

Separation of Concerns in Networked Service Composition

Emre Kıcıman and Armando Fox
{emrek, fox}@cs.stanford.edu

15th March 2001

Abstract

As networked services are increasingly being used as the core building blocks of today’s software programs, there is a growing need to apply the software engineering principles of separation of concerns to these service-based components. But most methods of separating concerns are not highly applicable to networked services because of their opaque nature—a composer has no control over the internals of these services. In this paper, we propose a model for how a software composer can take advantage of a graph model of service composition, and manipulate this representation to begin to separate service functionality from system-wide concerns such as performance, reliability, and manageability.

1 Introduction

As networked services are increasingly being used as the core building blocks of today’s software programs, there is a growing need to apply the software engineering principles of separation of concerns to these service-based components. Most methods of separating concerns assume that the components being used in development are under the control of the software composer. But this assumption does not hold when components are networked services—a composer has no control over the internals of these services. In this paper, we propose a model for how a software composer can take advantage of a graph model of service composition to separate service functionality from system-wide concerns such as performance, reliability, and manageability.

Network services are typically deployed and controlled by third-parties. They provide functionality that is difficult to encapsulate into redistributable components, or inconvenient to administrate except in a central location. This includes searches across large databases (web search engines, event calendars, etc.), complicated data transformers (web proxies for PDAs, and comprehensive document transformers), and “real world” services (e-commerce services, interfaces to hardware devices, etc.).

Since these services are deployed by third-parties, software composers do not have the ability to apply internal transformations to optimize the services they use. Effectively, the networked service model is enforcing a dominant decomposition of the application based on the administrative domains of the services. This makes it impossible to directly apply existing techniques for separation of concerns, such as aspect-oriented programming and hyperslices[10, 6].

Though composers cannot manipulate the internals of the networked services, they can manipulate how these services are used within the larger system. Specifically, service composers have the ability choose among competing service providers, insert arbitrary services within a composition, and rewrite the connections between services to change the flow of data between services. We concentrate on using these abilities to enhance the system-wide concerns of performance, reliability, and manageability.

1.1 Why Services?

Networked services can be difficult for a composer to customize, they impose a dominant dimension of separation which may not map well to other concerns.

Why would we want to use these services?

Simply put, networked services can provide functionality that traditional components cannot. They can be deployed with large amounts of data (street maps of the world, searchable index of the World Wide Web); data which cannot be feasibly distributed to potential users. The sheer volume of data involved in

Logically centralized services can be easier to administer, making it possible to provide functionality that would be overwhelming to provide as a traditional redistributable component. When functionality is centralized in one place, service and data upgrades are greatly simplified: minor (and sometimes even major) improvements to services can be deployed at a small number of controlled sites, rather than by distributing updates of a software component to potentially millions of people.

Services can also provide interfaces to “real-world” functionality (services such as printers, e-commerce). Since this functionality is physically located somewhere, it is by necessity required to be network accessible to be used from outside its immediate vicinity.

Finally, shared service infrastructure can provide a more efficient utilization rate than individual devices or applications. Individual personal devices are generally in use less than 4% of the time, whereas shared infrastructure services attempt to maintain an 80% utilization rate[2].

Today, network services are being deployed at a growing rate. The most popular framework for deploying networked services is the web, using HTTP-based servers and proxies, Microsoft’s .NET framework [7], Sun’s Jini[9], and BeComm’s String and Beads architecture [1]are all providing platforms for the deployment and composition of networked services.

1.2 Example

As an example, consider that a bookstore wants to deploy service to allow PDA users in their stores to compare prices of books on the shelves with the prices at various on-line stores. The desired composition, shown in Figure 1 on the following page, involves querying a number of on-line bookstores for the price

of the book, aggregating these results into a single document, and formatting this document for display on a handheld device. In this scenario, each of the on-line bookstore services and the data formatter is a third-party networked service. The disseminator and aggregator may or may not be third-party services, depending on the implementation.

The semantic functionality of this composition is easy to understand, but it is more difficult to comprehend how these services interact to provide a high-performance, high-availability, and easily-managed system. What happens when one of the on-line bookstore services fails, or responds slowly to a query?

The composer of this system can insert additional services into the system to compensate for possible failures of other services. For example, to increase performance, the composer can insert caching services to intercept data flowing in and out of slow services. To achieve a higher reliability, the composer can query multiple servers for each on-line bookstore, and run a voting algorithm to choose the “right” answer for each query. To manage and debug the system, the composer can insert logging and assertion operators into the composition. Each of these concerns can be addressed by making changes to the composition itself, without manipulating the internals of the networked services.

