國立交通大學

資訊科學與工程研究所

碩 士 論 文

資源導向計算及虛擬工作環境的設計與實作

Resource-Oriented Computing: Towards a Universal Virtual Workspace

研 究 生：洪秋榮

指導教授：陳俊穎　教授

中 華 民 國　九 十 五　年 八 月

資源導向計算及虛擬工作環境的設計與實作
Resource-Oriented Computing: Towards a Universal Virtual Workspace

研 究 生：洪秋榮　　Student：Chiou -Rung Hung

指導教授：陳俊穎　　Advisor：Jing-Ying Chen

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年八月

資源導向計算及虛擬工作環境的設計與實作

學生：洪秋榮　　指導教授：陳俊穎 博士

國立交通大學資訊科學與工程研究所

摘　　要

部落格以及維基百科等熱門網路應用程式的興起，意味著網際網路的發展趨勢正朝向一個大型的合作環境進行。在這個合作環境當中，使用者可以更輕易的同時扮演資訊提供者以及資訊使用者的角色。在可見的未來，當使用者可以在這環境中輕易的製作及分享更多樣的資源，從簡單的網頁到更複雜的軟體元件時，網際網路將成為一個能力更強的虛擬工作環境，能滿足更多不同背景的使用者的需求。然而，要實現這樣的虛擬工作環境，不僅僅要有健全的基礎架構來支援軟體元件的開發、佈署以及組合，同時還須提供讓一般使用者都可以接受的元件開發輔助技術。我們提出一個以資源為導向的架構來面對這些挑戰。在這個架構中，我們藉由使用者可定義的 XML 語法來描述各種不同的資源，而根據這些描述，相對應的動態服務可以在執行時期產生並加以管理。因此，在我們的虛擬工作環境中，不論是靜態的或是動態的服務組合，都可以透過連結這些不同資源的描述來達成。我們可以根據不同的問題領域，來建立適合的資源描述語法以及其對應的解譯與處理方法，來建造不同的網際網路應用程式。我們將會透過建立一個 e-Science 工作平台，來解釋如何運用我們所提出的方法幫助使用者更有彈性地使用分散各地的異質資源。

關鍵字: 網際網路服務, 服務導向架構, 服務組合, 軟體元件, Web 2.0

Resource-Oriented Computing: Toward a Universal Virtual Workspace

Student : Chiou-Rung Hung Advisors : Dr. Jing-Ying Chen

Institute of Computer Science and Engineering
National Chiao Tung University

ABSTRACT

The popularity of many emerging user-centric Web applications such as blogs and Wikipedia indicates the trend that the Internet is transforming into a global collaborative environment where most people can participate and contribute in ways not perceived before. It is foreseeable that when on-line resources created by and shared among people are not just simple Web contents but also more complex software artifacts, a more accommodating, universal, and virtual workspace may be formed that support people with much more diverse background and needs. To realize such goal, however, it requires not only comprehensive infrastructure support for the development, deployment, and assembly of diverse resources but also effective strategies and mechanisms that can handle the implied complexity. We propose a metaphoric, service-oriented architecture in which arbitrary resources are associated with syntactical descriptors, called metaphors, based on which run-time services can be instantiated and managed. Characterized as a universal development environment, our architecture permits static and dynamic service composition through syntactic metaphor composition. By defining metaphors for specific problem domains and by implementing suitable interpreters, either at a global scale or on a community basis, our architecture is intended to serve as the infrastructure holding multiple applications with substantial reuse. We demonstrate our idea by building a Web-based e-Science workbench, which allows user to utilize distributed computing and storage resources in a flexible manner.

Keywords: Web Services, SOA, Service Composition, Component, Web 2.0

誌　　謝

對於學位論文的完成，首先必須感謝我的指導教授陳俊穎老師，在研究所的兩年求學生涯中，總是耐心的給予我指導，並指引我正確的研究方向，對於思考解決問題的方法和態度上，也使我獲益良多；同時特別感謝口試委員林進燈教授與廖珠洲教授在百忙之中給予許多寶貴的指導與建議，使得論文的內容更加完備。

此外，感謝研究室一起奮鬥的伙伴們，亦超、宜涼，以及登揚、君翰、亦秋、景棠、嘉宏、嘉源諸位學長，以及學弟們，勝保、東潤，在研究進行時給與我許多的支持與鼓勵，並陪伴我度過研究生生涯。

最後，由衷地感謝我的家人，由於他們全力的支持，提供一個無後顧之憂的環境，讓我得以順利的完成學業，願將這份榮耀獻給我的家人。

洪秋榮 謹誌 2006 年 8 月

於交通大學協同合作實驗室

Table of Contents

摘　　要 ..... **i**

Abstract ..... **ii**

誌　　謝 ..... **iii**

Table of Contents ..... **iv**

List of Figures ..... **vi**

Chapter 1. Introduction - Motivation **1**

Chapter 2. Universal Virtual Workspace **4**

Chapter 3. An Abstract Service-Oriented Architecture **6**

Chapter 4. Metaphors for Resource Description and Composition **8**

4.1 Metaphor Format 8

4.2 A Component Model for Resource Composition 10

Chapter 5. Case Study: An e-Science Workbench **14**

5.1 The Workbench Architecture 14

5.2 Console User Interface 17

5.3 The Portal Interface 21

Chapter 6. Discussions and Related Work..... 28

Chapter 7. Conclusion..... 33

References..... 35

List of Figures

Figure 1. A resource-oriented architecture 6

Figure 2. An abstract component model 11

Figure 3. A file browser example..... 13

Figure 4. An e-Science workbench overview 14

Figure 5. Tasks and task types 17

Figure 6. A simple console-based client 18

Figure 7. A Deployment of our framework in a distributed environment 18

Figure 8. An ERP data analysis flow 21

Figure 9. Enterprise Portal Architecture Using WSRP..... 22

Figure 10. A typical usage scenario of WSRP..... 23

Figure 11. The whole portal page aggregated by the Jetspeed2 portal server 23

Figure 12. Adding a menu item in the project container 24

Figure 13. EEGLAB portlet 24

Figure 14. Use project manager portlet to control a workflow..... 25

Figure 15. Monitoring a workflow in action (debug mode) 25

Figure 16. Monitoring a workflow in action..... 27

Chapter 1. Introduction

The Internet has become indispensable nowadays as people rely on various kinds of Internet applications for their daily activities. In addition to performing personal tasks such as Web surfing or e-commerce activities, people also use the Internet to connect and collaborate with each other. Early Internet applications such as E-mail and FTP already provide basic information exchange mechanisms for people to collaborate. The Web also supports collaboration through information sharing, i.e. publishing personal Web pages and accessing others' Web pages.

Recently, the trend in user-centric collaborative computing has gained further momentum. Consider the widespread use of Web applications such as blogs and Wikipedia [1]. These applications provide easy-to-use interfaces that allow people to create contents such as opinions and photos for others to see. Equally importantly, they provide storage and content management facilities behind the scene to host these user-provided contents. Although the user interfaces are often limited (to increase ease of use), these applications already provide sufficient functionality people want. For example, blogs are often confined to certain Web page layouts prescribed by the designer, and people can only submit comments in terms of HTML documents. However, this simple design already provides sufficient functionality for most people wishing to share their thoughts. As a result, the simplicity helps these applications gain huge user base in a short period of time, which is commonly attributes by Web 2.0 [2] promoters as the network effect. Accordingly, it can be expected that further innovations along this trend will emerge in near future, due to their tremendous commercial potentials.

If the trend continues, one can envision many scenarios in which people create and share arbitrary resources – not just simple Web contents but also more complex software artifacts – on the Internet and make them accessible to others, making the Internet as an all-encompassing, universal, and virtual workspace (VW). In such a workspace, people can combine on-line resources flexibly in novel ways to form new applications for their own use.

Seemingly plausible, it remains questionable whether the virtual workspace vision will materialize and reach the mass audience. After all, the implied "programming" complexity is arguably best handled by highly skilled software engineers, who can hand craft complex

programming logics using powerful languages and tools. In fact, most users favor simplicity over flexibility, and they find novel uses of “primitive” tools that are simple in concept and easy to use, as long as they can achieve their goal.

Despite the feasibility concern, however, the Internet is in fact moving towards a universal virtual workspace gradually if we do not limit the audience to average users but also professionals. In the scientific community, for example, many e-Science portals (e.g. [3]) have been operational for a long time to help researchers perform scientific experiments and data analysis tasks on line. Frequently, when the scientists need to exercise new ways of data analysis and visualization schemes, they may write scripts using simpler languages without going further into the implementation details. However, when the available resources become insufficient or new tools and algorithms need to be developed, skilled programmers should still be handy.

From this perspective, it is more suitable to view the future VW as a complex ecosystem, where information and software artifacts are created and consumed among people with diverse background, skills, and needs. If the demands become substantial for tools that can help end users compose novel applications, developers with necessary background and skills may be motivated more. Similarly, new groups of people can emerge to help less skilled users create custom applications using resources available in the same VW. In other words, the ecosystem should promote both specialization and reuse, resulting in more efficient resource utilization when compared to custom application development practice that is more typical today.

In this thesis, we are concerned with requirements and challenges towards the VW goal, which requires not only robust infrastructure support for the development, deployment, and assembly of diverse software artifacts, but also effective strategies and mechanisms that can handle the implied complexity. Although many technologies exist today can relax some of the challenges, many issues remain to be tackled. For example, the Web Services [4, 5, 6] movement is pursuing a universal interoperability platform on top of which heterogeneous software systems can exchange information in terms of standard XML messages. However, the movement is developer-centric and standardizes only the minimal communication protocols among software systems. Aspects regarding service composition and hosting are either left for higher-level standards or to be supported by vendor-dependent mechanisms.

Instead of relying on existing distributed computing platforms, we believe it is beneficial to redesign a platform with the VW requirements in mind since the beginning, even if it is just for the sake of gaining better insights about what are essential and currently missing. One reason is that the WWW architecture, although powerful and indeed successful, are designed for information exchange rather than for VW. On the other hand, it is the simplicity that made WWW successful simply because it incurs much less efforts for companies and people to enter the WWW when compared to other mature or emerging, but less open and often heavy-weighted computing platforms.

We propose an abstract, protocol-based service-oriented architecture to overcome some of the obstacles mentioned thus far. Specifically, to handle the diversity in terms of problem domains and existing hardware/software resources, in our architecture, resources to be published over the Internet are associated with syntactical descriptors, called metaphors, based on which run-time services can be instantiated and managed. Because metaphors can be created and manipulated, and static and dynamic service composition can be achieved through syntactical metaphor composition, the resulting framework can be conceptualized as a universal development environment. By defining metaphors for specific problem domains and by implementing suitable interpreters on a community basis, the architecture is designed to serve as a common infrastructure hosting multiple P2P applications with substantial reuse.

