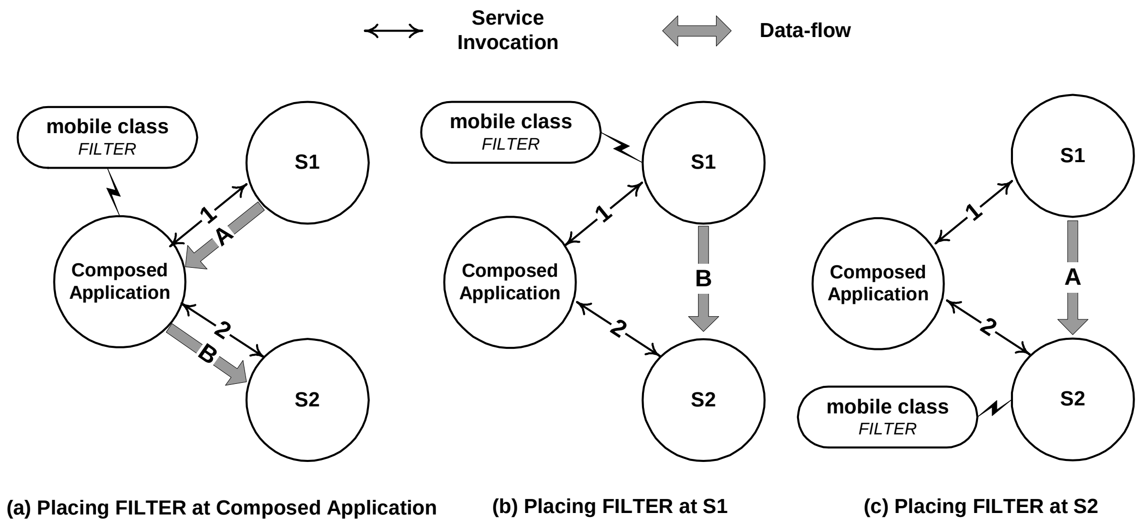

Figure 11: Execution Plans with Different Placements for the Mobile Class

Following the analysis of the above example, we derive an algorithm to determine the optimal placement of the mobile class. The algorithm seeks to locate the service that minimizes the data-flows among the services. Each input data element of the mobile class is modeled as a pair, (S_i, V_i), where S_i is the service that generates the *i*th input data element, and V_i is the volume of the data element. The output data element is modeled as (S_0, V_0), where S_0 is the service to which the result of the mobile class will be sent, and V_0 is the size of the output data element. Two observations are made. First, the sum of V_i remains the same regardless where the mobile class is executed. Second, by placing the mobile class on the service S_i, we can eliminate the corresponding data-flow volume V_i, since the data element is local to the service. Therefore, the optimal placement of the mobile class is the service S_i that has the largest aggregated V_i.

Figure 12 shows the Largest Data Set (LDS) algorithm that selects the service that generates and consumes the largest volume of data. The algorithm first computes the total amount of data associated with each service. Then, the service with the largest data volume is selected. The return of this algorithm is the service that represents the optimal placement of the mobile class.

```
INPUT:  input pairs($S_1$, $V_1$), …, ($S_n$,$V_n$)
        output pair ($S_0$, $V_0$)
OUTPUT: $S_{max}$
METHOD:
        $V_{max}$=0
        for every unique S in input and output pairs
              V=0
              for i=0,…,n
                    if $S_i$==S
                      V=V+$V_i$
              if V>$V_{max}$
                    $S_{max}$=S
                    $V_{max}$=V
```

Figure 12: LDS Algorithm for Optimal Placement of Mobile Class

5.3 Enabling Optimization for Mobile Classes

The LDS algorithm is applicable when the sizes of the input and output data elements are known for the mobile class. However, in many cases, the size of the output data element is only known after the execution. A mechanism to predict the size of the output data element is needed. This is handled by the *sizing function* of the mobile class. The sizing function is defined as $S_O = f(S_A, S_B, \ldots)$, where S_O is the size of the output data element, and S_A, S_B, and etc. are the sizes of the input data elements. The sizing function may be stored within the mobile class. The composed application controller uses the sizing function to calculate the size of the output data element based on the sizes of the input data elements.

Two special types of mobile classes have the simplified sizing functions. The first type is called the *expansion mobile class*, whose output data element is at least as large as the sum of the input data elements. Based on the LDS algorithm, the optimal mobile class placement would be the service that utilizes the result of the mobile class. In this case, the sizing function can be set to return infinity. The other special type is called the *compression mobile class*, whose output data element is smaller than at least one of the input data elements. The optimal mobile class placement is one of the services that generate the input data elements. In this case, the sizing function can be set to return zero.

