Composition of Engineering Web Services with Distributed Data Flows and Computations

David Liu[1], Jun Peng[2], Kincho H. Law[3], Gio Wiederhold[4], and Ram D. Sriram[5]

Abstract

This paper describes an experimental Flow-based Infrastructure for Composing Autonomous Services (FICAS), which supports a service-composition paradigm that integrates loosely-coupled software components. For traditional software service composition frameworks, the data-flows and control-flows are centrally coordinated, and the composed service operates as the hub for all data communications. FICAS, on the other hand, employs a distributed data flow approach that supports direct data exchanges among web services. The distributed data flows can avoid many performance bottlenecks attending centralized processing. The performance and flexibility of FICAS are further improved by adopting active mediation, which distributes computations within the service framework, and reduces the amount of data traffic significantly by moving computations closer to the data. A system has been prototyped to integrate several project management and scheduling software applications. The prototype implementation demonstrates that distributed data flow, combining with active mediation, is effective and more efficient than centralized processing when integrating large engineering software services.

Keywords

Engineering web services; service integration; distributed data flow; active mediation; mobile class; project management

[1] Ph.D. Candidate, 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] Professor, Department of Civil and Environmental Engineering, Stanford University, Stanford, CA 94305. E-mail: law@stanford.edu
[4] Professor, Computer Science Department, Stanford University, Stanford, CA 94305. E-mail: gio@db.stanford.edu
[5] Group Leader, Manufacturing Systems Integration Division, National Institute of Standards and Technology, Gaithersburg, MD 20899. E-mail: sriram@cme.nist.gov

1 Introduction

A software engineering paradigm where large software services are decomposed into cooperating components has been envisioned for over 30 years. Under this paradigm, software components are linked together through an integration framework to form composed software applications [1]. Software components are provided as processes managed by independent service providers. The components have clearly defined functions with accessible interfaces. These components are either existing applications or new programs developed for specific tasks. These software applications are wrapped into autonomous services whose functionalities are then composed together. The composed application is called a megaservice, which acts as a central controller for invoking, monitoring, querying, and terminating the autonomous services. With the rapid development of the Internet and networking technologies, the computing environment is evolving toward an interconnected web of autonomous services, both inside and outside of enterprise boundaries.

Performance remains to be an issue for most common service composition frameworks. In a typical composed application, all results from one web service are shipped to the megaservice, handled there, and then shipped to the next web service. Data are exchanged using a client-server model where the megaservice serves as the central hub of all data traffic. In most cases, this centralized data-flow approach is inefficient for integrating large-scale software services. This *centralized data-flow approach* is the default mode in many current software integration frameworks such as CORBA [2], J2EE [3], and Microsoft .NET [4].

To deal with the performance issue associated with the current service composition frameworks, we demonstrate a Flow-based Infrastructure for Composing Web Services (FICAS) [5]. FICAS is implemented as a collection of software modules that support the construction of web services, facilitate the functional composition of web services into composed application, and conduct the execution of the enhanced applications. A *distributed data-flow approach*, which allows data to be exchanged directly among the services, is adopted in FICAS framework to address three design concerns: (1) Scalability – integration and management of a large number of autonomous services in the service composition infrastructure; (2) Performance – high efficiency in the execution of megaservices; and (3) Ease of composition – effective and convenient specification of service compositions by the application programmers. FICAS uses

distributed data-flows to achieve better scalability and performance without sacrificing ease of composition.

FICAS also applies the concept of 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 [6]. Without being able to delegate such a capability to the remote service such incompatibilities have to be resolved at the application site. Active mediation exploits the notion of mobile code [7] to provide for 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. Active mediation distributes a class of computations within the service framework, and reduces the amount of data traffic significantly by moving computations closer to the data.

The paper is organized as follows. Section 2 gives an overview of FICAS, and then defines a metamodel to enable homogeneous access for autonomous services within FICAS. Section 3 describes the runtime environment of distributing data communications directly among autonomous services, and presents an empirical test to measure performance of centralized data flow to distributed data flow. Section 4 introduces active mediation techniques for dispatching software modules to client computers to reduce data traffic. Section 5 illustrates a prototype for a ubiquitous computing environment based on FICAS, and presents an example of project management on a building construction project. Section 6 summarizes the findings on the direct data flow among services and discusses future work.

2 Service Composition Infrastructures

Autonomous services are composed in a loosely coupled fashion to allow flexible integration of heterogeneous systems in a variety of domains. There have been many significant researches in service composition, particularly in creating uniform ways of describing, deploying, and accessing applications [8]. Research has also been reported on automated composition of web services [9]. While many standards have been proposed to represent processes using web services, such as BPEL4WS [10], WSCL [11], DAML-S [12], their implementations have not

yet been effectively demonstrated for distributed data flow applications. FICAS serves as a reference implementation for composing applications and to investigate the performance implications of a distributed data flow framework for service composition.

2.1 Autonomous Service Metamodel

In FICAS, autonomous services are specified as a homogeneous model that promotes communication and cooperation with each other. Figure 1 illustrates the autonomous service metamodel, which consists of a service core, an input event queue, an output event queue, an input data container, and an output data container:

- The service core represents the core functionality of the autonomous service. It is responsible for performing computation on the input data elements and generating resultant data elements. Existing software applications are wrapped into a service core.
- Events (messages) are exchanged between services to control the flow of autonomous service executions. Asynchronous execution of autonomous services is achieved by using queues for event processing. The default queuing protocol in FICAS is FIFO (first in first out), so event messages are processed in the order they arrive.
- The data containers are groupings of input and output data elements for the autonomous service. Input data elements are fetched from the input data container and processed by the service core. The generated data elements are put into the output data container. The data containers enable autonomous services to look up generated data elements.

Autonomous services export the service functionalities contained in the encapsulated software applications. Although the service functionalities differ, the way by which the functionalities are exported is similar for all the autonomous services. The autonomous 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 autonomous services can be significantly simplified by building the common components into a standard *autonomous service wrapper*, which facilitates the encapsulation of software applications into autonomous services.

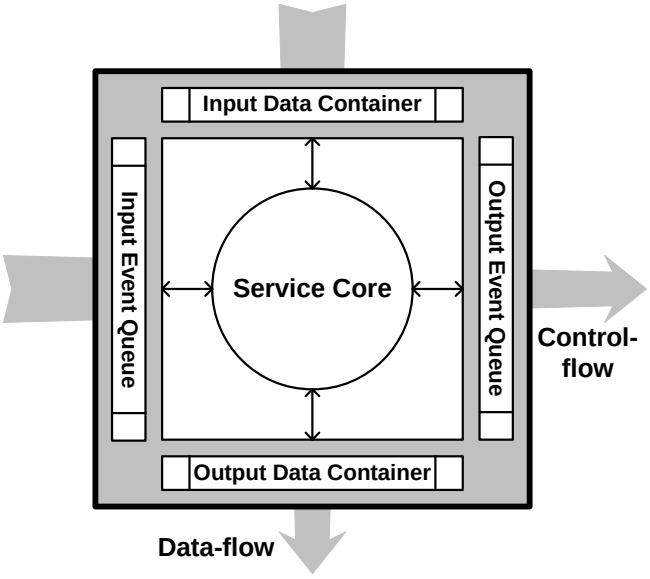

Figure 1: FICAS Autonomous Service Metamodel

In FICAS, the autonomous service wrapper has been implemented in Java. With the autonomous service wrapper provided as a standard module, the wrapping of a software application into an autonomous service is simplified to a matter of defining the *ServiceCore* interface, as shown in Figure 2. The application core connects to the autonomous service wrapper through three methods. The *setup()* method defines the actions of the application when the service is initialized; the *execute()* method is called when the service is invoked, triggering the application to process the data in the containers; and the *terminate()* method is called when the service is terminated. Each method takes three parameters. The *inputcontainer* provides the reference to the input data container of the autonomous service; the *outputcontainer* provides the reference to the output data container of the autonomous service; and the *flowid* identifies the flow to which the service request belongs. With the references to the data containers and the flow identifier of the request, the software application can look up the input parameters from the input data container and generate the results into the output data container.

In FICAS, the control-flow is primarily related to the event processing and the state management of the service core, and the data-flow is concerned with the exchange of data elements between the data containers and the processing of the data elements by the service core. While each component uses its own computational thread, the service core ties together the components into a coordinated entity.

```
public interface ServiceCore {

  public boolean setup(Container inputcontainer,
                Container outputcontainer, FlowId flowid);

  public boolean execute(Container inputcontainer,
                Container outputcontainer, FlowId flowid);

  public boolean terminate(Container inputcontainer,
                Container outputcontainer, FlowId flowid);
}
```

Figure 2: Class Interface of ServiceCore

2.2 Autonomous Service Access Protocol

Given the autonomous service metamodel, we define an autonomous service access protocol, ASAP, by which the autonomous services are accessed. ASAP manages control-flows and data-flows through a set of events. These events exist in the form of XML (eXtensible Markup Language) based messages that are used to interact with autonomous services. The hierarchical structure of XML provides a convenient method to define the composition of an event. ASAP is asynchronous and non-blocking, i.e., the sender of an event does not wait for a response. Instead, the sender continues to execute other activities that are independent on any response of the event. The protocol removes the barriers imposed by different megaservice programming languages and distribution protocols. For simplicity, the ASAP events are represented using their abbreviated functional representations instead of their full XML representations. The key ASAP events that are related to data-flow scheduling are listed below:

- SETUP (Service)

The SETUP event is used to initialize an autonomous service, which is to prepare necessary system resources for the actual invocations. A reply event is issued after initialization of an autonomous service.