To demonstrate our approach and to address many issues raised when tackling the VW challenge, we have also been developing an e-Science portal which allows user to utilize distributed computing and storage resources via familiar WIMP-style user interfaces. The rest of the thesis is organized below. In chapter 2 we clarify the concept of VW and reason about the infrastructure requirements. In chapter 3 we describe a resource-oriented architecture that is designed with the requirements described in chapter 2. In chapter 4 we describe the generic resource description language used to denote and/or annotate diverse resources. In chapter 5 we describe case studies employing the architecture. In chapter 6 we discuss additional characteristics and issues related to UVM in general, and also related work in this section. Finally, in chapter 7 we conclude the thesis.

Chapter 2. Universal virtual workspace

The vision of a universal, virtual workspace is inspired by the concept of “intercreativity” envisioned by Tim B. Lee [7] when architecting the WWW architecture, upon which people collaborate by creating and posting Web contents for others to see. This together with the emerging service-oriented computing trend have led us to the conclusion that Internet is transforming into a common medium for people to participate in, rather than just a consumer-producer platform where most people are restricted to access information and services provided by software developers.

The term workspace reflects the functional requirements for the user-centric VW. First of all, end users should be equipped with some kinds of virtual workbenches for personal use, just like using Web browsers to surf the Web. These workbenches should provide users with common facilities so that they can browse and assess available resources across the Internet. To support arbitrary problem domains that may require highly interactive user interfaces or larger network bandwidth, which are essential for scientific data analysis and visualization, both the types of resources and the communication channels among them should not be limited (e.g. to HTTP). Furthermore, it should be possible for users to access resources using interfaces with different capabilities under different circumstances.

In addition to accessing information and service in familiar, client-server ways, end users should also be able to combine resources into useful tools to help solving their own problems. Mechanisms for resource composition can vary dramatically requiring different degrees of programming skills; the resources to be assembled may be scattered across the Internet and can only be accessed through some kinds of communication channels. Certain virtual workbenches may also enable users to play the provider role such that the resources they create can become accessible to others. In short, virtual workbenches present end users the illusion of a software development and assembly environment, yet they are interchangeable in the sense that the resources being created or assembled remain independent even when the workbench changes.

The VW should also be a universal resource deployment platform. While some resources may only be accessed remotely, others may be downloaded into user’s machine and executed there locally. This capability is essential to permit sharing and reuse of highly interactive

GUIs without relying on proprietary deployment technologies, such as those supported by modern Web browsers.

When collaboration among people is concerned, VW should also support different collaboration schemes without “unnatural” workarounds. Possible collaboration schemes may range from real-time instant messaging or video conferencing, to event-driven workflows that schedule tasks to be performed among members in a team (e.g. for a development project).

Apart from the functional requirements outlined above, which already impose many design constraints when engineering a desirable architecture for VW, the “non-functional” requirements hinted by the adjectives “universal” and “virtual” stress that some form of virtualization over heterogeneous, dynamic, and distributed resources should be present, and be achieved through simple and universally accessible communication protocols. This essentially follows the hour-glass model that contributes to the success of the Internet and the Web.

Furthermore, to be scalable, and to promote community forming and resource sharing, the architecture should also adopt P2P concept and unite diverse resources in a decentralized manner. However, the architecture should permit multiple P2P applications running on the same set of peers. In other words, each peer may be overloaded with facilities supporting multiple P2P applications, ideally with substantial degree of reuse.

Chapter 3. An abstract service-oriented architecture

We propose a service-oriented architecture aiming at fulfilling the VW requirements described in the previous chapter. As shown in Figure 1, the architecture rests on a resource space in which resources are arbitrary artifacts identified with globally unique URIs. Different types of resources may contain information in binary or text forms. Because some resources such as Web pages may embed references to other resources, the resource space hence forms a global, interconnected resource network, much like the hyperlinked WWW. For the purpose of this thesis, it is sufficient to regard each resource as a file hosted inside a network addressable machine.

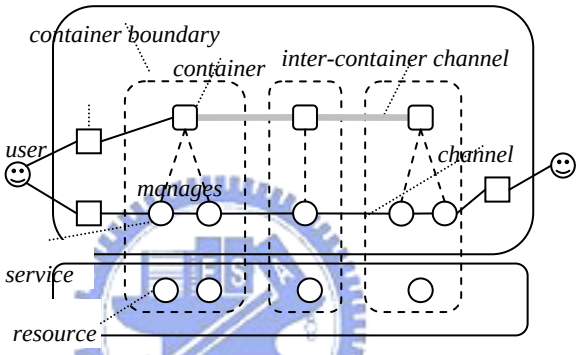


Figure 1. A service-oriented architecture

On top of the "persistent" resource space is the dynamic service space comprise of mutually interacting run-time objects, called services, which perform tasks upon requests. More importantly, each service is associated with exactly one resource, although the reverse is not necessarily true. Resources are partitioned into disjoint sets with respect to the authority component of their URIs, and a partition may be associated with a manager service, called container, which governs the definition, instantiation, customization, composition, and other lifecycle activities for the services within the partition. Note that although a container itself is also a service and hence is associated with a resource, how the container is instantiated is considered system-dependent.

Within the service space, it is possible that a service can access another service directly if both are in the same memory space; otherwise some kinds of communication channels should be established between them. In either case, the container is responsible of establishing suitable links among the services in question, rather than allowing the managed services to

establish links on their own.

Containers can be implemented differently, and they join and leave the service space continuously. Containers can be long running at server side accepting requests, or transient at client side interacting with users. A container may offer different levels of management capabilities to different classes of users – although there are basic operations all containers need to support.

To complete the architecture, we refer to clients that serve as intermediaries between end users and the services as agents, which are not managed by containers. Note that some agents and services may be equipped with GUIs interacting with users directly.

Chapter 4. Metaphors for resource modeling and composition

4.1. Metaphor Format

In the service-oriented architecture outlined previously, it is required that services are instantiated based on the resource contents. Containers are responsible of interpreting the resource contents and determining what services to create and how they are customized. When references to other resources are embedded inside a resource, the container may establish required connections among the respective services, according to the implied semantics implied by the resources (see below)..

Although resource interpretation and composition are container-specific or even proprietary, to increase the overall reusability and interoperability and to minimize unnecessary fragmentation, it is required that there exists a canonical resource description format to define new resources or annotate existing ones. These resource descriptors, called metaphors, are syntactical objects with structures derived from the contents of the resources they describe, respectively. A further requirement is that it should be possible to "syntactically" check a particular resource composition in a semantics-neutral manner, before actual run-time composition takes place.

Naturally, we choose a subset of well-formed XML documents as the canonical metaphor representation . In general, a metaphor can contain arbitrary structure without restrictions. However, we allocate a special namespace with some reserved elements and attributes. These meta-level constructs can be embedded inside metaphors and are used to constrain their structures without semantic implications. For simplicity, in what follows the prefix "m" is assumed to be bound to the meta constructs namespace. The reserved keywords are summarized below:

m:uri is an attribute whose value is a relative or absolute URI. When the root element of a metaphor has such an attribute, the value reflects the (intended) identity of the metaphor. When such an attribute is absent in the root element, the identity of the metaphor should be inferred from the context. More importantly, in case the attribute occurs elsewhere, it always serves as a reference to another existing resource.

m:is is an element that occur only at the top level, and its "m:uri" attribute should link to

another metaphor - which we refer to as the type of the metaphor - that may contain additional contents and further constraints. When a metaphor includes a type this way, it not only inherits all contents from the type, but also should conform to the constraints imposed by the type. When the type contains other types, the contents and constraints are collected recursively. In addition, when a metaphor contains multiple types, it should conform to all the constraints implied. As constraints may conflict with each other, we only consider valid types such that their inferred constraints do not conflict. In addition, metaphors trying to conform to conflicting types are considered invalid.

m:rel is also an element that occur only at the top level and is used to constrain the allowed links among metaphors. Syntactically, it states that the enclosing metaphor can have a certain number of child elements of a particular tag in its top level. The tag name and the multiplicity are prescribed by the “tag” and “card” attributes, respectively. Furthermore, the “m:uri” attribute also constrains the type of the metaphor can be linked to.

The m:elem element constrains the contents of metaphors. It states that a conforming metaphor can have child elements with element name specified by the “tag” attribute and their number of occurrences specified by the “card” attribute. The m:attr child elements, if any, also indicate the allowed attributes and their value types. In addition, m:elem can be declared recursively.

In other words, metaphors together form an ontology, or more precisely, multiple but possibly overlapping ontologies within the VW. Depending on the situations, precise semantics for a given ontology can also be defined in a foreign way. Metaphors can be created to annotate other actual resources, or they can be the resources themselves. In fact, is common to create metaphors to capture high-level concepts without explicitly describing their semantics, and use them immediately to annotate other existing resources. Unlike schema languages such as DTD and XML Schemas, metaphors mix meta-descriptors, links, and data contents together. Our goal is to confine ourselves to a subset of well-formed XML elements that are simple, sufficiently expressive, extensible, yet easy to check for consistency and validity. Therefore, we do not follow existing ontology languages such as RDF and OWL.

To illustrate the use of metaphors and schemas, consider a CASE environment we developed based on Eclipse. One of the core functionalities of the CASE environment is to support flexible design annotation: developers can create metaphors to annotate other

resources. Consider a scenario in which a developer wishes to document the design patterns used in his/her design, which may be embedded implicitly in the implementation source code. Suitable schemas can be created for various design patterns, respectively. The example below shows a possible schema definition that characterizes the Abstract Factory pattern [8].

```
<AbstractFactory m:uri="/pattern/AF">
  <m:rel tag="factory" type="/java/AbstractClass"/>
  <m:rel tag="product" card="1..*" type="/java/Interface"/>
  <m:rel tag="concreteFactory" type="/java/AbstractClass"
  <m:rel tag="concreteProduct" card="*" type="/java/Class"/>
</AbstractFactory>
```

The example below shows an application of the Abstract Factory pattern that conforms to the schema defined previously:

```
<myGUI>
  <m:is m:uri="/pattern/AF"/>
  <factory m:uri="/proj/src/gui/Factory.java"/>
  <product m:uri="/proj/src/gui/Widget.java"/>
  <product m:uri="/proj/src/gui/ScrollBar.java"/>
  <product m:uri="/proj/src/gui/Button.java"/>
  <concreteProduct m:uri="/proj/src/gui/ms/ScrollBar.java"/>
  <concreteProduct m:uri="/proj/src/gui/ms/Button.java"/>
</myGUI>
```

The example above is simple-minded because inconsistency can occur. However, for documentation purpose it suffices and the user is responsible of ensuring the consistency. On the other hand, more sophisticated schemas can be designed to fix potential conflicts, by introducing additional schemas. Also, more sophisticated GUIs to help developers create and maintain schemas and annotations can be supported.

As mentioned previously, there are some common protocols containers need to support. Here, specifically, all containers need to reply the metaphors of the resources they manage. Accordingly, it is straightforward for people to publish their own ontologies for the problem domains that concern them. On the other hand, public ontologies can also be created and maintained on a community basis so that containers or service developers joining the same community can interoperate and communicate.