The effectiveness of the LDS algorithm depends on the accuracy of the sizing functions. We demonstrate through examples of how the sizing function may be obtained. The relational mobile classes (as defined in Section 4.1) are used, because the relationships between the sizes of input and output data elements are well defined [Korth and Silberschatz 1991]. The sizing functions, as shown in Table 3, can be formulated in the following manner:

- The unary operators *select* and *project* return portions of the input relations. The mobile classes implementing the operators are by definition the compression mobile class. Hence, their sizing functions return zero.
- The *union* operator combines the two input relations. The mobile class is an expansion mobile class. Hence, the sizing function returns infinity.
- The *intersect* and *difference* operators return portions of the input relations. Their mobile classes are therefore compression mobile classes, and the sizing functions return zero.
- The result set of *cartesian product* operator contains all possible combinations of one tuple from each input relations. The result relation is generally larger than the input relations. Therefore, the sizing function returns infinity.
- The sizing function for the *join* operator is more complex. The sizing function depends on the characteristics of the input data and the predicate condition. For instance, if the join is an equality join with uniformly distributed values in input relations, the sizing function may be set to $S_0=c \times S_A \times S_B$, where c is the selection factor of the join. If the result relation is expected to be rather small, the sizing function can be set to zero to let the LDS algorithm choose one of the services that generate the input data elements. If the result relation is

expected to be large, the sizing function can be set to infinity to force the LDS algorithm to choose the service that utilizes the output data element.

Table 3: Sizing Functions for the Relational Mobile Classes

Mobile Class	**Sizing Function**
O = MCLASS(“select”, A)	$S_o = 0$
O = MCLASS(“project”, A)	$S_o = 0$
O = MCLASS(“union”, A, B)	$S_o = \infty$
O = MCLASS(“intersect”, A, B)	$S_o = 0$
O = MCLASS(“difference”, A, B)	$S_o = 0$
O = MCLASS(“cartesian”, A, B)	$S_o = \infty$
O = MCLASS(“join”, A, B)	$S_o = f(S_A, S_B)$

We have shown that the sizing function does not need to be precise for the LDS algorithm to be effective. The rules are similar to those seen in optimizing relational expressions [Ullman 1988]. In many cases, the sizing function is simplified to a constant. As the mobile class becomes more complex, the sizing function becomes harder to be represented in a mathematical formula. In some cases, the size of the output data element cannot be determined based on the size of the input data elements. As a future research direction, statistical methods may be used to adaptively estimate the correlations between the sizes of the output data element and the input data elements.

5.4 Performance Analysis

We analyze the performance of the composed application previously defined in Figure 10. The composed application is executed using different placements of the mobile class *FILTER*. We intend to measure the impact of the placement of the mobile class on the performance of the composed application. In addition, we replace the mobile class *FILTER* with a web service that implements the same functionality. Overall, three strategies are considered:

- Strategy 1: The mobile class *FILTER* is placed on service *S1*. The placement of the mobile class is generated by using the LDS algorithm.
- Strategy 2: We can also place the mobile class *FILTER* on service *S2*.
- Strategy 3: We implement a lightweight web service that replaces the mobile class *FILTER*. The string generated by *S1* is fed into the service, and the result is forwarded onto *S2* for further processing.

Figure 13 shows the execution times of these three different strategies. Different settings on the size of the string generated by *S1* are used. The following observations are made:

- The execution times of the composed applications increase with the size of the string. Three factors contribute to the increased execution times. First, longer time is taken to measure the size of the string. It results in the longer execution time for the LDS algorithm. Second, it takes longer to execute the mobile class or the utility service. Third, the larger string needs longer transmission time.
- The placement of the mobile class significantly impacts the performance of the composed application. Although both using mobile class, Strategy 1 performs significantly better than Strategy 2. Strategy 1 utilizes the LDS algorithm to minimize the amount of data-flows. In Strategy 2, *S1* transmits the original string to *S2*. Whereas in Strategy 1, *S1* only transmits the filtered string to *S2*. Strategy 1 causes significantly less amount of data traffic than Strategy 2.
- Both strategies involving the mobile class perform better than Strategy 3, which uses a lightweight web service. Strategy 3 incurs the most data-flow, as both the original string and the filtered string are transmitted among the services. In addition, the invocation of any remote service is more costly than the invocation of the mobile class within a service.

In summary, active mediation enabled by the mobile class is an effective approach in improving the performance of the composed application. The mobile class can be placed onto an appropriate service to minimize the amount of data communications. We have not considered here the effects of having the controlling node on a low bandwidth device.

However, mobile devices are very attractive to manage complex scenarios in engineering [Liu et al. 2003a], healthcare [Berners-Lee et al. 2001], and military situations [Gray 2000; Sheng et al. 1992]. In that case the benefits of distributed dataflow, enabled by active mediation, are even more striking [Liu 2003].

