

Merging Ccaffeine within Web Service Oriented Architecture

Wei Lu

Extreme! Lab, Computer Science Departement, Indiana University

April 21, 2006

1 Introduction

Command Component Architecture (CCA) [4] is a specification of component architecture designed for high performance scientific computing. It provides a way to build and manager large scale scientific software system by composing the “plug-and-play” components together. Ccaffeine [2] is one of the implementation frameworks of the CCA specification. It aims to providing high performance parallelism and allowing single component multiple data programming model.

However to run a scientific computing job in ccaffeine, only ccaffeine usually is insufficient. For example we may need to stage the data in from a Grid data manager before the running of the job and after the running to stage the result out. Also a scientific computing job may need lots of cooperation among different remote services which may not be running in the ccaffeine framework. So it would be desirable to merge the ccaffeine framework within the scientific workflow system (e.g. Kepler[3]), thus a ccaffeine framework is able to communicate and share the resource with the other services in the workflow context.

Our solution is wrapping the ccaffeine framework as a collections of web services. With the web service interface, it is easy and nature for those WS-specifications and toolkits which are built over the Web service stack to interoperate with the ccaffeine framework, therefore the “outside world” can share the powerful parallel component computation provided by ccaffeine while ccaffeine is able to share the huge resource from the “outside world”.

2 Architecture

Ccaffeine provides a command-line front end and a set of scripts, via which we are able to load the components, instantiate them , connect them and finally trigger the running by invoking the GoPort of one component, which is conventically called Driver. The commands will be fanned out to the multiple processors at background by a Muxer, and be executed parallelly.

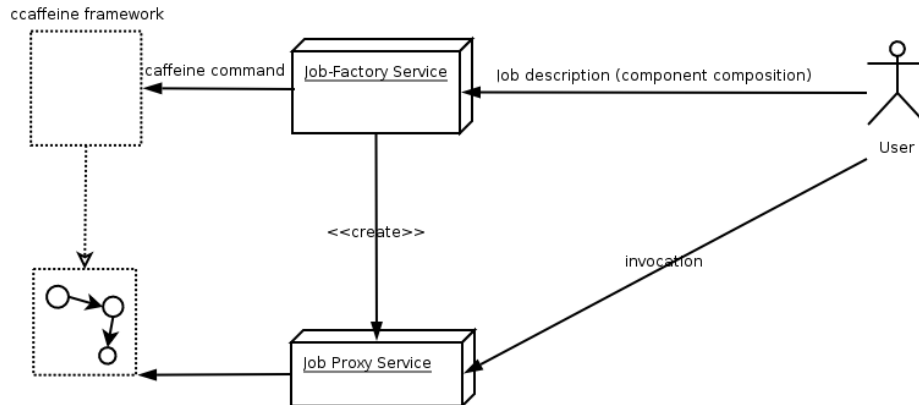


Figure 1: factory service

By the “job”, we mean a set of components connected together to implement a specific computing task (e.g. the calculation of Pi value). It is not hard to statically build a web service wrapper for a specific job in case we know the interface of its components. However it would be desirable to have a general *Job-Factory service* which can dynamically generate any job-specific service (called *Job-Proxy service*) based on users’ requirement. By the general Job-Factory service, users can dynamically design and modify how the job is going to work in term of components composition and then instantiate the design as a job-specific service. This is much like the run-time web service programming.

2.1 Job-Factory Service

The *Job-Factory service* basically provides a web-service front-end for the ccaffeine framework, and the IPC pipeline facilities is used for communication between them. However Job-Factory service doesn’t merely do the I/O redirection as the work in [6]. Since ccaffeine is not aware of the “Job”, it is the Job-Factory’s responsibility to create, maintain and monitor the Jobs and deploy the Job on the ccaffeine framework.