4.2. A Component Model for Resource Composition

Because service composition is done through metaphor composition, our architecture serves not only as a distributed computing platform but also as a global software development environment. Because metaphor syntax and semantics are domain- and container-specific, there is no restriction about how services should be instantiated, assembled, or managed, except that the containers perform these tasks based on the semantics intended by the metaphor designers. This feature is essential to fulfill the requirement that the platform can

support users of varying degree of skills. However, reusability and interoperability can be compromised when, for example, multiple, overlapping but incompatible languages are created.

What we want is a virtual component assembly platform that promotes component reuse. We adopt a component model that "encourages" manager-worker separation by ensuring that components are workers who concentrate only on the work they are designed for without worrying about how their supporting "colleagues" are created and accessed – all these tasks are the responsibility of the manager, thus making components more focused and reusable. As shown in Figure 2, a component is encapsulated inside its interface through which clients can interact with. The interface can further be divided, at least conceptually, into primary and customization interfaces. The former represents primary function the component provides, while the while the latter is used for configuration purpose before or when the component becomes functioning.

Note that the notion of component is abstract and is applicable not only to individual services, but also to groups of services. A composite component can be modeled as a set of collaborating services. Accordingly, the internal of a component is also divided into two parts – the subcomponents being used, and the connecting substance among these components. Naturally, we refer workers to the former while managers to the latter. Workers are well encapsulated and unaware of the actual functionality of the containing component.

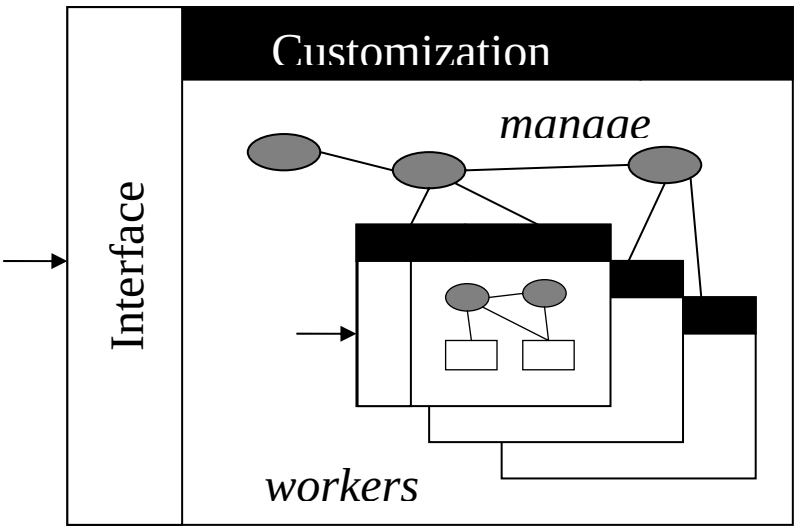


Figure 2. An abstract component model

To illustrate service composition and templates, consider a simplified file browser that

consists of basic components FileSystem, FileTreeView, and FileView. Related templates are given below:

```
<FileSystem.tmpl m:uri="/fs/FileSystem.tmpl">
  <m:is m:uri="/fs/FileSystem"/>
  <m:is m:uri="/java/JavaObject"/>
  <class name="fs.FileSystem"/>
  <m:rel tag="root" m:uri="/java/File"/>
</FileSystem.tmpl>
<FileTree.tmpl m:uri="/fs/FileTree.tmpl">
  <m:is m:uri="/fs/FileTree"/>
  <m:is m:uri="/gui/GUI"/>
  <m:is m:uri="/java/JavaObject"/>
  <class name="fs.FileTree"/>
  <m:rel tag="filesys" m:uri="/fs/FileSystem"/>
</FileTree.tmpl>
<FileView.tmpl m:uri="/fs/FileView.tmpl">
  <m:is m:uri="/fs/FileView"/>
  <m:is m:uri="/gui/GUI"/>
  <m:is m:uri="/java/JavaObject"/>
  <class name="fs.FileView"/>
  <m:rel tag="fileSys" m:uri="/fs/FileSystem"/>
</FileView.tmpl>
<Layout.tmpl m:uri="/gui/Layout.tmpl">
  <m:is m:uri="/gui/GUI"/>
  <m:is m:uri="/java/JavaObject"/>
  <class name="gui.LayoutService"/>
  <m:elem tag="layout">
    <m:attr name="style" type="String"/>
  </m:elem>
</Layout.tmpl>
```

Templates can be defined by developers or experienced users, even during run time. Once accepted by the container, instances of templates can be created. To compose a file browser at run time, for example, the following metaphors need to be created in proper places and conform to the templates stated above:

```
<fileBrowser m:uri="/user/fileBrowser">
  <m:is m:uri="/gui/Layout.tmpl"/>
  <layout style="hsplit">
    <left m:uri="./fileTree"/>
    <right m:uri="./fileView"/>
  </layout>
</fileBrowser>
<fileTree m:uri="/user/fileTree">
  <m:is m:uri="/fs/FileTree.tmpl"/>
  <filesys m:uri="./fileSystem"/>
</fileTree>
<fileView m:uri="/user/fileView">
  <m:is m:uri="/fs/FileView.tmpl"/>
  <filesys m:uri="./fileSystem"/>
</fileView>
<fileSystem m:uri="/user/fileSystem">
  <m:is m:uri="/fs/FileSystem.tmpl"/>
  <root m:uri="./root"/>
</fileSystem>
```

The composition example above only instantiates components and arrange the layout of the GUI ones. There is no connection among components. In this example, what is needed is that selection of a tree node in the FileTreeView component should cause the included FileView component to read data from FileSystem and display it. To achieve the goal, some rule-based composition logics can be supplied. For example, the instantiation of FileTree above is augmented with such rules, which essentially say that whenever FileTree responds to

change of selection from the user, it will issue a metaphor event (of some type published elsewhere), and the rule-based composition simply pattern matches the event and issue commands to other components.

```
<fileTree>
 <m:is m:uri="/fs/FileTree.tmpl"/>
 <filesys m:uri="./fileSystem"/>
 <env name="fview" m:uri="./fileView"/>
 <events>
   <rule>
     <if>
       <valueChanged>
         <file path="@path" folder="true"/>
       </valueChanged>
       <rw:call name="fview">
         <setText>This is a folder</setText>
       </rw:call>
     </if>
     <ok/>
   </rule>
   <rule>
     <if>
       <valueChanged>
         <file path="@path"/>
       </valueChanged>
       <rw:ctx name="fview">
         <viewFile path="@path"/>
       </rw:ctx>
     </if>
     <ok/>
   </rule>
 </events>
</fileTree>
```

Without going further into the details about the syntax and semantics of the composition format, which is application- and domain-specific, Figure 3 shows the result of such composition.

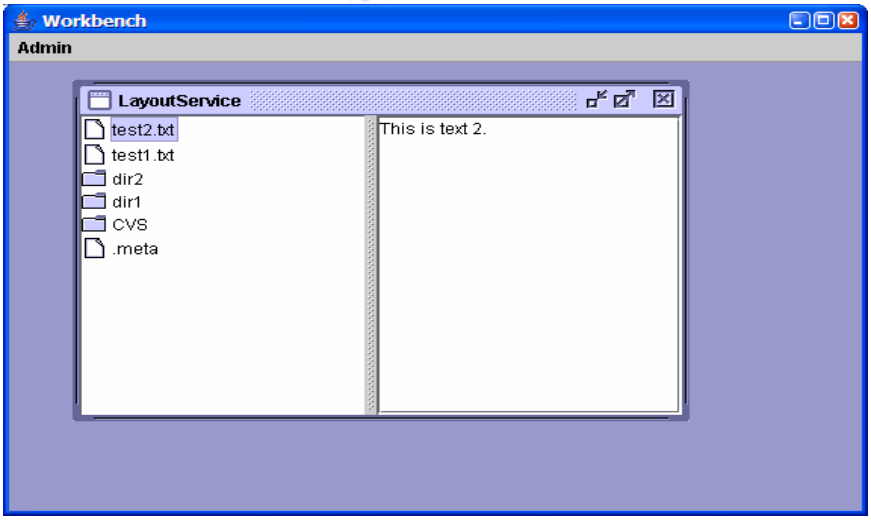


Figure 3. A file browser example

Chapter 5. Case Study: An e-Science Workbench

We have been developing an e-Science workbench as a proof of concept for the VW goal. The system is part of our efforts to develop a service-oriented infrastructure uniting heterogeneous, disparate computing resources across the Internet. The initial goal of the e-Science workbench is to develop a service-oriented version of an existing Matlab-based data analysis toolkit used by the Brain Research Center of NCTU. In the beginning, a single-machine version is developed. The required computing resources include Matlab [9] for its basic computational support as well as the toolbox EEGLAB [10] developed by UCSD. To wrap the Matlab computing power, as our implementation is Java-based, we use the open-source library, JMatLink [11], to work between the container and the Matlab engine.

5.1. The Workbench Architecture

The architecture of the e-Science workbench is depicted in Figure 4. As shown in the figure, one important objective of the system is to support multiple types of user interfaces for users to access the same set of resources and to perform flexible, dynamic service access, customization and composition.

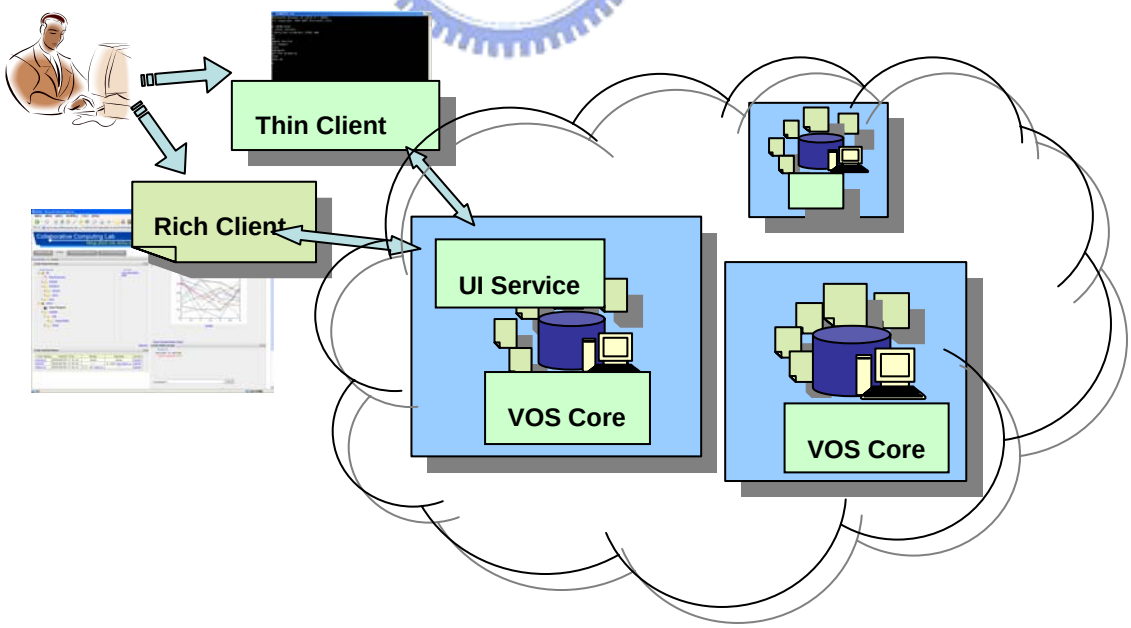


Figure 4. An e-Science workbench overview

Behind the user interfaces are a set of collaborating containers hosting various types of computational and storage resources. Although the containers form a P2P network, each

container may provide different set of resources. That is, each container may allow its user to define and create new resources in a way depending on the available administration interfaces provided by the container.

