Service Context Management for Exertion-oriented Programming

K.M.Divya Darshan
Spring 2007
Advisor: Dr.Michael Sobolewski

Table of Contents

1) Introduction……………………………………………..2
2) Literature Review………………………………………3
3) Methodology……………………………………………12
4) Result……………………………………………………30
5) Conclusions / Future Research……………………..31
6) References………………………………………………31

Table of Figures

1) Figure 1: Service Oriented Architecture…………………...3
2) Figure 2: Service Object Oriented Architecture…………..4
3) Figure 3: A job federation……………………………………..8
4) Figure 4: Lifecycle of Service Context……………………..9
5) Figure 5: Graphical depiction of Service Context……….10
6) Figure 6: Use Case Model for LCCM framework………..12
7) Figure 7: A simple diagram depicting the MVC framework……………………………………..23
8) Figure 8: Component Diagram for LCCM framework…..24
9) Figure 9: UML Interaction diagram between LCCM components……………………………………...27
10) Figure 10: UML Class Diagram for LCCM framework...28
11) Figure 11: Partial implementation of the GUI………….30

1) Introduction

Federated Service Object-Oriented paradigm [1] is built on the object-oriented distributed paradigm which is illustrated by the JINI architecture where in we have objects in the network coming together on the fly to play the roles they are built for. The same principle is applied to the Service-ORriented Computing EnviRonmet (SORCER) developed at Texas Tech University. In SORCER, a service provider is a remote object that accepts network requests—called *exertions*—from service requestors to execute an elementary item of work called a *service task* or a composite item of work called a *service job*. An exertion, either a task or job, can federate on multiple hosts according to its encapsulated data, operations, and control strategy. Service Context can be described in simple words as the data that the jobs and tasks work on or the input and output parameter of any provider method. As of today SORCER doesn't have any well defined framework for managing context from the time when it's created by a service provider to the time when it's destroyed. The main focus of this report will be to propose a framework to display and manage context in SORCER with emphasis on modularity and low network usage. The proposed framework would help in providing uniform and detailed service context views, standardized data structure for transferring service contexts and so on.

2) Literature Review

In the six generations of RPC systems only some like CORBA, Java RMI, and web/Globus services – support distributed objects [1]. Some of the common issues that an object wrapping approach does not help to cope with are network-centric messaging, invocation latency, object discovery, dynamic object Federation, fault detection, recovery, partial failure, etc. In contrast JINI architecture does not hide the network from the programmer it allows the programmer to deal with the network issues like leases for network resources, distributed events, transactions, and discovery/join protocols to form federations. The SORCER architecture [1] makes use of the JINI service management for exertion oriented programming facilitated by the Federated Method Invocation (FMI) explained later in this review. Currently FMI is the sixth generation RPC system which allows for network invocations on multiple federating hosts in the SORCER environment. Exertions as stated above are service requests from service requestors which can be submitted in the SORCER environment to any service provider. Once an exertion is submitted it becomes a Federated Service Object Oriented (FSOO) program and this exertion is dynamically linked to all the service providers on the network which are currently available. This type of phenomenon when the service providers come together dynamically to satisfy an exertion is called an exertion federation. In the SORCER environment exertions can be created from the user interfaces downloaded on the fly from the service providers.

Service oriented Architecture

A service-oriented architecture is a collection of services that communicate with each other.

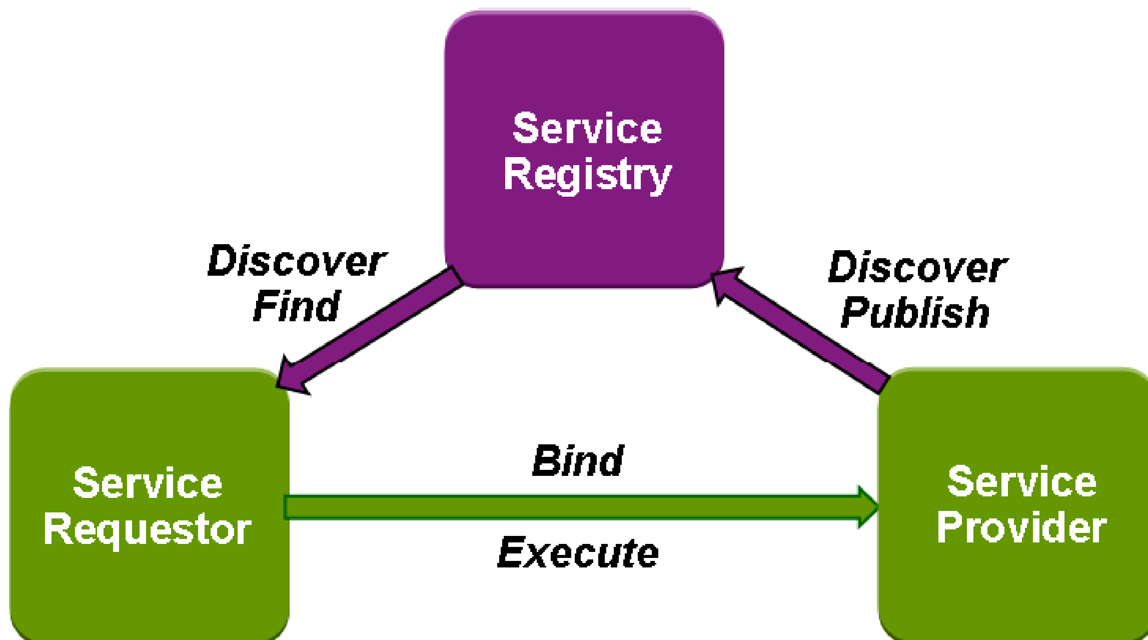

Figure 1: Service Oriented Architecture

The services are self-contained and do not depend on the context or state of the other service. They work within distributed systems architecture. Service providers in a SOA environment can be accessed independently by service requestors without prior knowledge of their implementation or platform. A SOA environment has service providers, requestors and a third component known as the Service Registry as shown below in figure 1: In comparison to client architecture we can note that the client in SOA is a service requestor and the server is a service provider. The service registry is responsible for registering the service providers available on the network by intercepting the announcements made by the providers and add the published services. The service requestor looks up a particular service by sending queries to the service registry; queries normally may contain search criteria related to the service name/type and quality of service. Discovery and Join protocols are used by the service providers and requestors to locate registries and then publish or acquire services on the network.

Service Object Oriented Architecture (SOOA) introduces us to an object known as proxy which implements the same service interfaces as the service provider and is registered with the service registry for use by the service requestors'.