Currently the Job-Factory provides two methods

- create the job and
- modify an existing job

To invoke the create method, the user should provide

- *gatewayComponentType*: the component interface which is going to be exposed to outside as a Job-Proxy web service;
- *composition*: the instructions about how the components are connected together to implement the *gatewayComponentType* interface

Since CCA uses the SIDL[7] to describe the interface of the component, the *gatewayComponentType* will be the name of interface defined in the SIDL file. Once Job-Factory service get the *gatewayComponentType*, it will find the SIDL definition in its interface definition repository and try its best to convert the SIDL definition to equivalent WSDL definition, which is going to be the interface of the created Job-Proxy service.

As for the *composition*, for sake of simplicity we adopt the ccaffeine “connect” and “disconnect” command syntax directly. The Job-Factory just simply check the command and forward the command to the ccaffeine framework. For instance the below script copied from the cca tutorial can be past to the Job-Factory directly as the argument *composition*.

```
connect driversCXXDriver IntegratorPort integratorsMonteCarlo IntegratorPort
connect driversF90Driver IntegratorPort integratorsMonteCarlo IntegratorPort
connect integratorsMonteCarlo FunctionPort functionsPiFunction FunctionPort
connect integratorsMonteCarlo RandomGeneratorPort \
    randomgensRandNumGenerator RandomGeneratorPort
```

(Here for simplicity we assume all the needed components have been loaded and instantiated. To be practical, We are designing more sophisticate scheme for this step, for example factory can stage, load and instantiate the needed components automatically if they are not loaded yet.)

Once the WSDL conversion and ccaffeine command running succeed, which means the composition of the Job is done, the Job-Factory generates the ID for the job, launches a new *Job-Proxy service*, whose interface will be the generated WSDL file and returns a URL of the WSDL file to the user. Those job related data, like Job ID, WSDL file, composition structure and driver component, will be saved in the Job-Factory indexed by the Job ID for later on modification and management. Once the user get the URL, he can invoke the service or later on modify the composition of the Job. To modify the composition of the job, user just need to give the job ID and the new composition script to the Job-Factory, which in turn looks for the saved records and call the ccaffeine to run the new composition script. The modification will not affect the interface of the job at all, but by the power of component system it do change the internal composition structure hence the implemeantion of the job at run-time and keep the user of the job service unaware of the change.

2.2 Convert SIDL to WSDL

The automatic conversion of the SIDL definition of the gateway-port type to the equivalent WSDL definition is the most crucial but challenging part of the system.

In some sense both SIDL and WSDL are the interface description languages. They have similar syntax structure to describe the interface of a object or a service. The SIDL interface corresponds the WSDL portType, which is renamed as interface in WSDL 2.0 [?], and the methods in the SIDL interface corresponds the operations in the WSDL portType. Under the superficial similar-

ity, however, SIDL, which roots in the object oriented CORBA IDL, is fundamentally distinct from the WSDL, which is inherently designed for message passing paradigm. The WSDL focus the message structure on the wire which server/client agree upon, whereas SIDL is about how the object are accessed through its interface. As a result, unlike the SIDL interface which is treated as the interface in Java or the abstract class in C++, the WSDL interface should be seen as a group of message exchanges in which a Web service is prepared to participate. Therefore the WSDL interfaces don't have the inheritance relationship.

Also the WSDL interface can't be refereed as type of parameters of the operations, nor as the element type defined in XML schema since we can infer little structural information merely from the interface. In this sense the WSDL is more one Contract Definition Language than one general IDL [?].

The message-oriented characteristic makes WSDL be the excellent interface language for loose-couple service oriented system since only on-wire format is enforced, eliminating the need for a common type system between client and server. Meanwhile SIDL is more suitable for the tight-coupled system, especially those with shared address space (e.g., Ccaffeine), in where a common type system is easier to maintain.