More complex analysis flow can be grouped into workflows in the e-Science workbench, and the processing logic can be performed by some workflow engine services. Workflows are in fact an important form of composition (i.e. process-oriented composition), especially in business context.

A workflow consists of multiple tasks that are interrelated with each other via control flow and data flow, in a form similar to common programming languages. In our case, workflows are expressed as XML documents with corresponding run-time workflow services. The enactment of a workflow is encapsulated in a process that contains the workflow instance and a scheduler, in which the process is responsible of maintaining the progress of the workflow while the scheduler decides which task processor a task is assigned to whenever the task is ready for processing. All these elements, that is, workflows, workflow instances, processes, schedulers and task processors are resources with corresponding metaphors in our system. A simple echo process is illustrated as follows:

```
<echo.proc>
  <m:is m:uri="//meta/java/JavaObject"/>
  <m:is m:uri="//meta/service/Process"/>
  <class name="cclab.grid.core.GridProcess"/>
  <flowInst m:uri="./echo"/>
  <scheduler
    m:uri="/bci/test.ps/go/grid/scheduler/simple.scheduler"/>
  <m:child name="echo">
     <echo.fi>
       <m:is m:uri="/task/FlowInstance"/>
       <flow m:uri="/bci/test.ps/go/taskflow/test/echo.flow"/>
       <var name="x" type="int" value="1000"/>
       <var name="y" type="int" value="2000"/>
       <var name="z" type="int" value="3000"/>
     </echo.fi>
  </m:child>
</echo.proc>
```

Note that what the metaphor states is that when the container interprets the resource and recognizes that it is a type identified with URI “//meta/java/JavaObject”, it can instantiate a Java object – the service associated with the resource – using the class named “cclab.grid.core.GridProcess” obtained from the metaphor content. The metaphor identified by “//meta/java/JavaObject” is also shown below:

```
<JavaObject>
  <m:elem tag="class">
   <m:attr name="name" type="String"/>
  </m:elem>
</JavaObject>
```

When the service associated with the echo process resource is created, it needs two component services; namely, the flow instance and the scheduler, which are indicated below. The flow instance in turn requires a workflow. All the references are represented as related URIs.

```
<simple.scheduler>
  <m:is m:uri="//meta/service/Scheduler"/>
  <m:is m:uri="//meta/java/JavaObject"/>
  <class name="cclab.grid.scheduler.SimpleScheduler"/>
  <grid m:uri="../test.grid"/>
</simple.scheduler>

<echo.fi>
  <m:is m:uri="/task/FlowInstance"/>
  <flow m:uri="/bci/test.ps/go/taskflow/test/echo.flow"/>
  <var name="x" type="int" value="1000"/>
  <var name="y" type="int" value="2000"/>
  <var name="z" type="int" value="3000"/>
</echo.fi>

<echo.flow>
  <m:is m:uri="/task/Flow"/>
  <flow>
    <var name="x" type="int"/>
    <var name="y" type="int" default="2000"/>
    <var name="z" type="int" default="3000"/>
    <task type="/task/type/EchoTask">
      <processor m:uri="bci/test.ps/go/grid/processor/a"/>
      <bind name="in" var="x"/>
      <bind name="out" var="y"/>
    </task>
    <task type="/task/type/EchoTask">
      <bind name="in" var="y"/>
      <bind name="out" var="z"/>
    </task>
  </flow>
</echo.flow>
```

As also being indicated above, a workflow may contain several tasks that are mutually related. Each task in the echo workflow above is associated with a task type that contains a set of parameters bound to the workflow's variable, and each of these parameters can be an input parameter, output parameter, or in-out parameter. You can assign a task processor for each task in the workflow, or a scheduler will assign one automatically according to its internal scheduling policies and algorithms.

Figure 5 shows the class diagram describing our implementation of the task type. In short, new task types can be created as Java classes that implement the ITaskType interface. Metaphors denoting these task types can be created afterwards and referenced in flows. Whenever a task is ready to proceed, the task type (service) will take all the input (XML) parameters passed from the process, process the inputs, and return (XML) outputs back to the process. The bindings of flow variables with the input/output parameters are indicated in the flow and handled by the process.

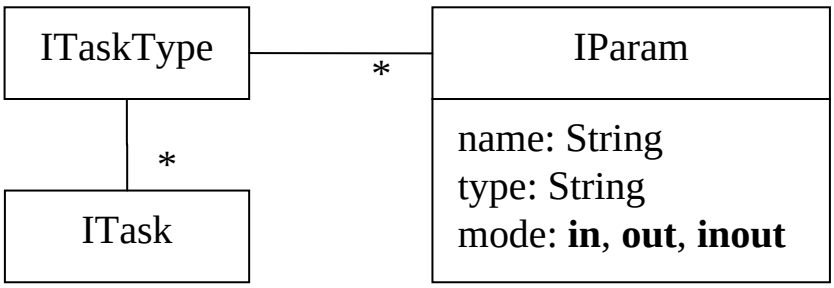


Figure 5. Tasks and task types.

5.2. Console User Interface

The text-based console interface is in fact the required interface because it permits many advanced and privileged management tasks, such as the creation of metaphors. To illustrate the use of the console, we consider an experiment that uses single-trial event-related potential (ERP) to recognize different brain potentials with five degrees of drowsiness. We will show how to design a resource relative to a function in the experiment with metaphor and its interpreter and then use workflow to composite the resource to perform the entire experiment.

With the support of JMatLink we can already perform the function of the experiment mentioned above, if we know the exactly correct command format for the function. But this will not be an acceptable and user-friendly user interface. With the workbench, we need to design metaphors for this basic function to increase reusability and user friendliness. Consider a basic function, called “loadset”, that is provided by EGLAB. Through JMatLink, the function can be invoked in Java as shown below:

```
JMatLink engine = new JMatLink();
engine.engOpen();
engine.engEvalString(“EEG = loadset(...);”);
```

To incorporate EEGLAB’s “loadset” function into our system, for example, we would like to have a service that handles requests in terms of XML (i.e. using XML as the standard message format) and translates them in the form recognized by JMatLink. A simple metaphor can be defined for the purpose:

```
<Loadset>
  <m:is m:uri="//meta/java/JavaObject"/>
  <class name="eeglab.Loadset”/>
</Loadset>
```

Whereas the Loadset class implementation is given below:

```
public class Loadset implements IMetaphorService
{
  public IMetaphor process(IMetaphor m) {
    if (!m.isTag(“loadset”)) return null;
```

```
    JMatLink engine = ...; // get engine service elsewhere
    String cmd = convertCommand(m);
    engine.engEvalString(cmd);
    return Metaphor.OK;
  }
}
```

In this case, the Loadset service can interpret input messages and dispatch them to the Matlab engine. Note that we also use metaphors to represent XML messages in our implementation, and services that process metaphors are modeled as IMetaphorService.

The metaphor and service above represent one of the resources managed by the container. How to access these resources depends on the available clients, which themselves can also be services managed by the container. For example, below is a “command-line” control service that serves as a client to invoke other resources.

```
<loadset>
  <m:is m:uri="/bci/core/EEGLabControl"/>
  <command tag="loadset"/>

  <m:is m:uri="//meta/java/JavaObject"/>
  <class name="cclab.matlab.eeglab.ctrl.LoadsetControl"/>

  <commandTemplate command="loadset">
    <loadset path="Path_Arg"/>
    <Path_Arg>
      <prompt label="Please input data set file path: "/>
      <default type="String"/>
    </Path_Arg>
  </commandTemplate>
</loadset>
```

The control is among the other similar objects managed by yet another console-based command dispatcher. Specifically, the dispatcher recognizes metaphors that have the type "/bci/core/EEGLabControl” and includes the associated control object (here a “LoadsetControl” object) into its own control pool. The dispatcher receives user’s requests as plain texts (in console mode), translates them into metaphors, and dispatches them to proper controls according to the filter information provided in the control metaphor. In the example above, the “loadset” control will accept metaphors with tag <loadset> and interacts with the user using a sequence of questions, as illustrated in Figure 6 below. It help user to complete the required parameters for the loadset function.

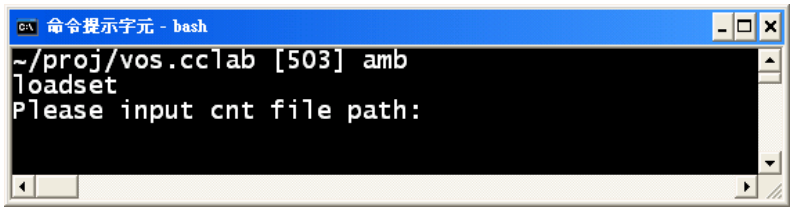


Figure 6. A simple console-based client

Similarly, we can design other resources/metaphors for the other functions needed in the EEGLAB analysis process, and put these resources in a container named Matlab container. Besides these, we need some other services for user to manage their resources.

To conduct an EEG signal data analysis, we first record the raw data from the instrument first, and then send the data to a computer with higher computational power for processing. Accordingly, we deploy our e-Science workbench in a distributed system that contains these two kinds of containers: project containers that maintain raw and analysis data, and Matlab container containing Matlab computing resources. As shown in Figure 7, all these containers have a common core called VOS that serves as the coordination engines, and through the console we can remotely control this container to perform actions. Interestingly, the VOS core is a resource tree that contains common definitions (metaphors) that are kept identical for each container in our implementation.

