5-2011

CHARM Components Monitoring System (CCMS)

Cemal Aktas
Utah State University

Recommended Citation

Aktas, Cemal, "CHARM Components Monitoring System (CCMS)" (2011). *All Graduate Plan B and other Reports*. 80.
https://digitalcommons.usu.edu/gradreports/80

CHARM COMPONENTS MONITORING SYSTEM (CCMS)

by

Cemal Aktas

A report submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Stephen Clyde Major Professor	Curtis Dyreson Committee Member

Nicholas Flann
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah

2011

ABSTRACT

CHARM Components Monitoring System

by

Cemal Aktas

Utah State University, 2011

Major Professor: Dr. Stephen Clyde
Department: Computer Science

Integrated systems need to be monitored regularly for operational problems such as network failures, loss of data, bottle necks, and other process failures. One of the best ways to monitor a system is through a graphical visualization of system activities. Humans are good at spotting patterns in graphic visualizations. This report describes a monitoring facility for integrated systems that can present wide range of graphics visualizations. As a proof of concept, it is implemented for the CHARM system, which is a distributed system used by the Utah Department of Health.

(73 pages)

ACKNOWLEDGMENTS

I would like to thank my major professor Dr. Stephen Clyde for his assistance, encouragement, directions, advice and guidance throughout my graduate level education. His recommendations and suggestions have been invaluable for both me and this project. I am also grateful for his patience throughout my graduate level education.

I am grateful to my committee members Dr. Curtis Dyreson and Dr. Nick Flann for their assistance, encouraging words, thoughtful criticism, time and attention during busy semesters.

I also thank the CHARM team leader Lan Hu for her support, guidance and patience. I have learned so many concepts from her.

Many thanks go to department secretaries Vicki Anderson and Genie Hanson. I thank Vicki and Genie for everything.

Words are not enough to express my thanks I owe to Hulya Aktas, my wife, for her endless support, love and encouragement. Without her love and support, it would be impossible for me to get my education. I am also grateful for my daughter, Izgi, and son, Beren, for their love and support. I am really sorry that I could not spend enough time with them during last five years because of my education.

Finally, I am grateful to my Heavenly Father for being with us all the time and hearing and answering our prayers.

Cemal Aktas

CONTENTS

Page

CHAPTER

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Distributed systems can consist of many independent services or components which interact with each other through message passing. Many factors, such as network traffic, transaction volume, local resource utilization, and network failures, etc., can affect the performance of a distributed system. It is difficult to identify performance problems quickly because the components can be hosted on multiple systems on separate networks. Because of this reason, monitoring the runtime performance of components can be effective way of spotting potential problems or inefficiency. Monitoring the performance of distributed components requires the following steps;

- Identifying meaningful metrics for each component that characterizes its performance.
- Determining the frequency and scales at which measurements for these metrics need to be gathered from the components.
- A way to gather these measurements into a center process that can process or aggregate them into visualization or query tool.
- Deciding what kind of visualizations would help system operators to identify potential problems.

This project, called *Charm Components Monitoring System* (*CCMS*), presents a monitoring system that monitors different components in real time. It allows a user to select components and monitor these selected components based on user defined time intervals. It provides visual graphs to make the monitoring easier. Requirements for this project can be found in chapter 2 and the system analysis, design and implementation details are explained in chapters 3, 4, and 5 respectively.

CHARM (Child Health Advanced Records Management) is a very big and complex system which has been used by Utah Department of Health (UDOH) since 2002. CHARM is a distributed system for integrating Child-Health Care Data for Utah Department of Health. “CHARM links child health information from several programs.”[5] CHARM has been using several components such as address cleaner, different agents for different resources, alerts, audit manager, core agent, matcher, query manager, and sync engine. Each component has a crucial role in the system.

Because CHARM has been using several components running on both the same and different machines, monitoring these components is important to observe its performance. Monitoring systems’ purpose is not debugging, they are not debugging tools. The main purpose of monitoring systems is to observe the performance of a system. Using monitoring systems, the weak points of a system can be discovered and necessary steps can be taken.

The Monitoring System for CHARM has the same purpose. Its purpose is to observe if any of the selected components need some improvements. Using the Monitoring System, we can see if any of the components becomes a bottle-neck.