- TERMINATE (Service)

The TERMINATE event unconditionally terminates an autonomous service. Garbage collection is conducted during the termination process to release any system resources involved with a service instance. A reply event is issued after the termination of an autonomous service.

- INVOKE (Service)

The INVOKE event requests an autonomous service. The service core of the autonomous service is started upon the processing of the INVOKE event. Upon completion of the service invocation, output data elements generated by the service core are placed onto the output data container. In addition, a reply event is sent.

- MAPDATA (DataElement, SourceService, DestinationService)

The MAPDATA event is used to establish a data-flow between two data containers. The event enables the distribution of data-flows within the service composition infrastructure. The sender of the MAPDATA event does not need to be the recipient of the data element. The events are usually 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 of this capability.

There are two forms of implementation for the MAPDATA event. The first is called "push MAPDATA," in which case the event is sent to the *SourceService*. The *SourceService* fetches the data element from its output data container and pushes the data element over to the *DestinationService*. Another implementation is called "pull MAPDATA," in which the event is sent to the *DestinationService*. The *DestinationService* pulls the data element from the *SourceService* and puts the data element onto its input data container. Both implementations are supported by FICAS.

2.3 Components in FICAS

The service composition infrastructure, FICAS, allows distributed software applications to hide heterogeneities in the network, platform, and language. FICAS is built upon a previously developed service composition infrastructure CHAIMS (Compiling High-level Access Interfaces for Multi-site Software) [13, 14], which focuses on the composition of services that are large distributed components. Residing on different computers, the services are inherently concurrent in nature, and the long duration of service execution necessitates asynchronous invocation and collection of results. CHAIMS developed a simple compositional language and runtime support

for applications composed from distributed modules. FICAS builds on the prior efforts of CHAIMS because the compositional language supports the goal for ease of composition.

Figure 3 illustrates the main components of FICAS. The buildtime components are responsible for specifying megaservices and compiling megaservice specifications into control sequences that serve as inputs to the runtime environment. For FICAS, a *Compositional Language for Autonomous Services* (CLAS) is defined to provide the application programmers the necessary abstractions to describe the behaviors of their megaservices. The CLAS language focuses on functional composition of autonomous services. A CLAS program is essentially a sequential specification of the relationships among collaborating autonomous services, without providing primitives to schedule or to coordinate control-flows and data-flows. The CLAS program is compiled by the buildtime component into a control sequence that can be executed by the runtime environment. The control sequence is language and platform independent, providing a bridge between the buildtime and runtime environments of FICAS.

The runtime environment of FICAS is responsible for executing control sequences. At its minimum, the runtime can consist of just one autonomous service, along with the service directory. The runtime environment can be expanded simply by plugging additional autonomous services into the communication network and registering the autonomous services with the service directory. The directory keeps track of available autonomous services within the infrastructure. While the directory is viewed logically as a centralized entity, it may be implemented as a distributed structure. In the prototype FICAS system, a centralized directory service is used.

A metamodel is defined to allow the construction of homogeneous autonomous services in a heterogeneous computing environment. The control-flows are coordinated by a megaservice controller, which is the centralized coordinator that carries out the execution of a megaservice. The controller generates an execution plan based on an input control sequence, and then follows the plan, coordinating control-flows among respective autonomous services. The controller is also responsible for optimizing the performance of the megaservice.

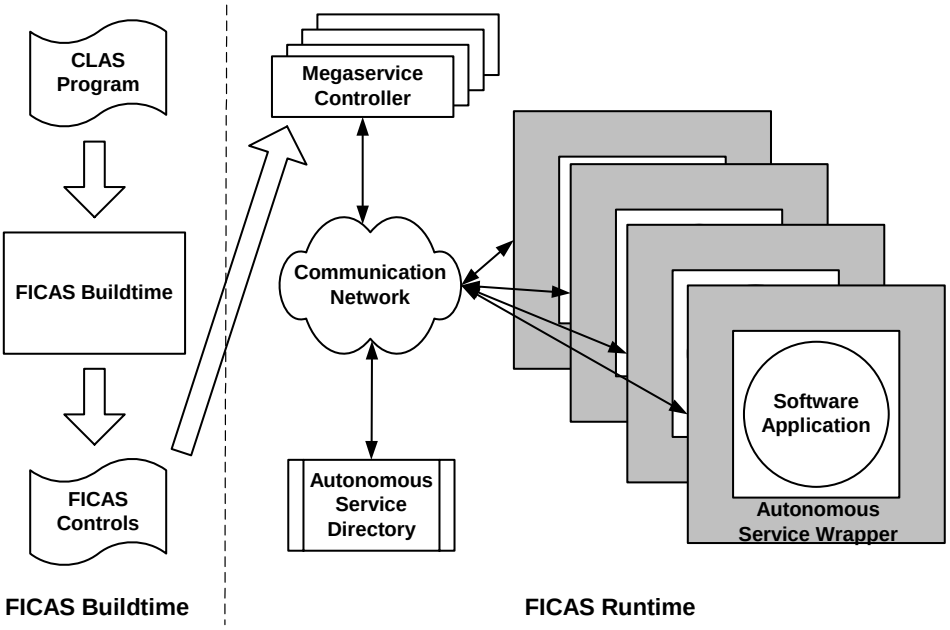

Figure 3: FICAS Architecture

3 Distributed Data-flow Planning

In FICAS, the megaservice controller has the sole responsibility for managing the control-flows for a megaservice. Based on an execution plan, the controller executes and schedules autonomous services by managing and coordinating the choice, timing, sequence, and dependencies of the outgoing ASAP events. The purpose of scheduling is to improve the Quality of Service (QoS) of the megaservice. Many techniques used to improve QoS for distributed workflows have been proposed for web service processes [15, 16]. The current implementation of FICAS focuses on minimizing the aggregate data communication cost among services.

One key characteristic of the FICAS service model is the explicit separation of control-flow and data-flow; such a separation is similar to the concept of separating the data-oriented view and the activity-oriented view [17]. The idea of separating data-flow from control-flow can also be seen in some distributed workflow environments. For instance, Exotica/FMQM adopts distributed workflow execution and data management for distributed workflow applications [18, 19]. However, data flow in a distributed workflow environment is often supported by a set of loosely synchronized replicated databases instead of direct messages, as supported by FICAS.

3.1 Distribution of Data-flows

Traditionally, both control-flows and data-flows are centrally coordinated, as illustrated in Figure 4(a). The megaservice requests information from *Service1* and passes the information onto *Service2* for further processing. The result of *Service2* is then forwarded to *Service3*. The central megaservice coordinates all the autonomous service invocations. Since the data-flows and the control-flows are not separated, the megaservice control serves as the hub for all the data communications. We call this runtime model the *centralized control-flow centralized data-flow model* (*1C1D) model*. The 1C1D model represents the simplest form of service composition runtime environment. Examples of the 1C1D model include the default usage of CORBA [2], J2EE [3], and Microsoft .NET architecture [4].

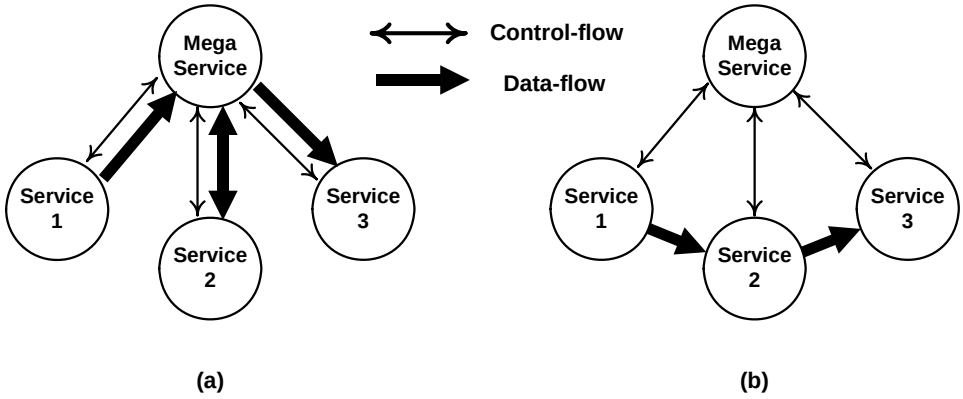

Figure 4: Centralized and Distributed Data-flows

There are performance and scalability issues associated with the 1C1D model. The megaservice must forward all data among autonomous services. Since data flows indirectly, there is extra communication traffic. The megaservice control becomes a communication bottleneck when large amounts of data are exchanged among the services. Furthermore, since all data traffic goes through the megaservice, the communication links of the megaservice become the critical system resource. It is especially problematic in an Internet environment, where the communication links between the megaservice and autonomous services likely suffer limited bandwidth. The centralized communication topology makes the 1C1D model difficult to scale.

The issues observed in the 1C1D model motivate the possible advantages to distribute the data-flows for the executions of megaservices. Figure 4(b) shows the control-flows and the data-

flows exhibited in a distributed data-flow infrastructure. The megaservice can inform two autonomous services to establish a direct data-flow. For instance, data are exchanged between autonomous services, from *Service1* to *Service2*, and from *Service2* to *Service3*, without going through the megaservice. This paper describes how FICAS distributes data-flows while maintaining centralized control mechanism as in the 1C1D model. This runtime model is called the *centralized control-flow distributed data-flow model* (*1CnD) model*. The decision to retain a centralized control-flow hinges upon ease of implementation and management. Applying distributed control-flow models effectively to conduct service composition is difficult in that it is hard to monitor the execution processes. In addition, there remain many technical challenges to convert a centralized megaservice specification of control sequences into distributed operational code segments.

