Active Mediation Technology for Service Composition

David Liu[a], Neal Sample[b], Jun Peng[c], Kincho Law[c], and Gio Wiederhold[b]

[a] Department of Electrical Engineering
[b] Computer Science Department
[c] Department of Civil and Environmental Engineering

Stanford University
Stanford, CA 94305 USA
{davidliu, nsample, junpeng, law, gio}@stanford.edu

Abstract. This paper discusses how active mediation allows information clients to modify the processing behavior of source information services to facilitate service composition. A new computing paradigm is enabled by incorporating active mediators into autonomous services to allow execution of mobile classes – dynamic information processing modules. We describe two key components of active mediation in the FICAS service composition infrastructure: (1) the programming language support for specifying mobile classes and using mobile classes in a composed service; and (2) the runtime support that enables the execution of mobile classes by autonomous services. The power of active mediation is demonstrated by a few examples of mobile classes that conduct complex information processing for service composition. Finally, we discuss the performance issues in deploying active mediation. Specifically, we introduce an algorithm that determines the optimal placement of mobile classes on alternative autonomous services, and discuss the applicability of the algorithm using example mobile classes.

1 Introduction

1.1 Background

We are seeing a continued increase in both the size and performance of computer networks. As networks become pervasive and ubiquitous, connectivity is assumed for all computing facilities, which can be accessed from any geographic location. Such phenomenon can be observed not only with the development of the Internet and the paradigm of web services [7, 21] but also from the growth of intra- and inter- enterprise networks. This development in communication infrastructure has significant effects on the structure of current and future large-scale software services. Rather than being constructed from ground up, software applications are expected to utilize functionalities provided by existing service components. Commercial off-the-shelf (COTS) software applications and other information services are the building blocks that provide pieces of functionalities. This vision of software composition [15] is echoed in the megaprogramming framework [5, 25], which builds on software components called megamodules [18] that capture the functionality of autonomous services provided by large organizational units. Autonomous services are linked together according to composition specifications [19, 23] to form megaservices.

We use the term "autonomous services" to describe the cooperating service components and emphasize their common characteristics. First, autonomous services are usually computational or data intensive, involving long running processes that desire asynchronous invocations. Second, autonomous services run on distributed hosts connected via a communication network, potentially producing large amounts of communication traffic. Third, autonomous services may have heterogeneous access interfaces, introducing a variety of information formats and types. Finally, autonomous services are managed without central administrative control; this makes it difficult to tailor the functionalities and the interfaces of the services according to the needs of the service clients.

Figure 1 illustrates the architecture of a typical megaservice composed of autonomous information services. A communication network provides physical connectivity to the hosts on which the autonomous information services reside. The operating systems provide native support for running the information service applications. In general, the native operating systems are heterogeneous, so are the data produced and consumed by the information services. Through a uniformly defined access interface, the functionalities of information services can be accessed with data mediated to share common formats and meanings among the services.

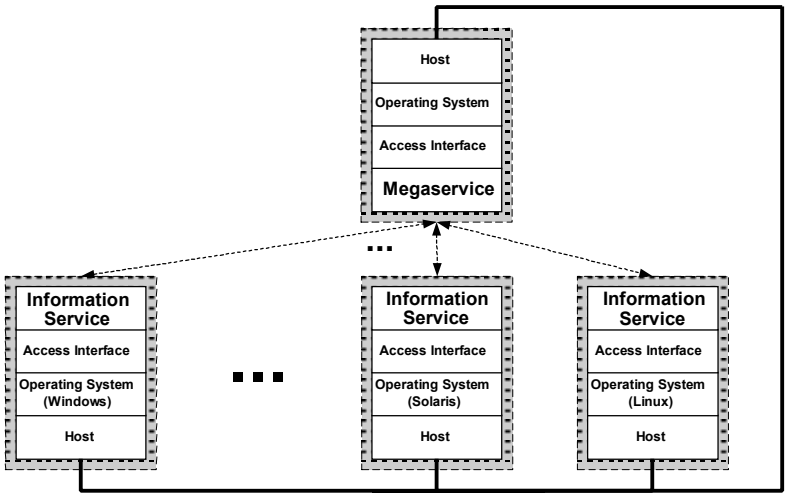

Figure 1: Composing Autonomous Services Into Megaservices

In our research of service composition infrastructures, we use the CHAIMS system (Compiling High-level Access Interfaces for Multi-site Software) [3, 23] as a point of departure. CHAIMS is a representative service composition system that is based on the concept of megaprogramming. It focuses on the composition of services that are provided by large, distributed components. CHAIMS provides a practical general-purpose compositional language. A purely compositional language CLAM has been defined in [19] to provide application programmers with the necessary abstractions to describe the behaviors of their megaprograms. CHAIMS also provides a runtime environment to hide heterogeneity in the underlying systems. The megaservice execution model is similar to that of Idealized Worker Idealized Manager (IWIM) model [1, 17], where a composed program selects appropriate processes to carry out sub-tasks. The composed program is known as the *manager*, and the processes are called *workers* for that manager. In the case of CHAIMS, megaservices are managers and autonomous services are workers. Autonomous services are entities with exposed methods. These methods can be accessed in a traditional way, invoked with input parameters then returning available values.

There are other compositional tools and frameworks that we could have chosen, such as Globus [8] or Ninja Paths [10]. The purely compositional nature of CHAIMS and CLAM allowed us to focus wholly on the mediation task without the distraction of the non-compositional (e.g., brokering, security) aspects of alternative frameworks.

