Composition of Autonomous 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 presents a Flow-based Infrastructure for Composing Autonomous Services (FICAS) that supports a software composition paradigm, where software components are linked together through an integration framework to form composed software applications called megaservices. The software components are provided as processes managed by independent service providers; we call these components autonomous services. FICAS employs a distributed data-flow approach that differs from the centralized data-flow approach adopted by many current service integration frameworks, such as CORBA, J2EE and SOAP. The distributed data-flow approach allows direct data exchange among the autonomous services and consequently facilitates the distribution of computations. FICAS is implemented as a collection of software modules that support the construction of autonomous services, facilitate the functional composition of autonomous services into megaservices, and carry out the execution of megaservices. We have built a prototype for an information service environment based on FICAS that incorporates a variety of construction project scheduling software. The prototype demonstrates that the distributed data-flow approach is more efficient than the centralized approach when integrating large engineering software services.

[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 [11]. Under this paradigm, software components are linked together through an integration framework to form composed software applications called megaservices [17]. Software components are provided as processes managed by independent service providers. The components have clearly defined functions with accessible interfaces. We call these software components 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.

Prior research has addressed the issue of composing megaservices based on autonomous services [16]. A megaservice acts as a central controller for invoking, monitoring, querying, and terminating the autonomous services. Autonomous services take turns to process the data supplied by the megaservice and return the processed results to the megaservice. Data are exchanged using a client-server model where the megaservice serves as the central hub of all data traffic. This *centralized data-flow approach* is used in many current software integration frameworks such as CORBA [13], J2EE [4], and Microsoft .NET [10].

This paper demonstrates that the centralized data-flow approach is inefficient for integrating large-scale engineering software services. A *distributed data-flow approach* is proposed to allow data to be exchanged directly among the services. This paper presents the Flow-based Infrastructure for Composing Autonomous Services (FICAS). FICAS is implemented as a collection of software modules that support the construction of autonomous services, facilitate the functional composition of autonomous services into megaservices, and conduct the execution of megaservices. There are three objectives in designing FICAS: (1) Scalability – integration and management of 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 aims to utilize the distributed data-flow approach to achieve better scalability and performance without sacrificing ease of composition.

The paper is organized as follows. Section 2 outlines the key issues in building service composition infrastructures and gives an overview of FICAS. Section 3 defines a metamodel to enable homogeneous access for autonomous services within FICAS. Section 4 describes the runtime environment of FICAS, focusing on distributing data communications among the services. Section 5 explores techniques in distributing computations among autonomous services. Section 6 illustrates a prototype for a ubiquitous computing environment based on FICAS. Section 7 summarizes our findings.

2 Service Composition Infrastructures

As software becomes more complex, there is a paradigm shift from coding as the focus of programming to a focus on software composition. Traditionally, large programs are partitioned into subtasks of manageable sizes. The subtasks are assigned to programmers who code the instructions in a programming language. The resulting subtasks are subsequently submitted for integration. Software composition, on the other hand, starts from an existing software base. Pre-existing software applications are wrapped into autonomous services, whose functionalities are then composed together. Many of the services may be distributed over the network and heterogeneous in nature.

2.1 Integration of Software Components

Software integration takes place in many forms. Early methods are based on code reuse. The simplest method is to copy the source code to wherever the desired functionality is needed. There are significant drawbacks to this approach, ranging from compiler incompatibility to difficulties in maintaining duplicate copies of program code. To deal with these drawbacks, software components written in a programming language are compiled into shared libraries. The libraries have public interfaces, through which the users invoke the functions contained in the libraries. The software components are executed on a single machine. The ownership of the reused software components belongs to the users of the software components.

The development of network computing allows software components to be distributed to multiple machines. Each software component runs as a separate process, communicating with each other by exchanging messages. With the proliferation of web services [9], more and more software components are provided in the form of autonomous services, each managed autonomously by its own providers [14]. Figure 1 illustrates an integration environment that consists of autonomous services connected by a communication network. Each autonomous service has four hierarchical layers:

- The "Host" layer represents the hardware platform the autonomous service runs on. This layer provides the hardware means for executing application instructions and routing data through the communication network.
- The "Operating System" layer provides software support for the system resource. It manages the processes of the software applications that perform the service. It also provides protocol support for the network intercommunications among different hardware platforms. For instance, the TCP/IP [7] protocol support belongs to this layer.
- The "Access Protocol" layer provides protocol support for accessing the data and the functionalities of the autonomous service. The access protocol defines how to encode a service request, and also specifies the manner in which the autonomous service responds to the request. The layer provides a level of abstraction to enable a service client to communicate with an autonomous service in a different operating system.
- The "Autonomous Service" layer is the application layer, which is concerned with the semantics of the autonomous service. Data integration is conducted at this layer so that autonomous services can exchange information in a mutually understandable fashion.

A megaservice is a conceptual composition of the functionalities exported by the autonomous services through the "Autonomous Service" layer. The execution of the megaservice is coordinated by a controller, which itself may be an autonomous service. The service integration environment can be conceptually viewed as a set of service nodes interconnected by a communication network. There are two types of messages: control messages and data messages, distinguished by their use at the message destinations. Control messages are mostly short messages that trigger state changes at the receiving services. Examples of control messages include service invocation requests and status polling requests. Data messages are mostly large data packets that are given to the receiving services for processing. Examples of data messages include engineering design, manufacturing and resource information to conduct simulation. To execute a megaservice, control and data messages need to be exchanged among autonomous services.

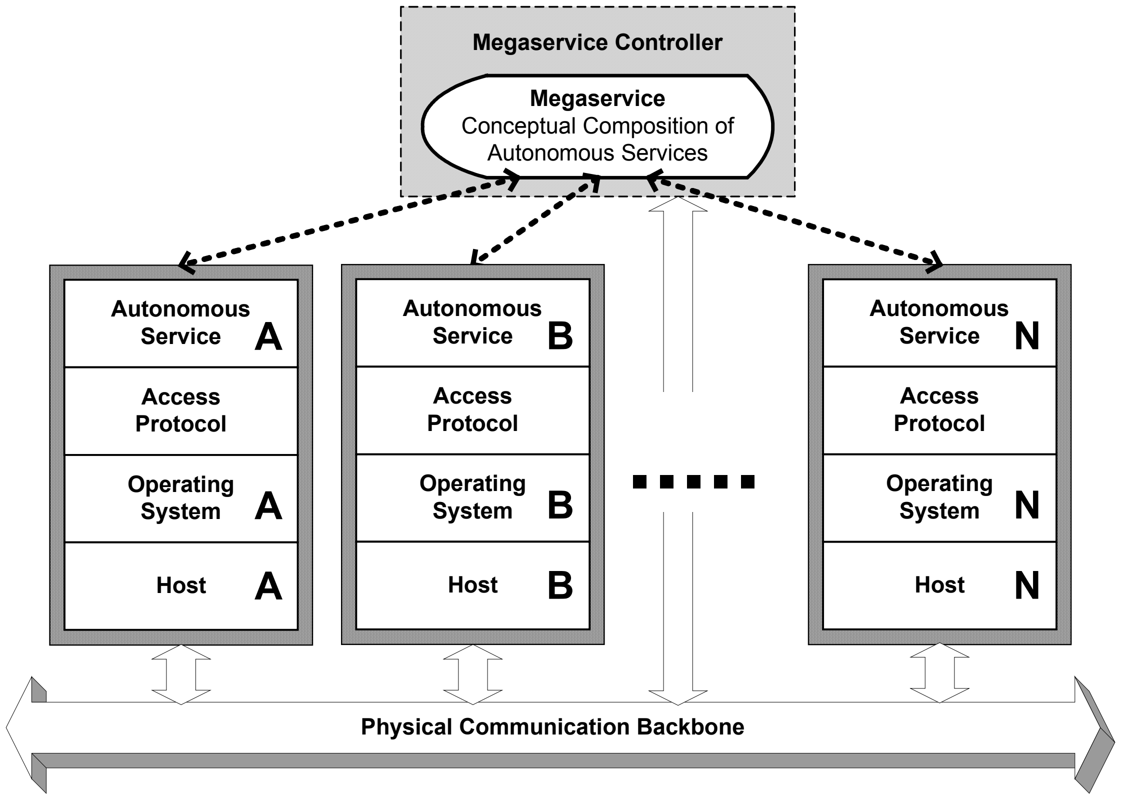


Figure 1: Hierarchical Model of Autonomous Services

2.2 Distribution of Data-flows

A control-flow is formed by a group of related and partially ordered control messages, and a data-flow is formed by a group of related and partially ordered data messages. Service integration environments differ in how control-flows and data-flows are formed and managed.

