Data-flow Distribution in FICAS Service Composition Infrastructure

David Liu
Dept. of Electrical Engineering
Stanford University
Stanford, CA 94305
davidliu@stanford.edu

Kincho H. Law
Dept. of Civil and Environmental Engineering
Stanford University
Stanford, CA 94305
law@cive.stanford.edu

Gio Wiederhold
Computer Science Department
Stanford University
Stanford, CA 94305
gio@db.stanford.edu

Abstract

This paper presents FICAS, a distributed data-flow infrastructure for composing software services into megaservices. We discuss the basic features of FICAS that enable the distribution of data-flows within megaservices. An autonomous service access protocol, ASAP, is defined to enforce the explicit separation of control-flows from data-flows of software services. We illustrate the procedure to construct and optimize megaservice execution plans that form distributed data-flows among collaborating services. The megaservice performance under FICAS is evaluated and compared with that under the centralized data-flow infrastructures. FICAS enhances the megaservice performance and is especially suitable for large-scale service composition.

1. INTRODUCTION

1.1. Background

As computation and communication technologies evolve, we are seeing a change in how large software applications are built. Rather than being constructed from ground up, applications are constructed by gluing together software services, each provides portion of functionalities. The megaprogramming framework [3, 12] and more recently Computational Grids computing systems [2] echo the vision of software composition that links together autonomous services to form megaservices. Though distributed and heterogeneous, autonomous services can be utilized as if they were locally available to the megaservices.

Service composition infrastructure is responsible for composing and executing megaservices. There are three goals in building service composition infrastructure: (1) Ease of composition – effective and convenient specification of service compositions by the application programmers; (2) Scalability – integration and management of large number of autonomous service in the service composition infrastructure; and (3) Performance – high efficiency in the execution of megaservices.

We build on prior systems such as CHAIMS (Compiling High-level Access Interfaces for Multi-site Software) [11] for autonomous service composition. The compositional language developed in CHAIMS supports well the goal for ease of composition. We intend to improve the scalability and the performance of the megaservices in CHAIMS via the distribution of data-flows at runtime. Given that many existing service composition infrastructures employ similar execution model as CHAIMS, our findings should also be broadly applicable to other systems.

1.2. Overview

A service composition runtime environment is conceptually viewed as a set of service nodes interconnected by a communication network. Messages are passed between pairs of service nodes. There are two types of messages: control messages and data messages, distinguished by their use at the recipients of the messages. Control messages are mostly short messages that trigger state changes at the receiving services. Data messages are mostly large data packets that are given to the receiving services for processing. We use control-flow to describe a group of related and partially ordered control messages, and use data-flow to describe a group of related and partially ordered data messages.

Service composition runtime environments differ in how control-flows and data-flows are formed and managed. Figure 1(a) illustrates the control-flows and the data-flows exhibited by a megaservice in the CHAIMS runtime. The megaservice control node 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, DCOM, Java RMI, and SOAP [10].

There are performance and scalability issues associated with the 1C1D model, where the centralized megaservice control node becomes the communication bottleneck when large amount of data are exchanged among autonomous services. The issues observed in the 1C1D model motivate us to distribute the data-flows for the executions of megaservices. Figure 1(b) shows the control-flows and the data-flows exhibited in a distributed data-flow infrastructure. The megaservice has the ability to inform two or more autonomous services to establish a data-flow through which data can be directly communicated. For instance, data are exchanged between autonomous services, from *Service1* to *Service2*, and from *Service2* to *Service3*, without going

through the megaservice. We call this runtime model the *centralized control-flow distributed data-flow model*, or *1CnD model*. This paper explores the techniques that support the distribution of data-flows within the service composition infrastructure.