1.2 Objectives

This paper discusses the use of active mediation in service composition infrastructures. Particularly, active mediation provides benefits in the following areas:

- Flexibility of autonomous services: Autonomous 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 potential information 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 autonomous services to satisfy the specific requirements from information clients. Information clients therefore need to work around the differences between the information they require and the information provided by autonomous services. Clients usually have to write customized codes (wrappers, filters, etc.) to work around mismatches. On the other hand, service providers generally find it difficult to alter the existing autonomous services – a modification for one class of users can have unexpected effects on other classes of users. As the number of users of an autonomous service increases, the service provider becomes more reluctant to make significant changes to the service. Active mediation increases the customizability and flexibility of autonomous services. Through active mediation, information clients can send dynamic routines to autonomous services to expand the functionalities of existing services.
- Communication load reduction: In CHAIMS, autonomous service invocations are carried out in a fashion similar to that of remote procedure calls [4]. Parameters are sent to the autonomous services and the results are returned back to the megaservice, which processes and dispatches the data to other autonomous services. We have shown in [13] that an execution model that employs centralized data-flows incurs significant performance penalties compared to an execution model that allows distributed data-flows. Active mediation allows

information processing to be distributed over autonomous services. It utilizes code transfer to reduce data traffic. Code segments are transferred to the most appropriate location for execution to reduce the amount of data communication between the autonomous services and the megaservice.

- Preserving the Power of a Compositional Language: It is important to have clear separation between computation and composition. CHAIMS enforces the separation by completely removing computational primitives from its compositional language, CLAM [19]. The drawback of this model is that it is difficult to specify application logic. Even for handling arithmetic operations, services need to be invoked. While keeping the design objective of CLAM to provide a simple language that separates composition from computation, we use active mediation to enable complex logical computations in megaservice specification. Mobile routines can express complex application logic without the complexity of constructing heavy weight autonomous services. The mobile routines are executed on autonomous services that are enabled with active mediation.

Active mediation applies the notion of mobile code [9] to facilitate dynamic information processing in service composition. We introduce an innovative architecture that enables autonomous service to smoothly adapt to a service composition infrastructure that utilizes active mediation. The capabilities of active mediation are discussed through a spectrum of application scenarios, which demonstrate how active mediation removes some of the otherwise insurmountable blocks in conducting effective and efficient service composition. We use active mediation to perform relational operations on the site of information services, providing extra value mediation functionality as well as improved performance. Dynamic type conversion can be conducted efficiently via active mediation, addressing an important issue in collaborating heterogeneous services. Furthermore, active mediation provides a solution to extraction model incompatibilities among autonomous services.

A significant benefit of active mediation is the flexibility of the location on which information processing can be carried out. Mobile code may be sent to alternative autonomous services, which allows performance optimization. We introduce an algorithm that determines the optimal location where active mediation should be conducted. The range of applicability of the algorithm is also discussed.

This paper is laid out in the following way. Section 2 introduces active mediation and describes how it is supported in FICAS (Flow-based Infrastructure for Composing Autonomous Services) by the compositional language and by the runtime environment. Section 3 illustrates three application scenarios of active mediation. Section 4 discusses performance optimization for active mediation. In section 5, we review the implications of active mediation, in FICAS and beyond.

2 Active Mediation in FICAS

As software becomes more complex and services become more powerful, it is essential to define a framework by which software can be constructed to serve clients with a wide range of computation and communication power. The challenge mixes the need to lower the complexity of software design with an attempt to minimize software maintenance costs. To face certain challenges of dynamic collaborative computing environments, mediators were introduced.

2.1 Mediation

Mediators are intelligent middleware that reside between information clients and information sources [22, 24]. They provide integrated information, without the need to integrate the data resources. Mediators perform functions such as integrating domain-specific data from heterogeneous sources, restructuring the results into object-oriented structures, and reducing the integrated data by abstraction to increase the information density.

A mediation architecture conceptually consists of three layers, as shown in Figure 2. The mediation layer resides between the base resource access interfaces and the service interfaces, incorporating value-added processing by applying domain-specific knowledge processing.

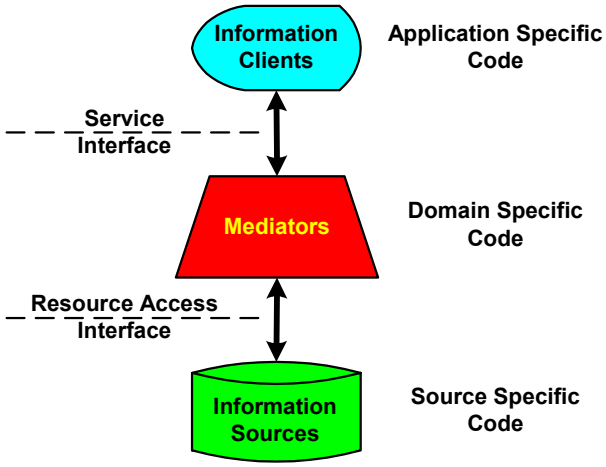

Figure 2: Mediation Architecture

In traditional mediators, code is written to handle information processing at the time the mediators are built. We call these types of mediators *static mediators*. As an extension to the static mediators, *active mediators* are introduced to allow information clients to specify client-defined actions in conducting information processing [12]. Active mediators have the ability to dynamically adapt their behaviors to the users of information sources. Through active mediation, autonomous services can manipulate data according to the specification of the information clients before the data is returned.

2.2 Mobile Classes

An information processing module dynamically loaded by the active mediators is called a *mobile class* (or *m-class*). Conceptually, a mobile class is a function that takes (multiple) input data elements, performs some operations on the input, then outputs a new data element. For instance, *Out = Func(param1, param2, param3)* represents a mobile class named *Func* that takes three data elements as input and produces an output data element *Out*.

In FICAS, mobile classes are specified as Java classes. Java [2] is chosen as the specification language because of Java's support for portability, its flexibility as a high-level language, and its support of dynamic linking/loading, multi-threading and standard API libraries. All m-classes are derived from *MobileClass*, whose interface definition is shown below:

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

The *execute* function takes a vector of data elements as input and generates a data element as output. M-classes overload the *execute* function to provide specific processing capability. The invoker of the mobile class fills in the input vector at runtime. Upon successful execution, the mobile class returns the output data element.

Mobile classes enable megaservices to include complex computational logic. For instance, m-classes are used for data compression, data expansion, aggregation, relational operations, etc. To invoke mobile classes with a megaservice program, we extend the compositional language CLAM with the *MCLASS* primitive:

```
variable = MCLASS (aclass, param1, param2, …)
```

The first argument *aclass* indicates the location from which the Java byte codes of the mobile class can be loaded. The location can be specified in two forms: (1) a fully qualified URL, or (2) a string, which is appended to a base repository URL to form the fully qualified URL. For example, if the base repository URL for the megaservice were http://mobile.class.repository, then the URL for loading the mobile class *aclass* would be http://mobile.class.repository/aclass.class.