2 Paths

As a testbed framework, we are using *Paths*, a service composition framework designed to simplify the composition of systems from service-based components[4]. A Paths composition is a pipe/filter stream through a graph of operators and connectors. Operators are services in the Paths framework, performing computations on data. Connectors transport data between services. Legacy services, such as existing web services, can be wrapped and used as operators in a composition.

Operators are loosely-coupled with each other, and do not directly name each other. Instead, operators name only their own inputs and outputs. A generic run-time system ensures that data from the output of one operator is correctly routed to the input of

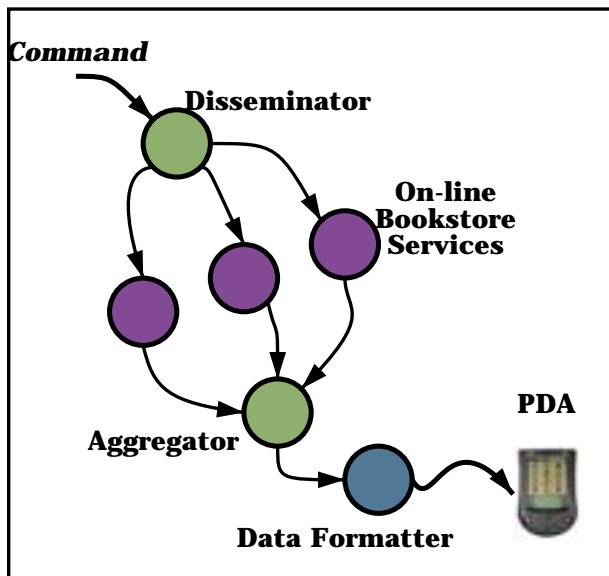


Figure 1: An example composition, aggregating data results from various services for display on a handheld device.

the next operator in the composition. The specification of a composition is defined external to the components, and, because of the indirection between operators, can be dynamically changed.

In addition, an explicit goal of the Paths framework is to eliminate the need to write complicated “glue-code” when composing services[5]. This has the effect of making automatic manipulation of a composition easier: rather than generating glue code and logic when we manipulate a composed system, we only need to incrementally modify a declarative specification of the composition.

3 Manipulating Compositions

To provide a separation of concerns in the composition of networked services, we use a model of pattern-recognition and rule-based manipulation. Once a the semantic functionality of a system has been composed, orthogonal concerns are added to the system by recognizing patterns in the composition, and ma-

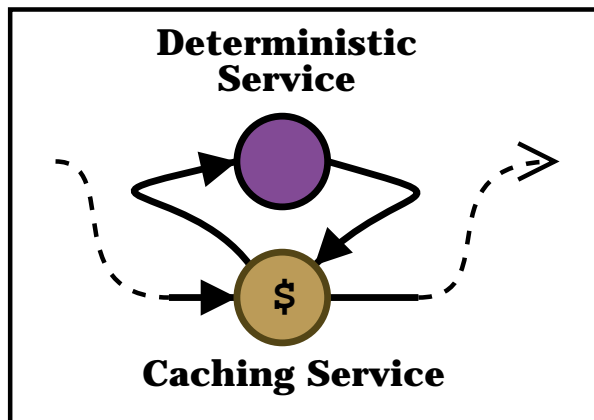


Figure 2: A caching service can be inserted to intercept the inputs and outputs to a deterministic service to improve system performance. The caching semantics (cache invalidation, etc.) can be set by the composer.

nipulating the pattern-matches in regular ways. Together, a pattern and rule implement a specific solution to a concern.

For example, to implement simplistic automatic cache placement to improve system-wide performance, one pattern might be to a string of deterministic, side-effect free services composed together. Once this pattern is found, a caching service can be inserted around the deterministic services to intercept service requests and attempt to fulfill them from a cache, or forward them to the services for completion.

Similar to aspect-oriented programming [6], our system matches patterns in a data-flow graph between components. However, because the internals of services are hidden by the time of composition, our pattern-matcher must rely on an explicit description of the service’s attributes

3.1 Overlapping Concerns

An important question is the degree to which there is an overlap of concerns between patterns-manipulations and the services themselves. Some concerns are almost completely orthogonal to the im-