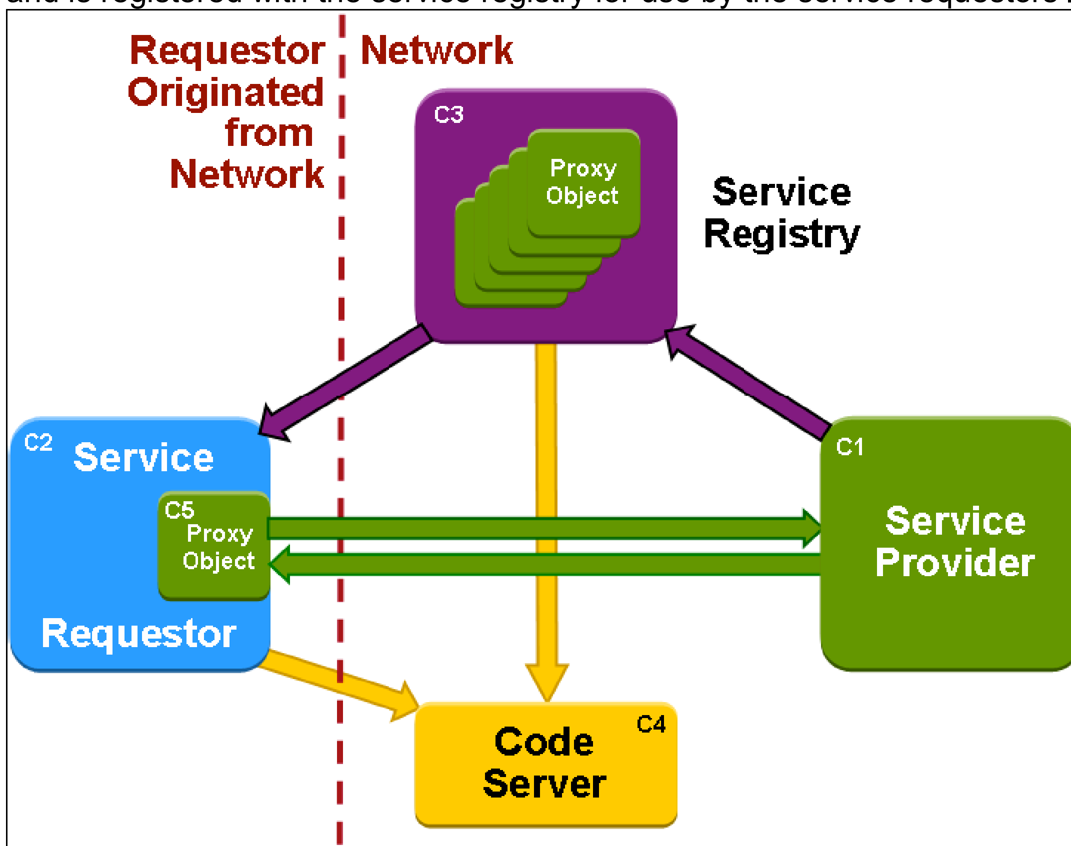

Figure 2: Service Object Oriented Architecture

In contrast to SOOA is the Service Protocol Oriented Architecture (SPOA) where a communication protocol is fixed and this protocol must be known both to the

service provider and requestor. In SPOA a provider registers itself with its name, the service requestors have to know the name of service beforehand to utilize a particular provider. A service requestor in SPOA will need to generate the proxy depending on the service description and the communication protocol being followed. However in SOOA the service requestor need not generate the proxy itself, since it gets the active proxy from the registry itself. In SOOA the communication between the proxy and its provider is a contract between them and also the implementation is defined by the provider. Another difference between the SPOA and SOOA is the way in which they discover the registry. SOOA need not know the specific location of the registry but in SPOA the specific location of the registry is to be known both by the service provider and service requestor. One of the weakness in SPOA has to do with the requestor creating and owning the proxy whereas in the SOOA the proxy is created and owned by the provider; due to the requestor creating and owning the proxy which meet the requirement agreed upon by the requestor and provider we get a generic protocol which leads to inefficient communication in some cases, whereas in SOOA we have the providers and requestors communicating with each other via a service registry and communicating through message passing. Coming to the network issues we know that it's not possible to take an object oriented program developed for local systems and turn into a program for distributed systems. We need to handle issues like network latency and so on. We also know that making local calls is not the same as making remote calls. Keeping all this mind exertion oriented programming was introduced in SORCER where we have no particular provider specified in the service request or the exertion.

Why JINI? [2] Jini has the potential to create large federations of electronic devices to which users can connect with a standard computer device, or they can get these services (memory, storage, and computation) from the network itself. As a result, Jini is different from traditional operating environments in at least ten important ways:

1. It's small. The Jini core requires only a Java virtual machine, 4 KB of resident memory, and some computational cycles. The Jini system is about 600 KB.

2. The system is fully object-based. That is, its components include methods that can be executed remotely, and when a method is incompatible with its execution environment (floating-point code, for example), it can load the correct method automatically.

3. Jini's basic control structures are simple. Objects communicate via the standard Java RMI interface through basic operations in JavaSpaces (a system that manages features such as object processing, sharing, and migration): read, write, and take.

4. All services are available through a simple lease facility. Leases are time-dependent (with renewal upon expiration) and allow multiple leases to a service.
5. Jini includes the standard Java security constructs. Given the distributed nature of the environment, this is very important, because security is in effect where the object is executed.
6. With JavaSpaces, Jini includes facilities for transactions (with two-phase commit) and persistency (initially with Object Design's ObjectStore PSE Pro for Java).
7. Within Jini, groups of basic information such as security policies are accessible. Those groups can be aggregated almost indefinitely into large networks.
8. Jini supports the Java programming model. Java services and protocols (JavaBeans, and so on) are available to the large base of existing programmers.
9. Source will be licensed for free (or something close to that).
10. Most importantly, Jini provides plug-and-execute capabilities where a new device can gain instant recognition (by a boot, join, and discover protocol) by the network and have access to any services for which it has authority. Sun compares this to the dial tone in the standard telephone network, and is considering using the JavaTone brand to describe the overall environment.

SORCER is a pure S2S framework based on JINI programming model. In SORCER service providers which do not have any association, federate dynamically to service an exertion. Once the exertion is satisfied the federation dissolves and the providers disperse to join other transactions. The three types of network objects are basically providers, requestors and registries. Providers are responsible for deploying the service on the network, publishing its services to the registry and allowing the service requestor to access its service. Service requestors query the registry to find a suitable service provider. Registries basically store the proxies and make them available to the requestors. Basically a service is a well known public interface. A service provider implements this interface or multiple interfaces in order to participate in federations.

A message from a service requestor to a service provider to execute an item of work is called a service exertion. The two types of exertion are job and task exertions. We can describe the task exertion as an elementary service request which is executed by a single service provider or a small federation. A task exertion is a composite exertion defined in terms of other jobs and tasks, which are executed by larger federations. To execute and exertion we need a service

oriented program that is bound to all the needed and currently available service providers on the network. This collection of service providers is known as an exertion federation also a virtual metacomputer with its metainstructions. Thus we have a service-oriented program having its own set of control strategy and also the service context; where service context is the data that the jobs and tasks work on.

Federated Method Invocation

With a brief introduction to FMI already given in the beginning, let us start by discussing about service providers. Service providers are network objects which have network state, network behavior, and network types. They also act a network peers which can replicate and compensate for other peers in a network often during network failures and so on. These providers come together on the fly to satisfy a particular exertion in the network. This federation is known as exertion federation. After a particular exertion for which they are federated is done, the federation is disbanded. It's also possible for the service providers to participate in more than one federation at the same time. In a component exertion there is sharing of data and the top level exertion is only complete if all the nested exertions are complete. In the object oriented environment the only means of passing control to an object is through messages since the object data is encapsulated. Each message specified has the name of the receiving object, the name of the operation to be invoked and any parameters for the operation. Sometimes due to network unreliability we have the problem of losing messages or the message not reaching the receiving object. Due to which it's better to send the messages which are related as a single request than as a number of requests. These messages are called the exertions which encapsulate multiple service signatures and the data service context.

A service signature is specified as below [1] -

- signature name
- service type – Java interface name
- selector of the service operation – operation name of the service type (Java interface)
- operation type – Signature.Type: PROCESS (default), PREPROCESS, POSTPROCESS
- service access type – using service accessor, catalog, or exertion space
- priority
- execution time flag – if true, the execution time is returned in the service context
- notifyees – list of email addresses (whom to notify when completed)
- service attributes – requestor's attributes matching provider's registration attributes.

PROCESS is the signature which that tells the binding provider for an exertion.