To convert the SIDL to the WSDL for the gateway-port type, currently we prohibit the SIDL interface type and its inheritance from the definition of method parameters. In other word, only primitive type and structure type are supported in the gateway-port type. With this restriction, it will be direct for the convection by referring the CORBA Web Service [?]. For instance a simple example of the SIDL definition and its converted WSDL definition.

```
interface IntegratorPort extends gov.cca.Port
{
    double integrate(in double lowBound, in double upBound, in int count);
}
```

can be converted the below equivalent WSDL directly.

```
<wsdl:message name="integrateInput">
  <wsdl:part name="lowBound" type="xsd:double"/>
  <wsdl:part name="upBound" type="xsd:double"/>
  <wsdl:part name="count" type="xsd:integer"/>
</wsdl:message>
<wsdl:message name="integrateOutput">
  <wsdl:part name="return" type="xsd:double"/>
</wsdl:message>
<wsdl:portType name="integrator.IntegratorPort_PortType">
  <wsdl:operation name="integrate">
    <wsdl:input message="integrateInput"/>
    <wsdl:output message="integrateOutput"/>
  </wsdl:operation>
</wsdl:portType>
```

Considering the support structure type in the SIDL is ongoing and SIDL interface is still main role in most SIDL definitions, we can leverage the language-specific reflection mechanism to alleviate the interface conversion problem somehow. User should provide the SIDL as well as the declaration file (head file in C++ or java file) of its implementation class by the reflection mechanism of C++ or Java [?, ?] we can figure out the structure of the implementation class, and thus its wire-format in the best effort.

2.3 Job-Proxy service

A Job-Proxy service is a web servicer wrapper specifically for a running job, so it has its own specific WSDL interface. The Job-Proxy receive and validates the incoming SOAP message; then extracts the typed argument values from message and feeds them to the ccaffeine.

CCA defines a special port, called GoPort, whose go() method is the start entry point of a Job and is analogous with the main() function in C. The component implementing the GoPort usually is called Driver. As a result, every job must have an associated Driver component, and by triggering the Driver component (i.e. calling go command in ccaffeine), we begin the running of the job. However the go() method of GoPort doesn't have any input arguments, and its return value is designed be to the indication of success or failure of execution rather than the meaningful result. So in order to pass the initial arguments and get back the return value between Job-Proxy and the Driver, we utilizes the port-property defined in CCA, which is a map of key and its typed value. For every input argument, the Job-Proxy set the corresponding port-property of GoPort of the Driver and Driver gets those argument value and begin the running; the result value will be set as a port-property by Driver again and fetched by Job-Proxy. (*Seems the ParameterPort is a better choice!*)

3 Build distributed workflow over the Job-Proxy service

With the help of the Job-Factory service and Job-Proxy service, user can create web services for various high performance jobs distributedly. and the web service interface enables we weave those distributed services together seamlessly by web service based workflow system(e.g. BPEL[1]).

This architecture provides a reasonable solution to merge the parallel computing within Web Service Oriented Architecture. For those computation intensive tasks, we deploy them in the ccaffeine as a Job-proxy services so that the parallel computation resource can be shared by remote users or services, whereas the workflow enable the sharing of the resource and cooperation of various services in the Grid.

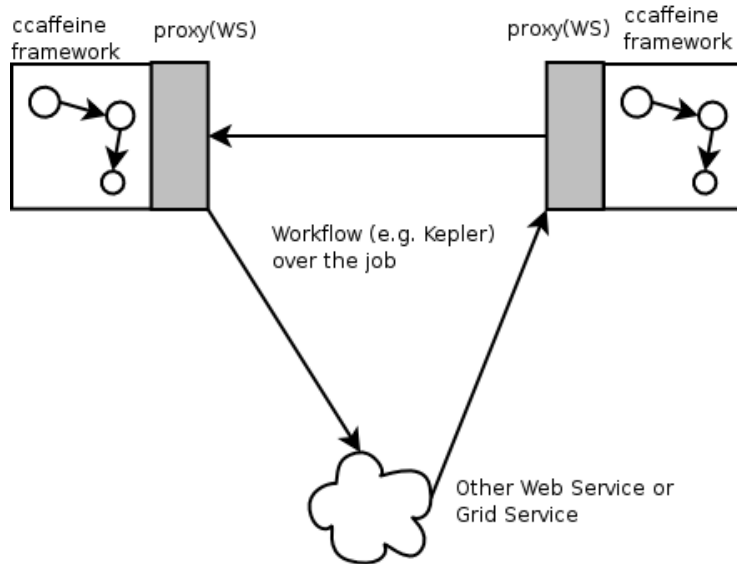


Figure 2: workflow over the jobs

3.1 Experience with Kepler

Kepler[3] is a workflow system designed for scientific application, it provides the orchestration over web service, Grid resource and numerous scientific applications packages. We takes the Kepler as the testbed for the merging idea. Below is a small demo to show how the ccaffeine Job-Framework and Job-Proxy are integrated in the Kepler developing environment.