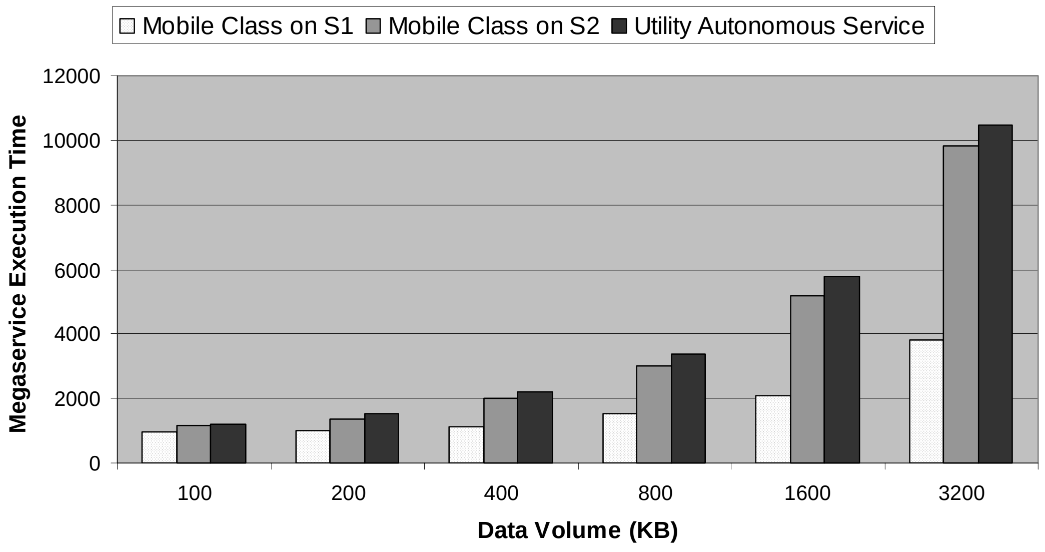

Figure 13: Performance Comparison Between Mobile Class and Service

6 Summary

Active mediation increases the customizability and flexibility of web services. It utilizes code mobility to facilitate dynamic information processing in service composition. This paper presents a conceptual framework for active mediation. An architecture is defined to enable smooth adoption of active mediation in service composition.

Active mediation allows data processing tasks to be specified for composed applications, at the same time separating computation from composition. This paper describes a spectrum of potential application scenarios that can be effectively addressed by active mediation. Specifically, valuable mediation functionalities such as relational operations, dynamic type conversion and extraction model mediation fit nicely into the active mediation framework. Through the discussion of the application scenarios, we present the effectiveness and flexibility of active mediation in conducting service composition.

The performance of active mediation in the service composition is analyzed. The fact that mobile classes may be executed on alternative locations enables us to conduct performance optimization. An algorithm that determines the optimal placement of mobile classes is introduced, and the applicability of the algorithm is discussed. Used appropriately, active mediation can greatly facilitate service composition, both in functionality and in performance.

Acknowledgement

This work is partially sponsored by the Center for Integrated Facility Engineering at Stanford University, the Air Force (Grants F49620-97-1-0339 and F30602-00-2-0594), and the Product Engineering Program headed by Dr. Ram D. Sriram at NIST (National Institute of Standards and Technology). The authors would also like to acknowledge an equipment grant from Intel Corporation for the support of this research.

References

Arbab, F., Herman, I., and Spilling, P. 1993. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5, 1, February, 23-70.

Arnold, K., Gosling, J., and Holmes, D. 2000. *The Java programming language*. Addison-Wesley, Boston, MA.

Berners-Lee, T., Hendler, J., and Lassila, O. 2001. The semantic web. *Scientific American*, 284, 5, May, 34-43.

Birrell, A. D., and Nelson, B. J. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2, 1, February, 39-59.

Boehm, B., and Scherlis, B. 1992. Megaprogramming. In *Proceedings of DARPA Software Technology Conference*. Los Angeles, CA, 68-82.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H., Thatte, S., and Winer, D. 2000. *Simple object access protocol (SOAP)*. W3C Note, http://www.w3.org/TR/SOAP.

Chandrasekaran, S., Madden, S., and Ionescu, M. 2000. *Ninja paths: An architecture for composing services over wide area networks*. Technical report, UC Berkeley, http://ninja.cs.berkeley.edu/dist/papers/path.ps.gz.

Comer, D. E. 2000. *Internetworking with TCP/IP, volume I, principles, protocols, and architecture*. Prentice Hall, Upper Saddle River, NJ.