Basically there are two types of exertion, Task and Job. A task can be compared to a statement in a conventional programming language. We should also note that the task is a batch of operations acting on the same service context. It is an elementary unit in exertion oriented programming. If a provider responds to a task's PROCESS signature that means that provider has the necessary method specified in the task's PROCESS signature. APPEND is the signature used to append the service context received from the provider in runtime to the task's currently processed service context. Appending a service context allows a requestor to use actual data in runtime not available to the requestor when a task is submitted [1]. A job is related to a procedure in programming language. It can consist of one or more tasks. It's a composition of exertions that make up a federated procedure [1]. A job federation is shown in the figure 3 below -

Service Context

In both computer science and information science, ontology is a data model that represents a set of concepts within a domain and the relationships between those concepts.

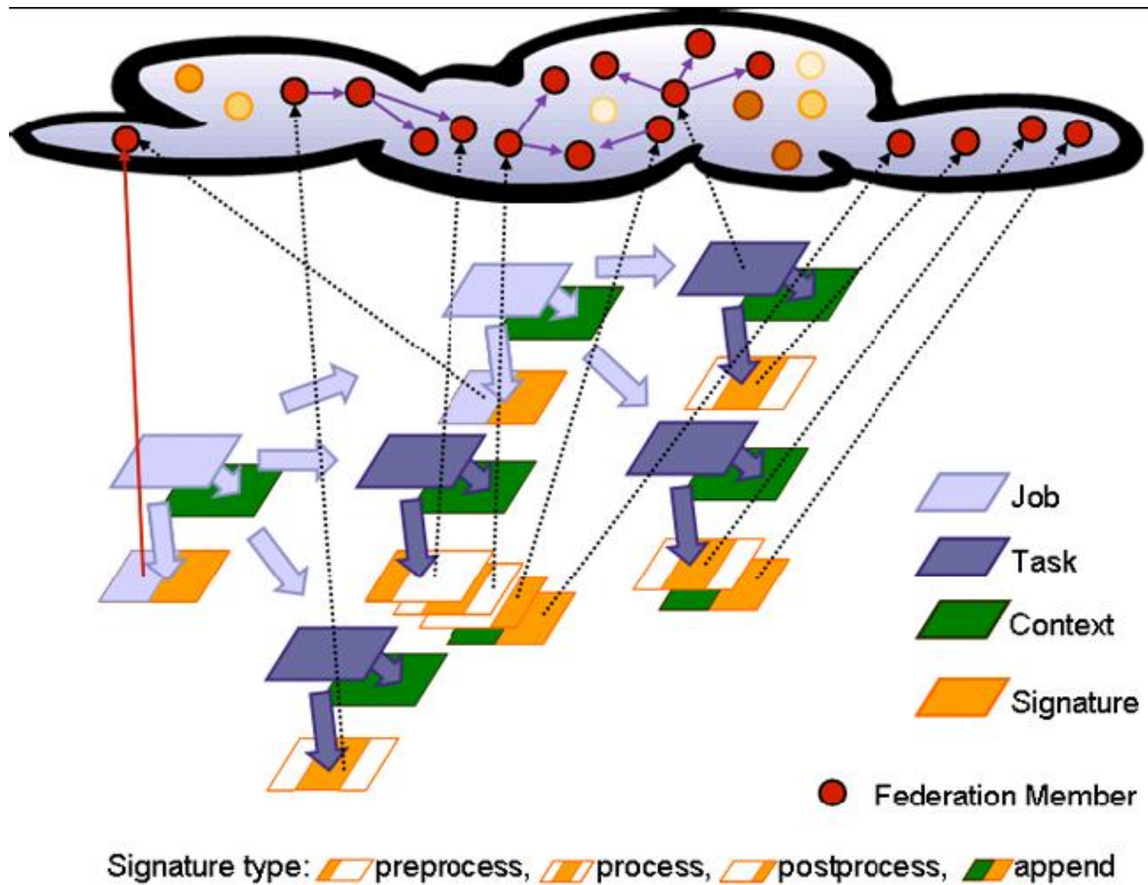

Figure 3 - A job federation.

It is used to reason about the objects within that domain [3]. A service context is a data model which defines the relationship between the service providers with the related data. The service provider’s vocabulary basically defines the objects and also the relationship between the objects of the service provider according to a specified domain of interest. For an exertion to be satisfied by a service provider from a service requestor, the requestor has to satisfy or comply with the ontology of the service provider. [1] In the percept conceptualization, attributes and their values are used as atomic conceptual
primitives and complements are used as molecular ones. A complement is an attribute sequence (path) with a value at the last position. An elementary percept property consists of a percept subject and a set of percept complements,
and usually corresponds to a simple sentence of natural language.

[1] A service context is a tree-like structure described conceptually in the EBNF conceptual syntax specification as follows:
1. context = [subject ":”] complement {complement}.
2. subject = element.
3. complement = element ";".
4. element= path ["=" value].
5. path = attribute { "/" attribute } [{ "<" association ">"}] [{ "/" attribute }].
6. value = object.
7. attribute = identifier.
8. relation = domain product.
9. association = domain tuple.
10. product = attribute {“|" attribute}.
11. tuple = value {"|" value}.
12. attribute = identifier.
13. domain = identifier.
14. association = identifier.
15. identifier = letter {letter | digit}.

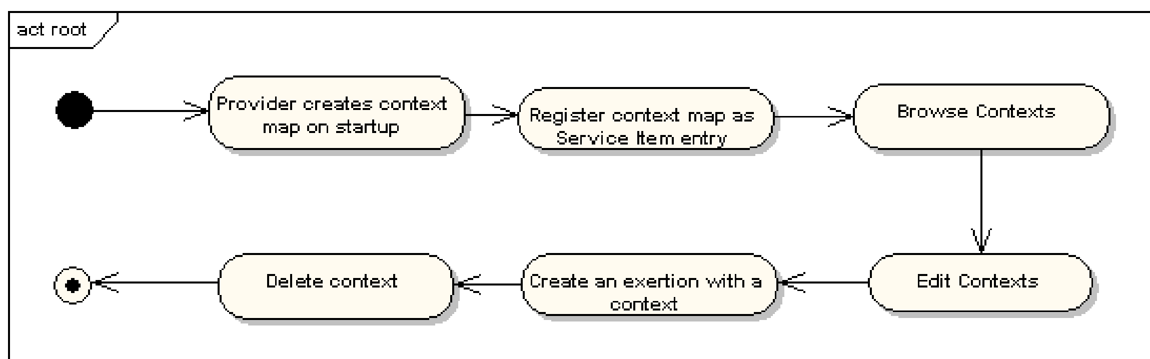

Figure 4: Lifecycle of Service Context

In figure 5, we have the lifecycle of a service context. A service provider creates the service context map when its started. A service context map is a key,value pair. The key consists of the interface and the method name with the value consisting of the context. The key can be presented in wild characters like

interface/*, denoting all the methods in a interface. Once the context is created, the next step is to register the context as an entry in the service item. After the registration process is done, we can must be able to browse all the contexts available. The contexts must also be editable. We must be able to create an exertion with the context and after the exertion is done, the context is destroyed.

An example taken from Dr.Mike Sobolewski's paper "**Metacomputing with Federated Method Invocation**" clears the idea about what exactly is a service context. The graphical depiction is shown in figure 5.

laboratory/name = SORCER: university=TTU;
university/department/name=CS;
university/department/room/number=20B;
university/department/room/phone/number=806-742-
university/department/room/phone/ext=237;
director <person | Mike | W | Sobolewski> /email=sobol@cs.ttu.edu;

person | firstname | initial | lastname.