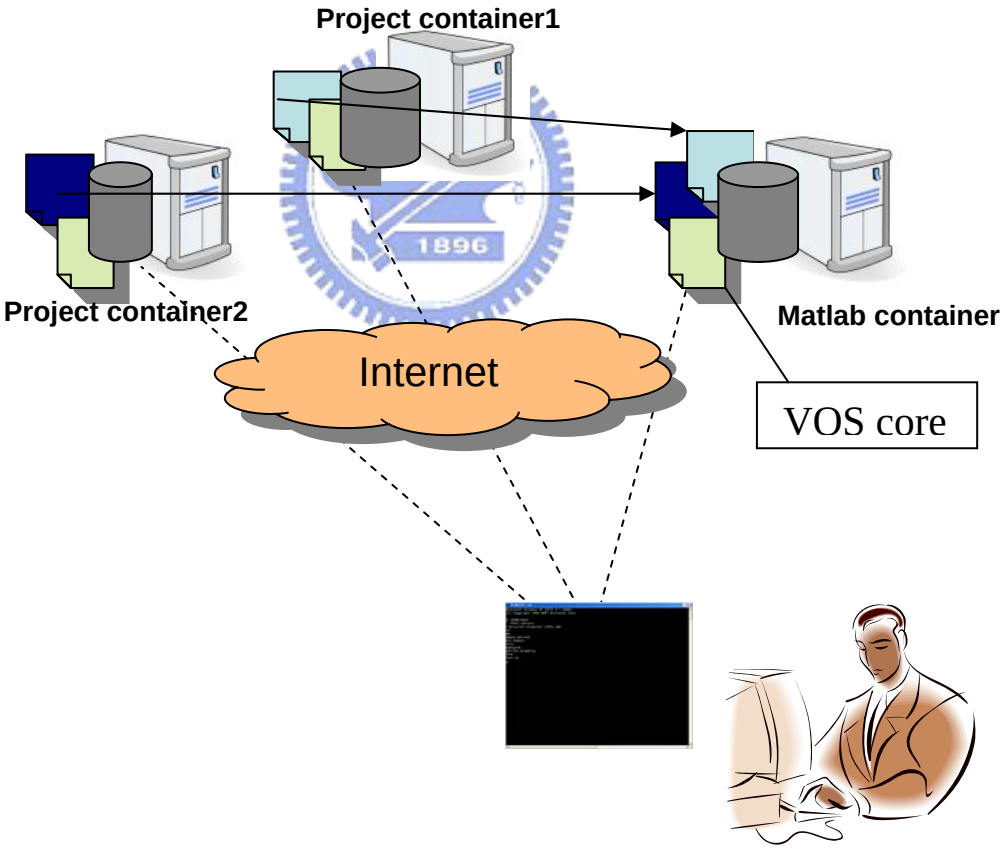


Figure 7. Deployment of our framework in a distributed environment

Through the console, we can manipulate the experiment process step by step manually. Using the services supported by the project container, we can send data to and obtain data from the Matlab container; using the services supported by the Matlab container we can

perform the analysis we need. In case the communication can be done through metaphor exchange, as in our implementation, standard channels among these services can be established. In case higher performance data transmission is required, special channels such as FTP combined with data compression techniques can be used.

Besides running manually the complete entire experiment step by step, we can also group required activities in a workflow and run it automatically. We break the analysis experiment into separate tasks such that each task takes an input, processes the input, and produces the output for the next task. As mentioned before, each task type can be defined and denoted as a metaphor to be used inside workflows. For example, below shows a task type declaration for the same "loadset" function described previously. Instead of being a function to be invoked by other clients directly, here it becomes "asynchronous" in the sense that the workflow engine should interpret such a task and invoke the "loadset" function at a proper time with required input/output data.

```
<LoadsetTask>
  <m:is m:uri="//meta/service/TaskType"/>
  <m:is m:uri="//meta/java/JavaObject"/>
  <class name="eeglab.LoadsetTaskType"/>
</LoadsetTask>
```

For illustration purpose, below we show a simple workflow that includes the Loadset task.

```
<loadset.flow>
  <m:is m:uri="/task/Flow"/>
  <flow>
    <var name="loadset" type="String" default="…"/>
    <var name="use_engine" type="String"/>
    <task type="./LoadsetTask">
      <bind name="path" var="loadset"/>
      <bind name="engine" var="use_engine"/>\
    </task>
  </flow>
</loadset.flow>
```

In this example, the overall flow requires two input variables that should be filled before it is activated, and the information will be passed to the Loadset task accordingly. The Loadset task will perform a call to the Loadset service described previously. The duty of LoadsetTask is to collect the needed parameters for the relative function in EEGLAB and send these information as metaphors to Loadset service for further process.

Figure 8 sketches a complete workflow that performs an ERP experiment for the study of drowsiness.

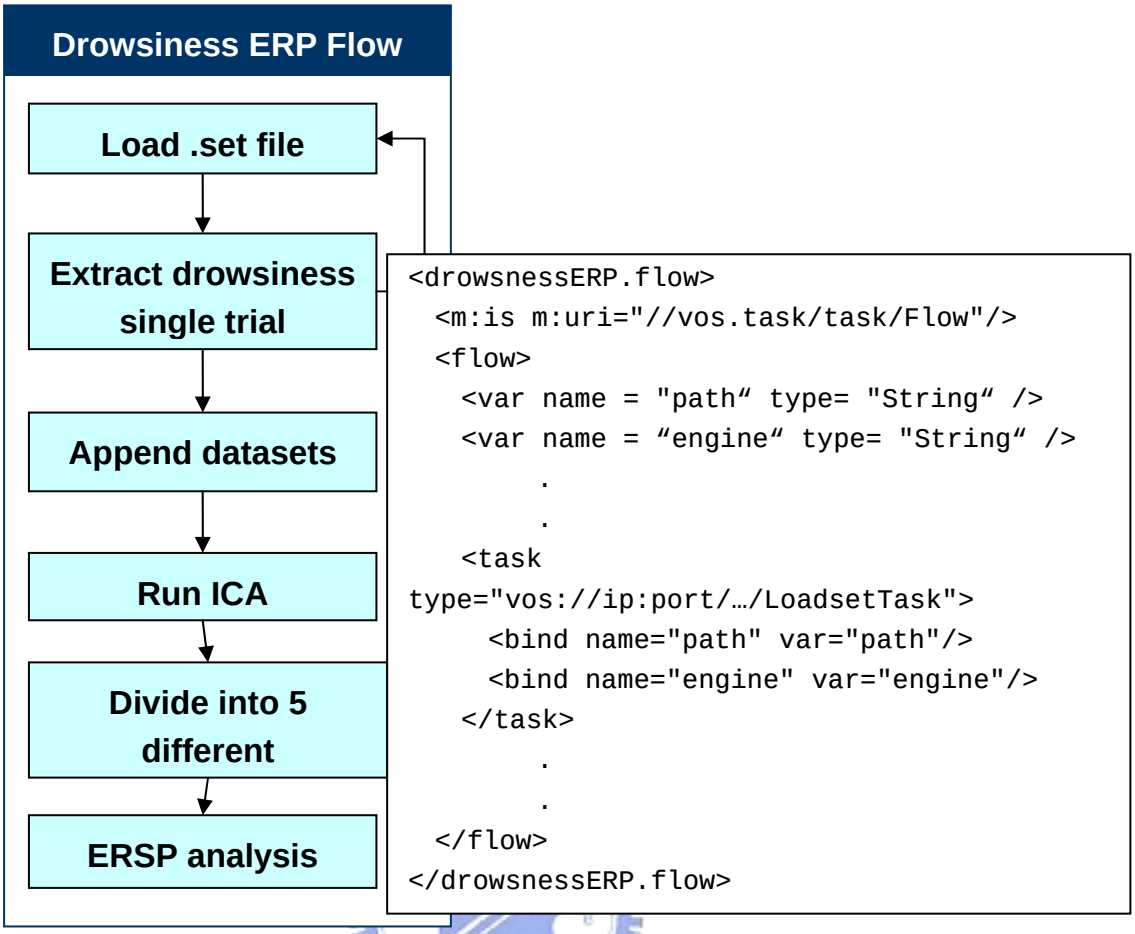


Figure 8. A ERP data analysis flow

5.3. The Portal Interface

The operations described previously can all be performed through consoles, locally or remotely, because what is needed is to interact with containers with proper metaphors. On the other hand, another major goal of our system is to provide similar functionality through more user-friendly, Web-based interfaces. For this purpose, we implement a portal interface using Java technologies, i.e. JSR 168 [12] and WSRP [13] (Figure 9). In general, a portal is a web site that acts as a point of entry or a gate to a large system.

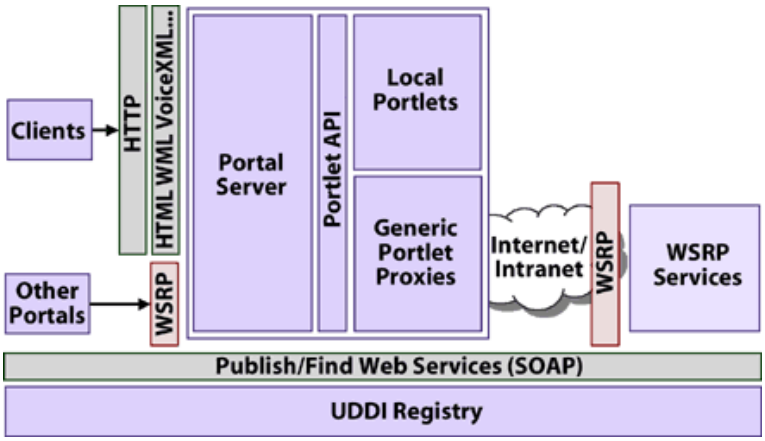


Figure 9. Enterprise Portal Architecture Using WSRP

As shown in Figure 9, a portlet is a Java-based web component that processes requests from clients and generates dynamic markup fragments. The portal server here manages the life cycle of portlets and aggregates markup fragment generated by portlets into complete portal pages. Also, portlets are not directly bound to a URL. Instead, a client interacts with portlets through a portal server. The markup fragments generated by portlets are not restricted to HTML, but depend on what kind of markup language supported by the browser used by the client. For example, WAP phones typically use WML while the well-known PC web browsers use HTML. Java Specification Request (JSR) 168 enables interoperability among portlet and portal server. More and more portal vendor claim that their product is completed compliant with JSR 168. As a result, more and more portlets can be reused again.

The WSRP specification is a product of the OASIS, and it enables interactive, presentation-oriented Web services [4] to be easily plugged into standards-compliant portals. The WSRP service in Figure 9 must implement the required interface defined in WSRP standard, and generate markup fragment packaged in SOAP [5] message. A WSRP standards-compliant portal can send clients' request to WSRP services and aggregate markup fragments from WSRP services through the common interface defined in WSRP. A typical usage scenario is shown in Figure 10. In the figure, a WSRP producer contains WSRP portlets in it, and performs the role WSRP services in Figure 9, and WSRP consumer is always a portal server.

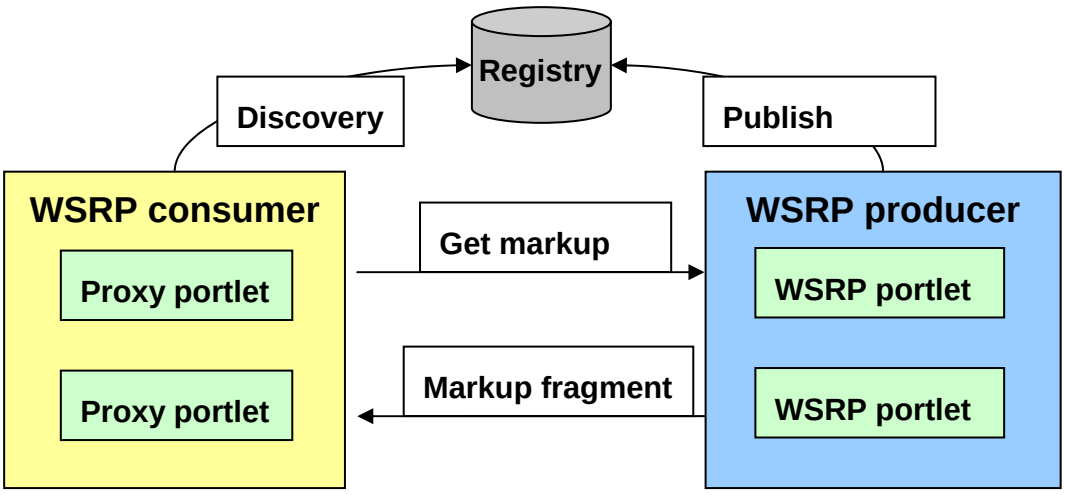


Figure 10. A typical usage scenario of WSRP

We choose Jetspeed2 from the Apache project (www.apache.org) as our portal server, and we have already designed a set of portlets that provide a Web-based user interface for the e-Science workbench. The whole portal page aggregated by the portal server is shown in Figure 11. There are five portlets in this portal page, including the project manager portlet, the image viewer portlet, the task flow viewer, the Matlab console, and the VOS console.