The mobile class may be loaded and executed on any one of the potential autonomous services that support active mediation. For instance, the mobile class *aclass* can be loaded either onto the autonomous service that generates *param1* or onto the autonomous service that generates *param2*. As we will discuss in Section 4, the decision is made at runtime to optimize the performance of the megaservice.

Once the mobile class is dynamically loaded onto an active mediator, the parameters in the MCLASS statement are fetched from individual autonomous services and used to fill the input vector *params* in the *execute* function of the mobile class. A data element will be returned as the result of the execution.

2.3 Active Mediation Architecture

Figure 3 illustrates the architecture of an active mediator, which consists of multiple functional components. The **data mediator** provides information integration for the underlying information sources. It provides the functionalities that a static mediator would provide, incorporating information processing logic specific to the application domain. The other components are added to provide the dynamic processing capability:

1. The **m-class fetcher** is responsible for locating and fetching appropriate mobile classes for information processing.
2. The **m-class cache** is a temporary storage for the Java byte codes of mobile classes. The cache is used to avoid duplicate loading of the mobile classes. The byte codes of a mobile class are first looked up from the m-class cache. When a cache miss occurs, the m-class fetcher is used to load the byte codes.
3. The **m-class API library** stores utility classes that make the construction of mobile classes more convenient. For instance, the JDK library [2] is provided as part of the m-class API library.
4. The **m-class runtime** is the module where m-classes are executed. The runtime invokes appropriate mobile classes to conduct dynamic information processing.
5. The **exception handling** module provides a comprehensive set of policies to handle abnormalities in loading or processing mobile classes. Our current implementation prohibits any results from getting through the mediator in the case of an exception. In addition, the conditions are logged for future maintenance.

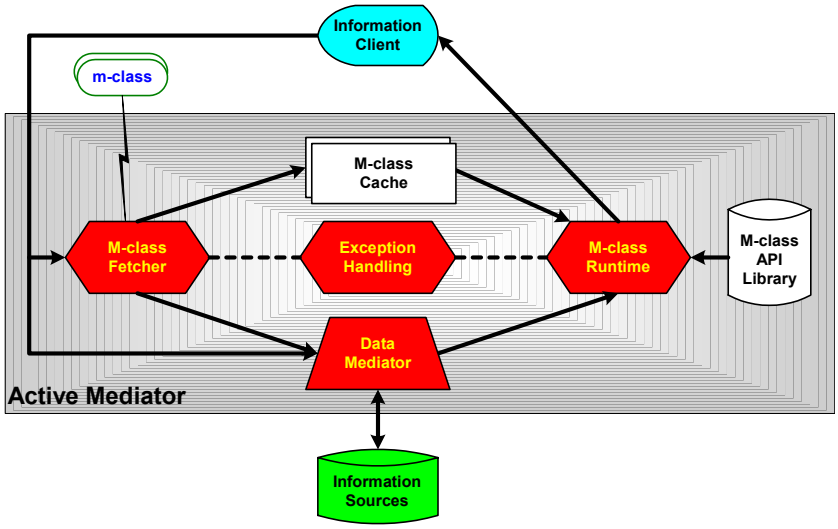

Figure 3: Active Mediation Architecture

Information clients can access the information sources via the active mediator in two forms, either as a data query or as a mobile class query. Data queries are handled directly by the data mediator, which issues requests to the information sources and integrates data from multiple sources. The integrated results pass through the m-class runtime module and are returned to the information client.

Mobile class queries are first handled by the m-class fetcher, which locates and loads the related mobile classes into the m-class cache. Simultaneously, data queries are issued to the data mediator to obtain data from the information sources. The returned data are further processed by the m-class runtime by invoking the mobile classes to conduct information processing. The processed results are then returned to the information client.

We highlight the fact that active mediators can be used in place of static mediators without changing either the information client or the source information service. This enables a smooth transition to an infrastructure that uses active mediation. When received by an active mediator, a data query not in the form of a mobile class will simply flow through the system without the invocation of any mobile classes, producing behavior identical to a static mediator.

2.4 Autonomous Service in FICAS

Autonomous services handle both immediate and deferred execution of mobile classes, which requires the separation of the control-flows from the data-flows. FICAS is an autonomous service metamodel (defined in [14]) that explicitly separates control-flows from data-flows. As shown in Figure 4, an autonomous service consists of an input event queue, an output event queue, an input container, an output container, and a service core. The service core represents the functionality of the autonomous service that is normally delivered by a software application. The event queues are used to handle service requests and to negotiate control-flows decisions with the megaservice controllers. The containers are used to store data elements used (and produced) by the service core.

Each autonomous service contains two flows. For control-flow, the autonomous service is primarily concerned with the state management of an autonomous service, namely the completion of a task, the termination of the service, the invocation of a mobile class, etc. Events are processed from the input event queue and generated onto the output event queue. For data-flow, the autonomous service primarily deals with performing services with the input data elements in the input container. As the results of a service execution, output data elements are generated in the output container. Data containers enable autonomous services to look up generated data elements and to fetch data elements from other autonomous services. The existence of data containers is essential to enable the asynchronous invocation of mobile classes.

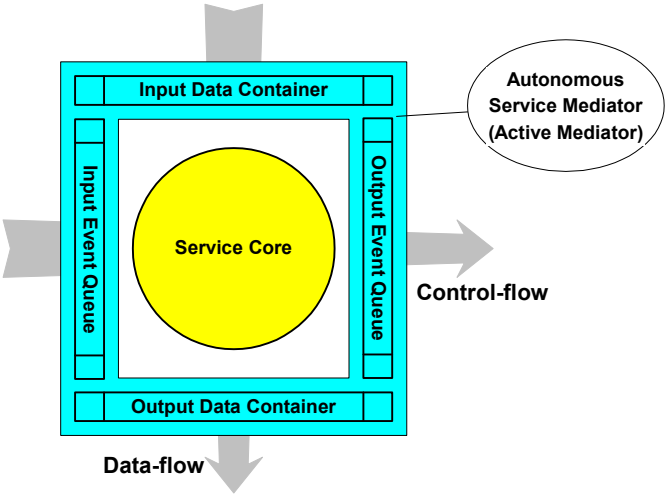

Figure 4: Autonomous Service Metamodel

We observe that the autonomous services differ only in their service cores. Event queues, data containers, and their processing logic are identical for all autonomous services. By building these structures into a standard component called an *autonomous service mediator*, we simplify the process of constructing autonomous services. We have implemented autonomous service mediators as active mediators, so that autonomous services can handle not only data queries but also mobile class queries.