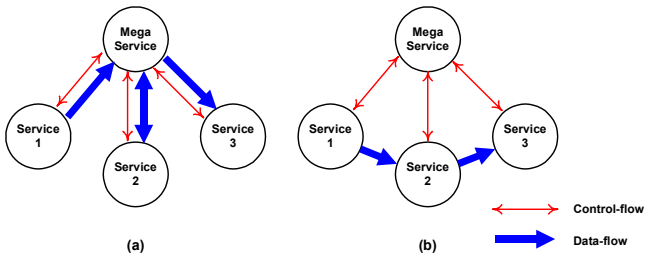

Figure 1: Centralized and Distributed Data-flows

2. FICAS

FICAS (Flow-based Infrastructure for Composing Autonomous Services) is a service composition infrastructure that supports distributed data-flows. FICAS consists of many interrelated components. As shown in Figure 2, FICAS is divided into buildtime and runtime. The buildtime components are responsible for composing megaservices and compiling megaservice specifications into control sequences that serve as inputs to the runtime environment. The runtime components are responsible for the executions of the control sequences.

Composition of autonomous services starts with the megaservice specification. For FICAS, we have defined CLAS (*Compositional Language for Autonomous Services*) to provide the application programmers the necessary abstractions to describe the behaviors of their megaservices [8]. CLAS focuses on functional composition of autonomous services. A CLAS program is essentially a sequential specification of the relationships among collaborating autonomous services. It does not provide any primitives to schedule and coordinate control-flows and data-flows. The CLAS program is translated 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 megaservice specification and megaservice execution.

The FICAS runtime environment is responsible for executing the control sequences. The megaservice controller is the entity that carries out the execution of a megaservice. The controller first converts an input control sequence into an execution plan, and then follows the plan to coordinate control-flows among the respective autonomous services. The controller serves as the centralized coordinator for all the control messages incurred by the megaservice. Since the megaservice execution is carried out with parallel invocations of autonomous services, the controller is also responsible for synchronizing control-flows and conducting performance optimization.

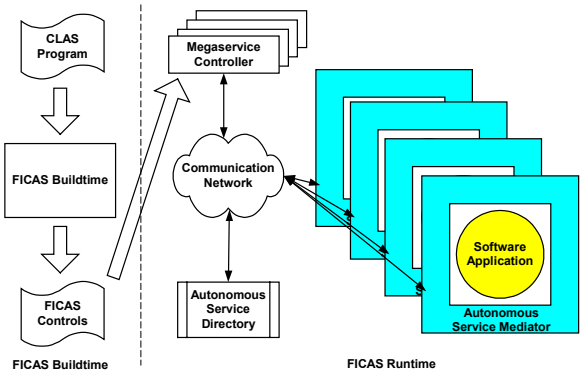

Figure 2: FICAS Architecture

Building a scalable runtime environment requires a mechanism to easily incorporate new software applications. This is achieved by wrapping each software application into an autonomous service with a mediator. The autonomous service mediator supports a common protocol that is developed to provide uniform access to the autonomous services. Autonomous services can join (or quit) the service composition infrastructure by directly connecting to (or disconnecting from) the communication network. The modularity of the autonomous services provides the infrastructure scalability and fault isolation.

The autonomous service directory is created to index the autonomous service parameters. It keeps track of available autonomous services within the infrastructure. The directory is viewed globally as a centralized entity, while it may be implemented as a distributed structure. The address of the directory is universally known to all the components within FICAS.

3. AUTONOMOUS SERVICE

The behavior of autonomous services is characterized by the autonomous service metamodel, based on which an access protocol is defined to provide uniform access to the autonomous services.

3.1. Autonomous Service Metamodel

Figure 3 illustrates the FICAS autonomous service metamodel, where 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 most important characteristic of the autonomous service metamodel is the explicit separation of control-flows from data-flows. For data-flow, the autonomous service primarily concerns about performing services on the data elements. For control-flow, the autonomous service primarily concerns about the state management of an autonomous service.

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 an existing software application 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 queue, where events are processed in the order by which they are received.

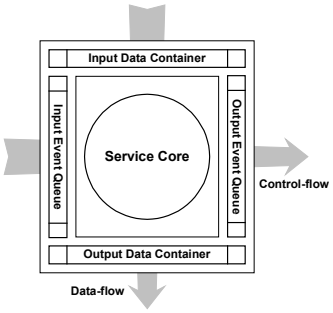

Figure 3: FICAS Autonomous Service Metamodel

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.