By distributing data-flows, FICAS eliminates the focused, redundant, and heavy data traffic caused by forwarding everything through the megaservice. The distributed data-flow model utilizes the communication network among autonomous services, and thus alleviates communication loads on the megaservice. Furthermore, FICAS allows computations to be distributed efficiently to where data resides, and in doing so the data can be processed on location with minimal communication traffic.

3.2 Planning Distributed Data-flows

Optimizing the placement of data processing to minimize data transfer has been a subject of interests for distributed database systems [20-23]. Query optimization on distributed database systems generally requires deciding where to send the data and where to perform query operations. A similar concept is adopted in FICAS to plan the distributed data-flows. There are three steps in generating an execution plan. First, the megaservice program is analyzed to discover data dependencies among autonomous services. Then, a data dependency graph is constructed to identify independent data-flows. Finally, based on the data dependency graph, the megaservice controller then builds an execution plan for the megaservice.

The data dependencies among the autonomous services are analyzed when the program is interpreted. The megaservice controller extracts from the statements the data dependencies among autonomous services. Figure 5 presents an example of a megaservice program segment, which shows implicit data dependencies between autonomous services. Invocation of *Service3*

takes *A* and *B* as input, which are the outputs of the invocations of *Service1* and *Service2*, respectively. Hence, *Service3* is data dependent on *Service1* and *Service2*. The dependencies are mapped into a data dependency graph (DDG) as shown in Figure 6. The nodes represent autonomous service invocations, and the directed arcs represent data dependencies between autonomous service invocations. Each directed arc points to the dependent autonomous service and is tagged with the data elements exchanged between the pair of autonomous services. For example, the arc between *Invocation1* and *Invocation3* represents that *Invocation3* is dependent on *Invocation1*, with *A* being the data element passed from *Invocation1* to *Invocation3*.

The megaservice execution plan is represented by an event dependency graph (EDG). The node in the EDG contains an outgoing ASAP event from the megaservice controller. The arc establishes a predecessor-successor relationship between a pair of ASAP events. The successor ASAP event cannot be sent until the action taken by the predecessor ASAP event is completed, i.e., the megaservice controller receives the response of the predecessor ASAP event. The megaservice controller uses the EDG to coordinate the execution of the megaservice. Invocation nodes in the DDG can be directly mapped onto the INVOKE event nodes in the EDG.

```
Invocation1 = Service1.invoke()
Invocation2 = Service2.invoke()

A = Invocation1.extract();
B = Invocation2.extract();

Invocation3 = Service3.invoke(A, B)

C = Invocation3.extract();

Invocation4 = Service4.invoke(C)
D = Invocation4.extract();
```

Figure 5: Example Megaservice Program Segment

Figure 7 shows the mapping scheme where data communications are directed between dependent autonomous services, resulting in the 1CnD execution model. The megaservice controller functions merely as a coordinator for the ASAP events that control the data communication activities. Each directed arc in the DDG is mapped onto a MAPDATA event node with arcs connecting the predecessor and successor event nodes. For instance, the arc tagged with *A* in the DDG (shown in Figure 6) is mapped onto the *MAPDATA(A, Service1,*

Service3) event node in the EDG (shown in Figure 7). In addition to the predecessor-successor statements as shown in this example, FICAS has also implemented some of the basic control constructs such as switching and looping. Details on these operations are discussed by Liu [5].

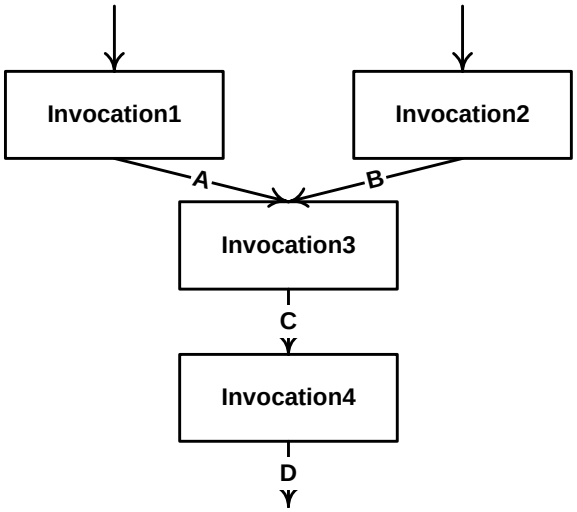

Figure 6: Sample DDG

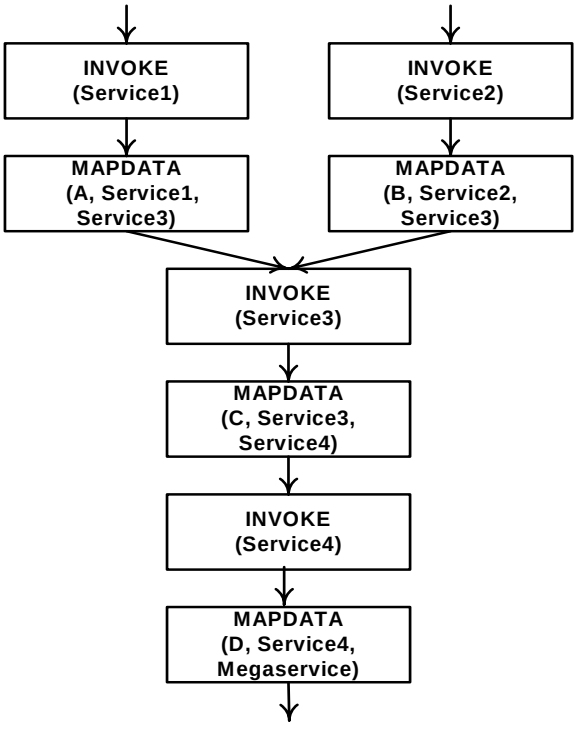

Figure 7: EDG with Distributed Data-flows

3.3 Performance Analysis

This section examines the performance of a sample megaservice supported by FICAS using different configurations of the computing environment. FICAS is compared with the centralized data-flow model by implementing the same megaservice under SOAP (Simple Object Access Protocol) [24]. As a lightweight protocol for exchanging information between applications in a distributed computing environment, SOAP has shown great potential for simplifying web service composition and the distribution of software over the Internet. There are several implementations of SOAP. They differ in their support for class binding, ease of use and performance [25]. As one of the popular choices for the SOAP implementations, Apache SOAP is selected to be the reference implementation.

Figure 8 illustrates the computing environment for the performance evaluation. Two autonomous services that focus on data communications are involved. No computational processing occurs on these autonomous services. Autonomous service *S1* randomly generates and returns a string whose size is specified by the input parameter. Autonomous service *S2* takes the string as input and immediately returns without doing anything. Two megaservices that utilize the autonomous services are constructed. The first megaservice, *MultiService*, forwards the string generated by the autonomous service *S1* to the autonomous service *S2*. This megaservice is designed to examine the impact of the data-flow distribution. The second megaservice, *SingleService*, simply invokes the autonomous service *S1*. This megaservice is used to measure the cost of a single service call.

The autonomous services and the megaservices are implemented for both SOAP and FICAS. The megaservices are implemented as Java applications that invoke the services using the Apache SOAP v2.2 API (Application Program Interface) library. For FICAS, the autonomous services are wrapped using our developed Java library. The service cores of the autonomous services are identical in functionality to their SOAP counterparts. The megaservices are specified as CLAS programs, which are compiled into control sequences by the FICAS buildtime environment. The megaservices are executed by sending the control sequences to a megaservice controller.

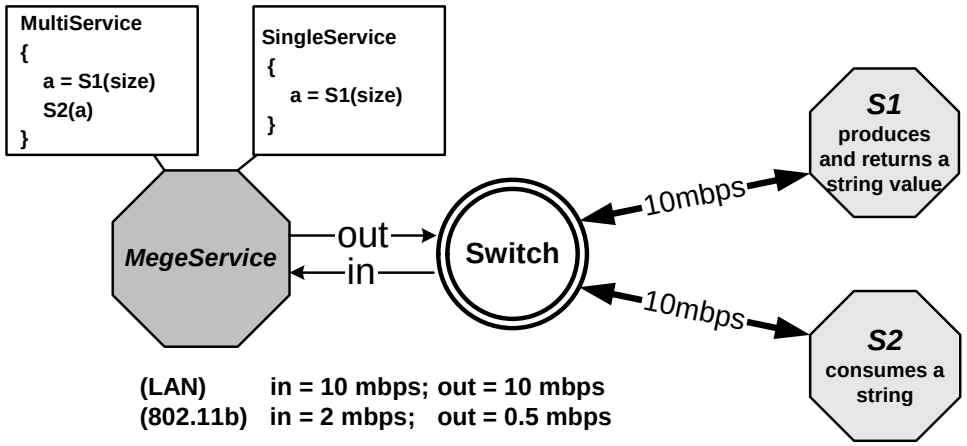

Figure 8: Test Environment for Comparing SOAP and FICAS

The tests are performed in a distributed computing environment. The machines are each configured with a Pentium-III 1 GHz processor and 256 MB RAM, running Windows 2000 Professional. The autonomous services run on two separate servers connected to a switch via a Local Area Network (LAN), whose bandwidth is 10 mbps each way. The megaservices run on the client machine. Two types of network connections are used to connect the client machine to the servers. The first connection uses LAN, whose communication bandwidth among all machines is 10 mbps each way. This type of connection resembles many corporate computing environments. The second connection uses an 802.11b wireless link. The downloading bandwidth is approximately 2 mbps, and the uploading bandwidth is approximately 0.5 mbps. This type of connection resembles a computing center environment, where servers are connected by high-speed communication links, but are accessed via relatively slower communication links.