1. Invoke the Job-Factory service to create the Job, whose task is simply calculating the value of Pi (*we uses the general on-line SOAP client <http://www.mindreef.net/tide/scopeit/start.do>*)

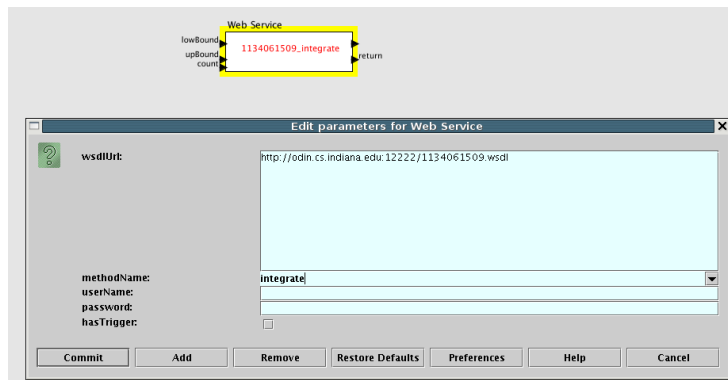
Request

```
create
(
  string gatewayComponentType = "integrator.IntegratorPort",
  string command = "
connect driversCXCDriver IntegratorPort integratorsMonteCarlo IntegratorPort
connect driversP90Driver IntegratorPort integratorsMonteCarlo IntegratorPort
connect integratorsMonteCarlo FunctionPort functionsPiFunction FunctionPort
connect integratorsMonteCarlo RandomGeneratorPort randomgensRandNumGenerator RandomGeneratorPort
"
)
```

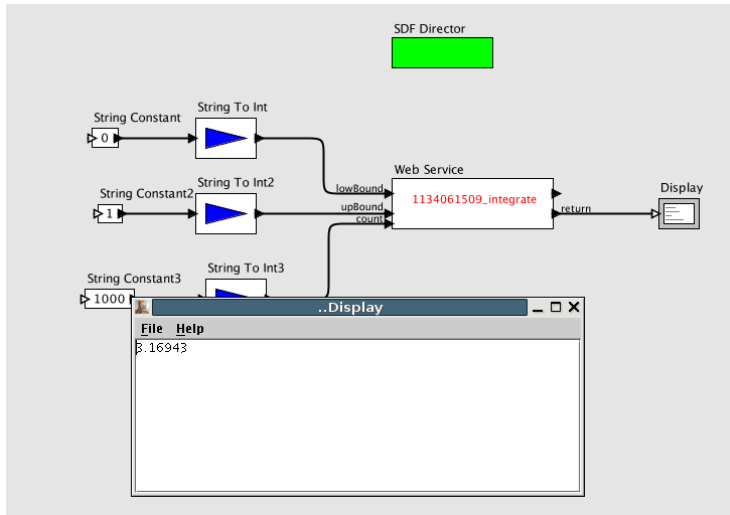
Response

```
serviceUri = http://odin.cs.indiana.edu:12222/1134061509.wsdl
```