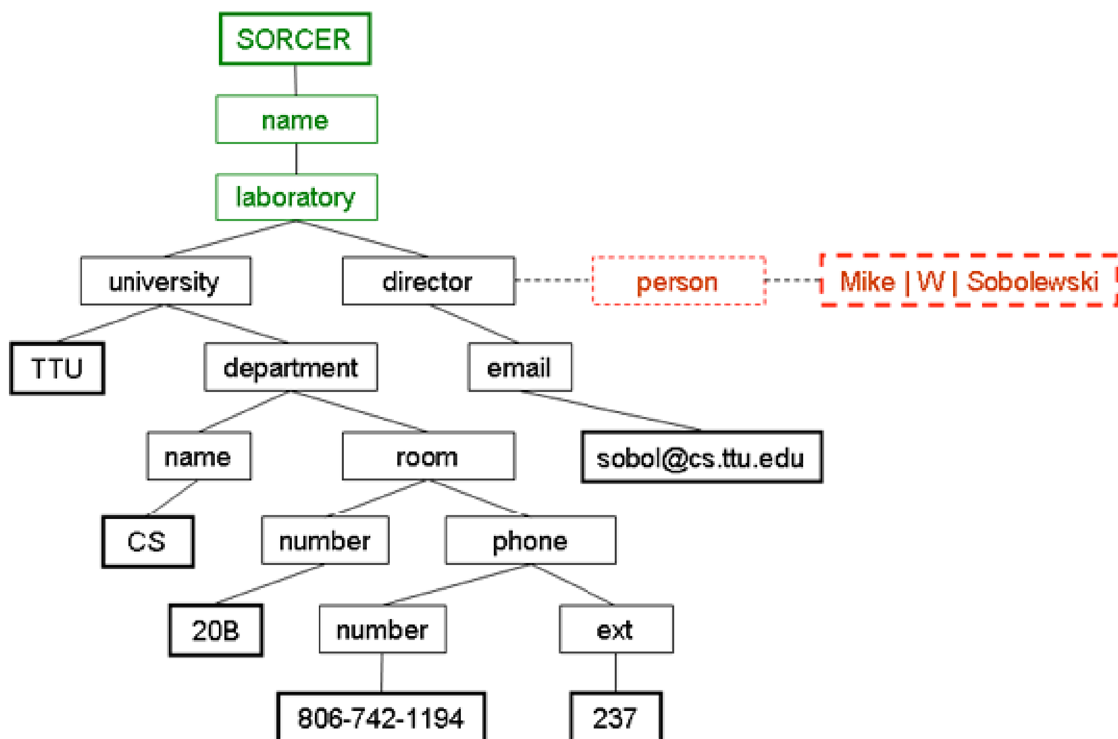

Figure 5: Graphical depiction of Service Context

The leaf node is where the data resides. This node is also known as the context node. The application domain namespace and a context model are denoted by the service context. The context is represented as a XML document for interoperability in SORCER. A data item in a leaf node can be reached with a

hierarchical name. Providers can execute and run the exertions using them itself as the data nodes to run complex interactive programs.

From the above discussion we have understood what exactly a service context is and where it comes into play. This report will focus on managing the service context for use by tasks and jobs. We will go into the details in the methodology section.

3) Methodology

A new framework Life-Cycle service Context Management (LCCM) for network centric exertion-oriented programming was created. The use case model is shown in figure 6.

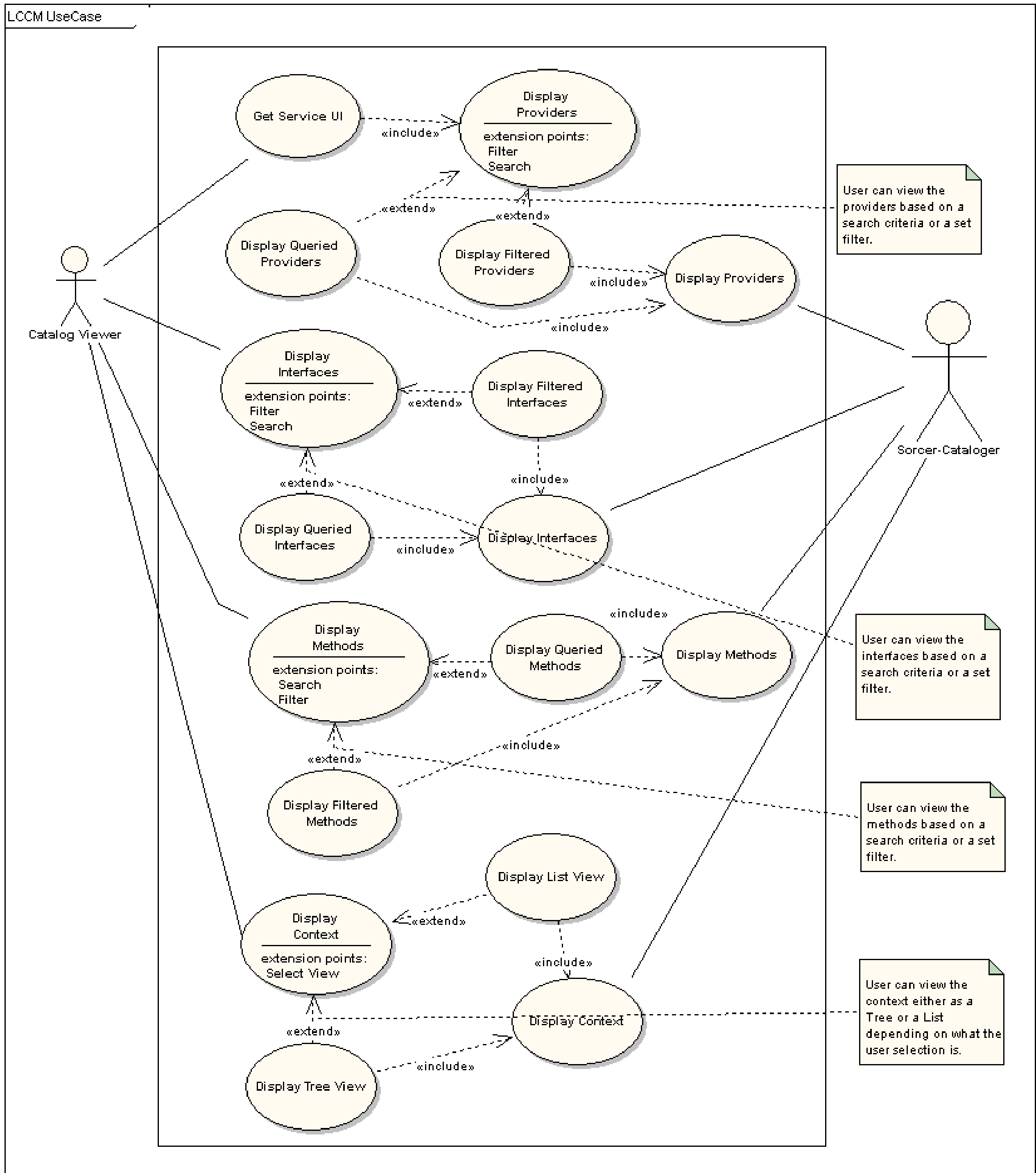

Figure 6: Use Case Model for LCCM framework.

The use case descriptions are given below.