The Vitruvian is a framework that handles serializing/deserializing objects, the distribution of objects, and all of the underlying communications, which includes taking care of the socket connections and ensuring their aliveness. The Vitruvian framework is used in this project because the CHARM system will use the Vitruvian framework in version 3.0. The usage of the Vitruvian framework is explained in chapter 6.

One of the purposes of using the Vitruvian framework is to test the Vitruvian framework's usage. Several test cases and different types of testing were applied to CCMS in order to make sure the Vitruvian framework and CCMS work together. Chapter 7 is the software testing section and this chapter explains what kind of tests we used for this project.

Using CCMS is pretty simple and it is easy to learn. It provides users a very easy to use user interface to monitor the components. Chapter 8 contains a quick user guide for the usage of CCMS.

There are still some other works to do in the future and these works and the conclusion can be found in section 9. This section explains what to do in the future to implement CCMS to the CHARM system.

CHAPTER 2

REQUIREMENTS

This chapter covers both functional and non-functional requirements. Even though this monitoring system will be used for Charm, it can be used by other distributed systems that use the Service Oriented Architecture. These requirements have been determined based on Charm structure and its components.

2.1. Functional Requirements

2.1.1. Settings:

2.1.1.1. The system shall allow users to change settings.

2.1.1.2. Users shall be able to save settings in the HDD.

2.1.1.3. The system shall allow users to change how many bars will be displayed on the screen for each graph.

2.1.1.3.1. Users shall be able to enter a number between 10 and 500 for the number of bars. Other values shall not be allowed.

2.1.1.4. The system shall allow users to change the graph refresh time.

2.1.1.4.1. Users shall be able to enter a number between 50 and 900,000 for the graph refresh time. Other values shall not be allowed.

2.1.1.4.2. These numbers shall be used by the system as millisecond.

2.1.1.5. The system shall allow users to change the time interval that components send their values.

2.1.1.5.1. Users shall be able to enter a number between 50 and 900,000 for time interval for components to send their values. Other values shall not be allowed.

2.1.1.5.2. These numbers shall be used by the system as millisecond.

2.1.1.6. The system shall allow users to change the colors of the bars in the graph(s).

2.1.1.6.1. Users shall be able to custom 10 different colors and see them all in the settings.

2.1.1.7. After making changes on the settings, new settings shall be used by the system immediately. Users shall not re-start the application to use new settings.

2.1.1.8. After making some changes on settings, users shall be able to cancel these changes.

2.1.1.9. After making changes, if users do not save the changes, next time when the application is loaded, previously saved values shall be loaded.

2.1.1.9.1. If there are no previously saved values, default values shall be loaded by the system.

2.1.2. Graph Form:

2.1.2.1. Users shall be able to see the list of the components that can be selected.

2.1.2.2. Users shall be able to select one or more or all of the components at a time.

2.1.2.3. Users shall be able to go to settings screen after selecting components.

2.1.2.4. After selecting component(s), users shall be able to see the graphs for selected components in one window.

2.1.2.5. If there are no selected components, the system shall warn users to make selection first.

2.1.2.6. The form shall support liquid layout pattern.

2.1.2.6.1. As the user resizes the window, the page contents shall be resized along with it so that the page is constantly filled.