Given the autonomous service metamodel, we define an Autonomous Service Access Protocol (ASAP) by which the autonomous services are accessed. ASAP is a simple protocol for exchanging information among autonomous services in a distributed environment. The protocol removes the barriers imposed by different megaservice programming languages and distribution protocols.

ASAP manages control-flows and data-flows through a set of events. These events exist in the form of XML based messages that are used to interact with autonomous services. The protocol defines how the receiving autonomous services would act and respond to each event. ASAP is asynchronous and non-blocking. The sender of an event does not need to wait for the response to the event. Instead, the sender can continue executing other activities that are not dependent on the response to the event.

For simplicity, we represent the ASAP events here using their abbreviated functional representations instead of their full XML representations. The key ASAP events related to data-flow scheduling are listed below. More complete information on the ASAP protocol is given in [14].

- **SETUP** (Service) – initializing an autonomous service. The autonomous service is informed to prepare necessary system resources for an actual invocation. A reply event is issued after the initialization of the autonomous service.

- **TERMINATE** (Service) – terminating an autonomous service. Garbage collection is conducted during a termination process, so that the system resources involved with an autonomous service instance are released. A reply event is issued after the termination of the autonomous service.
- **INVOKE** (Service) – requesting service from an autonomous service. The service core of the autonomous service is started upon the processing of an INVOKE event. After the completion of the service invocation, output data elements generated by the service core are placed into the output container. In addition, a reply event is issued.
- **MAPDATA** (DataElement, SourceService, DestinationService) – exchanging data between two autonomous services. The event enables the distribution of data-flows within the service composition infrastructure. The MAPDATA event requests a data element to be moved from the output container of the *SourceService* to the input container of the *DestinationService*. After completing the task, the *SourceService* issues a reply back to the originator of the MAPDATA event. It is important to note that the sender of the MAPDATA event does not need to be the recipient of the data element. The events can be sent from the megaservice controller that coordinates the autonomous service invocations, and the data elements are exchanged directly among the data containers of the autonomous services. While the support of the MAPDATA event makes it possible to have distributed data-flows, it is up to the megaservice controller to generate an execution plan that can take advantage this capability.
- **INVOKEMCLASS** (Service, AClass) – invoking a mobile class on an autonomous service. Input parameters of the mobile class are fetched from the input data container, and the result of the execution is put on the output container of the autonomous service. After the completion of the mobile class execution, a reply event is issued.

3 Applications of Mobile Classes

The key feature of the active mediators is their ability to dynamically configure the information processing behaviors. Information clients can send mobile classes to autonomous services where the source information resides. This section presents several sample applications that can be addressed by an active mediator.

3.1 Relational M-class

A large set of data operations can be represented as relational queries. We call this type of m-class a *relational m-class*. The relational m-class can be viewed as the implementation for its corresponding relational expression. The input and output data elements of a relational m-class are relations incorporated as objects. The input data elements are used as operands in the relational expression, and the output data element is the result of the relational operation. Complex expressions are built progressively by combining relational operators on sub-expressions. Table 1 lists the relational operators, their relational algebra representations, and their corresponding m-classes:

- Unary Operators (σ, π): The select operator σ selects tuples that satisfy a given predicate condition. The m-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 m-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 ($\cup$, $\cap$, –): The union operator $\cup$ returns the tuples that appear in either or both of the relations. The intersection operator $\cap$ returns only the tuples that appear in both of the relations. The difference operator – returns the tuples that appear in the first relation that are not in the second relation. The m-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 associates every tuple of the first relation with every tuple of the second relation. The theta join operator combines a selection with Cartesian product, forcing the resulting tuples to satisfy the specific predicate condition. The m-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 1: Relational Operators and Their Corresponding M-classes

Operator	Relational Representation	M-class Interface
Select	$O = \sigma_{cond} (A)$	O = SELECT (A)
Project	$O = \pi_{attr} (A)$	O = PROJECT (A)
Union	$O = A \cup B$	O = UNION (A, B)
Intersect	$O = A \cap B$	O = INTERSECT (A, B)
Difference	$O = A - B$	O = DIFFERENCE (A, B)
Cartesian product	$O = A \times B$	O = CARTESIAN (A, B)
Theta join	$O = A \bowtie_{cond} B$	O = JOIN (A, B)

3.2 Type Conversion M-class

Data generated by an autonomous service can be directly used by other autonomous services if the services share the same data types, formats, and granularities, etc. However, such homogeneity cannot be assumed within a large-scale service composition environment. Data enter with various types and will continue to appear in different types that are suited for different applications. The output data of one autonomous service needs to be converted to conform to the input data type of another autonomous service.

Traditionally, a type broker service or a distributed network of type brokers can be used to mediate the difference among data in various formats [16]. The type brokers can use data in unknown formats and convert them to known formats for the information client. The brokers serve as proxies connecting client requests with appropriate source services. A type graph is used to figure out the chain of conversions necessary. An example of automating this process can be seen in [6]. The issue of using type brokers for type conversion in service composition is its inefficiency. Large amount of data are forwarded among the brokers, especially when a chain of conversions is involved. Figure 5(a) presents an example of data flow in type-broker architecture. Data from the source service are represented in type T1, and the information client consumes data in type T3. Two type brokers are employed to convert source data from type T1 to type T3. Data are passed back and forth among the type brokers and the information client in order to convert the data into consumable format.