Traditionally, both control-flows and data-flows are centrally coordinated, as illustrated in Figure 2(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*, or *1C1D model*. The 1C1D model represents the simplest form of service composition runtime environment. Examples of the 1C1D model include CORBA [13], J2EE [4], and Microsoft .NET architecture [10].

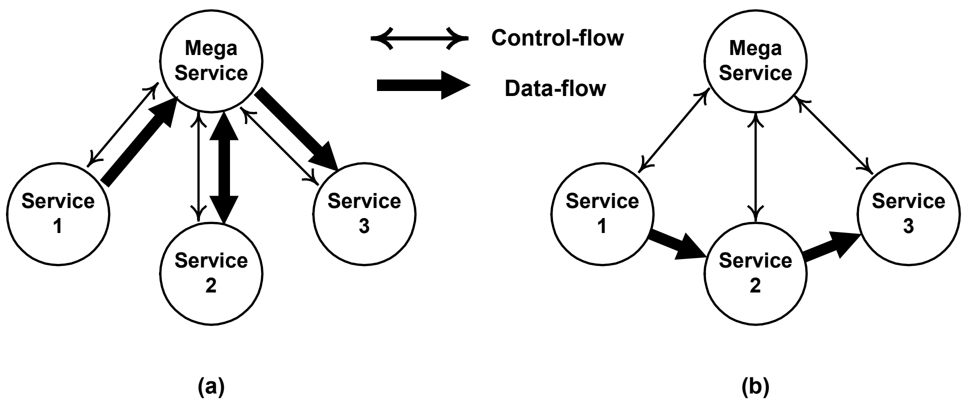


Figure 2: Centralized and Distributed Data-flows

There are performance and scalability issues associated with the 1C1D model. The megaservice forwards data between the two autonomous services when the data produced by one service is utilized by another service. Since the data is sent indirectly, redundant data traffic is resulted. The megaservice control becomes a communication bottleneck when large amount of data are exchanged among the services. Furthermore, since all data traffic go 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 are likely to be of limited bandwidth. The centralized communication topology makes the 1C1D model difficult to scale.

The issues observed in the 1C1D model motivate us to distribute the data-flows for the executions of megaservices. Figure 2(b) shows the control-flows and the data-flows exhibited in a distributed data-flow infrastructure. The megaservice has the ability to 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.

In this paper, we discuss how FICAS distributes data-flows while maintaining the same centralized control mechanism as in the 1C1D model. We call this runtime model the *centralized control-flow distributed data-flow model*, or *1CnD model*. The decision to retain a centralized control-flow is due to its ease of implementation and management. We find it difficult to effectively apply distributed control-flow models to conduct service composition. Because operational code segments would have to be distributed to relevant function units for execution, distributed control-flows require homogeneity in the underlying hardware platform. 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 redundant data traffic caused by the forwarding of data through the megaservice. The distributed data-flow model also utilizes the communication network among the autonomous service, and thus alleviates the communication load on the megaservice. Furthermore, FICAS allows computations to be efficiently distributed to where data resides, so that the data can be processed without incurring communication traffic.

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 [3, 16], 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 its compositional language supports the same 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, we have defined the CLAS (*Compositional Language for Autonomous Services*) 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.

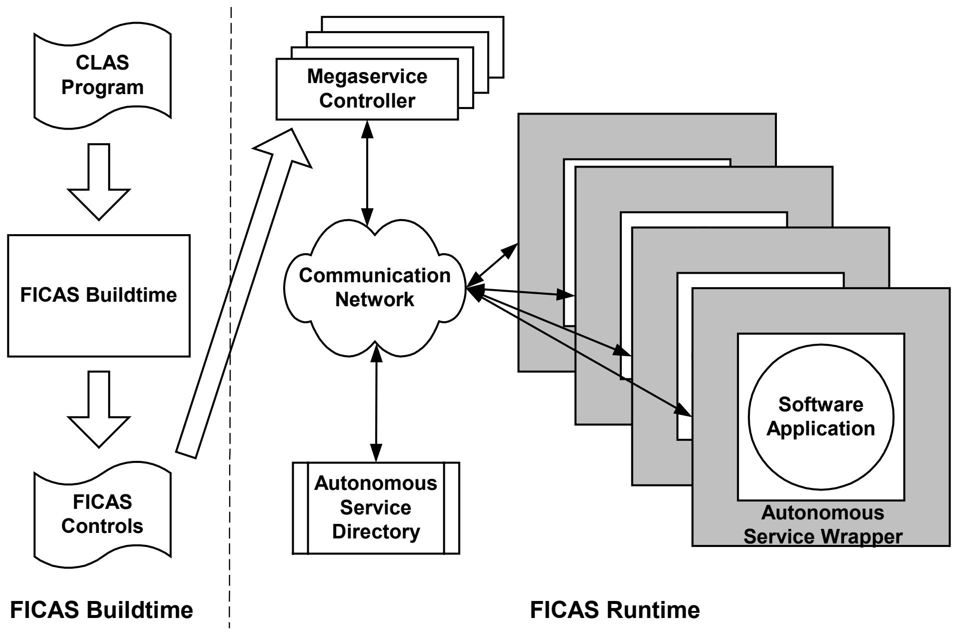


Figure 3: FICAS Architecture

The runtime environment of FICAS is responsible for executing the 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 by simply plugging additional autonomous services into the communication network and registering the autonomous services with the service directory. The autonomous service directory keeps track of available autonomous services within the infrastructure. While the directory is viewed globally as a centralized entity, it may be implemented as a distributed structure.

Autonomous services are formed by wrapping software applications. A metamodel is defined to allow the construction of homogeneous autonomous services in a heterogeneous computing environment. The key feature of the FICAS metamodel is the separation of the data-flows from the control-flows. 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 to coordinate the control-flows among the respective autonomous services. The controller is also responsible for optimizing the performance of the megaservice.

3 Autonomous Services

Autonomous services are running processes that involve one or more software applications along with the domain data they operate on. The client-server interaction model is used. The clients of the autonomous services make service requests, as autonomous services wait for service requests. To fulfill the service requests, autonomous services invoke processing routines to operate on their domain data. Autonomous services may also request information from other autonomous services. As the result of the services, desired information is returned to the clients.

3.1 Autonomous Service Metamodel

Autonomous services are specified under a homogeneous model in order to communicate and cooperate with each other. Figure 4 illustrates the autonomous service metamodel in FICAS. An autonomous service 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 the result data elements. We can usually wrap existing software applications into a service core.
- Events are exchanged between services to control the flow of autonomous service executions. Asynchronicity of autonomous service execution is achieved by using queues for event processing. Incoming events are placed at the tail of the input event queue, and outgoing events are placed at the tail of the output event queue. The default queuing system used in FICAS is the FIFO (first in and first out) queue, where events are processed in the order by which they are received.
- The data containers are groupings of input and output data elements for the autonomous service. The 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. The existence of data containers is essential for the distribution of data-flows. Under the 1CnD model, the data-flows can be formed between data containers of two autonomous services, while control-flows continue to go through the megaservice controller.