The execution times of the megaservices are measured with different settings on the data volume involved with the megaservices. The data volume is specified by the input parameter to the autonomous service *S1*. Figure 9 shows the measured performance of the megaservices when the client machine is connected to the LAN. The following observations can be made:

- FICAS performs worse than SOAP when the data volume is low. This is expected and can be explained for two reasons. First, FICAS has more complicated control-flows than SOAP. FICAS breaks down a single service call in SOAP into multiple control messages. FICAS also incurs more overhead in initializing and terminating the autonomous services. Second,

it is expected that Apache SOAP, being developed for quite some time, is better optimized than FICAS in terms of its Java source code.

- The performance of the FICAS megaservice *MultiService* is comparable to that of the SOAP megaservice *SingleService*. The megaservices are similar in performance because two megaservices incur the same amount of data-flows. For *SingleService*, the string generated by the autonomous service *S1* is sent to the megaservice. For *MultiService*, the same string is sent from the autonomous service *S1* to the autonomous service *S2*. The slight difference in the execution times of the megaservices can be attributed to the differences in control-flows.
- FICAS outperforms SOAP when the data volume is high. This is because the SOAP megaservice incurs twice as much data-flow as the FICAS megaservice. For the SOAP megaservice, two data messages are used to send the string from the autonomous service *S1* to the autonomous service *S2*, via the megaservice controller. For the FICAS megaservice, only one data message is used to send the string from the autonomous service *S1* to the autonomous service *S2*.

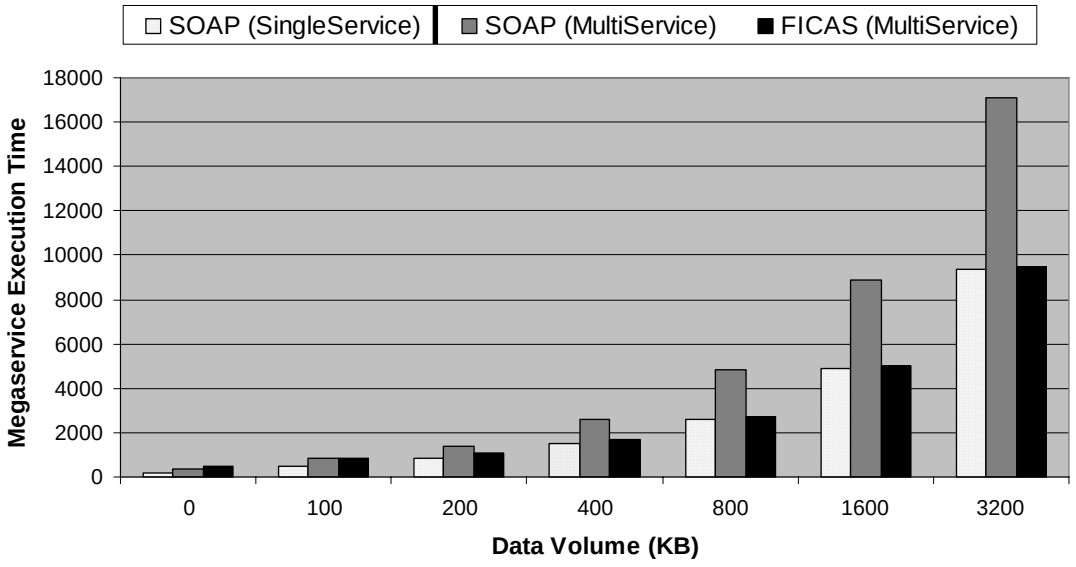

Figure 9: Comparison Between FICAS and SOAP on Local Area Network

Figure 10 compares the performance of the SOAP megaservice *MultiService* and the FICAS megaservice *MultiService* under various network settings. Under the LAN setting, the megaservices access the autonomous services through the 10 mbps LAN. Under the wireless setting, the megaservices access the autonomous services via a slower 802.11b access point. In both cases, communications with the megaservice have much lower bandwidth than communications among the autonomous services. Comparing the megaservice performance between the LAN and the wireless 802.11b settings, the following can be observed:

- The execution times for the SOAP megaservice increase significantly as the bandwidth of the communications with the megaservice decreases. Since all data-flows and control-flows go through the megaservice, the communications with the megaservice become the bottleneck of the system.
- The execution times for the FICAS megaservice increase only slightly when comparing the wireless and the LAN settings. As the data-flows are distributed among the autonomous services, communications with the megaservice are only used for the control-flows. Because the control messages are small and compact in nature, the control-flows place little burden on the network. Thus, the performance of the megaservice is barely affected.

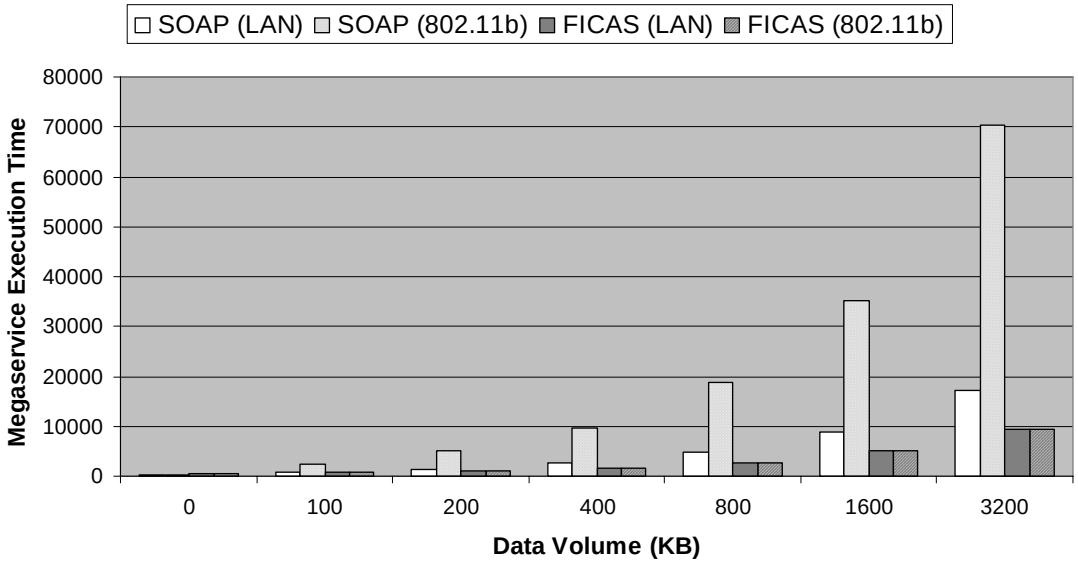

Figure 10: Megaservice Performance Under Different Network Configurations

To summarize, 1CnD model responds better than 1C1D when the data volume is large or when the bandwidth is limited for communicating with the megaservice. All network traffic in 1C1D goes through the megaservice, and thus places heavy burden on its communication links. In contrast, 1CnD distributes the data-flows and takes advantage of the communication network among the autonomous services.

4 Mobile Classes and Active Mediation

While the FICAS approach gains from direct data communication among the services, it does not have the capability to directly map incompatible sources or to integrate information from diverse sources. For example, in the semantic web setting [26, 27], where there are a large collection of autonomous and diverse providers, it cannot be expected that each service can deliver results that can be fully compatible and useful to other services that the composed application may need to invoke. Active mediation is introduced to provide client-specific functionalities so that services can be viewed as if they were intended for the specific needs of the client [6]. This section describes a mediation architecture that supports the execution of mobile classes. In addition, an algorithm is presented that determines the optimal location to carry out the execution of a mobile class.

4.1 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. The mobile class supported in FICAS is similar to the mobile agent technology [28-30]. Both approaches utilize executable programs that can migrate during execution from machine to machine in a heterogeneous network. However, mobile agents are self-governing in that they decide when and where to migrate on their own. On the other hand, the mobile class is an integral part of the service composition framework. Since mobile classes are controlled by megaservices, their management and deployment become easier.

Mobile classes can be implemented in many general-purpose programming languages [31-33]. In this work, Java is chosen as the specification language for mobile classes [34]. 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.

Mobile classes enable megaservices to perform computations with greater efficiency. Figure 11 shows an example where mobile classes are used in place of type broker services to conduct type conversions. Traditionally, an autonomous service serving as a type broker or a distributed network of type brokers can be used to mediate the difference among data in various formats [35]. A type graph is used to figure out the chain of necessary conversions. An example of automating this process can be seen in the Ninja project [36]. Figure 11(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. A potentially large amount of data is passed among the type brokers. Alternatively, mobile classes can be used in place of type brokers to handle type mediation. Rather than forwarding data among the type brokers, the megaservice loads the mobile classes onto the autonomous services to provide the type mediation functions. Multiple mobile classes for type mediation can be utilized together, forming a network of type brokers. As shown in Figure 11(b), two mobile classes are used to convert data from type T1 to type T3. The type mediation is conducted at the source autonomous 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 autonomous service. Since the mobile classes are invoked on the source autonomous service, the multiple interim data transfers are eliminated and data traffic is limited to essential transmissions.