3.2. Protocol Support for Data-flow Distribution

Given the autonomous service metamodel, we define an autonomous service access protocol, ASAP, by which the autonomous services are accessed. The protocol removes the barriers imposed by different megaservice programming languages and distribution protocols.

ASAP manages control-flows and data-flows through a set of events. These events exist in the form of XML based messages that are used to interact with autonomous services. The 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.

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. More complete information on the ASAP protocol is given in [8].

- 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.

- TERMINATE (Service)

The TERMINATE event unconditionally terminates an autonomous service. Garbage collection is conducted during the termination process, when system resources involved with an autonomous service instance are released.

- 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.

- 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.

4. DISTRIBUTED DATA-FLOW SCHEDULING

FICAS assigns the megaservice controller the sole responsibility in coordinating control-flows for a megaservice. The controller is responsible for issuing the ASAP events and monitoring their results. An execution plan is generated to determine the choice, timing and sequence of ASAP events.

There are three steps in generating an execution plan. First, the megaservice program is analyzed to discover data dependencies among the invocations of 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 4 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*.

```
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 4: Sample Megaservice Program Segment

The data dependencies are mapped into a data dependency graph (DDG) as shown in Figure 5. 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*.

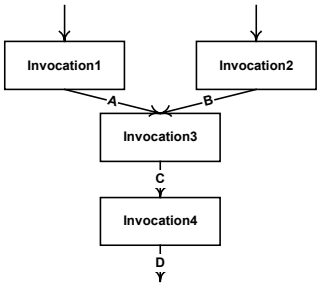

Figure 5: Sample DDG

The DDG can be further converted into a megaservice execution plan in the form of an event dependency graph (EDG). The megaservice controller uses the EDG to coordinate the execution of autonomous services. In an EDG, the nodes represent the ASAP events managed by the megaservice controller, and the arcs represent the dependency between a pair of related ASAP events. Invocation nodes in the DDG can be directly mapped into the INVOKE event nodes in the EDG. The mapping of the directed arcs in the DDG is more complex. Different mapping schemes may produce different data-flow models for the megaservice.

Figure 6 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 5) is mapped into the *MAPDATA(A, Service1, Service3)* event node in the EDG (shown in Figure 6).

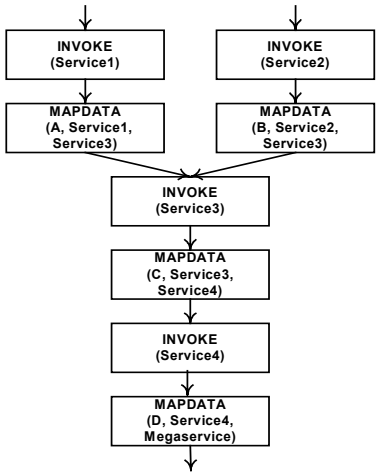

Figure 6: EDG with Distributed Data-flows

5. PERFORMANCE ANALYSIS

The distribution of data-flows improves performances and scalability by avoiding the data communication bottleneck at the megaservice controller. In this section, we study the performance characteristics of megaservices under different system settings.

5.1. FICAS vs. SOAP

We study the performance of megaservices in an example engineering service environment as shown in Figure 7. The megaservice is specified to retrieve a specific project model using the *ModelRetriever* service, then conduct scheduling on the model using the *Scheduler* service, and finally notify the related parties about the change via the *ChangeManager* service.

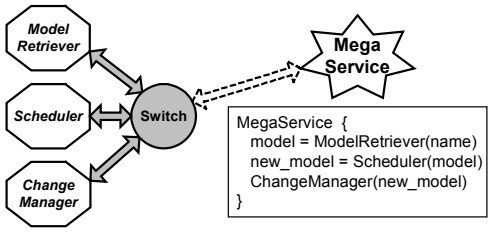

Figure 7: A Megaservice for Engineering Services

The autonomous services run on distributed servers that are connected via a switch with 10mbps bandwidth on each port. The megaservice runs on a client machine that is

connected to the servers either directly via the switch or via an 802.11 wireless access point. The two network settings facilitate our comparison of megaservice performances.

We implement the megaservice with two different integration models: (1) SOAP [10] is used as the reference platform for the 1C1D model, where each service invocation is a remote procedure call initiated from the megaservice; and (2) FICAS is used as the reference platform for 1CnD model, where data-flows are distributed.