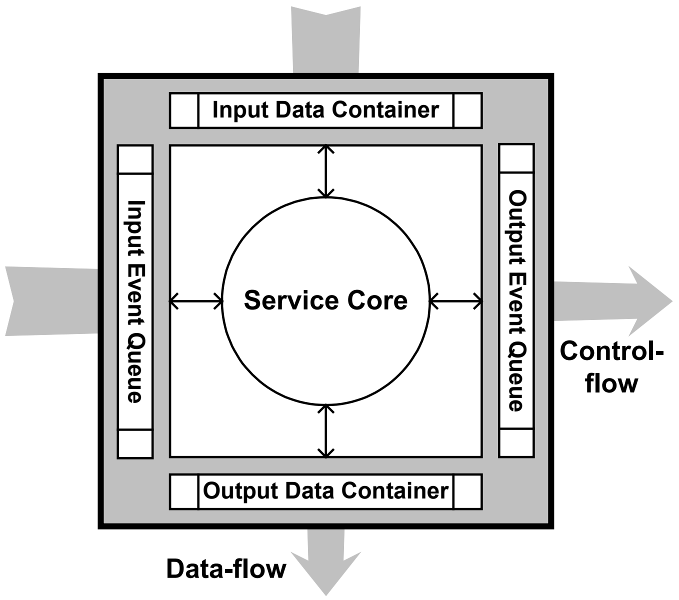


Figure 4: FICAS Autonomous Service Metamodel

The key characteristic of the FICAS autonomous service model is the explicit separation of control-flow and data-flow. For control-flow, the autonomous service primarily concerns about the event processing and the state management of the service core. For data-flow, the autonomous service primarily concerns about the exchange of data elements between the data containers and the processing of the data elements by the service core. The control-flows and the data-flows are managed by asynchronous components of the autonomous service. While each component uses its own thread, the service core ties together the components into a coordinated entity.

3.2 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 based messages that are used to interact with autonomous services. The hierarchical structure of XML provides a convenient method of defining the composition of an event. ASAP is asynchronous and non-blocking. The sender of an event may not wait for the response of the event. Instead, the sender can continue to execute other activities that are not dependent on the response of the event. The protocol removes the barriers imposed by different megaservice programming languages and distribution protocols. For simplicity, we represent the ASAP events using their abbreviated functional representations instead of their full XML representations. The key ASAP events that related to data-flow scheduling are listed below:

- SETUP (Service)

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

- TERMINATE (Service)

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

- INVOKE (Service)

The INVOKE event is used to request an autonomous service. The service core of the autonomous service is started upon the processing of the INVOKE event. After the completion of the service invocation, output data elements are generated by the service core and are placed onto the output data container. In addition, a reply event is issued.

- 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 case the event is sent to the *DestinationService*. The *DestinationService* pulls the data element from the *SourceService* and put the data element onto its input data container. Both implementations are supported by FICAS.

3.3 Autonomous Service Wrapper

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 module. We call such a module *autonomous service wrapper*. The wrapper provides the support for the ASAP protocol, and facilitates the encapsulation of software applications into autonomous services.

In FICAS, the autonomous service wrapper has been implemented in Java. The Java classes and interfaces are incorporated into a Java library. 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 5. 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 autonomous service wrapper fills in the values for these parameters when it actives the connector. 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 generates the results into the output data container.

```
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 5: ServcieCore Interface

4 Distributed Data-flow Planning

In FICAS, the megaservice controller has the sole responsibility for managing the control-flows for a megaservice. The controller executes and coordinates autonomous services by controlling the choice and the timing of ASAP events. We characterize the coordination as an execution plan, which defines the choice, timing, sequence, and dependencies of the outgoing ASAP events.

4.1 Planning 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 can build an execution plan for the megaservice.

The megaservice program segment in Figure 6 shows implicit data dependencies between autonomous services. For instance, 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 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. The dependencies are mapped into a data dependency graph (DDG) as shown in Figure 7. 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 the 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 into the INVOKE event nodes in the EDG. The mapping from the directed arcs in the DDG to the event nodes in the EDG is more complex. Different mapping schemes can produce different data-flow models for the megaservice.

```
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 6: Sample Megaservice Program Segment

Figure 8 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 into 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 7) is mapped into the *MAPDATA(A, Service1, Service3)* event node in the EDG (shown in Figure 8).

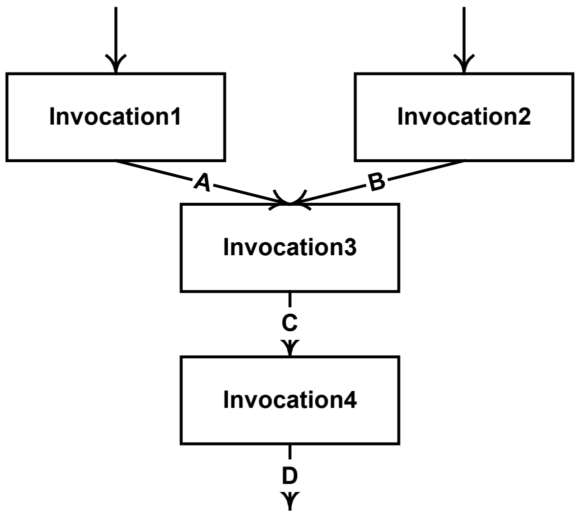


Figure 7: Sample DDG

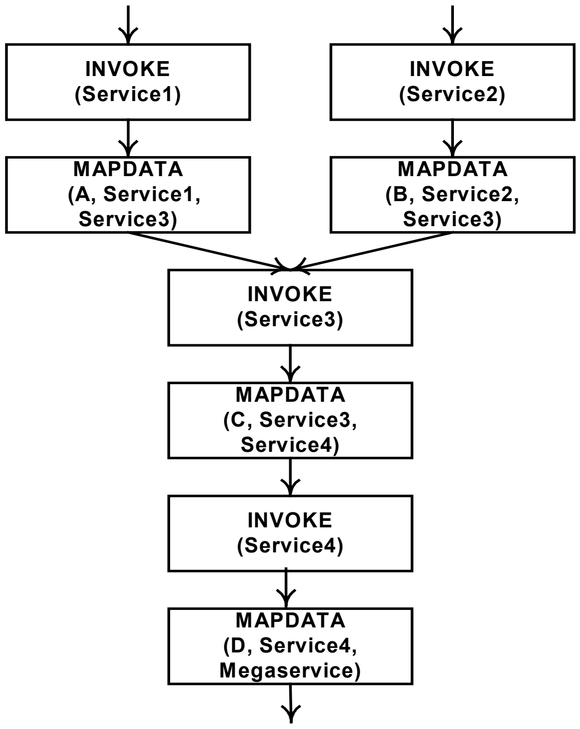

INVOKE (Service1)
INVOKE (Service2)
MAPDATA (A, Service1, Service3)
MAPDATA (B, Service2, Service3)
INVOKE (Service3)
MAPDATA (C, Service3, Service4)
INVOKE (Service4)
MAPDATA (D, Service4, Megaservice)

Figure 8: EDG with Distributed Data-flows

4.2 Performance Analysis

In this section, we measure the performance of a sample megaservice supported by FICAS. Different configurations of the computing environment are used to examine the performance of the megaservice. We also compare FICAS with the centralized data-flow model by implementing the same megaservice under SOAP [5]. 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 [8]. As one of the popular choices for the SOAP implementations, Apache SOAP [1] is selected to be the reference implementation.

Figure 9 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. All Java programs are written and compiled with Sun's JDK 1.3.0 for the Microsoft Windows operating system. For SOAP, the autonomous services are implemented as Java methods whose interfaces are registered with the Apache Tomcat application server v4.0. The megaservices are implemented as Java applications that invoke the services using the Apache SOAP v2.2 API 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.