2.1.3. Components:

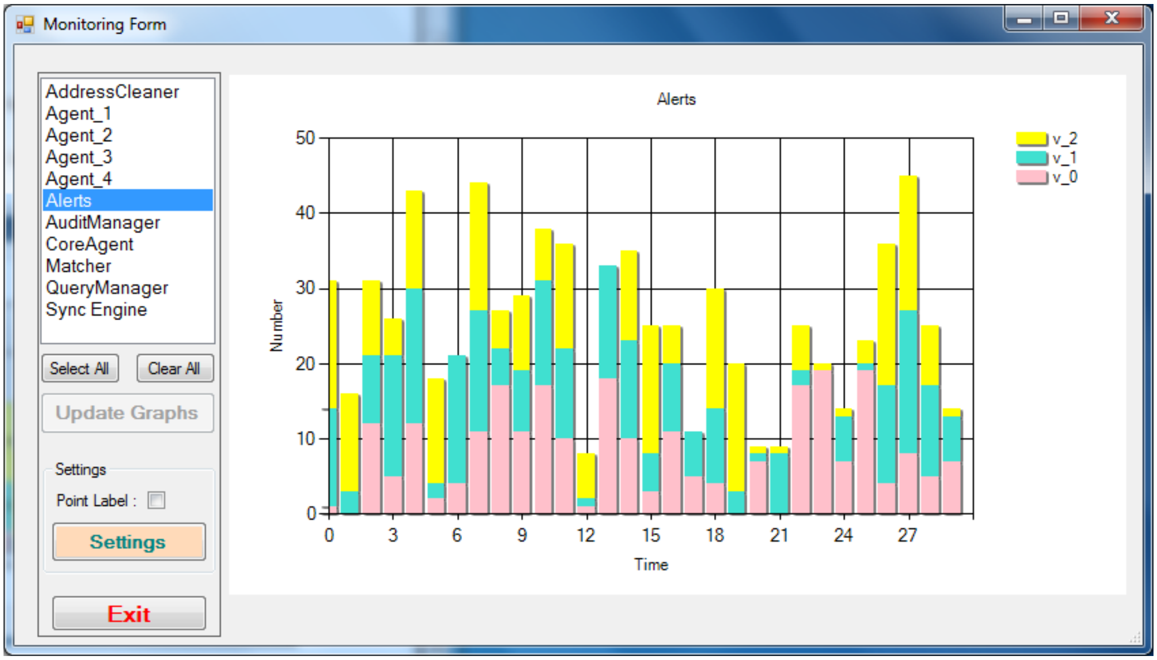
2.1.3.1. Components shall keep track of numbers that will be sent to the monitoring service.

2.1.3.2. These numbers shall be set to zero, once the monitoring system tells a component to stop sending numbers.

2.1.4. Graphs:

2.1.4.1. Graphs shall use bars to display values.

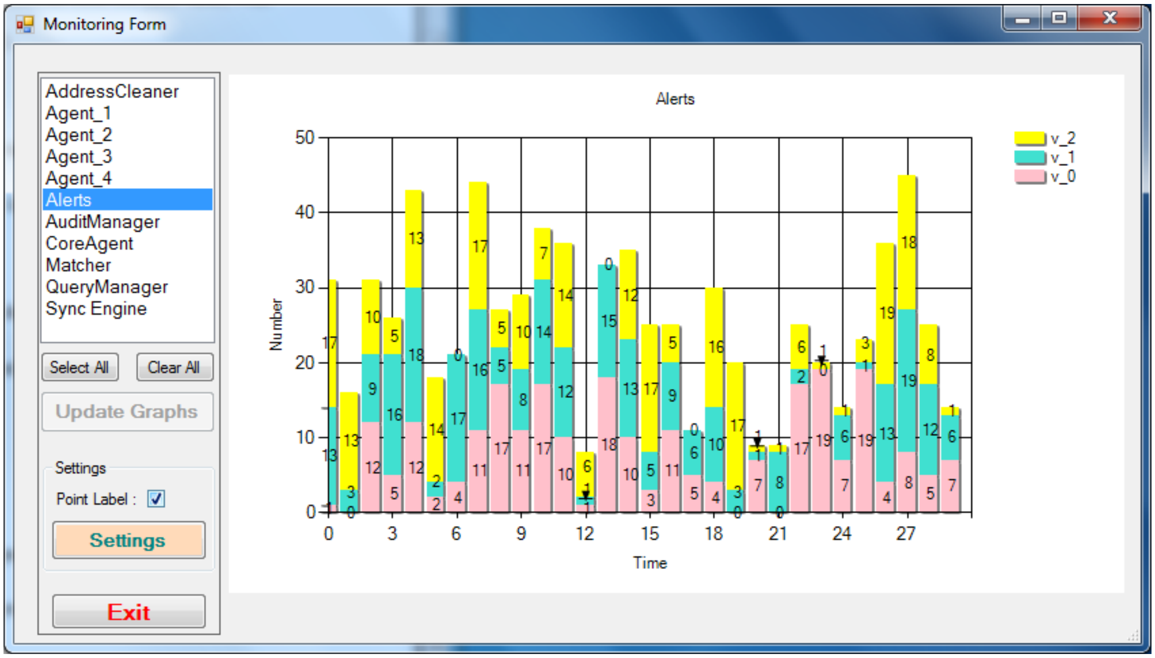
2.1.4.2. If a component sends more than one value at a time, these values set shall be displayed on a multi-bar graph with a different color for each value, with one axis representing the time (Figure 2.1).