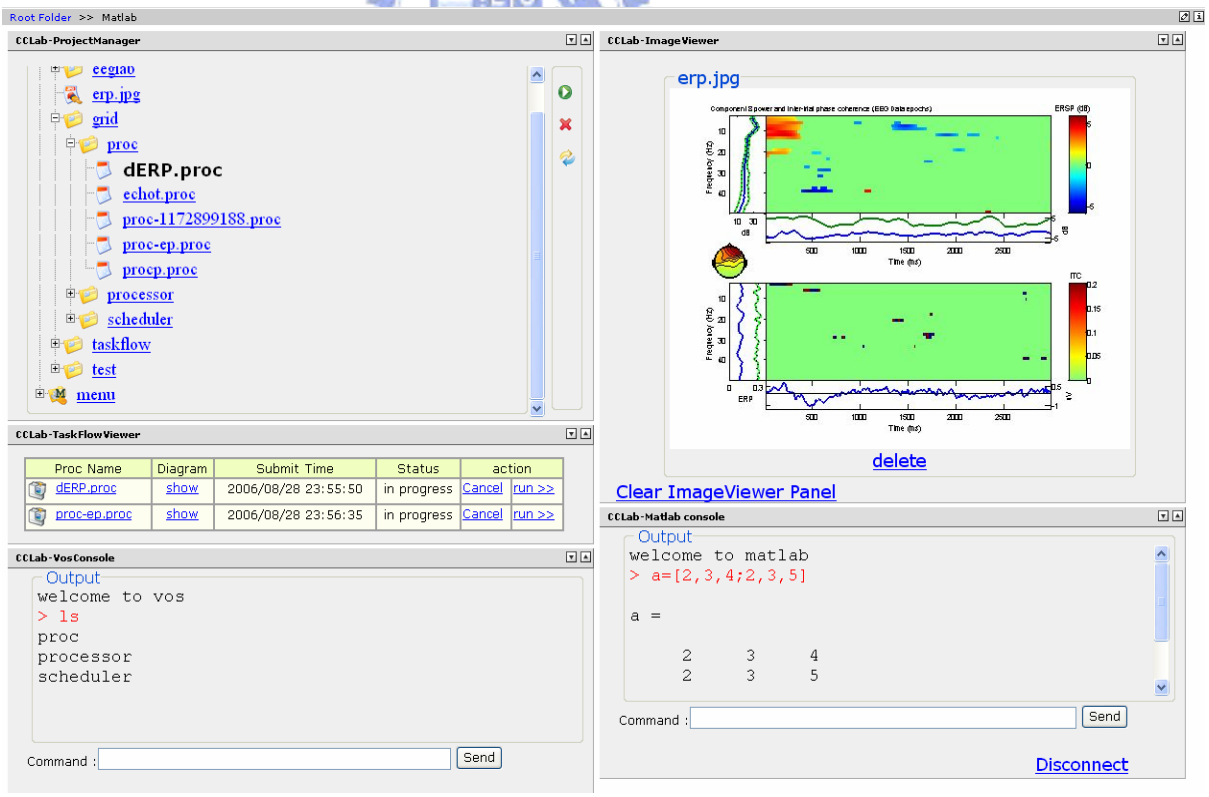


Figure 11. The portal page aggregated by the Jetspeed2 portal server

As shown in Figure 12, a user can modify the resource for the project container to add

some frequently used services as a menu item in project manager portlet. The image viewer portlet gives an easy way for user to view the graphical results online. The task flow viewer portlet shows a list of submitted jobs. The Matlab console portlet lets user access Matlab like using Matlab in its console mode. Finally, the VOS console portlet lets us manipulate VOS like what we did in previous section.

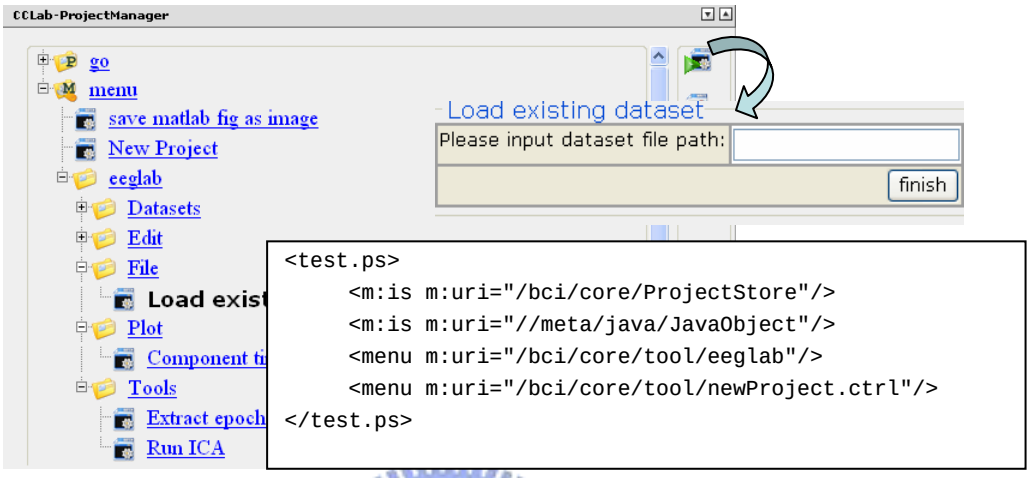


Figure 12. Adding a menu item in the project container

In Figure 12, we add some frequently used function as a menu item for later use, which also separates the function from the actual service. Take the EEGLAB function in Figure 13 as an example, you do not need to know where the service is, and you can simulate the manipulation in EEGLAB environment in the portlet.

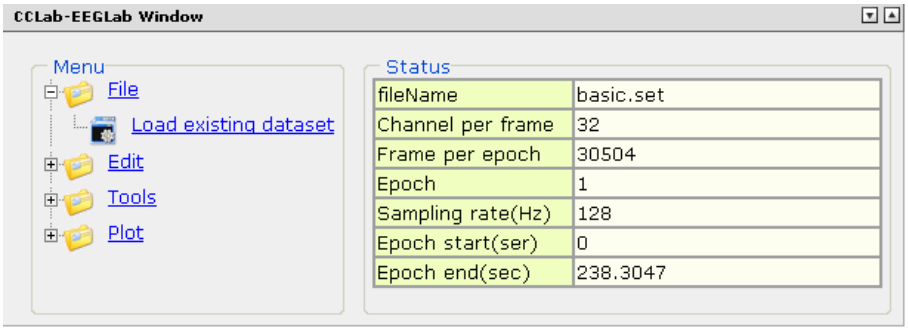


Figure 13. EEGLAB portlet

In a nutshell, we implement a resource manage system with WIMP (Window, Icon, Menu, Pointer) style Web interface. That is, the interface provides a workspace that looks like multiple directories of resources. These resources include data files, computing resources, and other resources – some of which are in metaphor forms. Different resources may be represented with different icons and associated with basic operations such as content viewing,

editing, or execution. More interestingly, menus can also be represented as metaphors and interpreted by the container (and related portlets) to emulate the menu interfaces.

The portlet we access in portal server is only a user interface for transferring user's requests into metaphors that are acceptable by the back-end services, although the user interface may provide additional help for user to construct such requests. The interface itself does not record any user-specific information. However, each user is in fact associated with a particular resource such that the user can get his own objects based on his/her identity.

When workflows are concerned, creating workflows and executing them are straightforward. As described previously, one simply creates workflows as metaphors with syntax prescribed by the container, and invoke them using the Web-based interface. The workflow managing portlet will show a form for user to input all the needed arguments for the workflow. We can see that some fields of the arguments have default values, which are defined in the flow metaphor. The process with scheduler will be created afterwards, and will be added into the task flow viewer portlet for monitoring (Figure 14).

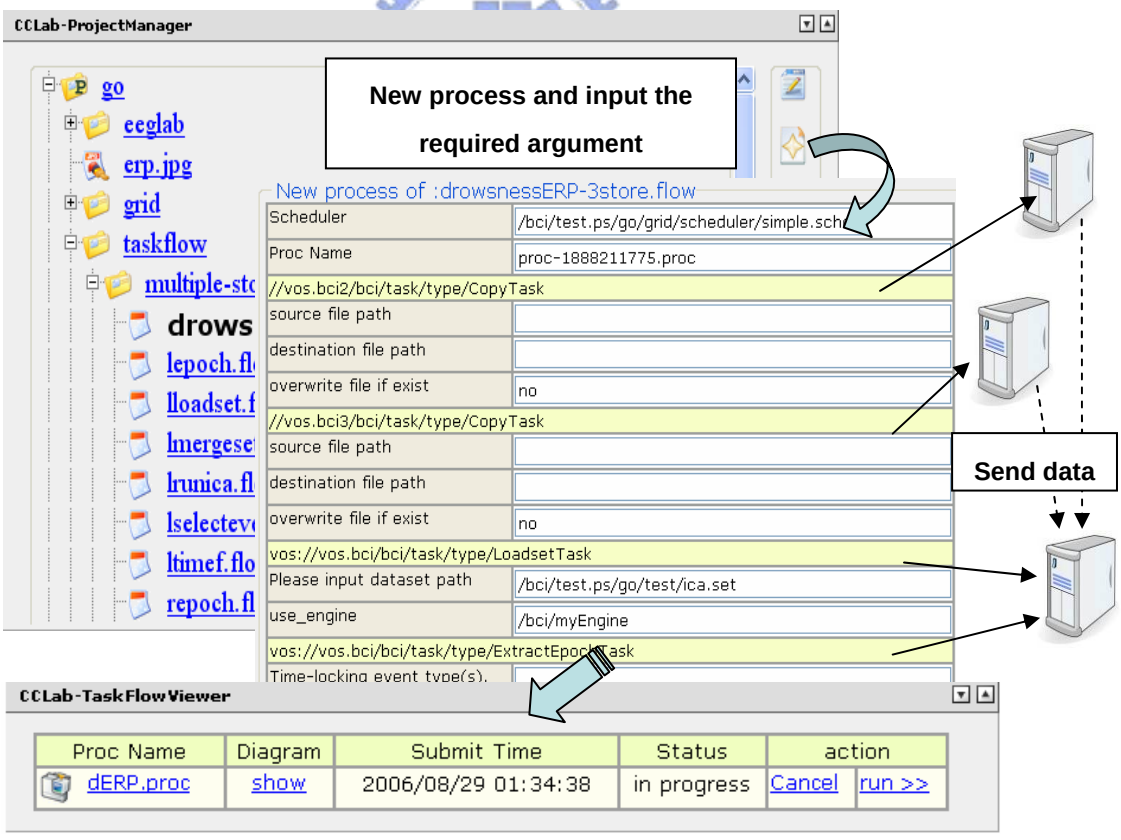


Figure 14. Use project manager portlet to control a workflow

The items in the task flow viewer means that the related process has been submitted, and

can be started anytime. We can obtain more information about the task flow by clicking the process name in task flow viewer portlet. In this view of detail information (Figure 15), you can run the task flow step by step and check if each task brings you a correct result. When you run the entire task automatically, you still can check detailed information such as the current status of the task flow. If the entire task flow needs more time to complete, you can close the connection between you (the browser) and the portal without worrying about losing your jobs.