USE CASE 1	**Build UI**	
Goal in Context	Build the UI that has all the data to display the providers, interfaces, methods and contexts.	
Scope & Level	User, primary function.	
Preconditions	Information must be received from the lookup services about the available providers.	
Success End Condition	Provider list is properly displayed.	
Failed End Condition	Provider list is not properly displayed. Build UI is unable to access lookup service.	
Primary, Secondary Actors	Cataloger viewer. Lookup service.	
Trigger	User selects the cataloger browser in service browser.	
DESCRIPTION	Step	Action
	1	User views the catalog browser.
	2	A list containing providers available on the network is displayed.
	3	The user may select one of the providers.

USE CASE 2	**Select Providers List**	
Goal in Context	User selects a provider from a list, which then displays a list of interfaces offered by that provider.	
Scope & Level	User, primary function.	
Preconditions	Information must be received from the lookup services about the available providers.	
Success End Condition	Interface list is properly displayed.	
Failed End Condition	Interface list is not properly displayed. Build UI is unable to access lookup service.	
Primary, Secondary Actors	Cataloger viewer. Lookup service.	
Trigger	User selects a provider from the list provided.	
DESCRIPTION	Step	Action
	1	User selected provider is highlighted.
	2	Interface list with respect to the selected provider is displayed.

USE CASE 3	**Search Provider**	
Goal in Context	Allows the user to search through the list of providers to find one particular provider.	
Scope & Level	User, primary function.	
Preconditions	A menu with all providers must be built by Build UI first.	
Success End Condition	Provider is found by search.	
Failed End Condition	Provider is not found by search.	
Primary Actor	User	
Trigger	User types the name of a provider in the search bar.	
DESCRIPTION	Step	Action
	1	The user types the name of the desired provider into the search bar
	2	If a match is found, the provider is selected.
EXTENSIONS	Step	Branching Action
	2a	If an exact match is not found, the closest match is selected instead.

USE CASE 4	**Set Provider Filter**	
Goal in Context	User sets the filter according to which the provider list is displayed in the cataloger browser.	
Scope & Level	User , primary function	
Preconditions	Information must be received from the lookup services about the available providers.	
Success End Condition	Provider list is displayed according to the filter.	
Failed End Condition	Provider list is not displayed according to the filter set by the user.	
Primary, Secondary Actors	Catalog Viewer. Lookup Service	
Trigger	User clicks on the provider filter button in the cataloger browser.	
DESCRIPTION	Step	Action
	1	A pop up window is opened for entering the filter criteria.
	2	User enters the filter criteria and closes the filter window.
	3	The providers are displayed according to the filter criteria entered in the filter window.

USE CASE 5	**Select Interface**	
Goal in Context	User selects an interface from a list, which then displays a list of methods offered by that interface.	
Scope & Level	User, primary function.	
Preconditions	User has selected a provider from the UI	
Success End Condition	All the methods with respect to the interface selected are displayed.	
Failed End Condition	No methods with respect to the interface selected are displayed.	
Primary actor	User.	
Trigger	User Selects an interface from Interface list	
DESCRIPTION	Step	Action
	1	Methods of the selected interface are displayed

USE CASE 6	**Search Interface**	
Goal in Context	Allows the user to search through the list of interfaces to find one particular interface.	
Scope & Level	User, primary function.	
Preconditions	A menu with all interfaces must be built by Build UI first.	
Success End Condition	Interface is found by search.	
Failed End Condition	Interface is not found by search.	
Primary Actor	User	
Trigger	User types the name of an Interface in the search bar	
DESCRIPTION	Step	Action
	1	The user types the name of the desired interface into the search bar

	2	If a match is found, the interface is selected.
EXTENSIONS	Step	Branching Action
	2a	If an exact match is not found, the closest match is selected instead.

USE CASE 7	**Set Interface Filter**	
Goal in Context	User sets the filter according to which the Interface list is displayed in the cataloger browser.	
Scope & Level	User , primary function	
Preconditions	Information must be received from the lookup services about the available Interfaces.	
Success End Condition	Interface list is displayed according to the filter.	
Failed End Condition	Interface list is not displayed according to the filter set by the user.	
Primary, Secondary Actors	Catalog Viewer. Lookup Service	
Trigger	User clicks on the Interface filter button in the cataloger browser.	
DESCRIPTION	Step	Action
	1	A pop up window is opened for entering the filter criteria.
	2	User enters the filter criteria and closes the filter window.
	3	The Interfaces are displayed according to the filter criteria entered in the filter window.

USE CASE 8	**Select Method**	
Goal in Context	A list showing all methods is offered to the user, and the user selects one.	
Scope & Level	User, primary function.	
Preconditions	An Interface has been selected previously.	
Success End Condition	Selected methods context is displayed.	
Failed End Condition	Context is not properly displayed.	
Primary actor	User.	
Trigger	User Selects a method	
DESCRIPTION	Step	Action
	1	Selected method is highlighted
	2	The context for the selected method is displayed in either list or tree form.

USE CASE 9	**Search Method**
Goal in Context	Allows the user to search through the list of Methods to find one particular method.
Scope & Level	User, primary function.
Preconditions	A menu with all Methods must be built by Build UI first.
Success End Condition	Method is found by search.
Failed End Condition	Method is not found by search.
Primary actor	User
Trigger	User types the name of a Method in the search

DESCRIPTION	Step	Action
	1	The user types the name of the desired method into the search bar
	2	If a match is found, the method is selected.
EXTENSIONS	Step	Branching Action
	2a	If an exact match is not found, the closest match is selected instead.

USE CASE 10	**Set Method Filter**	
Goal in Context	User sets the filter according to which the Method list is displayed in the cataloger browser.	
Scope & Level	User , primary function	
Preconditions	Information must be received from the lookup services about the available Methods.	
Success End Condition	Method list is displayed according to the filter.	
Failed End Condition	Method list is not displayed according to the filter set by the user.	
Primary, Secondary Actors	Catalog Viewer. Lookup Service	
Trigger	User clicks on the Method filter button in the cataloger browser.	
DESCRIPTION	Step	Action
	1	A pop up window is opened for entering the filter criteria.
	2	User enters the filter criteria and closes the filter window.
	3	The methods are displayed according to the filter criteria entered in the filter window.

USE CASE 11	**Toggle List/Tree View**	
Goal in Context	A button is added that will alternates between displaying the context in tree or list format to the GUI.	
Scope & Level	User, primary function.	
Preconditions	The context must be displayed with respect to a selected method.	
Success End Condition	Context display is changed to the selected view.	
Failed End Condition	Context display does not change.	
Primary actor	User.	
Trigger	User selects either the List or Tree button on the bottom of the menu.	
DESCRIPTION	Step	Action
	1	The user may switch between tree or list view.
	2	The context display is changed to whichever option is selected.

USE CASE 12	**Tree Display**
Goal in Context	Displays the context for the selected service and method in a tree format.
Scope & Level	Cataloger, subfunction.
Preconditions	Method must be selected, and the tree button must be clicked.
Success End Condition	Context is displayed in tree format.
Failed End Condition	Context is not properly displayed.
Trigger	Tree Display is called by the cataloger browser.

DESCRIPTION	Step	Action
	1	The context is displayed in tree format.

USE CASE 13	**List Display**	
Goal in Context	Displays the context for the selected service and method in a list format.	
Scope & Level	Cataloger, subfunction.	
Preconditions	Method must be selected, and the list button must be clicked.	
Success End Condition	Context is displayed in list format.	
Failed End Condition	Context is not displayed.	
Primary actor	User.	
Trigger	Tree Display is called by the cataloger browser.	
DESCRIPTION	Step	Action
	1	The context is displayed in list format.

USE CASE 14	**Display Context**
Goal in Context	Displays the context for the selected service and method.
Scope & Level	User, primary function.
Preconditions	Method must be selected.
Success End Condition	Context is displayed.
Failed End Condition	Context is not displayed.