Active mediation avoids data transmission among type brokers by performing type conversion at the source mediator. As shown in Figure 5(b), rather than data being forwarded among type brokers, type conversion functions specified in the form of mobile classes are forwarded to where the data is located. Similar to the network of type brokers, multiple type conversion m-classes can be chained together. Type conversion is conducted using mobile classes at the source mediator. Data in a consumable format is directly returned to the information client. Collocating the conversion mechanism with the data avoids the obvious cost of multiple interim data transfers, reducing data traffic to essential transmissions.

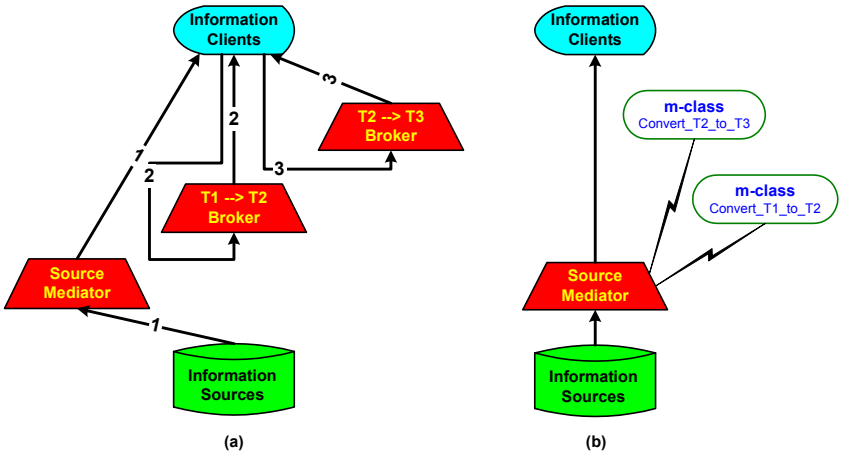

Figure 5: Type Conversion for Autonomous Services

3.3 Extraction Model Mediation

Autonomous services can produce data with a wide variety extraction models [20]. Interesting megaservices will have a set of "upstream" autonomous services generating data that is consumed by a set of "downstream" autonomous service (the sets are generally not disjoint). When there is a mismatch between how data is produced and how that data is later consumed, extraction mediation can play an important role. For example, an upstream service might produce data progressively (as a sequence of tuples), while the downstream service requires that the data arrive as a whole relation. Active mediation can be used to prepare the data for different extraction models.

A taxonomy of extraction models for autonomous services that produce outputs based on specific inputs is presented in [20]. This taxonomy is based on three binary factors: partial extraction, runtime service status, and runtime result status. These three binary factors combine to form eight (2^3) basic types of data extraction methods, including familiar and obvious methods (e.g., SQL cursors and RPC), and some less obvious methods (e.g., semantic partial extraction and progressive extraction). These eight extraction formats have meaningful matches with respect to input formats, but do not fully describe all possible extraction models.

One extraction model not captured by the taxonomy in [20] is less interesting from the perspective of an autonomous service, but that nonetheless has utility, is the autonomous data stream. An example of an autonomous data stream is one that continually generates tuples, regardless of client input data. A stock ticker is this type of autonomous data stream. There is no input data to the stock ticker, yet it produces output data that may be used by other autonomous services and megaservices.

Active mediators play a critical role in extraction model mediation, ensuring that format inconsistencies do not prevent service composition. Some m-classes used for extraction model mediation are equivalent to type conversion or relational m-classes, others are externally indistinguishable (yet functionally distinct), while some are radically different from the m-classes that we have seen so far.

The simplest m-classes for extraction model mediation are functionally equivalent to relational m-classes used for projection operations. For instance, consider an autonomous service that produces three outputs, *A*, *B*, and *C*. The *partial extraction* model presented in [20] may not be fully implemented in the autonomous service, so *A*, *B*, and *C* are delivered as an opaque object *X*. By extracting only the components of *X* desired by downstream services, m-classes can mimic partial extraction, a behavior not directly supported by the service itself. This simple process can be coupled with selection and reordering (e.g., sorting) predicates for producing many different types of behaviors. With basic m-classes, opaque data can be filtered, set reordered, transmission delayed, projected, split, recombined, and/or extracted multiple times.

A more complex m-class for extraction model mediation is not externally distinguishable from simpler m-classes, but must be uniquely tailored to data extraction. For example, the stock ticker service is an instance of extraction model mediation that looks like a value-based filter, but is implemented in a very different way. A typical value-based filter (like a select) operates on an input set, and produces an output set of an equal or smaller size, after examining all the values in the set. However, how can an m-class with a selection predicate examine *all* the tuples in a continuous *stream* before returning any data? With an understanding of the how the data is produced and how it is composed, streams become meaningful inputs to downstream services.

One downstream service might be interested in all elements coming from the ticker within a certain window of time; the corresponding m-class generates tuples until the data generated by the ticker falls outside that window, at which time the m-class has completed its task. (Simpler selection m-classes would not understand a termination condition beyond "there are no more tuples.") Another downstream service might be interested in a complete cycle of all elements coming from the ticker. An m-class can sample the stream at an arbitrary point, finding a first ticker symbol, say "QXYZ." The m-class can pass through all further symbols until seeing "QXYZ" again. Having observed an entire cycle, the data is then ready for the downstream consumer service.

Finally, certain extraction models are simply incompatible with other input models. These can be the most difficult to generate m-classes for, but are a place where m-classes are indispensable. An example of this incompatibility was mentioned earlier: an upstream service delivers an SQL cursor, but the downstream service expects a relation. The logic behind the m-class is transparent enough: scroll the cursor to fill a complete relation. However, such a fundamentally simple extraction model mismatch will stymie any upstream/downstream pair without mediation. The power of extraction model mediation does not end with upstream/downstream couplings. The separation of control-flow from data-flow can also be achieved through collocation of m-classes for extraction mediation, even for autonomous services that do not otherwise support such a separation. The models seen in [13] can all be realized with an intelligent application of the appropriate m-classes, even where legacy services do not have such support.

Autonomous services will invariably feature multiple dependencies on both the nature of the service and the expected audience. But not all producers and consumers are created equally. Even when there is a type and domain

match between the upstream and downstream services, the extraction model (or the corresponding input model) may provide and seemingly insurmountable block. Active mediation through the application of m-classes is a solution to extraction model incompatibilities.