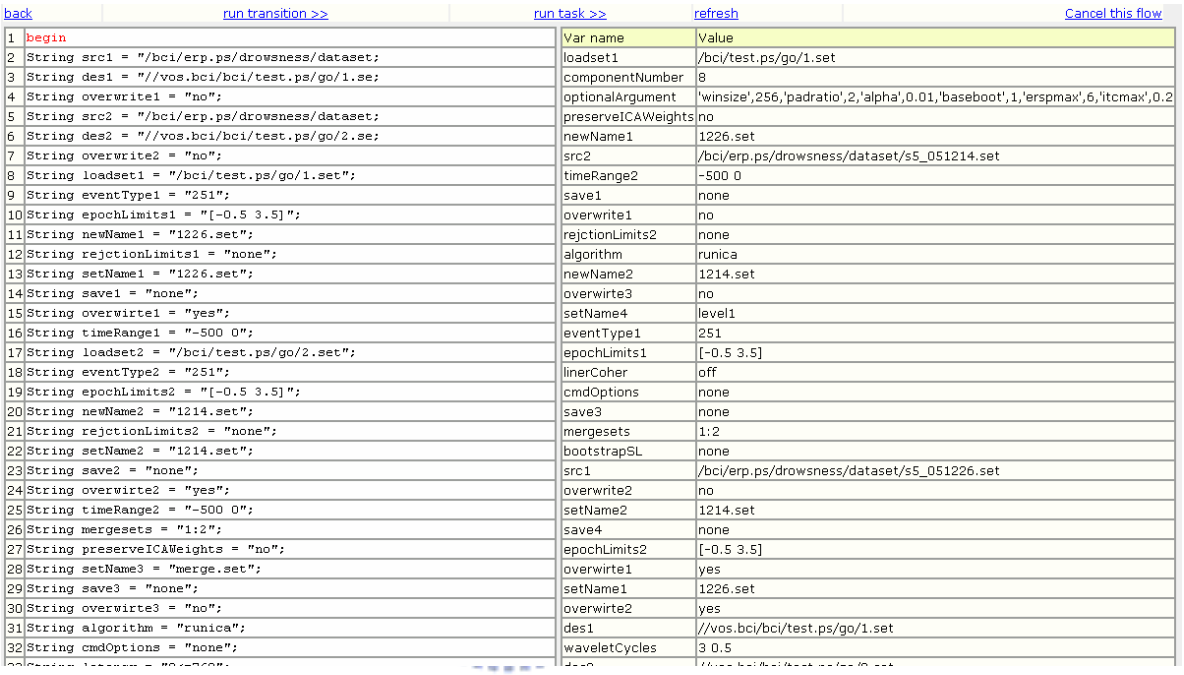


Figure 15. Monitoring a workflow in action (debug mode)

We check the workflow status in Figure 15, but it's not easy for users who are not familiar with the program-like format. Actually the purpose of this view is to help a workflow designer to debug step by step, who needs to monitor the statement in progress and check for details immediately.

For ordinary people, they can check the workflow status in a diagram format as shown in Figure 16. In the left we show the workflow structure as a tree for user to navigate easily. The workflow diagram is shown in the middle and shows both variable block and tasks differently. Finally, at the right-hand side are details of the flow variables which are bind to task for input or output purpose.

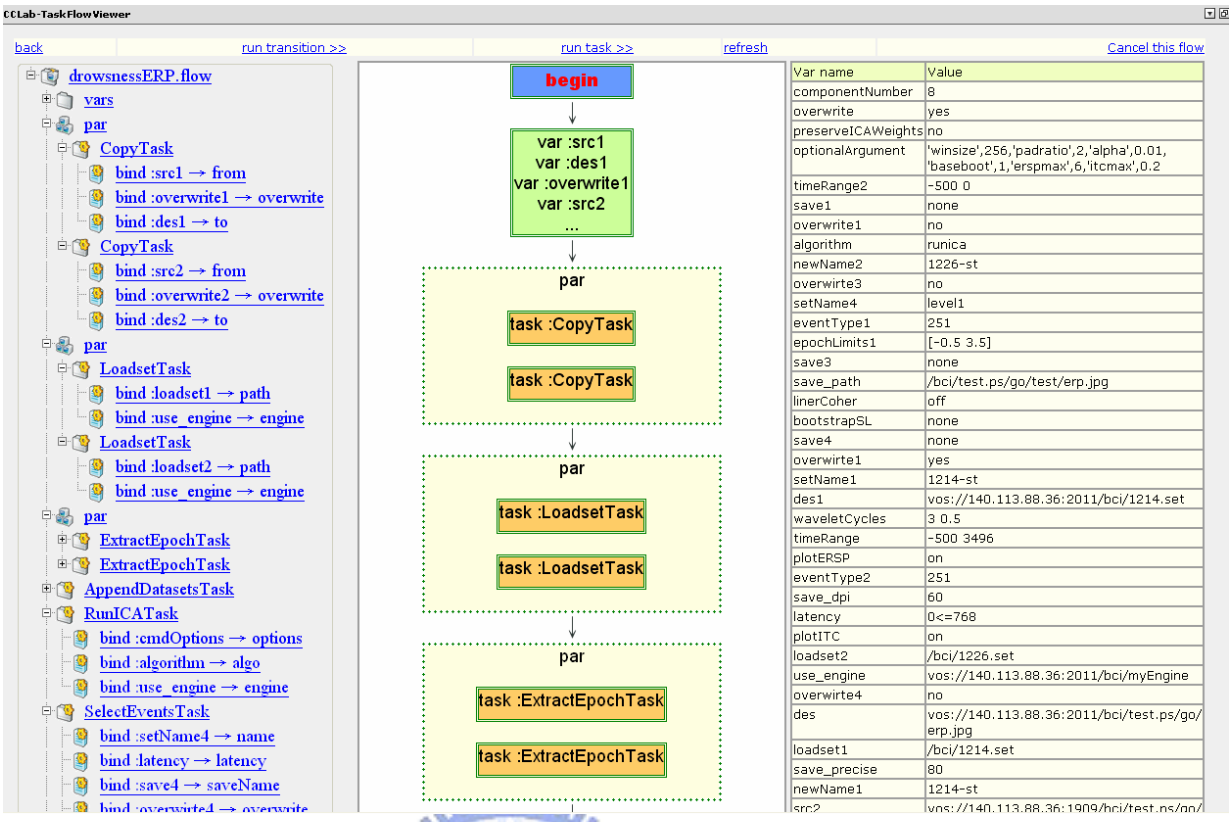


Figure 16. Monitoring a workflow in action

Chapter 6. Discussions and Related Work

Many concepts and technologies for VW as outlined previously are not new. Our goal is to reconcile these different, sometimes mutually competing concepts and technologies into a compact yet comprehensive computing model that satisfies the emerging collaborative computing requirements. Without going into specific research work, in this section we touch on some of the most relevant research areas and discuss their differences from ours at the architecture level. Note that some of the points raised below are part of the future plan for our system and are not realized yet in our system.

Virtualization has been one of the fundamental principles underpinning computer science and information technology. The virtual machine concept has been formulated since the invention of assembly and high-level languages. The Java technology, including the language and the virtual machine standards, also attempts to lay down a uniform programming and execution platform. The operating systems research is also another discipline that emphasizes virtualization by establishing a higher-level layer of abstractions over the underlying hardware resources. Modern researches on distributed operating systems [14, 15, 16] further attempt to virtualize heterogeneous hardware/software resources across the network.

Virtualization of distributed, heterogeneous resources is recently signified by the grid computing [16-19] research and closely related peer-to-peer computing [20-25]. The goal is to utilize otherwise idle computing resources by joining them into workhorses that approximate super computers. Furthermore, the concept of virtual organization also stresses that the primary emphasis is on effective utilization of distributed resources across organizational boundaries while respecting the authority and policies of individual organizations. This is what differentiates grid computing from distributed operating systems research. From a slightly different angle, many P2P applications have been developed to connect multiple machines for community sharing. P2P has two aspects, for heterogeneous resource sharing in decentralized manner, and to provide resource sharing among end users through some form of social network.

In contrast to these approaches most of which achieve virtualization by adopting a bottom-up, layered architecture, i.e. by successively basing higher-level abstractions on the layer below, our approach to virtualization is to standardize resource denotation scheme via

metaphors and the protocol for exchanging them. With the persistent, passive resources as the basis, we attempt to build a syntactic infrastructure with some (limited) forms of syntax checking at the global level, but leave actual interpretation of metaphors and the instantiation of services open, and hope that the resulting infrastructure can hold a self-evolving ecosystem.

One aspect that relates grid computing to our approach is the emphasis on the ubiquitous Internet as the common backbone. The hour-glass model as campaigned by I. Foster that underpins the Internet and the Web makes it possible that with simple and robust "waist", be it TCP/IP or HTTP/HTML, diverse applications from above or heterogeneous hardware/software resources from below can converge and reused in a multiplying manner. Take the Web for example, it is relatively cheap to establish a simple Web server, at least in the first few years, and to make oneself visible in the Web. The relatively low entry cost has helped propel the Web to what it is today. Our architecture is a straightforward extension from the Web architecture, but with the explicit VW objective in mind. In some sense, the simplicity principle is also behind the REST style Web services when compared to the more "heavy-weight" Web Service standards stack, except that they are concerned with turning the Web into an interoperability platform for software systems, not a VW.

Web Services is another movement that attempts to take advantage of the WWW by basing XML-based message exchanges on top of the ubiquitous HTTP protocol, so as to enable technology-neutral interoperability among software systems. It is arguable that Web Services concept is a combination of protocol-centric WWW architecture and earlier distributed computing platforms such as CORBA.

However, Web Services is not another distributed computing platform. Similar to the grid computing versus distributed operating system, the business world concerns, pragmatically, that there are open standards that enable software systems in one company to communicate with software systems in other companies without concerning which languages, technologies, operating systems, and so on are internally used. Furthermore, business entities want to automate cross-organization business processes as much as possible. From this perspective, service composition plays the central role in the service oriented architecture, because the traditional client-server model is no longer the main computing model for SOA, but instead peer-to-peer collaboration among services.