Primary actor	User.	
Trigger	User selects a method. User can select either the list/tree view.	
DESCRIPTION	Step	Action
	1	The current display is cleared.
	2	The currently selected method's context is displayed in either tree or list display, depending on whether the user has clicked the list/tree view button.

The component diagram in figure 8 represents the main components Catalog Browser, Catalog Dispatcher, Provider Dispatcher, Provider View, Context Browser Model, Context View and Context Dispatcher. The LCCM framework is developed on the MVC framework for displaying the context. [3] MVC stands for Model-View-Controller.

The MVC paradigm is a way of breaking an application, or even just a piece of an application's interface, into three parts: the model, the view, and the controller. MVC was originally developed to map the traditional input, processing, and output roles into the GUI realm.

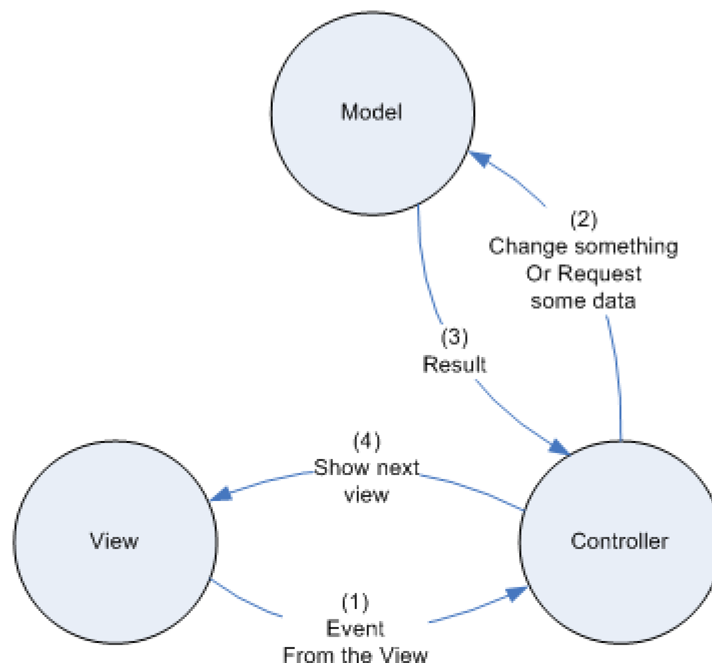

Figure 7: A simple diagram depicting the MVC framework [4]

The user input, the modeling of the external world, and the visual feedback to the user are separated and handled by model, viewport and controller objects. The controller interprets mouse and keyboard inputs from the user and maps these user actions into commands that are sent to the model and/or viewport to effect the appropriate change. The model manages one or more data elements, responds to queries about its state, and responds to instructions to change state.

Figure 8: Component Diagram for LCCM framework. The components in blue are the components added to form the LCCM framework.

The viewport manages a rectangular area of the display and is responsible for presenting data to the user through a combination of graphics and text.

The model is used to manage information and notify observers when that information changes. It contains only data and functionality that are related by a common purpose. If you need to model two groups of unrelated data and functionality, you create two separate models. A model encapsulates more than just data and functions that operate on it. A model is meant to serve as a computational approximation or abstraction of some real world process or system. It captures not only the state of a process or system, but how the system works. This makes it very easy to use real-world modeling techniques in defining your models. For example, you could define a model that bridges your computational back-end with your GUI front-end. In this scenario, the model wraps and abstracts the functionality of a computation engine or hardware system and acts as a liaison requesting the real services of the system it models. The [view or viewport] is responsible for mapping graphics onto a device. A viewport typically has a one to one correspondence with a display surface and knows how to render to it. A viewport attaches to a model and renders its contents to the display surface. In addition, when the model changes, the viewport automatically redraws the affected part of the image to reflect those changes. There can be multiple viewports onto the same model and each of these viewports can render the contents of the model to a different display surface. A viewport may be a composite viewport containing several sub-views, which may themselves contain several sub-views.

A controller is the means by which the user interacts with the application. A controller accepts input from the user and instructs the model and viewport to perform actions based on that input. In effect, the controller is responsible for mapping end-user action to application response. For example, if the user clicks the mouse button or chooses a menu item, the controller is responsible for determining how the application should respond. The model, viewport and controller are intimately related and in constant contact. Therefore, they must reference each other.

In figure 7 above shows the basic lines of communication among the model, viewport and controller. In this figure 7, the model points to the viewport, this allows it to send the viewport weakly-typed notifications of change. The model should know nothing about the kind of viewports which observe it. By contrast, the viewport knows exactly what kind of model it observes. The viewport also has a strongly-typed pointer to the model, allowing it to call any of the model's functions. In addition, the viewport also has a pointer to the controller, but it should not call functions in the controller aside from those defined in the base class. The reason is you may want to swap out one controller for another, so you'll need to keep the dependencies minimal. The controller has pointers to both the model and the viewport and knows the type of both. Since the controller

defines the behavior of the triad, it must know the type of both the model and the viewport in order to translate user input into application response.

The components in red in figure 8 depict the JINI components. The roles of the components Lookup service and Service Description are described below. The catalog service will actually be an object (or set of objects) which is living within a server. The server performs various tasks on behalf of the service. Primarily the server registers service with the lookup service/service locator. In order to perform this registration, the server must first find a lookup service. [5] This can be done in two ways: if the location of the lookup service is known, then the server can use unicast TCP to connect directly to it. If the location is not known, the server will make UDP multicast requests, and lookup services will respond to these requests. Lookup services will be listening on port 4160 for both the unicast and multicast requests. When the lookup service gets a request on this port, it sends an object back to the server. This object, known as a registrar, acts as a proxy to the lookup service, and runs in the service's JVM (Java Virtual Machine). Any requests that the server needs to make of the lookup service are made through this proxy registrar. Any suitable protocol may be used to do this. What the server does with the registrar is to register the service with the lookup service. The component in orange is the cataloger service.

Provider View, Context View, Context Browser Model, Context Dispatcher, Provider Dispatcher, Catalog Dispatcher and Catalog Browser form the LCCM framework which is shown in blue color in figure 8. The Catalog Dispatcher gets the service from the component Catalog Service. This component is responsible for interacting with the remote model. Coming to the component model depicted in figure 8 for the LCCM framework we have a single model which is context browser model. The views are the Provider View and the Context View with the Catalog Dispatcher and Context Dispatcher being the controllers. The Catalog Dispatcher is for the remote model. The context dispatcher listens to the events from the context view. The catalog browser window is responsible for displaying the entire LCCM framework encompasses provider view and context view. The catalog dispatcher is responsible for getting the service from the component catalog service. The provider view displays the list of providers, interfaces and methods. The data for displaying these details is got from the Context Browser Model. The Provider Dispatcher gets a change event from the Provider view. This change of event might be a change in the list of providers, interfaces or methods. It can also encompass the filtering and searching of providers, interfaces and methods. After the event is got by the provider dispatcher it is passed on to the Context Browser Model which evaluates the event and processes the data accordingly and gets the result. The nest step would be to update the views regarding the change in data and the views updating themselves accordingly. This also applies to the context browser model which handles the service context.