4 Performance Optimization for Active Mediation

FICAS is a distributed data-flow infrastructure in that data can be passed directly between autonomous services without going through the megaservice. As shown in [13], distributed data-flow infrastructures offer significant performance improvement over centralized data-flow infrastructures in executing megaservices, in terms of both aggregated cost and response time. The improvement in aggregated cost comes from the savings in data passed between autonomous services and the megaservice in the distributed data-flow model. The improvement in response time comes from alleviating the communication bottleneck at the megaservices.

By enabling active mediation at autonomous services, information processing tasks may be dispatched to autonomous services, enabling further performance optimization.

4.1 Placement of Mobile Classes

The placement of a mobile class can greatly affect the performance of the megaservice it serves. We focus our effort on optimizing the megaservice performance on the communication aspects rather than the computational aspects. We assume that the cost of loading and executing an m-class remains the same regardless of which autonomous service the m-class is loaded onto. The megaservice controller makes the decision about the placement of the m-classes based on the cost that the resulting data-flows would have on the megaservice. For simplification, we assume a uniformly connected processor network, where the network bandwidths between any node pairs are the same.

Figure 6 shows a simple example of using m-classes. The result generated by *Invocation1* is a list of numbers. The m-class *sum* will add up the numbers in the list and pass the summation to *Service2* for further processing.

```
/* ... */

Invocation1 = Service1.INVOKE()
A = Invocation1.EXTRACT();

B = MCLASS (“sum”, A)

Invocation2 = Service2.INVOKE(B)
C = Invocation2.EXTRACT();

/* ... */
```

Figure 6: Sample Megaservice Program Segment with M-classes

We can first analyze the example intuitively. Figure 7 illustrates three potential execution plans. The plans differ in the runtime placement of the m-class *Sum*. Consequently, these plans form different data-flow structures for the megaservice, as shown in Figure 8. The three plans are listed as follows:

- **Plan 1**: By placing *sum* at the megaservice controller, we can construct the execution plan as shown in Figure 7(a). *Service1* generates data element *A* and passes it to megaservice. The m-class *Sum* processes the data element *A* at the megaservice controller. The processed result *B* is then sent to *Service2* for further processing.
- **Plan 2**: By placing *sum* at the source autonomous service *Service1*, we can construct the execution plan as shown in Figure 7(b). *Service1* generates data element *A* and processes it locally using the m-class *Sum*. The processed result *B* is then sent to *Service2* for further processing.
- **Plan 3**: By placing *sum* at the destination autonomous service *Service2*, we can construct the execution plan as shown in Figure 7(c). *Service1* generates data element *A* and passes it to *Service2*. *Service2* processes the data locally using the m-class *Sum* and then uses the processed result *B* for further processing.

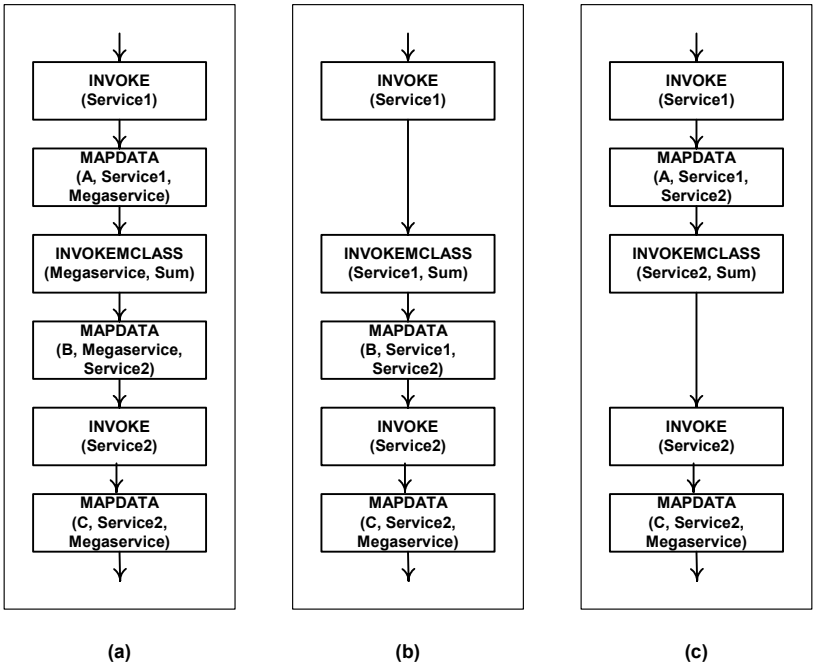

Figure 7: Execution Plans for the Sample Megaservice using M-class *Sum*

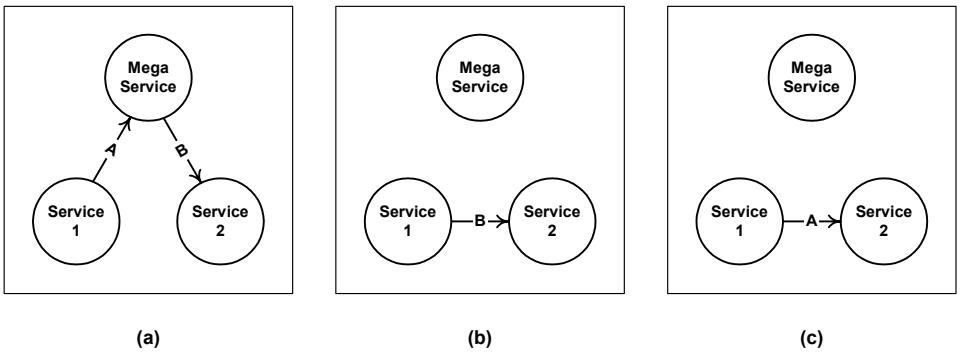

Figure 8: Data-flow Diagrams for the Sample Megaservice using M-class *Sum*

Our objective is to choose the plan with the best performance based on the cost criteria established in [13] – aggregated cost and response time. It is clear from the data-flow diagram that Plan 1 incurs the most communication traffic compared to the other two plans. Both the input data element *A* and output data element *B* are sent between the megaservice controller and the autonomous services. The plan has the worst performance in terms of both aggregated cost and response time.