Point Label : ☐

Figure 2.1 - For each value, a different color is used on a bar.

2.1.4.3. The user shall be able to turn the value labels on/off (Figure 2.2).



Point Label : ☑

Figure 2.2 – Value labels are displayed on the bar for each color.

2.1.4.4. The meaning of colors shall be displayed on the graph(s) (Fig. 2.3).

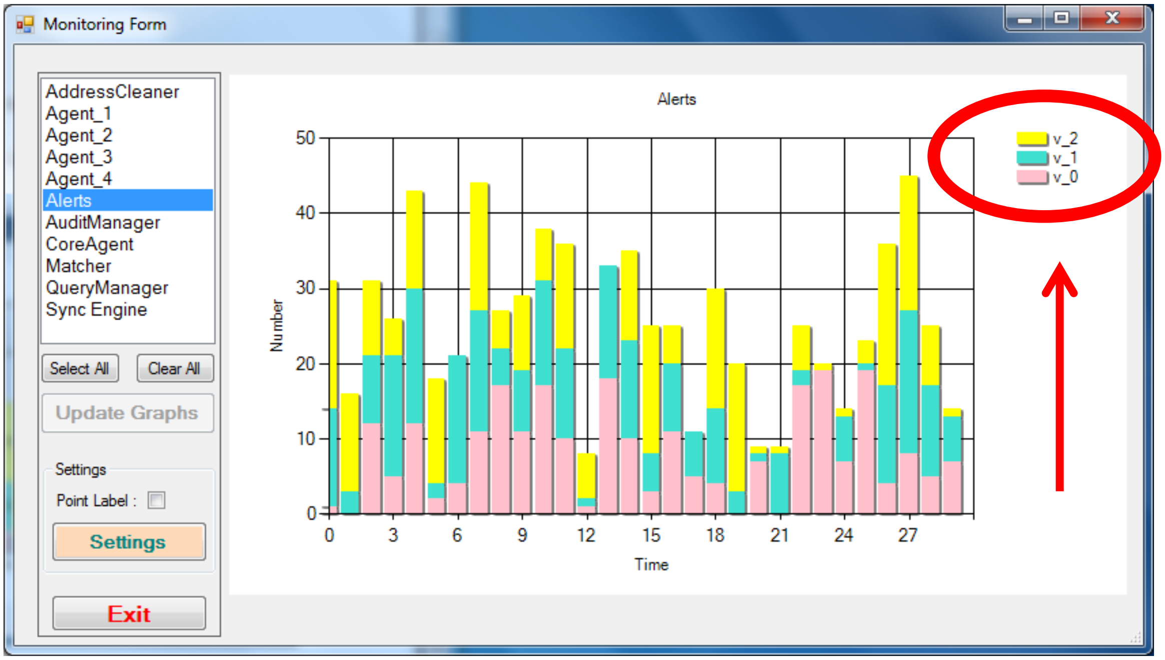


Figure 2.3 - Meaning of colors are displayed on the graph(s).

2.1.4.5. The component's name shall be displayed on top of the graph.

** Figures 2.1, 2.2, and 2.3 are just for an illustration of a multi-bar graph and the actual display does not need to use the same color scheme and does not have the same size or portions.

2.2. Non-functional Requirements:

2.2.1. Programming language shall be C# (C-Sharp).

2.2.2. The system shall use Vitruvian for distribution.

2.2.3. The system shall use windows operating system.

2.2.4. The system shall have dummy components for testing and these components shall send random numbers to the Monitoring System for testing.

2.2.5. The system shall have comprehensive unit tests for each class.

2.2.6. Design documents and a report about the system shall be given.

2.2.7. System Architecture:

2.2.7.1. The system shall use *Service Oriented Architecture*.

2.2.7.2. GUI shall not be allowed to communicate with *Components* directly.

2.2.7.3. GUI shall communicate with a *service* and this *service* will communicate with other components.

CHAPTER 3

SYSTEM ANALYSIS

3.1. Overview

System analysis is the breaking of a system into its components (or pieces) to see how these components (or pieces) communicate each other and work. It is the process of investigating a system, identifying problems, and searching possible improvements about the system [7]. System Analysis is concerned with the classes, objects and mechanisms that are present in the problem domain. The classes are identified along with their relationships to each other, and described in UML class diagrams [4].