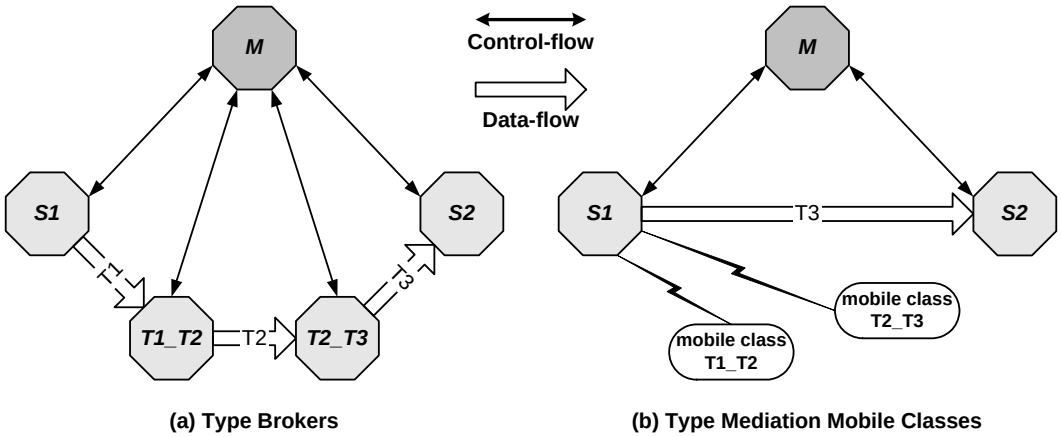

Figure 11: Type Conversion Using Type Broker Services and Mobile Classes

4.2 Active Mediation

An active mediator is an information-processing engine that resides between source information services and information clients. Incorporation of an active mediator allows an autonomous service to support the execution of mobile classes. Active mediator processes the source information by executing mobile classes specified by information clients. Figure 12 illustrates the architecture of an active mediator:

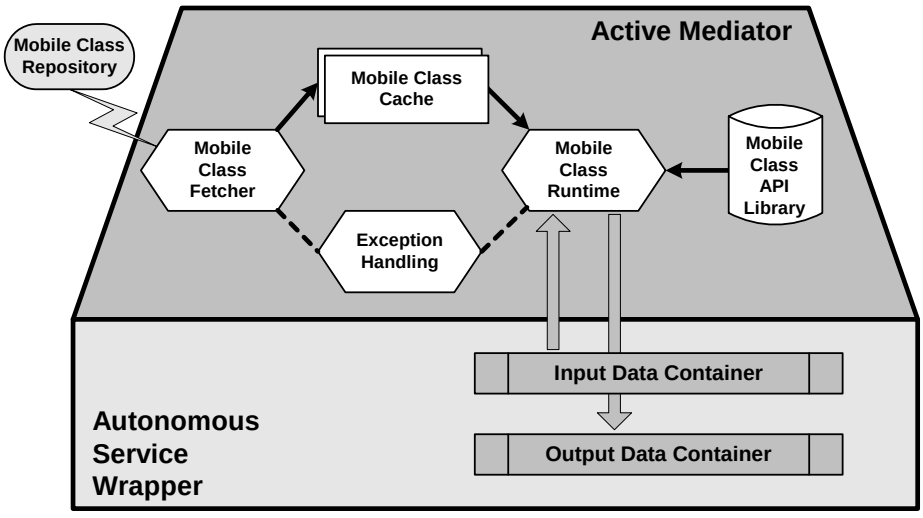

Figure 12: Active Mediation Architecture

- The Mobile Class Fetcher is responsible for loading the Java code of the mobile class. The name of the mobile class indicates where the Java class file can be found.

- The Mobile Class Cache is a temporary storage for the loaded Java class. The Mobile Class Cache is used to avoid the duplicate loading of a mobile class. The cache is looked up first before any Java classes are loaded. Only when a cache miss occurs is the Mobile Class Fetcher used.
- The Mobile Class API (Application Program Interface) Library stores the utility classes that make the construction of mobile classes more convenient. For instance, the Java Development Kit library [37] 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 runtime uses the data containers of the autonomous service to manage the input and output data of the mobile class. The parameters for invoking the mobile class are loaded into the input data container by the megaservice controller before the invocation of the mobile class. The parameters are looked up and supplied to the *execute()* function. The result of the *execute()* function is put into the output data container, and can then be utilized by the megaservice controller.
- The Exception Handling module provides error handling for the loading and the execution of mobile classes.

4.3 Placement of Mobile Classes

The choice of which autonomous service executes the mobile class affects how the data-flows are formed for the megaservice to which the mobile class belongs. The placement of the mobile class therefore has significant impact on the performance of the megaservice. An example megaservice, as shown in Figure 13, is used to demonstrate such an impact. The megaservice involves two autonomous services and one mobile class. The autonomous services, *S1* and *S2*, are the same as the ones in the example illustrated in Figure 8. The mobile class *FILTER* takes a large string as input, filters through the content, and returns a string that consists of every 10th 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 autonomous services involved in the megaservice, there are three potential placement strategies, as shown in Figure 14:

- Strategy 1: By placing the mobile class *FILTER* at the autonomous service that hosts the megaservice controller, we can construct the execution plan as shown in Figure 14(a). *S1* generates the data element *A* and passes it to the megaservice. The mobile class processes *A* at the megaservice, and the result *B* is then sent to *S2* for further processing.
- Strategy 2: By placing the mobile class *FILTER* at *S1*, the execution plan as shown in Figure 14(b) can be constructed. *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.
- Strategy 3: By placing the mobile class *FILTER* at *S2*, we can construct the execution plan as shown in Figure 14(c). *S1* generates the data element *A* and passes it to *S2*. *S2* processes *A* locally using the mobile class to generate the result *B*.

To compare the strategies, it is assumed that the performance of loading and executing the mobile class is the same on all autonomous services. Strategy 1 requires both the input data element *A* and the output data element *B* to be transmitted among the megaservice and the autonomous 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 sent between the autonomous services. The data element *B* is sent from *S1* to *S2* in Strategy 2, and the data element *A* is sent from *S1* to *S2* in Strategy 3. 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.

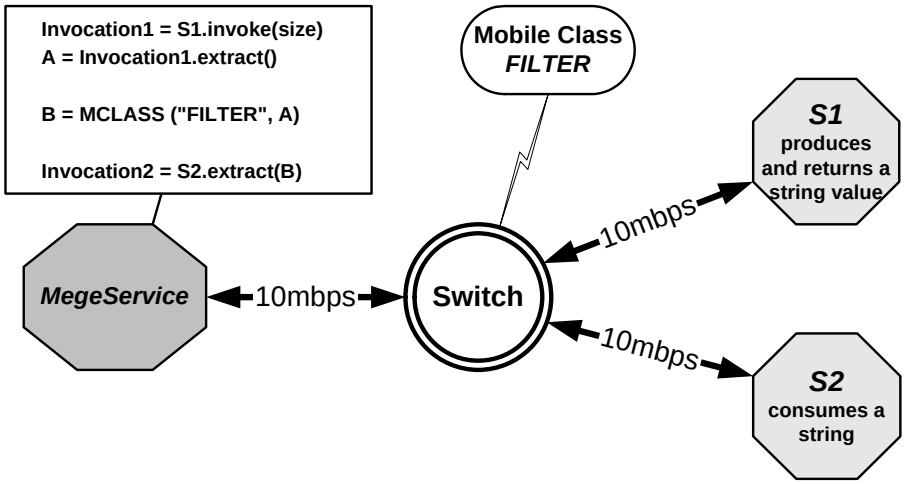

Figure 13: Example Megaservice that Utilizes the Mobile Class *FILTER*

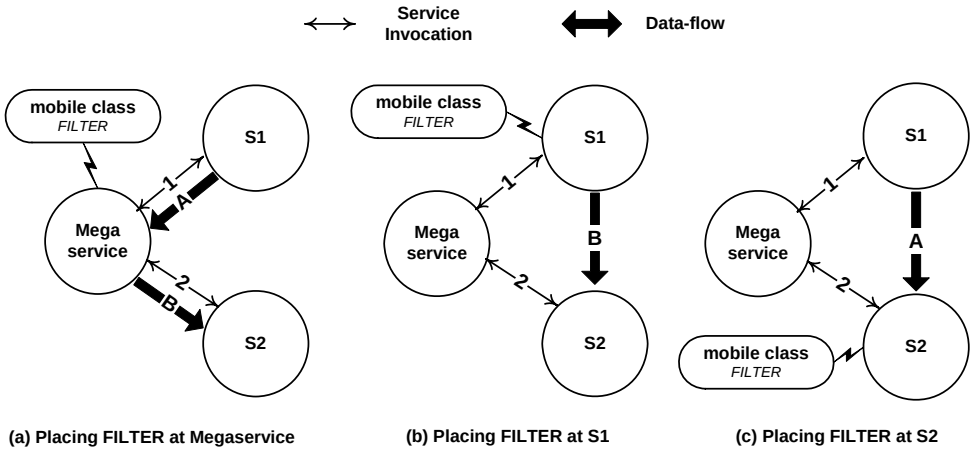

Figure 14: Execution Plans with Different Placements for the Mobile Class

The optimal placement of a mobile class should usually minimize the data-flows among related autonomous services. For a mobile class, each input data element to the mobile class is represented as a pair, (S_i, V_i), 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 mobile class will be sent, and V_0 is the size of the data element. Two observations can be made. First, the sum of V_i remains the same regardless where the mobile class is executed. Second, placing the mobile class on the autonomous service S_i can eliminate the corresponding data-flow volume V_i as the data element is local to the autonomous service. Therefore, the optimal placement of the mobile class is the autonomous service S_i that has the largest aggregated V_i.