One of the service composition approaches that receive most attentions today are business process oriented standards such as BPEL [26] and CDL [27]. In this model, the composition is usually described in a way similar to control flows in common programming languages, and participating Web services are often modeled as servers that receive requests and reply accordingly.

In contrast, our composition model via metaphor composition resembles hyperlinks in the WWW architecture, and there is no standard composition language syntax or semantics except some basic syntactic checks. Furthermore, existing composition standards often leave service creation and management unspecified and to be filled by service providers. Although the WSRF [28] family of specifications touch on some of the fundamental issues, in short, these approaches are still technology-centric without considering the enablement of generic virtual workbenches that help users create and maintain resources in a uniform manner.

Our approach to describing and interpreting resources via metaphors also provides a composition framework that promotes language-based, ontology-driven component reuse, in the sense that new types of resources are conceived with corresponding languages defined and interpreters developed. Specifically, at the resource level, new resources can be created for a given domain (generic or domain specific); in this case, custom, resource (type) specific language syntax can be defined for the customization of individual resource (type). At the container level, new communities or domains can be created by equipping containers with differentiating interpreters. The access interface to the container, the composition mechanisms, and the corresponding assembly languages are all extensible.

Another issue that is closely related to service instantiation and management is resource/service deployment. To enable sharing of services that are best executed in the user's local machine environment, some forms of binary deployment are needed. Deployment mechanisms are also an active area that receives many research and development efforts. Popular Web browsers, for example, often provide some plug-in mechanisms that download executable resources such as Java Applets or Flesh applications and manage them behind the scene. Other deployment mechanisms outside Web browsers are also common, such as the plug-in management in the popular Eclipse IDE, or Java-based standard OSGi, or the Maven project that streamlines software building process by acquiring required libraries across network.

However, these deployment mechanisms focus on managing downloaded modules which often depend on each other in a static way, and they are not designed for users to assemble novel applications. In other words, deployment mechanisms are separated from component or service composition frameworks, not to mention the VW requirements.

The recent Web 2.0 initiative as proposed by Tim O'Rielly in late 2005 has also confirmed our VW vision. One aspect regarding these Internet applications are the relatively limited roles their users can play. Whether the applications are based on n-tier or P2P technologies, end users use services via interfaces prescribed by the application developer. To put in another words, what will be available for end users to use depend indirectly through the market mechanism, driven by multiple factors such as users' or community need, research or business innovations, or other strategic agenda behind companies and organizations. The difference between blogs and personal web pages is the fact that the former provide hosting capability, while the latter leave the aspect open.

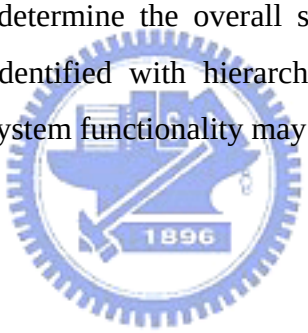
Specifically, Web 2.0 is a term roughly distinguishes the kinds of Web applications that place end users - and the collaboration among them - at the center. As mentioned before, many Web 2.0 applications provide facilities for users to contribute contents collectively, on a community basis, in a way similar to open-source projects where software artifacts receive continuous inspection and modification by the global community.

Nevertheless, there are still some significant differences between VW and Web 2.0 initiative. First, most Web 2.0 applications are still client-server architecture and focus on Web-based interfaces, which is evidenced by the closely related AJAX technology that concerns more fine-grained interaction between browser (web pages) and remote servers. In our resource-oriented architecture, we abstract the acceptable protocols away without limiting ourselves to Web-based interfaces. However, to encourage reuse and reduce fragmentation, we also provide a canonical inter-container protocol that serves as the middle ground for end users, service designers, and service "composers" in between.

Secondly, the Web 2.0 is essentially a set of guidelines about applications. The actual forms can vary a lot, ranging from traditional Web-based applications to P2P ones, although the underlying the infrastructure is still the Internet. Without a commonly acceptable but higher-level infrastructure among them, the development cost remains high. Furthermore, Web 2.0 does not consider too much about the assembly of third-party modules and argues

the irrelevancy of software deployment and evolution issues. However, whenever reusable modules emerge and become commodities that people can use to assemble their own applications, these unavoidable, transitional software engineering issues recur immediately. Although many issues remain to be exploited, our resource-oriented architecture already models the development, deployment, and assembly aspects in terms of resources, metaphors, containers, and a canonical communication protocols among containers.

Research in coordination models and languages [29-31] distinguish between computation and coordination, and propose languages with different coordination models. For example, Linda provides a shared tuple space where computational processes can insert and retrieve data using simple pattern matching rules. Compared to coordination languages, our approach intends to complement existing application programming languages, and pays attention on fundamental reusability issue as well as the flexibility and usability of customization interface. VW can also be regarded as a kind of coordination model. In Linda, it is still the (distributed) computational processes who determine the overall system behavior. In VW, in contrast, computational processes are identified with hierarchical URLs and concentrate on their dedicated jobs and the overall system functionality may be reshaped by the coordinator.



Chapter 7. Conclusion

We have stated the VW vision towards a global collaboration environment with desirable properties and requirements. The idea is essentially to extend the information sharing Web to include software sharing, in a way even average people can contribute to the global village. With so many valuable open source projects contributed by numerous developers and made freely available to the public, it would save a lot of efforts if there are ways, even for non-technical persons, to utilize these software resources by mixing and matching rather than continuously re-inventing the wheel. Of course, there are far more obstacles than what we have discussed in this thesis. Some of the major shortcomings discussed in this thesis include the lack of standardized notion of resource and service instantiation and hosting – which is partially due to the implied technical and security complexity. Another important issue is the research and development of "bullet-proof" software engineering environment that even non-technical persons can become productive. We have introduced the notion of virtual workbenches but without further discussing what it implies. Apparently, substantial efforts are needed in order to make the workbench sufficiently intelligent, robust, self-diagnosing, and self-healing. These kind of developing environments are indeed being approached and reflected in modern IDEs, but more work is needed when considering not just software engineers but the rest. To think about it, there is essentially no difference between developers creating software artifacts and Web 2.0 users creating simple diaries. The differences are the skilled needed and tools involved, and together all participating persons and resources form an ecosystem with collaboration, competition, coalition, and so on occurring in different part of the VW.

We have proposed the metaphoric service-oriented architecture and establish necessary foundation that may enable a universal development, deployment, and assembly platform across distributed, heterogeneous resources. Specifically, we make explicit the description format of resources and role of containers that are responsible of managing corresponding run-time services associated with the resources.

We showed that dynamic service composition can be achieved though metaphor composition, and demonstrated the validity of our approach using a functioning e-Science portal that provides users with conventional desktop interface for resource management – using the Internet as the computer.

Nevertheless, the step we take towards VW is just a beginning. As a universal IDE, further research to ensure the robustness, consistency, and continuous evolution of the IDE as a whole is necessary. A comparable IDE monitoring and ensuring development progress and error handlings, 7/24, is not available yet. The framework we implemented so far requires the developers to perform syntax checking from time to time. A sound foundation, for example, rooted from configuration management technologies, but extended to the global level, is needed.

Finally, we purposely leave the security issue unattended. Certainly, may be the most important requirements for any distributed computing platforms are the support for security and privacy. For example, the security issue is placed among the highest priority requirements in the grid computing community. In this thesis, however, we focus more on the functional aspects of VW and the corresponding infrastructure support for flexible composition of distributed, heterogeneous resources. Security issues related to VW are considered future work.

References

[1] Wikipedia - The Free Encyclopedia. http://www.wikipedia.org

[2] Tim O'Reilly, "What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software". http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html

[3] The myGrid Consortium, "myGrid: Middleware for in silico experiments in biology". http://www.mygrid.org.uk/

[4] W3C, "Web Services Architecture". http://www.w3.org/TR/ws-arch/

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

[6] W3C, "Web Services Description Language (WSDL) ". http://www.w3.org/TR/wsdl

[7] T. Berners-Lee, "Realising the Full Potential of the Web", W3C notes. http://www.w3.org/1998/02/Potential.html

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[9] Matlab, http://www.mathworks.com/

[10] EEGLAB, http://www.sccn.ucsd.edu/eeglab/

[11] JMatLink, http://www.held-mueller.de/JMatLink/

[12] JSR 168, http://www.jcp.org/en/jsr/detail?id=168

[13] Web Service for Remote Portlet (WSRP), http://www.oasis-open.org/committees/wsrp/

[14] A.S. Tanenbaum and R. van Renesse, "Distributed Operating Systems", Computing Surveys, Vol. 17, No. 4, December 1985, pp. 419-470

[15] S. J. Vaughan-Nichols, "Developing the Distributed-Computing OS", IEEE Computer, Vol. 35, No. 9, September 2002, pp. 19-21

[16] J. Kubiatowicz and D. P. Anderson, "The Worldwide Computer: An operating system spanning the Internet would bring the power of millions of the world's Internet-connected PCs to everyone's fingertips", Scientific American, March 2002, pp. 40-47

[17] L. Smarr and C.E. Smarr, "Metacomputing", Communications of the ACM, 35(6), (1992), pp. 74-84.

[18] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", International J. Supercomputer Applications, 15(3), 2001.

[19] Ian Foster, "Service-Oriented Science", Science 6 May 2005, vol. 308, no. 5723, pp. 814 – 817.

[20] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications", IEEE/ACM Transaction on Networking, vol. 11, no. 1, 2003, pp. 17–32.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content Addressable Network", Proc. ACM SIGCOMM, 2001, pp. 161–72.

[22] A. Rowstron and P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems", Proc. Middleware, 2001.

[23] Xiaohui Gu, Klara Nahrstedt, and Bin Yu, "SpiderNet: An integrated peer to peer service composition framework", 13th IEEE International Symposium on High-Performance Distributed Computing (HPDC-13), 2004.

[24] Pitoura, E., S. Abiteboul, D. Pfoser, G. Samaras and M. Vazirgiannis, "DBGlobe: a Service-oriented P2P System for Global Computing", SIGMOD Record 32, 2003.

[25] P Maheshwari, S Kanhere, N Parameswaran, "Service-oriented middleware for peer-to-peer computing", Proceedings of the 2005 3rd international conference on industrial informatics IEEE, Piscataway, NJ, 2005, pp. 98 – 103.

[26] R. Khalaf, N. Mukhi, S. Weerawarana, "Service–Oriented Composition in BPEL4WS", World Wide Web Conference, 2003. http://www2003.org/.

[27] W3C, Web Services Choreography Working Group. http://www.w3.org/2002/ws/chor/

[28] WSRF, OASIS Web Services Resource Framework TC, http://www.oasis-open.org/

[29] P. Ciancarini, "Coordination models and languages as software integrators", ACM Computing Surveys, 28(2), June 1996, pp. 300-302.

[30] D. Gelernter, "Generative communication in Linda", ACM Transactions on Programming Languages and Systems, 7(1), Jan. 1985, pp. 80-112.

[31] D. Gelernter and N. Carriero, "Coordination languages and their significance", Communications of the ACM, 35(2), Feb. 1992, pp. 97-107