The use case diagram captures the user goals. The use case tells developers with a high-level overview of who will use the Charm Components Monitoring System (CCMS) [12]. It also tells what can be done using the CCMS. The use case diagram is explained in section 3.2. and section 3.3 contains the class diagrams. The class diagrams describe the key objects and their relations in the system. When used as part of a systems analysis, they can help developers to solidify understanding of system's components and thus set the stage for a more informed design [2].

3.2. Use Case (User Goals)

As mentioned earlier, the purpose of the CCMS is not debugging but to spot potential problems or inefficiency in real-time. For example, it can help system operators determine if a component has become a bottle-neck and then monitor any corrective actions.

The use-case diagram in Figure 3.1 shows that the user can select the components to monitor. The user also has an option to make changes on *settings* (Figure 3.2). Using settings, a user can change;

- *Number of Bars* - max # of bars for each graph
- *Graph Refresh Time* – Time period for refreshing the screen for graphs
- *Component's Send Values Time* – Time period components use to send values
- *Colors* – Colors of bars in graphs
- *Save* – Saves settings on the *HDD* (Figure 3.2).

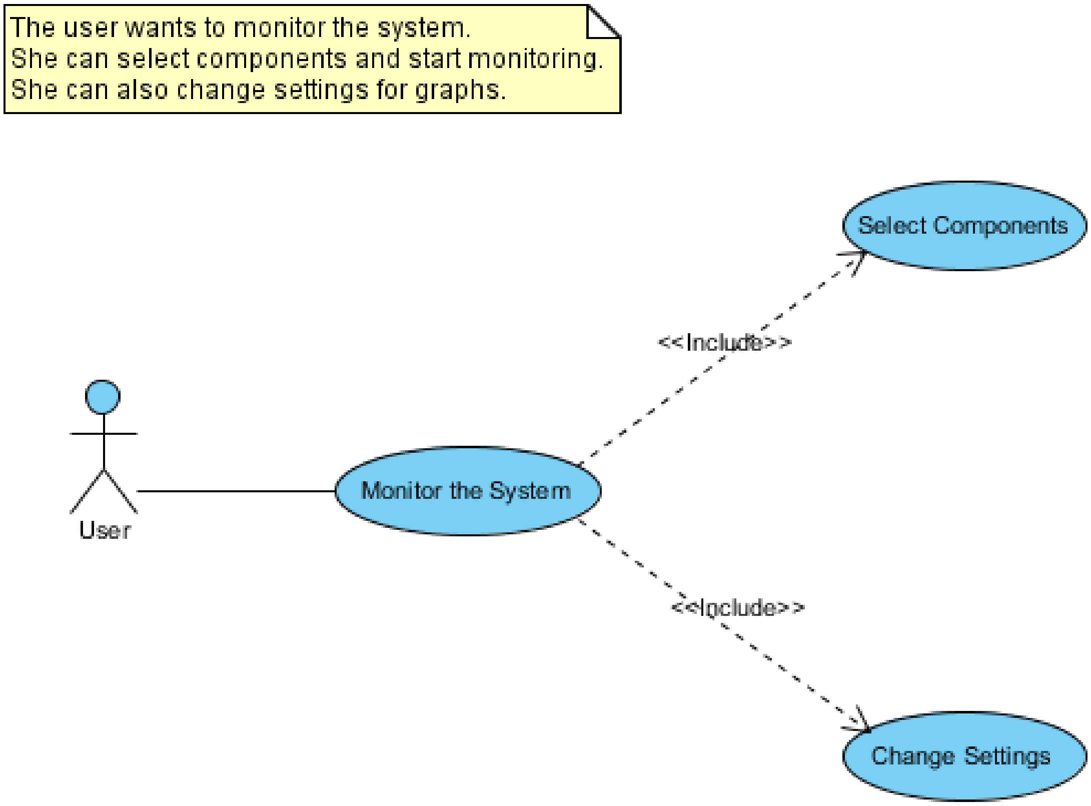

The user wants to monitor the system.
She can select components and start monitoring.
She can also change settings for graphs.
Select Components
<<Include>>
User
Monitor the System
<<Include>>
Change Settings