The response times of megaservices are measured with different settings on the size of the project model. Since the computational elapsed times contributed to the autonomous service executions are identical under both integration models, we compare only the communication elapsed time, which is calculated as the megaservice response time minus the sum of processing elapsed times of autonomous services. Figure 8 shows the megaservice performances measured with various network settings and integration models.

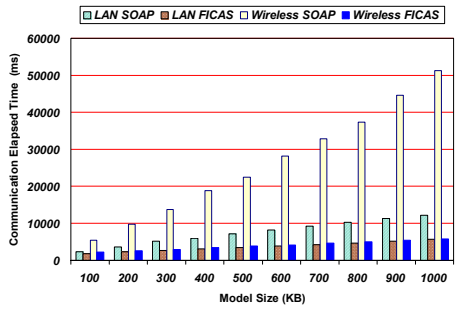

Figure 8: Performance of the Megaservice

A couple of observations can be made that are consistent with our mathematical analysis conducted in [7]. First, the response times under FICAS are better than their counterparts under SOAP for all load settings. The larger the project model size, the more significant the performance improvement the FICAS model has over the SOAP model. Secondly, the response time increases linearly with respect to the volume of the data-flows. The response times under SOAP increase at much faster rates than the response times under FICAS. The rate of increase is especially significant in the wireless connection scenario, when data communications between client machines and servers become a bottleneck in SOAP. On the other hand, we observe small increase in response time in FICAS with large model sizes. The FICAS (1CnD) model alleviates the bottleneck by distributing network traffic among the autonomous services.

5.2. Performance Impact of Control-flows

When the control message size is comparable to the data message size, the impact of control-flows on the performance of megaservices needs to be accounted for.

Under the 1C1D model, a SOAP service invocation consists of two messages: (1) an invocation request message sent from the megaservice controller to the autonomous service, and (2) a message returned from the autonomous service containing the result of the service invocation. Under the 1CnD model, FICAS breaks up a service invocation into multiple stages, introducing a sequence of messages: (1) an invocation request message sent from the megaservice controller to the autonomous service, (2) an acknowledgement message sent back to notify the completion of the service, (3) a MAPDATA message sent from the megaservice to notice the autonomous service where the result data should be forwarded, (4) a data message forwarding the data content between two autonomous services, and (5) an acknowledgement message notifying the megaservice controller the completion of the MAPDATA task.

We model the cost of messages as a linear function to their sizes. Each message has a fixed setup cost of λ_0 (e.g. the cost of initialization, buffering, etc.). In addition, a control message has a payload of 1, and a data message has a payload of λ_d. Hence, each control message incurs a cost of (λ_0+1), and each data message incurs a cost of $(\lambda_0+\lambda_d)$.

The aggregated communication cost for a service invocation can be calculated by adding up the costs of all messages. Under the 1C1D model, the aggregated cost is $(2\lambda_0+1+2\lambda_d)$. Under the 1CnD model, the aggregated cost is $(5\lambda_0+4+\lambda_d)$. Figure 9 illustrates the costs under both models with different λ_0 and λ_d settings.

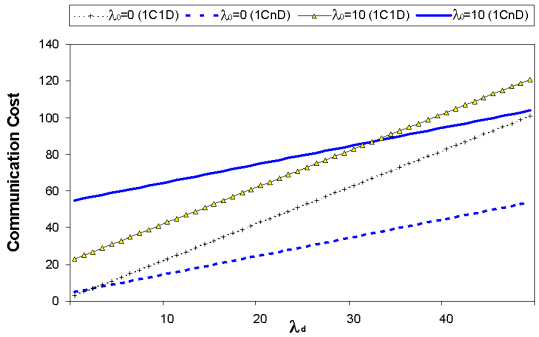

Figure 9: Communication Costs for Service Invocations

We observe that higher message setup cost λ_0 attributes to higher communication cost for service invocations. The performance in the 1CnD model is more adversely affected than in the 1C1D model. Since the 1CnD model incurs more messages for each service invocation, its performance is more sensitive to the message setup cost. Furthermore with a small data payload, the 1C1D model may perform better that the 1CnD model. However, the communication cost for 1C1D model scales up much faster than for the 1CnD model. The 1CnD model outperforms the 1C1D

model with a larger data payload. The 1CnD model becomes a preferred environment for the composition of autonomous services when exchanged data are much larger than the control messages.

6. RELATED WORK