Figure 15 shows the LDS (Largest Data Size) algorithm that selects the autonomous service that generates and consumes the largest volume of data for a given mobile class. The algorithm first computes the total amount of data attributed to 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 mobile class. S_{max} is returned as the output of the algorithm.

The LDS algorithm is applicable when the input and output data sizes are known for the mobile classes. For a situation where the output data size of a mobile class is only determined after the execution of the mobile class, the output data size needs to be estimated. The output data size of a mobile class can be viewed as a function of the input data sizes of the mobile class: $S_O = f(S_A, S_B, \ldots)$. The function f is called the sizing function of the mobile 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 code in the mobile class repository. The megaservice controller can then use the sizing function to estimate the output data size for running the LDS 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 15: LDS Algorithm for Optimal Mobile Class Placement

The LDS algorithm assumes that the network links among all autonomous services are of comparable performance. When this assumption does not hold, a more complicated model can be adopted to minimize the aggregated time. Various network parameters, such as topology of network and bandwidth of network channels, can have impact on the performance of the megaservice, and other algorithms (for example, see [22]) can also be implemented in FICAS.

5 Example Application

The examples shown in the previous section show that FICAS is well suited for composing autonomous services that exchange large amounts of data. The distribution of data-flows and the use of mobile classes facilitate service composition and improve the performance of the megaservice. To demonstrate the effectiveness of FICAS, an engineering service infrastructure for construction project management applications has been implemented [38]. The process of building the service infrastructure includes: (1) wrapping software applications into autonomous services, (2) implementing mobile classes, and (3) constructing megaservices to accomplish the engineering tasks.

5.1 Building Autonomous Services

The first step in building the engineering service infrastructure is to wrap each software application into an autonomous service. The service core of the autonomous service is created by defining the *ServiceCore* interface based on the software application. The service core is then linked to an autonomous service wrapper (ASW). Figure 16 shows an example of wrapping the Primavera P3™ application software into an autonomous service that supports project scheduling. The *P3Service* class implements the three methods in the *ServiceCore* interface. The *setup()* method and the *terminate()* method specify that no action is performed for the initialization and the termination of the autonomous service. The *execute()* method defines the actions for the invocation of the autonomous service. The method starts by fetching the input parameters from the input data container. The first parameter specifies the service request, and the second parameter contains the input data for a schedule, based on which the Primavera P3™ application is utilized to conduct scheduling. The result of the scheduling is encapsulated into a data element and put into the output data container. The *P3Service* class is provided as an input to the constructor of the *ASW* class to connect the Primavera P3™ application with the autonomous service wrapper. After the autonomous service is built, it is registered with the autonomous service directory. The registration entry specifies the name, the IP address, and the port number of the autonomous service. Once registered, the autonomous service is ready to be used for composition.

5.2 Constructing Mobile Classes

Lightweight information processing routines are specified as mobile classes, whose executions are determined by megaservices during runtime. Figure 17 shows a sample mobile class that converts data from Process Specification Language (PSL) format [39] into Microsoft Excel format. The *psltoexcel* class implements the *MobileClass* interface, whose definition is provided in the *FICAS.zip* class library. The *execute()* function takes the first argument for the mobile class as the input data in PSL, converts the data into Microsoft Excel format, and returns the converted data as the output data element.

In the engineering information service infrastructure, mobile classes are compiled and their byte codes are stored in a repository that is accessible from the web. Megaservices locate a mobile class by attaching a base URL to the mobile class name. For instance, if the base URL

for the mobile class repository is *http://ficas.stanford.edu/mcrepo*, then the byte codes for *psltoexcel* can be located at *http://ficas.stanford.edu/mcrepo/psltoexcel.class*.

```
public class P3Service implements ServiceCore
{
  public boolean setup(Container inc, Container outc, FlowId inf) {
    return true;
  }

  public boolean terminate(Container inc, Container outc, FlowId inf)
  {
    return true;
  }

  public boolean execute(Container inc, Container outc, FlowId inf) {
    /* Fetch the desired operation from the input data container */
    String operation = inc.fetch(inf, 0).getStringValue();

    if (operation.equals("reschedule")) {
      /* Fetch the input schedule from the input data container */
      String input = inc.fetch(inf, 1).getStringValue();

      /* Invoke P3 to conduct rescheduling */
      String output = P3Schedule(input);

      /* Put regenerated schedule on the output container */
      outc.put(inf, 0, new DataElement().setValue(output));
    }

    return true;
  }

  private String P3Schedule(String schedule) {
    /* Invokes the Primavera P3 software to process the input,
       the result of the rescheduling is returned */
    ...
  }

  public static void main(String argv[]) throws Exception {
    if (argv.length != 1) {
      System.err.println("Usage: java P3Service port");
      return;
      }

    /* Creating the autonomous service */
      new ASM(Integer.parseInt(argv[0]), new P3Service());
  }
}
```

Figure 16: Example Autonomous Service that Utilizes Primavera P3

```
public class psltoexcel implements MobileClass
{
  public DataElement execute(Vector params) {
    /* Fetch the input data, in PSL format */
    String p3 =
      ((DataElement) params.firstElement()).getStringValue();

    /* Convert the input data to excel format */
    String excel = Convert_PSL_To_Excel(p3);

    /* Return the converted data, in Excel format */
    return new DataElement().setValue(excel);
  }

  private String Convert_PSL_To_Excel(String p3) {
    ...
  }
}
```

Figure 17: Example Mobile Class that Converts Data from PSL to Microsoft Excel

5.3 A Sample Megaservice

Figure 18 shows an example megaservice that utilizes multiple autonomous services and mobile classes to perform rescheduling of project plans. The megaservice is specified as a CLAS program. Three autonomous services are utilized by the megaservice: (1) the *PSLService* that handles the access of the project models, (2) the *P3Service* that conducts the scheduling of a project plan, and (3) the *ExcelService* that displays the project plan. In addition, the mobile class *psltoexcel* is used to convert data between the PSL format and the Microsoft Excel format. The megaservice is compiled into a control sequence in FICAS. The invocation of the megaservice causes the *PSLService* to fetch the project model, which is then rescheduled by the *P3Service*. The update schedule is stored back to the database using the *PSLService* and shown to the project personnel using the *ExcelService*.

A sample scenario is presented to demonstrate how the engineering service infrastructure helps facilitate personnel, from different functional groups, conduct collaborations. The data for the test case model is part of the Mortenson Ceiling project (part of the construction of the Disney Concert Hall, designed by Frank Gehry). Figure 19 shows the view of the scheduling information using Primavera P3™. The project data is stored in a relational database. The data is shared between the relational data model and the proprietary Primavera data model using the *PSLService*. The project schedule can also be reviewed using a handheld Palm device to directly access the relational database. This capability can be helpful for the on-site personnel of the

construction project. Suppose that the duration for the activity 18T1-33201, for erecting a roof element, is hypothetically changed from 1 day to 40 days, as shown in Figure 20. The change can be made remotely using the Palm device. The update will then trigger the *SchedulingDemo* megaservice, which updates the project schedule. As part of the *SchedulingDemo* megaservice, the project schedule is also automatically updated in Excel to notify the project personnel, as shown in Figure 21. The updated schedule can also be displayed using MS Project either retrieving the data from the relational database or directly exchanging with Primavera P3™ via a PSL data exchange service. Figure 22 shows that not only the activity 18T1-33201 is updated, but the dependent activities are also updated as well.

The example infrastructure involves software applications (Primavera P3™, Excel, Microsoft Project, Oracle database, PALM service, etc.) that exchange large amount of data. The applications are conveniently wrapped into autonomous services. Computational tasks are easily specified using mobile classes. Engineering processes are systematically defined as megaservices. This example demonstrates the applicability of FICAS model for the composition of large-scale engineering web services. Other examples of using FICAS for project management applications can be found elsewhere [5, 40].