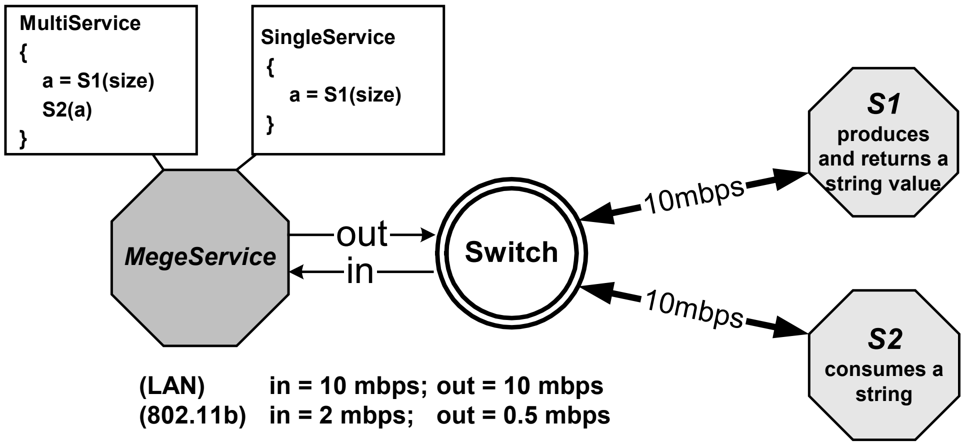


Figure 9: Test Environment for Comparing SOAP and FICAS

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 10 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 by 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 overheads in initializing and terminating the autonomous services. Although FICAS achieves performance gains by distributing the data-flows, the gains are not enough to offset the extra overheads in the control-flows. Second, it is expected that Apache SOAP, being under development for quite some time, is better optimized than FICAS in terms of its Java source codes.

- 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.
- The execution times of the megaservices increase linearly with respect to the data volume. Since there is no computational processing on either the autonomous services or the megaservices, the increase in execution times comes from the increased data-flows. The execution times approximately double each time the data volume doubles.
- FICAS outperforms SOAP when the data volume is high. The larger the data volume, the bigger is the difference between the execution time of the FICAS megaservice *MultiService* and that of the SOAP megaservice *MultiService*. This is because the SOAP megaservice incurs twice as much data-flows 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*.

To summarize, Apache SOAP and FICAS are similar in many aspects, while their most significant difference is in how they deal with data-flows. Apache SOAP incurs the centralized data-flows, and FICAS distributes the data-flows among the autonomous services. When the data volume is low, Apache SOAP outperforms FICAS since Apache SOAP has simpler control-flows. When data volume is high, FICAS outperforms SOAP by taking advantage of the data-flow distribution.

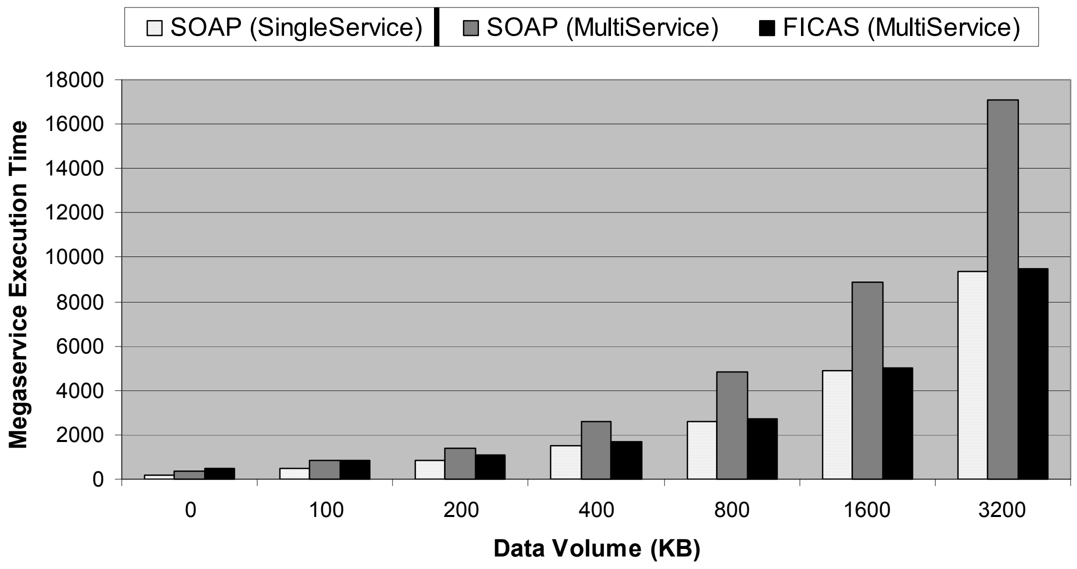


Figure 10: Comparison Between FICAS and SOAP on Local Area Network

Figure 11 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. The communications with the megaservice have much lower bandwidth than the communications among the autonomous services. Comparing the megaservice performance between the LAN and the wireless 802.11b settings, we observe the following:

- 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. Hence, when deploying a SOAP service composition infrastructure, it is important to ensure the high quality of the network connections between the megaservice and the autonomous services
- 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.

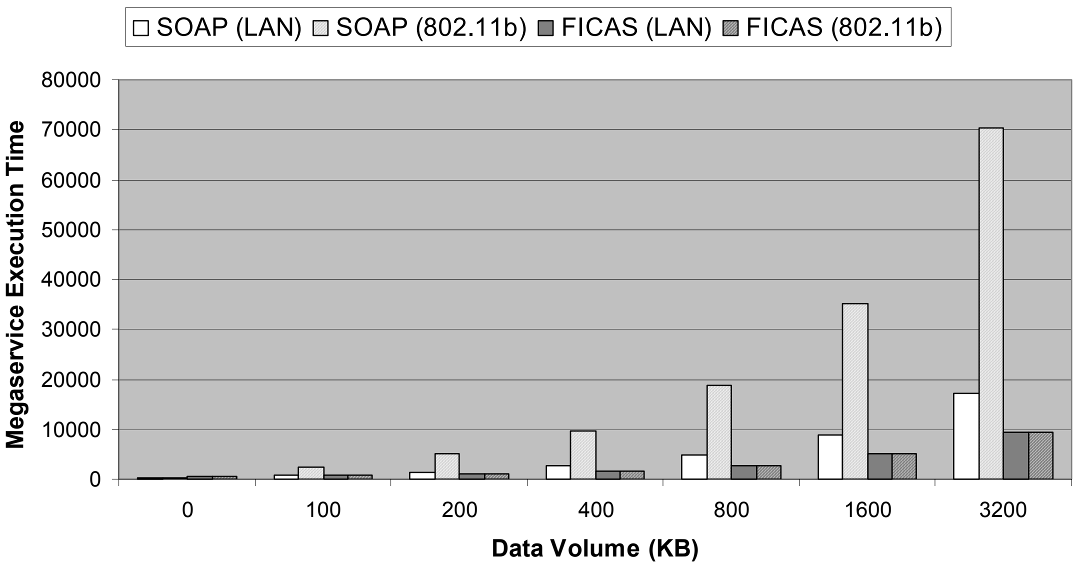


Figure 11: Megaservice Performance Under Different Network Configurations

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

5 Mobile Classes and Active Mediation

The distribution of data allows computations to be distributed to minimize data traffic. We describe in this section how distributed computation is enabled by the mobile class. We define an architecture that supports the execution of mobile classes. We also outline an algorithm that determines the optimal location to carry out the execution of a mobile class.

5.1 Mobile Classes

A mobile class is an information-processing module that can be dynamically loaded. Conceptually, the mobile class is a function that takes some input data elements, performs certain operations, and then outputs a new data element. For instance, $y = f(x_1, x_2, x_3)$, represents a mobile class named *f* that takes three data elements as input and produces an output *y*.

Java is chosen as the specification language for mobile classes in FICAS. Such selection is made for a few reasons. First, Java is a general programming language that 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 senses. 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.

All mobile classes need to implement a common interface named *MobileClass*, whose interface is shown in Figure 12. The interface contains a single function that represents the functionality of a mobile class. The *execute()* function takes a vector of data elements as the input and generates a data element as the output. The *execute()* function is overloaded by a mobile class to provide specific processing functionality. Once coded, a mobile class is compiled into a Java class and put into the mobile class repository. The Java class will be looked up later when the mobile class is invoked by a megaservice.

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

Figure 12: Sample Megaservice Program Segment

Mobile classes enable megaservices to perform computations with greater efficiency. Figure 13 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 [12]. The type brokers can convert data in unknown formats to known formats. A type graph is used to figure out the chain of necessary conversions. An example of automating this process can be seen in [6]. Figure 13(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. Potentially large amount of data are 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, similar to the network of the type brokers. As shown in Figure 13(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 the data traffic is limited to essential transmissions.