Plan 2 and Plan 3 differ in the data content sent between the autonomous services. Plan 2 places the m-class on *Service1*. The data element *A* generated by *Service1* is processed locally by the m-class *Sum*. Thus, only data element *B* is sent from *Service1* to *Service2*. On the contrary, Plan 3 sends the data element *A* to the destination autonomous service. The plan avoids the communication traffic of sending data element *B*, which is processed by the m-class *Sum* locally on *Service2*.

Evaluating Plan 2 against Plan 3 is a matter of comparing the sizes of data elements *A* and *B*. Due to the fact that the sample m-class *Sum* performs aggregation on the input data content, the output data *B* has smaller volume than the input data *A*. Therefore, Plan 2 incurs the least amount of communication traffic, and it would be chosen as the best execution plan.

4.2 LDS Algorithm

Now, we can construct execution plans for generic megaservices by placing m-classes at optimal locations to minimize communication traffic. For an m-class with known input and output data sizes, each input or output data element to the m-class can be characterized as a pair. Each input is a (S_i, V_i) pair, where S_i is the autonomous service that generates the ith input data element, and V_i is the volume of the data element. The output is a (S_0, V_0) pair, where S_0 is the destination autonomous service to which the result of the m-class will be sent, and V_0 is the size of the data element.

Figure 9 shows the LDS (Largest Data Size) algorithm, which targets the autonomous service that generates and consumes the largest volume of data for a given m-class. The algorithm first computes the total amount of data connected with the m-class for each unique autonomous service. Then, the autonomous service with the largest data volume is selected as S_{max}, which represents the optimal placement for the m-class. S_{max} is returned as the output of the algorithm.

```
INPUT: input pairs(S1, V1), …, (Sn,Vn)
       output pair (S0, V0)
OUTPUT: Smax
METHOD:
    Vmax=0
    for every unique S in input and output pairs
        V=0
        for i=0,…,n
            if Si==S
                V=V+Vi
        if V>Vmax
            Smax=S
            Vmax=V
```

Figure 9: LDS Algorithm for Optimizing m-class Placement

Two special types of m-classes simplify the LDS algorithm. The first type of m-class is called *expansion m-class*, whose output data size is at least as large as the sizes of its input data. Based on the LDS algorithm, the optimal m-class placement would be the destination autonomous service to which the result of the m-class would be sent. The other special type of m-class is called a *compression m-class*, where the output data size is smaller than the sizes of its input data. The optimal m-class placement can be chosen as the input autonomous service that has the largest input data size.

The LDS algorithm is only applicable when the input and output data sizes are known for the m-classes. For a situation where the exact output data size of an m-class is unknown until after the execution of the m-class, we need a method to estimate the output data size.

We view the output data size of an m-class as a function on the input data sizes of the m-class: $S_O = f(S_A, S_B, \ldots)$. The function f is called the sizing function of the m-class, where S_O is the output data size and S_A, S_B are the input data sizes. The sizing function may be stored along with the Java byte codes in the m-class repository. The megaservice controller can then use the sizing function to estimate the m-class output data size for running the LDS algorithm.

Clearly, the effectiveness of the mediation based data-flow optimization technique depends on the accuracy of the characterization of m-classes using the sizing functions. In the following section, we discuss how to obtain the sizing functions for a special class of m-classes. In addition, we offer insights into how sizing functions may be formulated for general m-classes.

4.3 Data-flow Optimization for Sample M-classes

The relational operators have well defined relationships between the input and output data sizes. Their sizing functions can be mathematically formulated. Research results are borrowed from the field of query processing to estimate the result size of a relational query [11]. This section shows that the sizing functions do not need to be precise for the LDS algorithm to generate an optimal result. In many cases, sizing functions can be simplified to constant values, in which case the optimal m-class placement can be quickly determined.

Table 2 lists the simplified sizing functions for the relational m-classes. The unary operators *select* and *project* return only portions of the input relations. The m-classes implementing the unary operators are by definition compression m-classes. Their sizing functions return zero, guaranteeing that the output data size should be no larger

than the input data size. Hence, the SELECT and PROJECT m-classes are always placed on the source autonomous service that generates the input data.

The *union* operator combines the two input relations. Hence, its m-class is an expansion m-class. The sizing function returns infinity, guaranteeing that the output data size should be greater than the input data sizes. The m-class is always placed on the destination autonomous service that uses the output data.

The *intersect* and *difference* operators return portions of the input relations. Their m-classes are compression m-classes, and thus the sizing function return zero. The m-class would be placed on one of the source autonomous services. The choice of the source autonomous services will made when the size of the input data elements are evaluated at runtime. The optimal placement is the input autonomous service with larger data size.

The result set of *Cartesian product* operator contains all possible combinations of one tuple from each input relations. The result relation is clearly larger than the input relations. Similar to the *union* operator, the sizing function of *Cartesian product* returns infinity. The m-class is placed on the destination autonomous service.

The sizing function for the *theta join* operator is more complex. The exact function usually depends on the characteristics of the input data and the types of predicate conditions. For instance, the sizing function may be set to $S_0=c\times S_A\times S_B$ if the *theta join* is an equality join with uniformly distributed values in input relations; if the result relation is expected to be rather small, the sizing function can be set to zero to enforce the LDS algorithm choose the input autonomous services. The sizing function may be set to infinity to have the LDS algorithm choose the output autonomous service if the result relation is expected to be larger than the input relations.

Table 2: Sizing Functions for Relational M-classes

M-class Interface	Sizing Function
O = SELECT (A)	$S_O = 0$
O = PROJECT (A)	$S_O = 0$
O = UNION (A, B)	$S_O = \infty$
O = INTERSECT (A, B)	$S_O = 0$
O = DIFFERENCE (A, B)	$S_O = 0$
O = CARTESIAN (A, B)	$S_O = \infty$
O = JOIN (A, B)	$S_O = f(S_A, S_B)$

Relational expressions that are certain combinations of relational operators may have simple mathematical representations of their sizing functions. For instance, the O=PROJECT(INTERSECT(A,B)) has a simple sizing function of $S_O=0$.