2. Create a web service actor for the Job-Proxy in Kepler



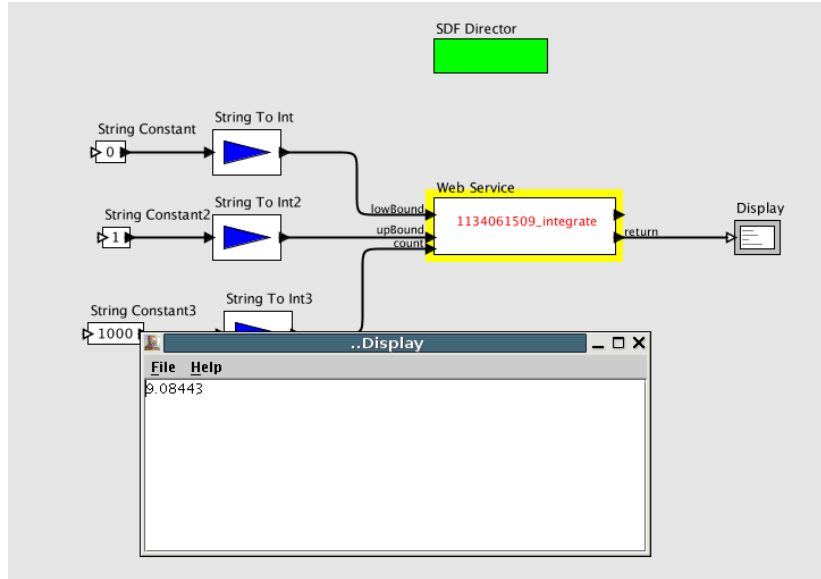
- Run the Job-Proxy in the context of workflow, and get the calculated Pi value



- Modify the Job by calling Job-Factory again, now we replace the PiFunction component with LinearFunction component

Request	Pseudocode
	<p>HTTP Headers</p> <pre> modify (string jobId = "1134061509", string command = " disconnect integratorsMonteCarlo FunctionPort functionsPiFunction FunctionPort connect integratorsMonteCarlo FunctionPort functionsLinearFunction FunctionPort ") </pre>
Response	Pseudocode
	<p>HTTP Headers</p> <pre> modificationResult = true </pre>

- Run the modified Job-Proxy again, the calculated value is changed due to the use of LinearFunction complement



4 Future work

- It is desirable to have a unified developing environment for both workflow and ccaffeine composition. We are working a special Kepler Actor for ccaffeine. By the actor we can specify, modify the composition (even in GUI) of a Job directly in the Kepler developing environment. This will eliminate the need of their-part SOAP client
- Job-Factory might be able to be more powerful for components manager, data staging and other work to set up and tear down the context of job running.
- Every job should have a Driver. However writting a dummy Driver , which just pass the argument and result between the Job-Proxy and other components, will be tedious. We are considering the feasibility of automatically generation of the Driver component for the each Job while the Job-Factory creates the Job.

References

- [1] Business process execution language for web services. <http://www-128.ibm.com/developerworks/library/specification/ws-bpel>, 2003.

- [2] Benjamin A. Allan, Robert C. Armstrong, Alicia P. Wolfe, Jaideep Ray, David E. Bernholdt, and James A. Kohl. The cca core specification in a distributed memory spmd framework. *Concurrency and Computation: Practice and Experience*, 14(5):323–345, 2002.
- [3] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludscher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows. In *16th Intl. Conf. on Scientific and Statistical Database Management (SSDBM'04)*, 21-23 June 2004 2004.
- [4] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott R. Kohn, Lois McInnes, Steve R. Parker, and Brent A. Smolinski. Toward a common component architecture for high-performance scientific computing. In *HPDC*, 1999.
- [5] The Common Component Architecture Forum Tutorial Working Group. A hands-on guide to the common component architecture. "http://www.cca-forum.org/download/tutorial/guide-html-0.3.1 ,c2", 2005.
- [6] Victor P. Holmes, Wilbur R. Johnson, and David J. Miller. Integrating web service and grid enabling technologies to provide desktop access to high-performance cluster-based components for large-scale data services. In *Annual Simulation Symposium*, number 167-174, 2003.
- [7] Gary Kumpf, Tamara Dahlgren, Thmoas Epperly, and James Leek. Babel 1.0 release criteria: A working document. <http://www.llnl.gov/CASC/components/docs/BabelReleaseCriteria.pdf>, December 2003.