Interaction Diagram

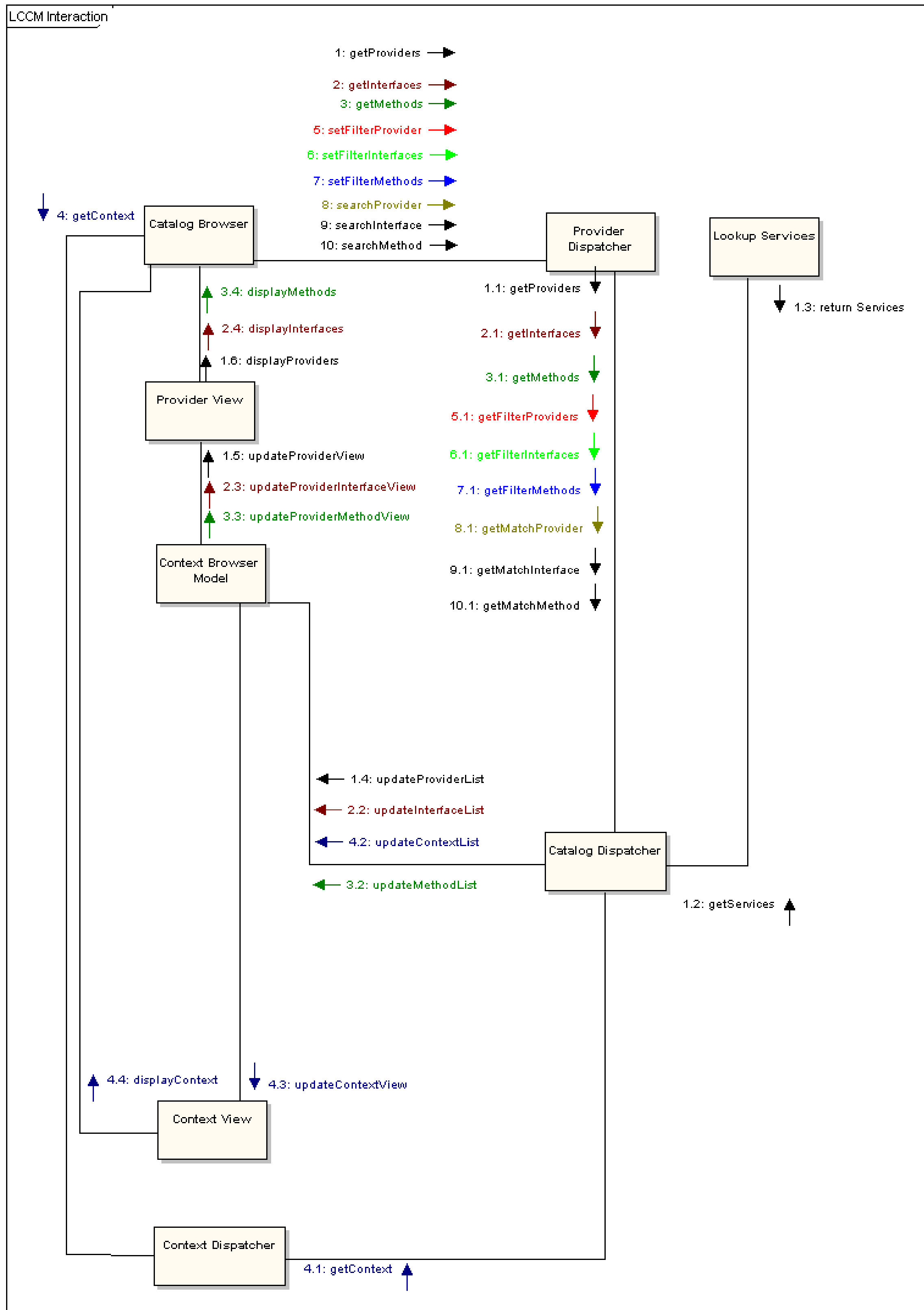

Figure 9: UML Interaction diagram between LCCM components

When the catalog browser window is opened for viewing the service context, the catalog dispatcher contacts the Lookup service and gets the service item. After the service is got, the provider list is updated in the context browser model and then the updated provider list is displayed in the provider view. Once all the service providers are displayed, we have the facility to search the provider list for a single provider or similar provider with the query string being the name of the provider. Sometimes it becomes a problem when lot many irrelevant service providers are displayed. Therefore we have the option to even filter out the service providers which is not relevant to our work. This search and filter option applies to Interfaces and Methods also. The context can only be displayed. It cannot be searched. The class diagram is shown in figure 10.