As m-classes get more complex, the relationship between output data sizes and input data sizes becomes harder to represent with simple mathematical formulae. In some cases, the output data sizes may not be determined based on the input data sizes, as the content of the input data affects the output data size. We are exploring techniques to estimate the output size of arbitrary m-classes. As a future research direction, we are looking at statistical methods to adaptively estimate the correlations between the m-classes' input and output data sizes.

5 Conclusions

Active mediation increases the customizability and flexibility of autonomous services. It utilizes code mobility to facilitate dynamic information processing in service composition. In this paper, we offer a conceptual framework for active mediation. An innovative architecture is defined to enable smooth adoption of active mediation in autonomous services. Our evolutionary approach allows coexistence of active mediation and static mediation, making it feasible to build a service composition infrastructure that supports active mediation. We survey a spectrum of 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 reveal the importance of active mediation in conducting service composition.

We discuss the performance issues in deploying active mediation in the service composition runtime environment. The fact that mobile classes may be executed on alternative locations enables us to conduct performance optimization. We introduce an algorithm that determines the optimal placement of mobile classes, and discuss the applicability of the algorithm.

6 Acknowledgement

This research is partially sponsored by the National Institute of Standards and Technology, National Science Foundation, and Center for Integrated Facility Engineering at Stanford University.

References

[1] F. Arbab, I. Herman, and P. Spilling, "An Overview of Manifold and its Implementation", *Concurrency: Practice and Experience*, vol. 5(1), Feb 1993, pp. 23-70.

[2] K. Arnold, J. Gosling, and D. Holmes, "The Java Programming Language", Java Series, Addison-Wesley, 2000.

[3] D. Beringer, C. Tornabene, P. Jain, and G. Wiederhold, "A Language and System for Composing Autonomous, Heterogeneous and Distributed Megamodules", Proceedings of DEXA International Workshop on Large-Scale Software Composition, Vienna Austria, August 1998.

[4] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls", *ACM Transactions on Computer Systems*, vol. 2(1), February 1984, pp. 39-59.

[5] B. Boehm and B. Scherlis, "Megaprogramming", Proceedings of DARPA Software Technology Conference, Los Angeles, April 1992, pp. 68-82.

[6] S. Chandrasekaran, S. Madden, and M. Ionescu, "Ninja Paths: An Architecture for Composing Services over Wide Area Networks", UC Berkeley, Technical Report, 2000, http://ninja.cs.berkeley.edu/dist/papers/path.ps.gz.

[7] D. F. Ferguson, "Web Services Architecture: Direction and Position Paper", Proceedings of W3C Web Services Workshop, San Jose, CA, April 2001.

[8] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit", *International Journal of Supercomputer Applications*, vol. 11(2), 1997, pp. 115-128.

[9] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding Code Mobility", *IEEE Transactions on Software Engineering*, vol. 24(5), 1998, pp. 342-361.

[10] S. Gribble, M. Welsh, R. von-Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. H. Katz, Z. M. Mao, S. Ross, and B. Zhao, "The Ninja Architecture for Robust Internet-Scale Systems and Services", U.C. Berkeley, To appear in a Special Issue of Computer Networks on Pervasive Computing, 2002, http://ninja.cs.berkeley.edu/dist/papers/ninja.ps.gz.

[11] H. F. Korth and A. Silberschatz, "Database System Concepts", 2nd ed, McGraw-Hill, 1991.

[12] D. Liu, K. Law, and G. Wiederhold, "CHAOS: An Active Security Mediation System", Proceedings of International Conference on Advanced Information Systems Engineering, LNCS, vol.1789, B. Wangler and L. Bergman (eds.), Springer-Verlag, 2000, pp. 232-246.

[13] D. Liu, K. Law, and G. Wiederhold, "Analysis of Integration Models for Service Composition", Proceedings of Third International Workshop on Software and Performance, Rome, Italy, July 2002.

[14] D. Liu, K. Law, and G. Wiederhold, "FICAS: A Distributed Data-Flow Service Composition Infrastructure", Stanford University, Unpublished Report, 2002, http://mediator.stanford.edu/papers/FICAS.pdf.

[15] M. D. McIlroy, "Mass Produced Software Components", *Software Engineering, NATO Science Committee*, January 1969, pp. 138-150.

[16] J. Ockerbloom, "Mediating Among Diverse Data Formats", Carnegie Mellon University, Pittsburgh, PA, PhD. Thesis, 1998.

[17] G. A. Papadopoulos and F. Arbab, "Coordination of Distributed Activities in the IWIM Model", *International Journal of High Speed Computing*, vol. 9(2), 1997, pp. 127-160.

[18] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", *P. Freeman and A. I. Wasserman*, Tutorial on Software Design Techniques, IEEE Computer Society Press, 1983.

[19] N. Sample, D. Beringer, L. Melloul, and G. Wiederhold, "CLAM: Composition Language for Autonomous Megamodules", Proceedings of Third International Conference on Coordination Models and Languages, Amsterdam, April 1999.

[20] N. Sample, D. Beringer, and G. Wiederhold, "A Comprehensive Model for Arbitrary Result Extraction", Proceedings of ACM Symposium on Applied Computing, Madrid, Spain, March 2002.

[21] W3C, "Simple Object Access Protocol (SOAP)", 2000, http://www.w3.org/TR/SOAP.

[22] G. Wiederhold, "Mediators in the Architecture of Future Information Systems", *IEEE Computer*, March 1992, pp. 38-49.

[23] G. Wiederhold, D. Beringer, N. Sample, and L. Melloul, "Composition of Multi-site Services", Proceedings of IDPT'99, Kusadasi, Turkey, June 1999.

[24] G. Wiederhold and M. Genesereth, "The Conceptual Basis for Mediation Services", *IEEE Expert, Intelligent Systems and Their Applications*, vol. 12(5), October 1997, pp. 38-47.

[25] G. Wiederhold, P. Wegner, and S. Ceri, "Towards Megaprogramming", *Comm. ACM*, vol. 35(11), Nov 1992, pp. 89-99.