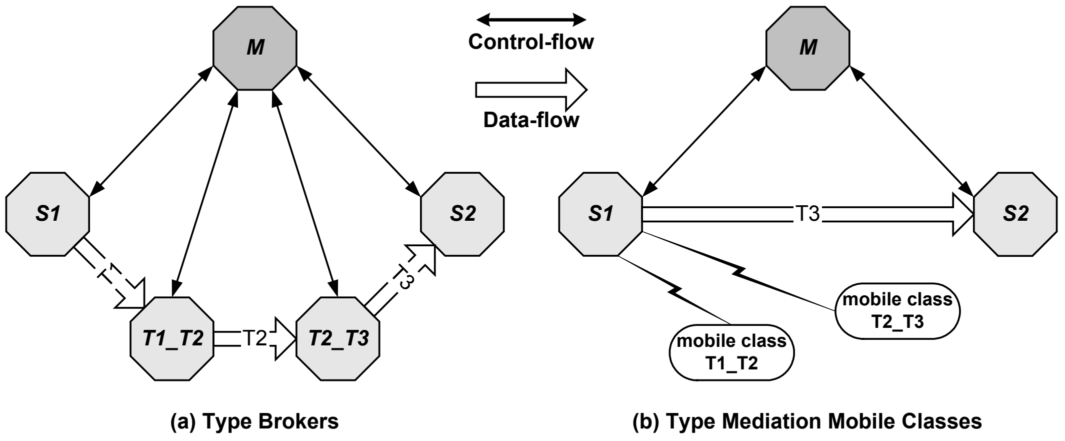


Figure 13: Type Conversion Using Type Broker Services and Mobile Classes

5.2 Active Mediation

Active mediator is the 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 14 illustrates the architecture of an active mediator:

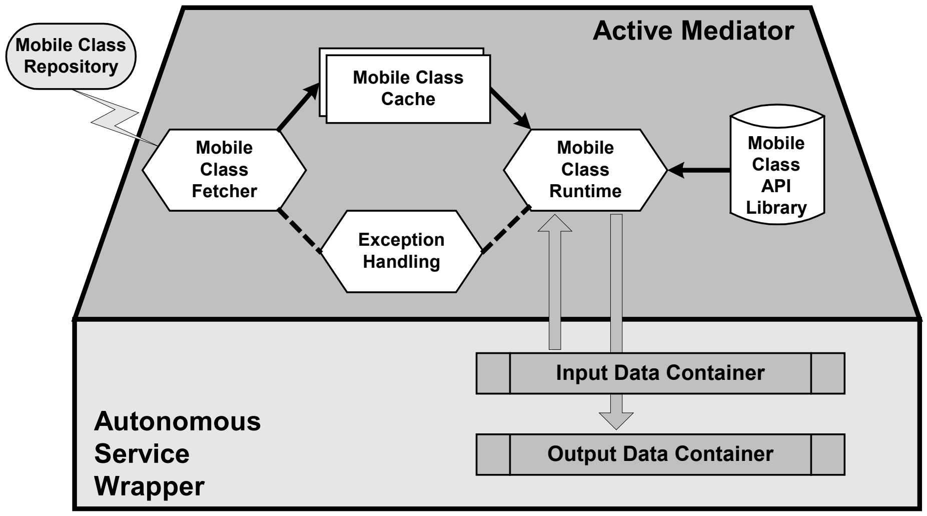


Figure 14: Active Mediation Architecture

- The Mobile Class Fetcher is responsible for loading the Java class of the mobile class. The name of the mobile class indicates where the Java class file can be found. If the name of the mobile class starts with "http://", then the URL for loading the Java byte codes can be obtained by appending ".class" to the name of the mobile class. For example, the Java class file for the mobile class "*http://mobile.class.repository/int2float*" can be found at "*http://mobile.class.repository/int2float.class*". If the name of the mobile class is a normal string, then the URL can be obtained by prefixing a base URL and appending ".class" to the name. For example, if the base class path for a megaservice is "*http://mobile.class.repository*", then the Java class file for the mobile class "*int2float*" can be located at "*http://mobile.class.repository/int2float.class*".
- 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 the cache miss occurs, the Mobile Class Fetcher is used.
- The Mobile Class API Library stores the utility classes that make the construction of mobile classes more convenient. For instance, the Java Development Kit library [2] 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.

5.3 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 15, is used to demonstrate such 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 9. 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, we have three potential placement strategies, as shown in Figure 16:

- <u>Strategy 1</u>: 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 16(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.

- <u>Strategy 2</u>: By placing the mobile class *FILTER* at *S1*, we can construct the execution plan as shown in Figure 16(b). *S1* generates the data element *A* and processes it locally using the mobile class. The result *B* is sent from *S1* to *S2* for further processing.

- <u>Strategy 3</u>: By placing the mobile class *FILTER* at *S2*, we can construct the execution plan as shown in Figure 16(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, we assume 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 content sent between the autonomous services. For Strategy 2, the data element *B* is sent from *S1* to *S2*. For Strategy 3, the data element *A* is sent from *S1* to *S2*. Since the data element *B* is one tenth in size compared to the data element *A*, Strategy 2 incurs the least amount of communication traffic. Therefore, Strategy 2 is the placement strategy that has the best performance.

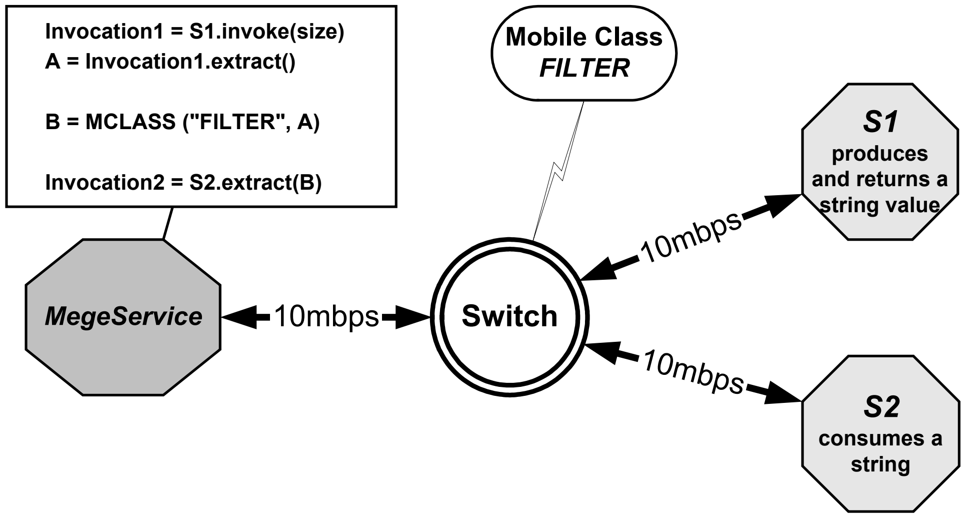

Invocation1 = S1.invoke(size)
A = Invocation1.extract()
B = MCLASS ("FILTER", A)
Invocation2 = S2.extract(B)
Mobile Class FILTER
S1 produces and returns a string value
MegeService
10mbps
Switch
10mbps
10mbps
S2 consumes a string

Figure 15: Example Megaservice that Utilizes the Mobile Class *FILTER*

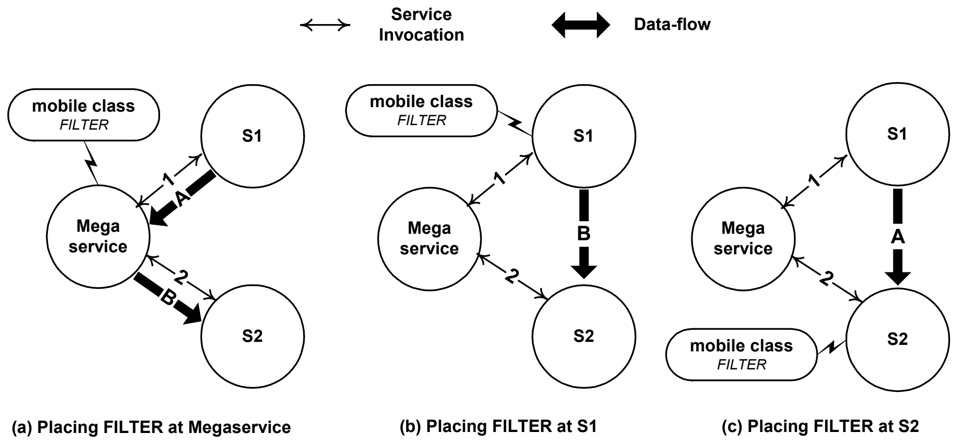


Figure 16: Execution Plans with Different Placements for the Mobile Class

The optimal placement of a mobile class should 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 *i*th 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, by placing the mobile class on the autonomous service S_i, we 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 17 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.