Eddon, G., and Eddon, H. 1998. *Inside Distributed COM*. Microsoft Press, Redmond, WA.

Foster, I., and Kesselman, C. 1997. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11, 2, 115-128.

Fuggetta, A., Picco, G. P., and Vigna, G. 1998. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24, 5, 342-361.

Gray, R. S. 2000. Soldiers, agents and wireless networks: A report on a military application**.** In *Proceedings of the Fifth International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*, Manchester, England, April.

Gribble, S. D., Welsh, M., von-Behren, J. R., Brewer, E. A., Culler, D. E., Borisov, N., Czerwinski, S. E., Gummadi, R., Hill, J. R., Joseph, A. D., Katz, R. H., Mao, Z. M., Ross, S., and Zhao, B. Y. 2001. The Ninja architecture for robust Internet-scale systems and services. *Computer Networks*, 35, 4, 473-497.

Kirtland, M. 2000. The programmable web: Web services provides building blocks for the Microsoft .Net framework. *MSDN Magazine*, September.

Korth, H. F., and Silberschatz, A. 1991. *Database system concepts*. McGraw-Hill, Columbus, OH.

Liu, D. 2003. *A distributed data flow model for composing software services*. Ph.D. Thesis, Stanford University, Stanford, CA.

Liu, D., Cheng, J., Law, K. H., Wiederhold, G., and Sriram, R. D. 2003a. An engineering information service infrastructure for ubiquitous computing. *Journal of Computing in Civil Engineering*, 17, 4, 219-229.

Liu, D., Law, K. H., and Wiederhold, G. 2002. Data-flow distribution in FICAS service composition infrastructure. In *Proceedings of* the *15th International Conference on Parallel and Distributed Computing Systems*. Louisville, KY.

Liu, D., Sample, N., Peng, J., Law, K. H., and Wiederhold, G. 2003b. Active mediation technology for service composition. In *Proceedings of Workshop on Component-Based Business Information Systems Engineering (CBBISE'03)*. Geneva, Switzerland.

Mcllroy, M. D. 1969. Mass produced software components. *Software Engineering, NATO Science Committee*, January, 138-150.

Ockerbloom, J. 1998. *Mediating among diverse data formats*. Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA.

Papadopoulos, G. A., and Arbab, F. 1997. Coordination of distributed activities in the IWIM model. *International Journal of High Speed Computing*, 9, 2, 127-160.

Peng, J., Liu, D., and Law, K. H. 2003. An engineering data access system for a finite element program. *Journal of Advances in Engineering Software*, 34, 3, 163-181.

Perrochon, L., Wiederhold, G., and Burback, R. 1997. A compiler for composition: CHAIMS. In *Proceedings of the Fifth International Symposium on Assessment of Software Tools and Technologies*, 44-51, Pittsburgh, PA.

Roy, J., and Ramanujan, A. 2001. Understanding web services. *IT Professional*, 3, 6, 69-73.

Sample, N., Beringer, D., Melloul, L., and Wiederhold, G. 1999. CLAM: Composition language for autonomous megamodules. In *Proceedings of the Third International Conference on Coordination Models and Languages*, 291-306, Amsterdam, Netherland.

Sample, N., Beringer, D., and Wiederhold, G. 2002. A comprehensive model for arbitrary result extraction. In *Proceedings of the ACM Symposium on Applied Computing*. Madrid, Spain.

Sheng, S., Chandrakasan, A., and Brodersen, R. W. 1992. A portable multimedia terminal. *IEEE Communications Magazine*, 30, 12, December, 64-75.

Ullman, J. 1988. *Principles of database and knowledge-base systems*. Computer Science Press, Rockville, MD.

Vinoski, S. 1997. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine,* 14, 2, February.

Wiederhold, G. 1992. Mediators in the architecture of future information systems. *IEEE Computer*, March, 38-49.

Wiederhold, G. 2002. Information systems that also project into the future. In *Proceedings of the Databases in Networked Information Systems (DNIS 2002)*. Aizu, Japan, 1-14.

Wiederhold, G. 2003. The product flow model. In *Proceedings of the 15th Conference on Software Engineering and Knowledge Engineering (SEKE)*. Skokie, IL, 183-186.

Wiederhold, G., and Genesereth, M. 1997. The conceptual basis for mediation services. *IEEE Expert, Intelligent Systems and Their Applications*, 12, 5, October, 38-47.

Wiederhold, G., and Jiang, R. 2000. Information systems that really support decision-making. *Journal of Intelligent Information Systems*, 14, March, 85-94.

Wiederhold, G., Wegner, P., and Ceri, S. 1992. Towards megaprogramming: A paradigm for component-based programming. *Communications of the ACM*, 35, 11, November, 89-99.