For the current implementation of FICAS, sophisticated concurrent control is not fully supported. When a process sends a write request to the project data, the data will be locked for further write requests, but read requests are still permitted. To improve system performance and flexibility, granular lock [41] may be implemented.

```
SchedulingDemo "http://ficas.stanford.edu/mcrepo"
{
  psl_svc = SETUP("PSLService")
  p3_svc = SETUP("P3Service")
  excel_svc = SETUP("ExcelService")

  /* Fetch project data from database */
  psl = psl_svc.INVOKE("to-psl", "%%")
  original_schedule = psl.EXTRACT()

  /* Reschedule project */
  p3 = p3_svc.INVOKE("reschedule", original_schedule)
  updated_schedule = p3.EXTRACT()

  /* Store the updated project data into database */
  oracle = psl_svc.INVOKE("to-oracle", updated_schedule)
  status1 = oracle.EXTRACT()

  /* Populate Excel Service with updated project data */
  excel_data = MCLASS("psltoexcel", updated_schedule)
  excel = excel_svc.INVOKE("populate", excel_data)

  psl_svc.TERMINATE()
  p3_svc.TERMINATE()
  excel_svc.TERMINATE()
}
```

Figure 18: Sample Megaservice Specified in CLAS

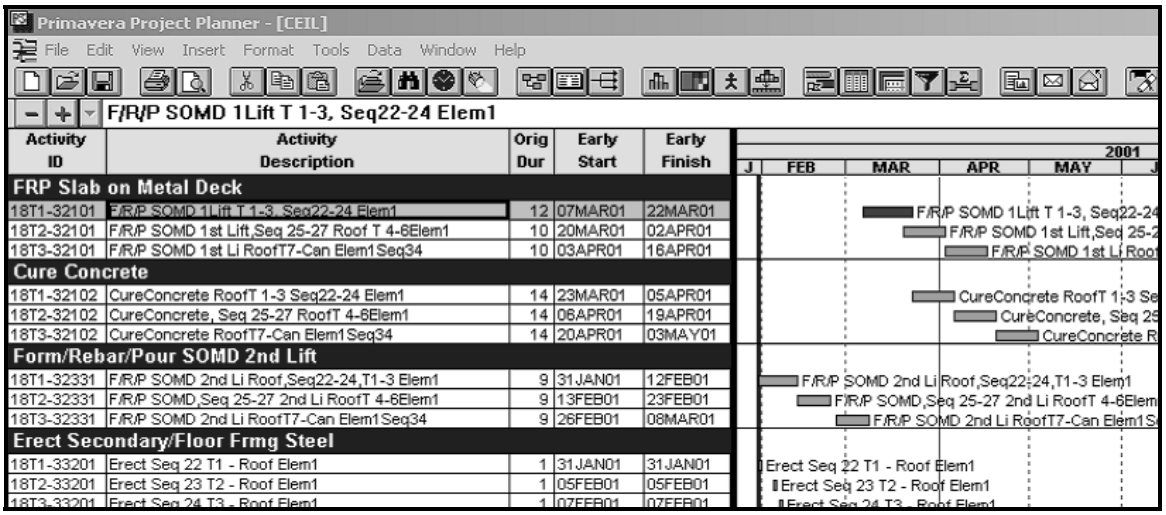

Figure 19: Reviewing the Project Schedule in Primavera P3

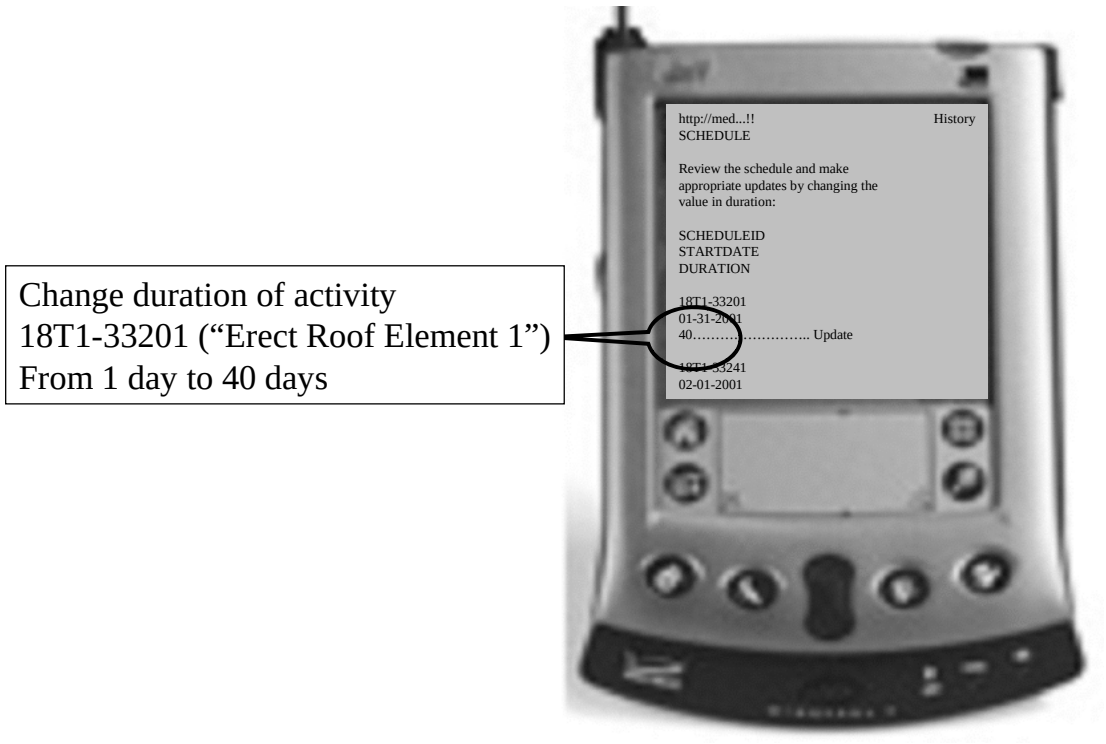

Figure 20: Revising the Project Schedule via a Palm Device

Microsoft Excel - exceldata1043198044210.csv

File Edit View Insert Format Tools Data Window Help

C158 = 40

	A	B	C	D	E	F	G	H	I	J
148	17T7-61511	8/30/2001	28	0	0					
149	17T7-61900	4/26/2001	3	0	83					
150	17T8-61501	5/5/2001	2	10	186					
151	17T8-61511	10/10/2001	15	60	60					
152	17T8-61900	5/1/2001	4	0	154					
153	1800-71151	1/31/2001	6	0	315					
154	1800-71201	2/1/2001	8	0	315					
155	18T1-32101	5/1/2001	12	0	0					
156	18T1-32102	5/17/2001	14	0	315					
157	18T1-32331	1/31/2001	9	0	298					
158	18T1-33201	1/31/2001	40	0	0					
159	18T1-33401	4/20/2001	7	0	0					
160	18T2-32101	5/14/2001	10	0	23					
161	18T2-32102	5/31/2001	14	0	315					
162	18T2-32331	2/13/2001	9	0	298					
163	18T2-33201	3/30/2001	1	0	0					
164	18T2-33401	5/1/2001	9	0	23					
165	18T3-32101	5/29/2001	10	0	144					
166	18T3-33201	4/3/2001	1	0	0					

Figure 21: Reviewing the Updated Project Schedule in Microsoft Excel

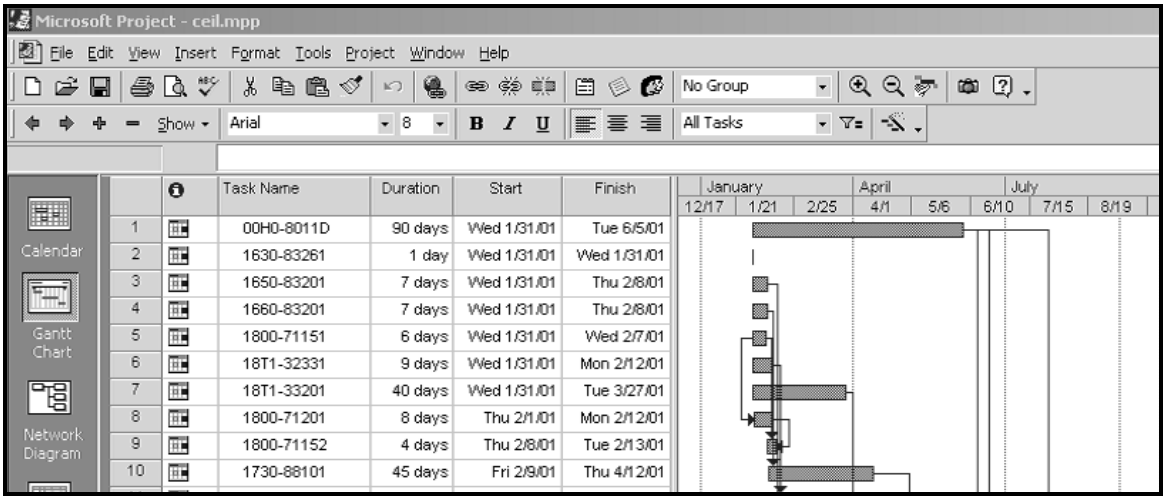

Figure 22: Reviewing the Updated Schedule in Microsoft Project

6 Summary

This paper investigates the integration of web services that communicate large volumes of data. Traditionally, a megaservice resides at the central hub that handles all the data traffic, while each web service processes data supplied by the composed application. This centralized data flow is shown to be inefficient when data are substantial. To improve effective use, the distributed data flow approach is introduced which allows direct data exchange among the web services. The FICAS architecture is defined to enable smooth adoption of distributed data-flow and active mediation in services composition. Programmers or users can specify in FICAS a metamodel for autonomous services, based on which services can be accessed and composed in a homogeneous manner. The metamodel leads to the ASAP protocol that separates the data communications from the control processing in autonomous services. The analysis shows that the distribution of data communications improves megaservice performance, especially when large volumes of data are exchanged among the services. The distributed data-flow approach also eliminates the bottleneck on the communication links of the megaservice by taking advantage of the communication network among the services.

Used appropriately, active mediation will greatly facilitate service composition, both in functionality and in performance. Active mediation increases the customizability and flexibility of web services. Specifically, active mediation enables interoperation of web services without requiring that heterogeneous data be transmitted via central nodes. It utilizes code mobility to

facilitate dynamic information processing in service composition. Delegating the maintenance of software is an important benefit of the services model [42]. Active mediation allows data-processing tasks to be specified for composed applications, at the same time separating computation from composition.