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

Figure 17: LDS Algorithm for Optimal Mobile Class Placement

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, we need to estimate the output data size. We view the output data size of an mobile class as a function on 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 codes in the mobile class repository. The megaservice controller can then use the sizing function to estimate the mobile class output data size for running the LDS algorithm.

5.4 Performance Analysis

We now analyze the performance of the megaservice previously defined in Figure 15. The megaservice is executed using different placements of the mobile class *FILTER*. We intend to measure the impact of the placement of the mobile class on the performance of the megaservice. In addition, we replace the mobile class *FILTER* with an autonomous service that implements the same functionality. The performance of the megaservice utilizing the autonomous service is compared with the megaservice utilizing the mobile class. We consider the following scenarios:

- Strategy 1: The megaservice conducts active mediation on *S1* by executing the mobile class *FILTER* on *S1*. The placement of the mobile class is generated by the LDS algorithm.
- Strategy 2: The megaservice conducts active mediation on *S2* by executing the mobile class *FILTER* on *S2*.
- Strategy 3: We implement a utility autonomous service that replaces the mobile class *FILTER*. The string generated by *S1* is fed into the autonomous service, and the result is forwarded onto *S2* for further processing.

Figure 18 shows the execution times of the megaservice. Different settings on the size of the string generated by *S1* are used. The following observations are made:

- The execution times of the megaservices increase with the size of the string. Three factors contribute to the increased execution times. First, longer time is taken to measure the size of the string. It results in the longer execution time for the LDS algorithm. Second, it takes longer to execute the mobile class or the utility autonomous service. Third, the larger string results in longer transmission time for the data elements.
- The placement of the mobile class significantly impacts the performance of the megaservice. Strategy 1 performs significantly better than Strategy 2. Strategy 1 utilizes the LDS algorithm to minimize the amount of data-flows incurred by the megaservice. In Strategy 2, *S1* transmits the original string to *S2*. Whereas in Strategy 1, *S1* only transmits the filtered string to *S2*. Strategy 1 causes significantly less amount of data traffic than Strategy 2.

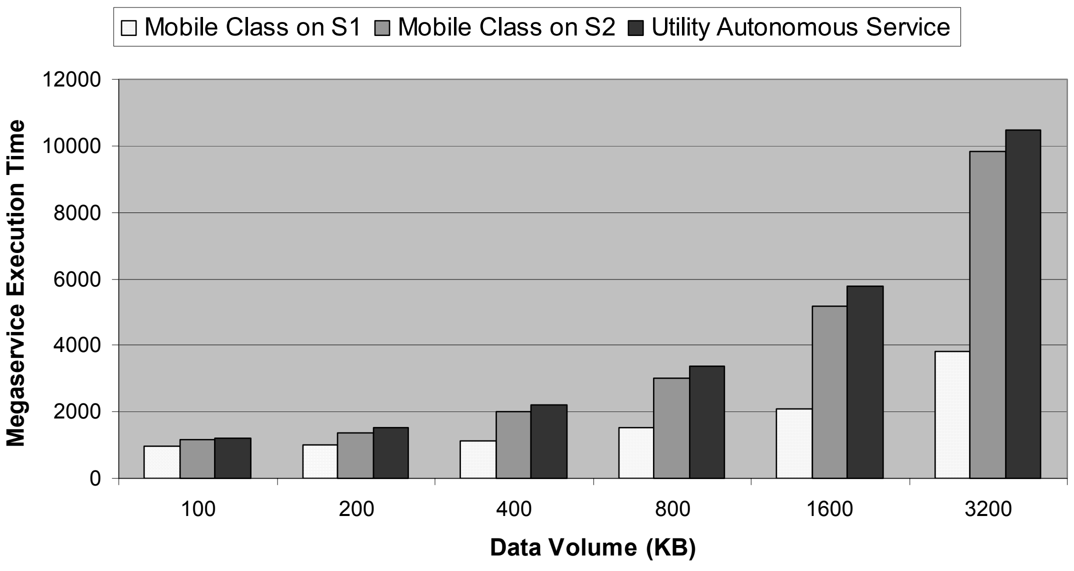


Figure 18: Comparison Between Mobile Class and Autonomous Service

- Both strategies involving the mobile class perform better than Strategy 3, which uses the utility autonomous service. Strategy 3 incurs the most amount of data-flows, as both the original string and the filtered string are transmitted among the autonomous services. In addition, the invocation of the autonomous service is more costly than the invocation of the mobile class.

In summary, active mediation enabled by the mobile class is an effective approach in improving the performance of the megaservice. The mobile class can be placed onto the appropriate autonomous service to minimize the amount of data communications.

6 Information Service Infrastructure

We have shown through simple examples that FICAS is well suited for composing autonomous services that exchange large amount 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, we implement an engineering service infrastructure for construction project management applications. We illustrate the process of building the service infrastructure by: (1) wrapping software applications into autonomous services, (2) implementing mobile classes, and (3) constructing megaservices to accomplish the engineering tasks.

6.1 Building Autonomous Services

The first step in building the engineering service infrastructure is to wrap each software application into an autonomous service. We create the service core of the autonomous service by defining the *ServiceCore* interface based on the software application. The service core is then linked to an autonomous service wrapper (ASW). Figure 19 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.

6.2 Constructing Mobile Classes

Lightweight information processing routines are specified as mobile classes, whose executions are determined by megaservices during the runtime. Figure 20 shows a sample mobile class that converts data from Process Specification Language (PSL) format [15] into Microsoft Excel format. The *psltoexcel* class implements the *MobileClass* interface, whose definition is provided in the *FICAS.zip* class library. The *execute()* function take the first argument for the mobile class as the input data in PSL, convert the data into Microsoft Excel format, and return the converted data as the output data element.

In our 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 19: Example Autonomous Service that Utilizes Primavera P3

6.3 A Sample Megaservice

Figure 21 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, which is accessible on the web at *http://ficas.stanford.edu/CLASParser/SchedulingDemo.xml*. 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*.

We now look at a sample scenario to demonstrate how the engineering service infrastructure helps facilitate personnel from different functional groups conduct collaborations. We use the model of the Mortenson Ceiling project (part of the construction of the Disney Concert Hall) as the test case. Figure 22 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 is particularly important for the on-site personnel of the construction project. Suppose that the duration for the activity 18T1-33201, for erecting a roof element, is changed from 1 day to 40 days, as shown in Figure 23. The change can be made remotely using the Palm device. The update will 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 24. The updated schedule can also be retrieved from the relational database using MS Project. Figure 25 shows that not only the activity 18T1-33201 is updated, but the dependent activities are also updated as well.