Class Diagram

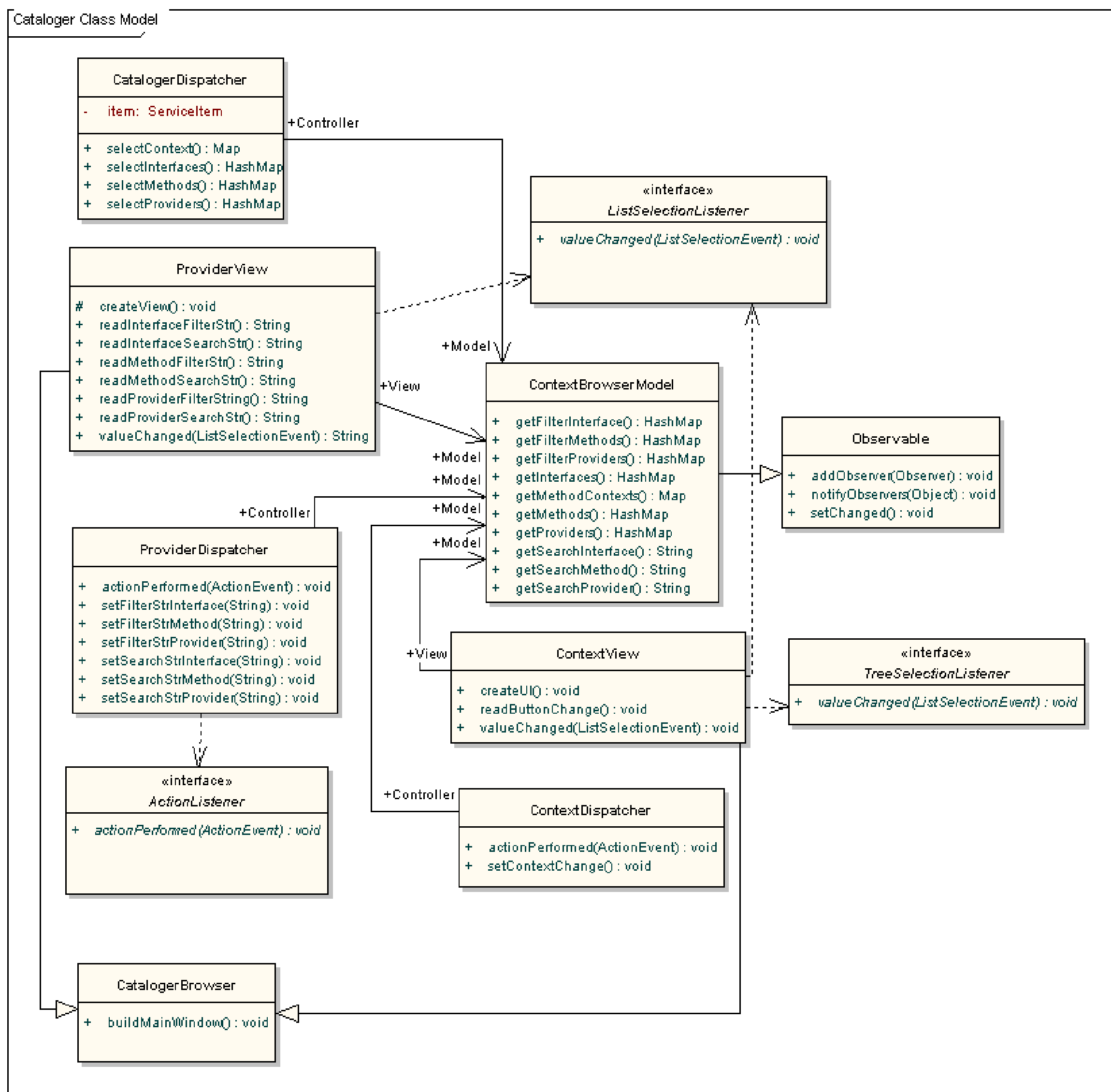

Figure 10: UML Class Diagram for LCCM framework

For implementing the GUI for LCCM framework, Java Swing was used. [6] Swing was used because it's a platform independent, Model-View-Controller GUI framework for Java. It follows a single-threaded programming model, and possesses traits such as platform independence. Swing platform independent both in terms of its expression (Java) and its implementation (non-native universal rendering of widgets).Swing is a highly partitioned architecture which allows for the 'plugging' of various custom implementations of specified framework interfaces: Users can provide their own custom implementation(s) of these components to override the default implementations. In general, Swing users can extend the framework by: extending existing (framework) classes; providing alternative implementations of core components. Swing is a component-based framework. The distinction between objects and components is a fairly subtle point: concisely, a component is a well-behaved object with a known/specified characteristic pattern of behavior. Swing objects asynchronously fire events, have 'bound' properties, and respond to a well known set of commands (specific to the component.) Specifically, Swing components are Java Beans components, compliant with the Java Beans Component Architecture specifications. Given the programmatic rendering model of the Swing framework, fine control over the details of rendering of a component is possible in Swing. As a general pattern, the visual representation of a Swing component is a composition of a standard set of elements, such as a 'border', 'inset', decorations, etc. Typically, users will programmatically customize a standard Swing component (such as a JTable) by assigning specific Borders, Colors, Backgrounds, etc., as the properties of the component. The core component will then use this property (settings) to determine the appropriate renderers to use in painting its various aspects. However, it is also completely possible to create unique GUI controls with highly customized visual representation. (Swing components support transparent rendering.)

Swing's heavy reliance on runtime mechanisms and indirect composition patterns allow it to respond at runtime to fundamental changes in its settings. For example, a Swing-based application can change its look and feel at runtime (from say MacOS look and feel to a Windows XP look and feel). Further, users can provide their own look and feel implementation, which allows for uniform changes in the look and feel of existing Swing applications without any programmatic change to the application code. The magic of Swing's configurability is also due to the fact that it does NOT use the native host OS's GUI controls for representation, but rather 'paints' its controls programmatically, through the use of Java 2D APIs. Thus, a Swing component does NOT have a corresponding native OS GUI 'peer', and is free to render itself in any way that is possible with the graphics APIs.

As shown in figure 10 the class provider dispatcher which is one of the controller implements the interface ActionListener. The cataloger dispatcher class is responsible for getting the service item from the lookup services and the initial set of providers. Based on the selection of a single provider the query is send to

the model to get the updated list of interfaces and so on. The Provider View implements the listselectionlistener which is necessary to listen to events in the provider list, interface list or the method list. The context view implements the treeselectionlistener in addition to the listselectionlistener since it has to be displayed in both the list format and the tree format.

4) Results

The LCCM framework helps in managing the service context for exertion-oriented programming. In a network where there are thousands of service providers it becomes difficult to manage them. Moreover it becomes very difficult to find the provider we need and also to work with the service context associated with one of the methods of the provider. Now due to this framework we are able to display all the providers in a network, search for a particular provider and also able to filter out the service providers not necessary through the filter method. The search and filter option is there for Interfaces and Methods.

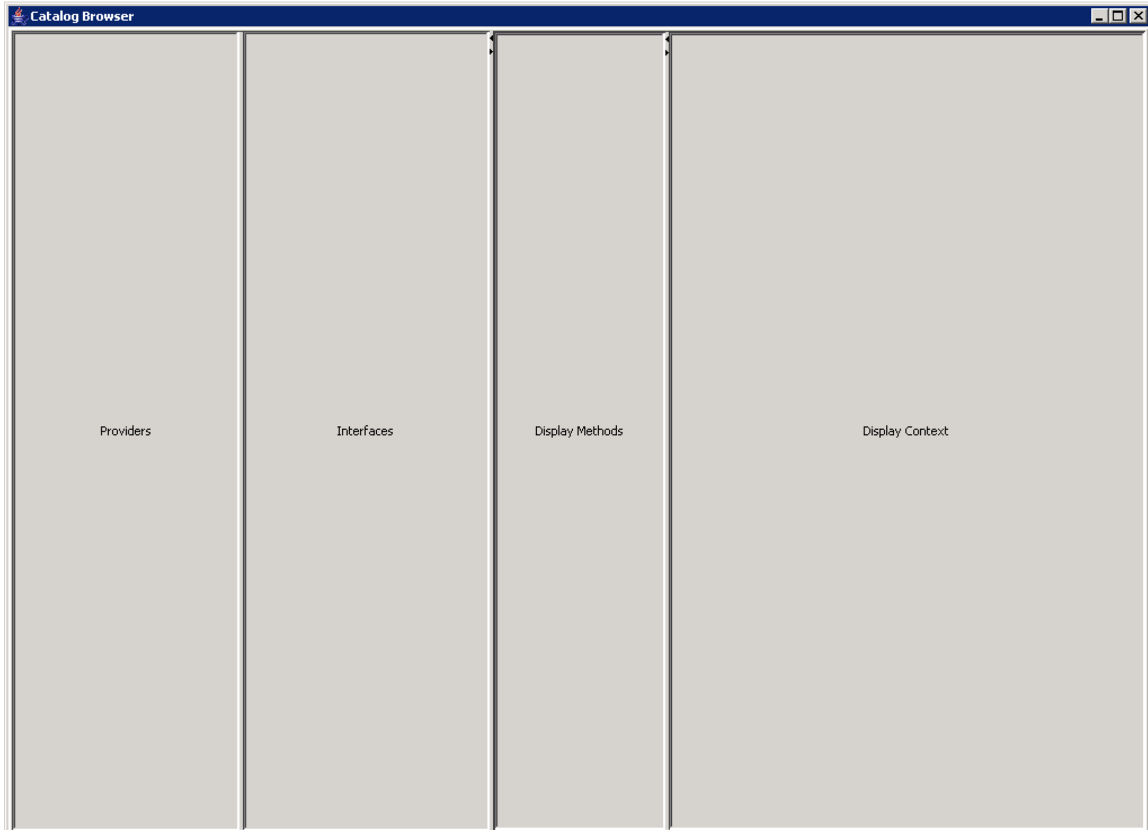

Figure 11: Partial implementation of the GUI showing where the provider, interfaces, methods and the context will be displayed.

5) Conclusions / Future Research

The advantages of the LCCM framework being but not limited to having a Uniform method context viewer, User friendly cataloger for viewing the context with respect to each method, Standardized data structure for transferring the context for service and so on. Due to this framework one will have a clear idea about the service context. Future work on this area will consist of editing the service context in the GUI itself. This will greatly help in exertion-oriented programming since we can input data to tasks and jobs and also view the output in the GUI itself.

6) References:

[1] Metacomputing with Federated Method Invocation by Dr.Michael Sobolewski.
[2] http://www.javaworld.com/javaworld/jw-10-1998/jw-10-blundon.html
[3] http://ootips.org/mvc-pattern.html
[4] http://www.stannard.net.au/blog/index.cfm/2006/9/1/Writing-Your-Own-Simple-MVC-Framework-in-ColdFusion
[5] Jan NewMarch tutorial
[6] www.wikipedia.org