An application scenario is presented to demonstrate the process by which services are built and integrated using FICAS. Legacy engineering applications are tied together to form integrated work processes. This example shows that distributed data flow, combining with mobile classes, is effective and more efficient than centralized processing when integrating large engineering software services. In the example engineering application, the controlling node can run on a low bandwidth device and thus has tremendous effects on performance. Typically, mobile devices are attractive to manage complex scenarios in dealing with governmental regulation [43], engineering [38], healthcare [26], and military situations [44]. These cases can be benefited significantly by distributed dataflow and active mediation model, such as FICAS.

Acknowledgement

This work was partially sponsored by the Center for Integrated Facility Engineering at Stanford University, a Stanford Graduate Fellowship, the Air Force (Grant F49620-97-1-0339, Grant F30602-00-2-0594), and the Product Engineering Program at National Institute of Standards and Technology (NIST). The Product Engineering Program gets its current support from the NIST's SIMA (Systems Integration for Manufacturing Applications) program. The authors would like to acknowledge an equipment grant from Intel Corporation for the support of this research. The Mortenson Ceiling Project data was provided by Professor Martin Fischer and his research group at Stanford University, and the wrappers for the various software were developed by Mr. Jim Cheng; their contributions are greatly acknowledged. Last but not least, the authors would like to thank Gordon Lyon and Edward J. Barkmeyer of NIST for their valuable suggestions, which substantially improved the exposition of this paper. No approval or endorsement of any commercial product by the National Institute of Standards and Technology or by Stanford University is intended or implied.

References

[1] Wiederhold G, Wegner P and Ceri S. Towards megaprogramming: A paradigm for component-based programming. Communications of the ACM 1992;35(11):89-99.

[2] OMG. The common object request broker: Architecture and specification version 2.0, Object Management Group, Report 95-3-10, 1995.

[3] Bodoff S, Green D, Haase K, Jendrock E, Pawlan M and Stearns B. The J2EE tutorial, Boston, MA: Addison-Wesley Professional, 2002.

[4] Platt DS. Introducing Microsoft .Net, 3rd Edition, Microsoft Press, 2003.

[5] Liu D. A distributed data flow model for composing software services. Ph.D. Thesis. Department of Electrical Engineering, Stanford University, 2003.

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

[7] Fuggetta A, Picco GP and Vigna G. Understanding code mobility. IEEE Transactions on Software Engineering 1998;24(5):342-61.

[8] Curbera F, Khalaf R, Mukhi N, Tai S and Weerawarana S. The next step in web services. Communications of the ACM 2003;46(10):29-34.

[9] Narayanan S and McIlraith SA. Simulation, verification and automated composition of web services, Proceedings of International World Wide Web Conference, Honolulu, Hawaii, 2002;77-88.

[10] Andrews T, Curbera F, Dholakia H, Goland Y, Klein J, Leymann F, et al. BPEL4WS specification: Business process execution language for web services version 1.1. http://www-106.ibm.com/developerworks/library/ws-bpel/, 2003.

[11] Banerji A, Bartolini C, Beringer D, Chopella V, Govindarajan K, Karp A, et al. Web services conversation language (WSCL) 1.0. http://www.w3.org/TR/2002/NOTE-wscl10-20020314/, 2002.

[12] Ankolekar A, Burstein M, Hobbs JR, Lassila O, Martin DL, McIlraith SA, et al. DAML-S: Semantic markup for web services, Proceedings of the International Semantic Web Working Symposium, Stanford, CA, 2001;411-30.

[13] Beringer D, Tornabene C, Jain P and Wiederhold G. A language and system for composing autonomous, heterogeneous and distributed megamodules, Proceedings of DEXA International Workshop on Large-Scale Software Composition, Vienna, Austria, 1998.

[14] Wiederhold G, Beringer D, Sample N and Melloul L. Composition of multi-site services, Proceedings of 4th World Conference on Integrated Design and Process Technology IDPT'99, Kusadasi, Turkey, 1999.

[15] Blake MB. Coordinating multiple agents for workflow-oriented process orchestration. Information Systems and e-Business Management 2003;1:1-18.

[16] Zeng L, Benatallah B, Dumas M, Kalagnanam J and Sheng QZ. Quality driven web services composition, Proceedings of the 12th International World Wide Web Conference (WWW2003), Budapest, Hungary, 2003;411-21.

[17] Scherer RJ. AI methods in concurrent engineering. I. Smith (Eds), Artificial intelligence in structural engineering: Information technology for design, collaboration, maintenance and monitoring, Springer, Germany, 1998;359-83.

[18] Alonso G, Reinwald B and Mohan C. Distributed data management in workflow environments, Proceedings of the 7th International Workshop on Research Issues in Data Engineering (RIDE'97), Birmingham, England, 1997;82-90.

[19] Mohan C, Alonso G, Gunthor R and Kamath M. Exotica: A research perspective on workflow management systems. Data Engineering 1995;18(1):19-26.

[20] Thomas G, Thompson GR, Chung C-W, Barkmeyer E, Carter F, Templeton M, et al. Heterogeneous distributed database systems for production use. ACM Computing Surveys 1990;22(3):237-66.

[21] Ceri S, Pernici B and Wiederhold G. Distributed database design methodologies. Proceedings of the IEEE 1987;75(5):533-46.

[22] Sheth AP, Singhal A and Liu MT. An analysis of the effect of network parameters on the performance of distributed database systems. IEEE Transactions on Software Engineering 1985;11(10):1174-84.

[23] Neuhold EJ and Walter B. An overview of the architecture of the distributed data base system, Proceedings of the International Symposium on Distributed Data Bases (DDB), Berlin, Germany, 1982;247-90.

[24] Box D, Ehnebuske D, Kakivaya G, Layman A, Mendelsohn N, Nielsen H, et al. Simple object access protocol (SOAP). http://www.w3.org/TR/SOAP, 2000.

[25] Davis D and Parashar M. Latency performance of soap implementations, Proceedings of 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2002), Berlin, Germany, 2002;407-12.

[26] Berners-Lee T, Hendler J and Lassila O. The semantic web. Scientific American 2001;284(5):34-43.

[27] Hendler J, Berners-Lee T and Miller E. Integrating applications on the semantic web. Journal of the Institute of Electrical Engineers of Japan 2002;122(10):676-80.

[28] White JE. Mobile agents. J. M. Bradshaw (Eds), Software agent MIT Press, 1997;437-72.

[29] Gray RS. Agent Tcl: A flexible and secure mobile-agent system, Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96), Monterey, CA, 1997;9-23.

[30] Brewington B, Gray R, Moizumi K, Kotz D, Cybenko G and Rus D. Mobile agents in distributed information retrieval. M. Klusch (Eds), Intelligent information agents, Springer-Verlag, 1999;355-95.

[31] Cardelli L. A language with distributed scope. Computing Systems 1995;8(1):27-59.

[32] Douglis F and Ousterhout J. Transparent process migration: Design alternative and the sprite implementation. Software: Practice and Experience 1991;21(8):757-85.

[33] Muhugusa M. Implementing distributed services with mobile code: The case of the messenger environment, Proceedings of the IASTED International Conference on Parallel and Distributed Systems, Austria, 1998;1-31.

[34] Wollrath A, Riggs R and Waldo J. A distributed object model for the Java system. Computing Systems 1996;9(4):265-90.

[35] Ockerbloom J. Mediating among diverse data formats. Ph.D. Thesis. Computer Science Department, Carnegie Mellon University, 1998.

[36] Gribble SD, Welsh M, von Behren JR, Brewer EA, Culler DE, Borisov N, et al. The Ninja architecture for robust Internet-scale systems and services. Computer Networks, Special Issue on Pervasive Computing 2001;35(4):473-97.

[37] Arnold K, Gosling J and Holmes D. The Java programming language, Boston, MA: Addison-Wesley, 2000.

[38] Liu D, Cheng J, Law KH, Wiederhold G and Sriram RD. Engineering information service infrastructure for ubiquitous computing. Journal of Computing in Civil Engineering 2003;17(4):219-29.

[39] Schlenoff C, Gruninger M, Tissot F, Valois J, Lubell J and Lee J. The process specification language (PSL): Overview and version 1.0 specification, National Institute of Standards and Technology, Gaithersburg, MD, Report 6459, 2000.

[40] Cheng J. A simulation access language and framework with applications to project management. Ph.D. Thesis. Department of Civil and Environmental Engineering, Stanford University, 2004.

[41] Gunaseelan L and LeBlanc RJ. Distributed Eiffel: A language for programming multi-granular objects, Proceedings of the 4th International Conference on Computer Languages, IEEE, San Francisco, CA, 1992.

[42] Wiederhold G. The product flow model, Proceedings of 15th Conference on Software Engineering and Knowledge Engineering (SEKE), Skokie, IL, 2003;183-6.

[43] Lau GT, Kerrigan S, Law KH and Wiederhold G. An e-government information architecture for regulation analysis and compliance assistance, Proceedings of ICEC'04: Sixth International Conference on Electronic Commerce, Delft, The Netherlands, 2004.

[44] Sheng S, Chandrakasan A and Brodersen RW. A portable multimedia terminal. IEEE Communications Magazine 1992;30(12):64-75.