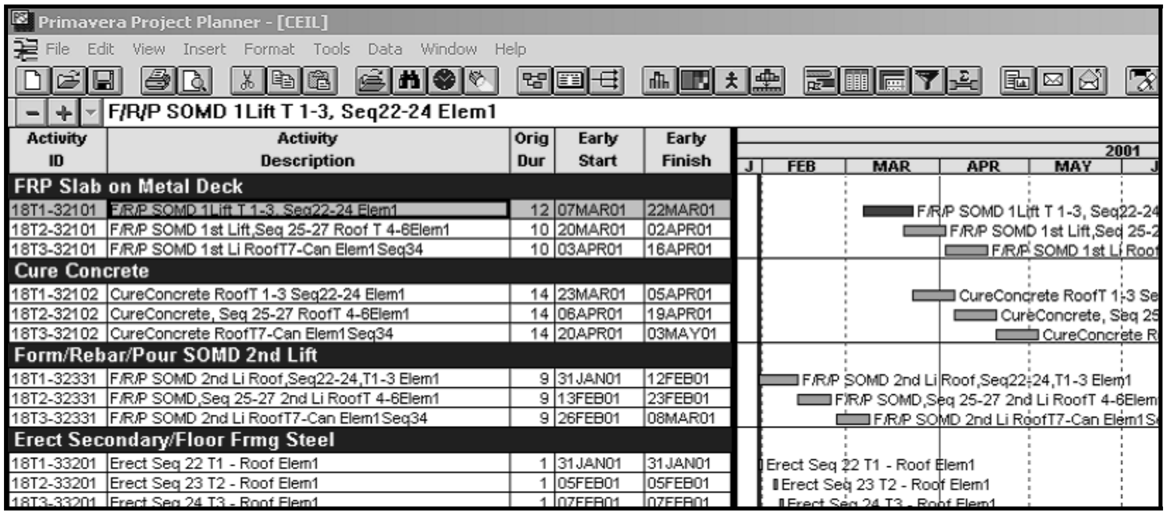
The example infrastructure involves software applications that exchange large amount of data. The applications are conveniently wrapped into autonomous services. Computational tasks are easily specified using mobile classes. Engineering processes are systematically defined as megaservices. Our example demonstrates that FICAS model is suitable for the composition of large-scale autonomous services.

```
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 20: Example Mobile Class that Converts Data from PSL to Microsoft Excel

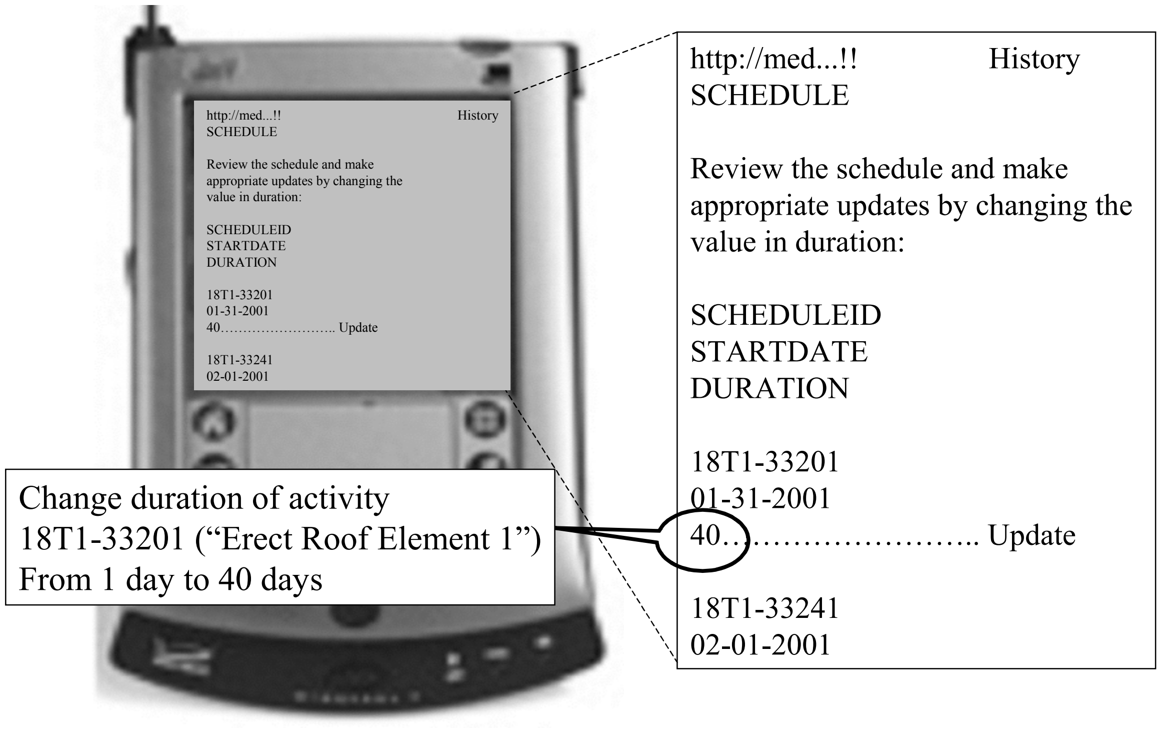
```
SchedulingDemo "http://ficas.stanford.edu/mcrepo"
{
  psl_svc = SETUP("PSLService")
  p3_svc = SETUP("P3Service")
  excel_svc = SETUP("ExcelService")

  /* 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 21: Sample Megaservice Specified in CLAS


Primavera Project Planner - [CEIL]
File Edit View Insert Format Tools Data Window Help
F/R/P SOMD 1Lift T 1-3, Seq22-24 Elem1
2001
J FEB MAR APR MAY J

Activity ID	Activity Description	Orig Dur	Early Start	Early Finish
FRP Slab on Metal Deck				
18T1-32101	F/R/P SOMD 1Lift T 1-3, Seq22-24 Elem1	12	07MAR01	22MAR01
18T2-32101	F/R/P SOMD 1st Lift,Seq 25-27 Roof T 4-6Elem1	10	20MAR01	02APR01
18T3-32101	F/R/P SOMD 1st Li RoofT7-Can Elem1Seq34	10	03APR01	16APR01
Cure Concrete				
18T1-32102	CureConcrete RoofT 1-3 Seq22-24 Elem1	14	23MAR01	05APR01
18T2-32102	CureConcrete, Seq 25-27 RoofT 4-6Elem1	14	06APR01	19APR01
18T3-32102	CureConcrete RoofT7-Can Elem1Seq34	14	20APR01	03MAY01
Form/Rebar/Pour SOMD 2nd Lift				
18T1-32331	F/R/P SOMD 2nd Li Roof,Seq22-24,T1-3 Elem1	9	31JAN01	12FEB01
18T2-32331	F/R/P SOMD,Seq 25-27 2nd Li RoofT 4-6Elem1	9	13FEB01	23FEB01
18T3-32331	F/R/P SOMD 2nd Li RoofT7-Can Elem1Seq34	9	26FEB01	08MAR01
Erect Secondary/Floor Frmg Steel				
18T1-33201	Erect Seq 22 T1 - Roof Elem1	1	31JAN01	31JAN01
18T2-33201	Erect Seq 23 T2 - Roof Elem1	1	05FEB01	05FEB01
18T3-33201	Erect Seq 24 T3 - Roof Elem1	1	07FEB01	07FEB01

Figure 22: Reviewing the Project Schedule in Primavera P3



http://med...!! History
SCHEDULE

Review the schedule and make appropriate updates by changing the value in duration:

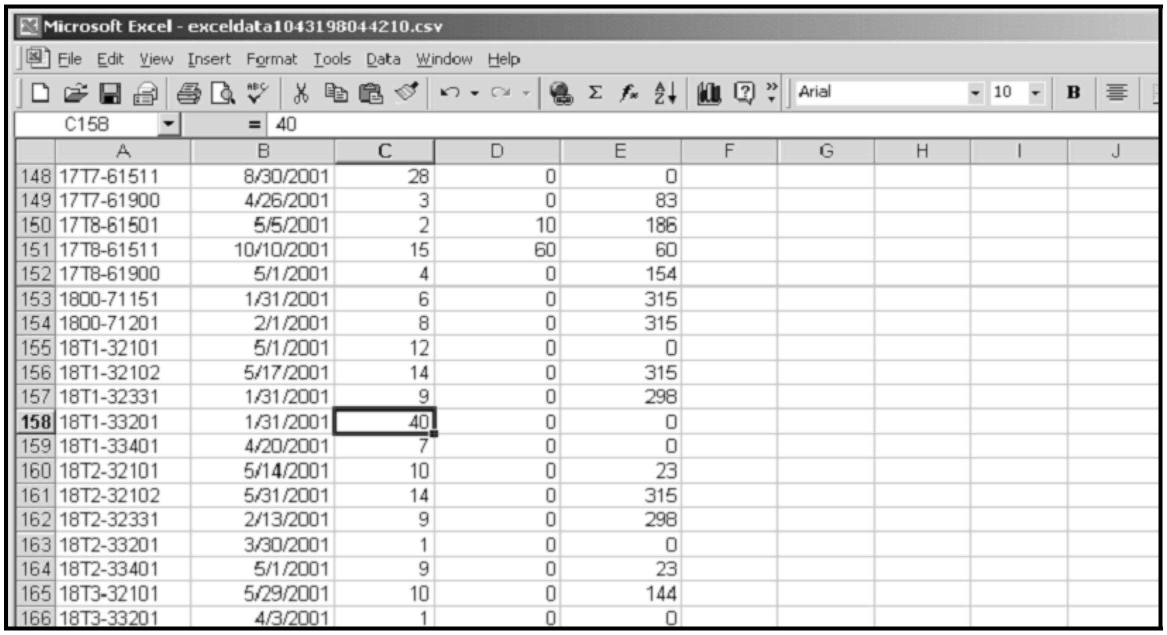
SCHEDULEID
STARTDATE
DURATION

18T1-33201
01-31-2001
40…………………….. Update

18T1-33241
02-01-2001

Change duration of activity 18T1-33201 (“Erect Roof Element 1”) From 1 day to 40 days

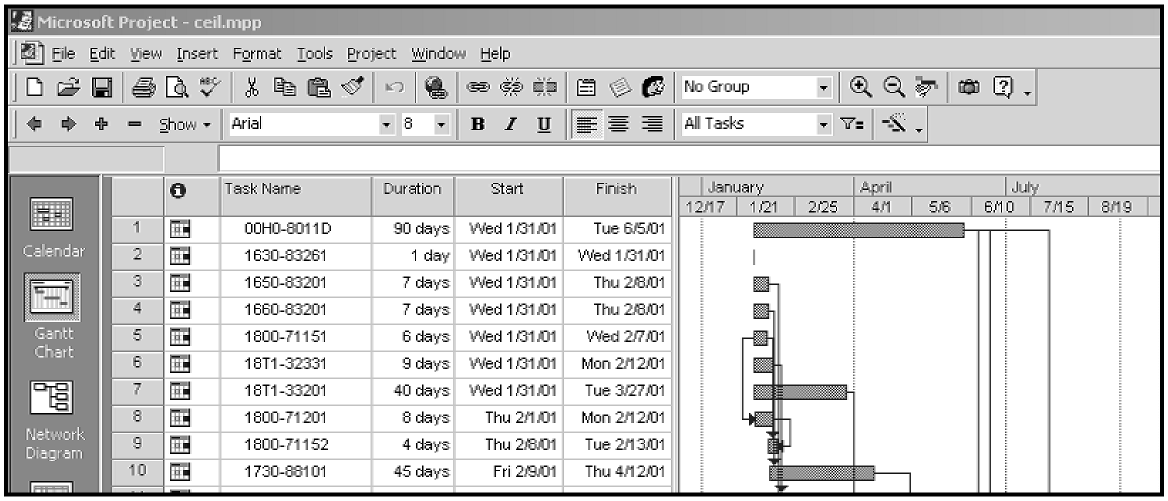
Figure 23: Revising the Project Schedule via a Palm Device


Microsoft Excel - exceldata1043198044210.csv
File Edit View Insert Format Tools Data Window Help
Arial
10

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 24: Reviewing the Updated Project Schedule in Microsoft Excel



	Task Name	Duration	Start	Finish
1	00H0-8011D	90 days	Wed 1/31/01	Tue 6/5/01
2	1630-83261	1 day	Wed 1/31/01	Wed 1/31/01
3	1650-83201	7 days	Wed 1/31/01	Thu 2/8/01
4	1660-83201	7 days	Wed 1/31/01	Thu 2/8/01
5	1800-71151	6 days	Wed 1/31/01	Wed 2/7/01
6	18T1-32331	9 days	Wed 1/31/01	Mon 2/12/01
7	18T1-33201	40 days	Wed 1/31/01	Tue 3/27/01
8	1800-71201	8 days	Thu 2/1/01	Mon 2/12/01
9	1800-71152	4 days	Thu 2/8/01	Tue 2/13/01
10	1730-88101	45 days	Fri 2/9/01	Thu 4/12/01

Figure 25: Reviewing the Updated Schedule in Microsoft Project

7 Summary

This paper investigates the integration of services that communicate large volumes of data. Traditionally, a megaservice is the central hub for all the data traffic, while the autonomous services process the data supplied by the megaservice and return the processed results to the megaservice. This centralized data-flow approach is shown to be inefficient. To improve performance, the distributed data-flow approach is introduced to allow direct data exchange among the autonomous services. The distribution of data also enables computations to be more effectively distributed.

FICAS is a service composition infrastructure that utilizes the distributed data-flow approach. We define 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. Autonomous services conforming to the ASAP protocol can be coordinated by a centralized controller, while data communications are distributed among the services. The conducted performance 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.

We introduce active mediation and mobile classes that enable computations to be distributed among autonomous service. A mobile class, which implements specific information processing functionality, can be dynamically loaded onto an autonomous service to process data local to the autonomous service. We discuss how autonomous services support the execution of mobile classes with an active mediator. By moving computations closer to data, we can significantly reduce the amount of data traffic for a megaservice. The algorithm to determine the optimal location for the execution of mobile classes is discussed.

An information service infrastructure is described at the end. We use construction project scheduling software to illustrate the process by which services are built and integrated using FICAS. Legacy engineering applications are tied together to form integrated work processes. FICAS, based on the distributed data-flow approach, is shown to be suited for integrating large-scale engineering services.

8 Acknowledgement

This work is 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 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 4D Viewer and the 4D model of the Mortenson Ceiling Project are provided by Professor Martin Fischer and his research group at Stanford University. No approval or endorsement of any commercial product by the National Institute of Standards and Technology or by Stanford University is intended or implied. Certain commercial equipments, instruments, or materials are identified in this paper in order to facilitate better understanding. Such identification does not imply recommendations or endorsement by the National Institute of Standards and Technology or by Stanford University, nor does it imply the materials or equipment identified are necessarily the best available for the purpose.

9 References

[1] *Apache SOAP*, Apache Software Foundation, http://xml.apache.org/soap/, 2002.

[2] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*, Java Series, Boston, MA, 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] S. Bodoff, D. Green, K. Haase, E. Jendrock, M. Pawlan, and B. Stearns. *The J2EE Tutorial*, Addison Wesley Professional, 2002.

[5] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer. *Simple Object Access Protocol (SOAP)*, W3C Note, http://www.w3.org/TR/SOAP, 2000.

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

[7] D. E. Comer. *Internetworking with TCP/IP, Volume I, Principles, Protocols, and Architecture*, 4th ed, Prentice Hall, 2000.

[8] D. Davis and M. Parashar. "Latency Performance of SOAP Implementations," *Proceedings of 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2002)*, Berlin, Germany, pp. 407-412, May 2002.

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

[10] M. Kirtland. "*The Programmable Web: Web Services Provides Building Blocks for the Microsoft .NET Framework*," *MSDN Magazine*, September 2000.

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

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

[13] OMG. *The Common Object Request Broker: Architecture and Specification Version 2.0*, Object Management Group, Report # 95-3-10, July 1995.

[14] L. Perrochon, G. Wiederhold, and R. Burback. "A Compiler for Composition: CHAIMS," *Proceedings of Fifth International Symposium on Assessment of Software Tools and Technologies*, Pittsburgh, June 1997.

[15] C. Schlenoff, M. Gruninger, F. Tissot, J. Valois, J. Lubell, and J. Lee. *The Process Specification Language (PSL): Overview and Version 1.0 Specification*, National Institute of Standards and Technology, Gaithersburg, MD, Report # 6459, 2000.

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

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