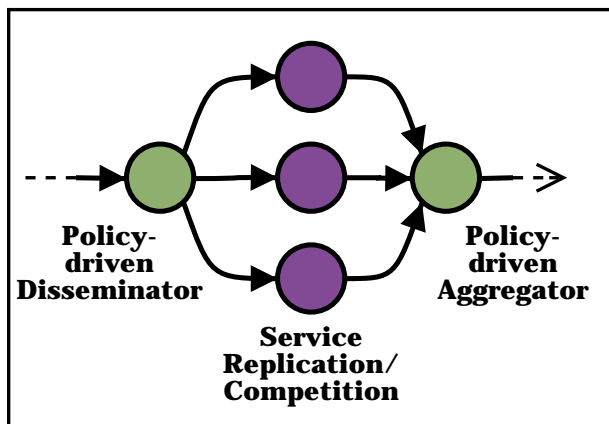


Figure 3: By inserting policy-driven aggregators and disseminators around services, and replicating services (or using competing services), we can implement a large number of common distributed systems patterns.

plementation of a service, while other concerns require intimate knowledge and/or trust of the service implementation. The caching example above, for example, searches a composition based on attributes of determinency. Another concern, such as automatic checkpointing of service state, might require much greater knowledge to be shared between the service and the pattern-manipulation.

We are currently investigating what properties of services are the most useful for recognizing patterns in compositions.

3.2 Building Blocks for Manipulation

In designing manipulation patterns for performance and reliability concerns, we have come across a small number of manipulations that can be used to instantiate a relatively large number of concerns. The most common manipulations are the replication of a service, and the placement of policy-driven data disseminators and aggregators before and after service(s) in a composition.

With these two simple manipulations, we can implement many common distributed system patterns. To improve performance, they can be used to imple-

ment parallelization of some kinds of computations, and various data striping schemes in storage systems. To improve reliability, these same manipulations can be used to implement Byzantine voting schemes, hot failovers, and mirrored communications. To aid in the management of a system, we can disseminate data to logging and monitoring services.

4 Related Work

The patterns and manipulations we use are most similar to the aspects and weavers used in aspect-oriented programming[6]. The main difference between the two is our emphasis on applying manipulations to coarse-grained objects, rather than the sub-component manipulations of aspect-oriented programming.

Hyperslices and hyperspaces assumes that multiple views on a system will be integrated at composition-time.

Injecting ilities by controlling communication: many of the actual implementations of “ilities” are similar. Main difference is that it is based on an implicit composition model, where as we manipulate an explicit, separate description of services.

Silva proposes a system to support the separation of concerns for the development of distributed applications [8]. Though related, this system does assume that the composer has control over the components being composed, and applies aspects to these components to manipulate their internals.

5 Conclusions

We have outlined a model for manipulating compositions of networked services to iteratively add support for the system-wide concerns of performance, reliability, and manageability. Modeling the composition as a data-flow graph of services, we can search for patterns in the composition, and apply manipulation rules to address these system-wide concerns, without requiring any internal changes to services.

We are currently implementing the pattern-recognizer and manipulator to manipulate composi-

tions, and are exploring the issues of service-attribute description for pattern recognition. We have delineated patterns and manipulation rules for about two dozen common distributed systems-style manipulations, and are working on building a test-application in the Paths composition framework, leveraging both new and legacy network services.

References

- [1] BeComm, Strings Overview. *Information sheet*. <http://www.becomm.com/strings.asp>
- [2] Eric A. Brewer, Lessons from Giant-Scale Services. *Draft*.
- [3] Filman, R.E., Barret, S., Lee, D. D., and Linden, T., Inserting Ilities by Controlling Communications. *To Appear in Comm. ACM*.
- [4] Emre Kıcıman and Armando Fox, Using Dynamic Mediation to Integrate COTS Entities in a Ubiquitous Computing Environment. In *Proceedings of the Second International Symp. on Handheld and Ubiquitous Computing (HUC2k)*, Bristol, UK, 2000.
- [5] Emre Kıcıman, Laurence Melloul, and Armando Fox, Towards Zero-Code Software Composition. *Submitted to Hot Topics in Operating Systems (HotOS VIII)*.
- [6] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin, Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. number 1241 in Lecture Notes in Computer Science. Springer-Verlag, June 1997. 14.
- [7] Microsoft, Microsoft .NET Remoting: A Technical Overview. *Technical Paper*. <http://msdn.microsoft.com/library/techart/hawkremoting.htm>
- [8] Antonio Rito Silva, Pedro Sousa and Jose Alces Marques, Development of Distributed Applications with Separation of Concerns. In *Proceedings of the 1995 Asia-Pacific Software Engineering Conference*, pages 168–177, Brisbane, Australia, December 1995.
- [9] Sun Microsystems. Jini Technology: Architectural Overview. *White Paper*. <http://www.sun.com/jini/whitepapers.architecture.html>
- [10] Peri Tarr, Harold Ossher, William Harrison, and Stanley Sutton, Jr., *N* Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, May 1999.