Dataflow network based systems [6, 9] are similar to FICAS in how data-flows are distributed. However, there are several important differences: (1) Computational nodes in dataflow networks usually handle fine-grained tasks and require homogeneity in the underlying hardware platform, whereas autonomous services in FICAS are coarser-grained and heterogeneous in nature. (2) The dataflow networks use the flow of information as the only control mechanism. State transitions within a node are caused by arrivals of its input data. FICAS adopts an event driven paradigm where control logic is centrally specified and executed, greatly simplifying the programming and execution model. (3) The dataflow network is established at initialization time, prior to the program execution. This lack of ability to dynamically establish links between computational nodes limits the use of dataflow networks in realistic applications.

MANIFOLD [1] introduces event driven control paradigm to complement the dataflow like control mechanism. Events and the logic to handle the events are explicitly specified in MANIFOLD programs. Facilities are provided to explicitly manage synchronization, proper ordering, and timing of activities involved in a program. Compared to MANIFOLD, FICAS has a much simpler programming model that is intended for application specialist with minimum programming experience. Furthermore, MANIFOLD programs explicitly specify data communication links between concurrent processes, whereas megaservices in FICAS rely on the runtime system to perform optimization and schedule dynamic data-flows between autonomous services.

In our research, we use the CHAIMS system as a point of departure. There are other compositional tools and frameworks that we could have chosen, such as Globus [4] or Ninja Paths [5]. The purely compositional nature of CHAIMS allowed us to focus wholly on data-flow distribution without the distraction of the non-compositional (e.g., brokering, security) aspects of alternative frameworks.

7. CONCLUSIONS

This paper presents FICAS, a service composition infrastructure with distributed data-flows. Autonomous services are built to support the service access protocol ASAP, which enforces the explicit separation of data-flows from control flows. ASAP serves as the basis for building the high-performance, scalable, and distributed data-flow service composition runtime environment.

We illustrate the construction of the megaservice execution plan that takes advantage of the distributed data-flows. The performance of megaservices is analyzed and compared between the 1C1D model and the 1CnD model. We conclude that the distribution of data-flow in FICAS enhances megaservice performance and thus is especially suitable for large-scale autonomous service composition.

8. ACKNOWLEDGEMENT

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

REFERENCES

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

[2] M. Baker, R. Buyya, et al., "The Grid: International Efforts in Global Computing", Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR 2000), Italy, 2000.

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

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

[5] S. Gribble, M. Welsh, et al., "The Ninja Architecture for Robust Internet-Scale Systems and Services", U.C. Berkeley, To appear in a Special Issue of Computer Networks on Pervasive Computing, 2002, http://ninja.cs.berkeley.edu/dist/papers/ninja.ps.gz.

[6] J. Herath, N. Saiko, et al., "Dataflow Computing Models, Languages and Machines for Intelligence Computations", *IEEE Transactions on Software Engineering*, vol. 14, 1988, pp. 1805-1828.

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

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

[9] P. Suhler, J. Bitwas, et al., "TDFL: A Task-level Dataflow Language", *Journal of Parallel and Distributed Computing*, vol. 9, 1990, pp. 103-115.

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

[11] G. Wiederhold, D. Beringer, et al., "Composition of Multi-site Services", Proceedings of IDPT'99, Kusadasi, Turkey, June 1999.

[12] G. Wiederhold, P. Wegner, et al., "Towards Megaprogramming", *Comm. ACM*, vol. 35(11), Nov 1992, pp. 89-99.