Figure 3.1 - Use case of the *CCMS*

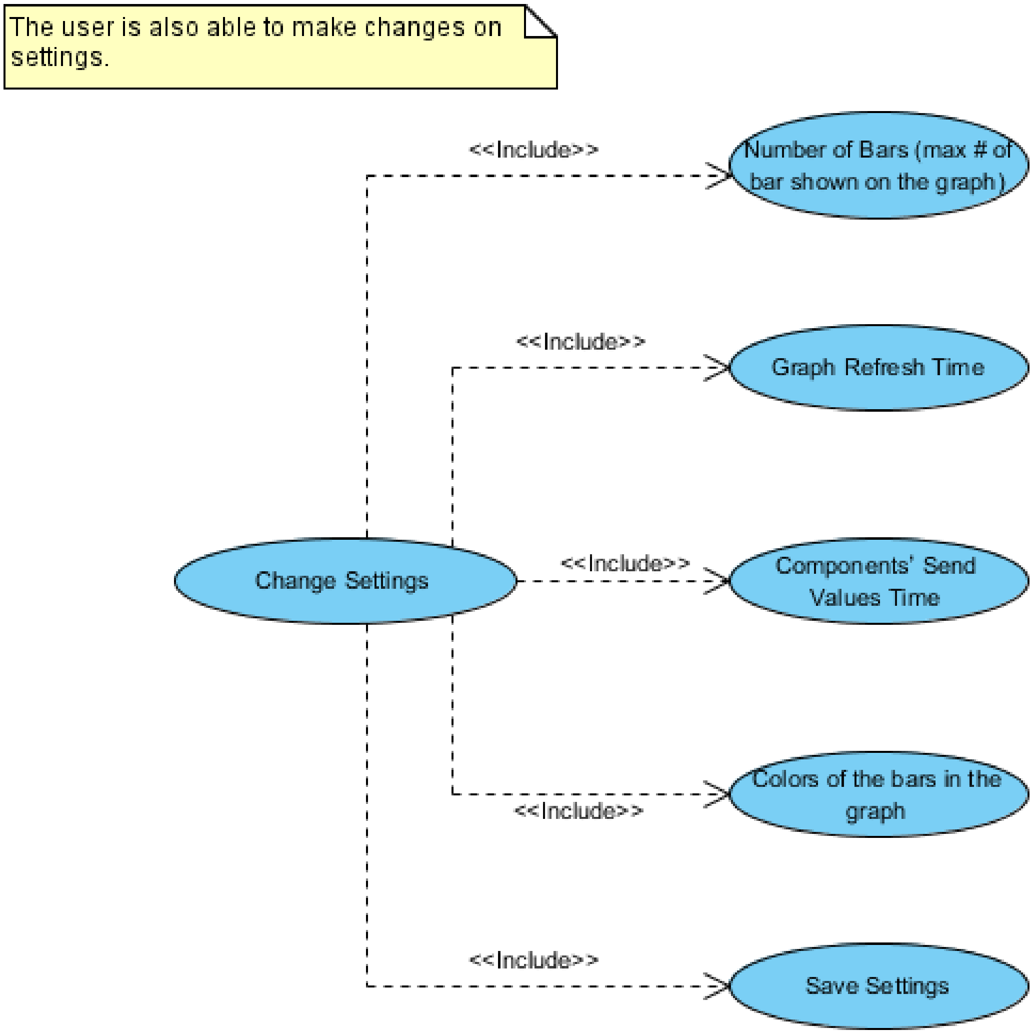

The user is also able to make changes on settings.
<<Include>>
Number of Bars (max # of bar shown on the graph)
<<Include>>
Graph Refresh Time
Change Settings
<<Include>>
Components' Send Values Time
Colors of the bars in the graph
<<Include>>
<<Include>>
Save Settings

Figure 3.2 – A user can also change the settings of the *CCMS*

3.3. Class Diagrams

The class diagrams in Figures 3.3 and 3.4 show domain classes and communication between objects. Because of the purpose of the system analysis, the class diagrams in Figures 3.3 and 3.4 do not include solution-domain details.

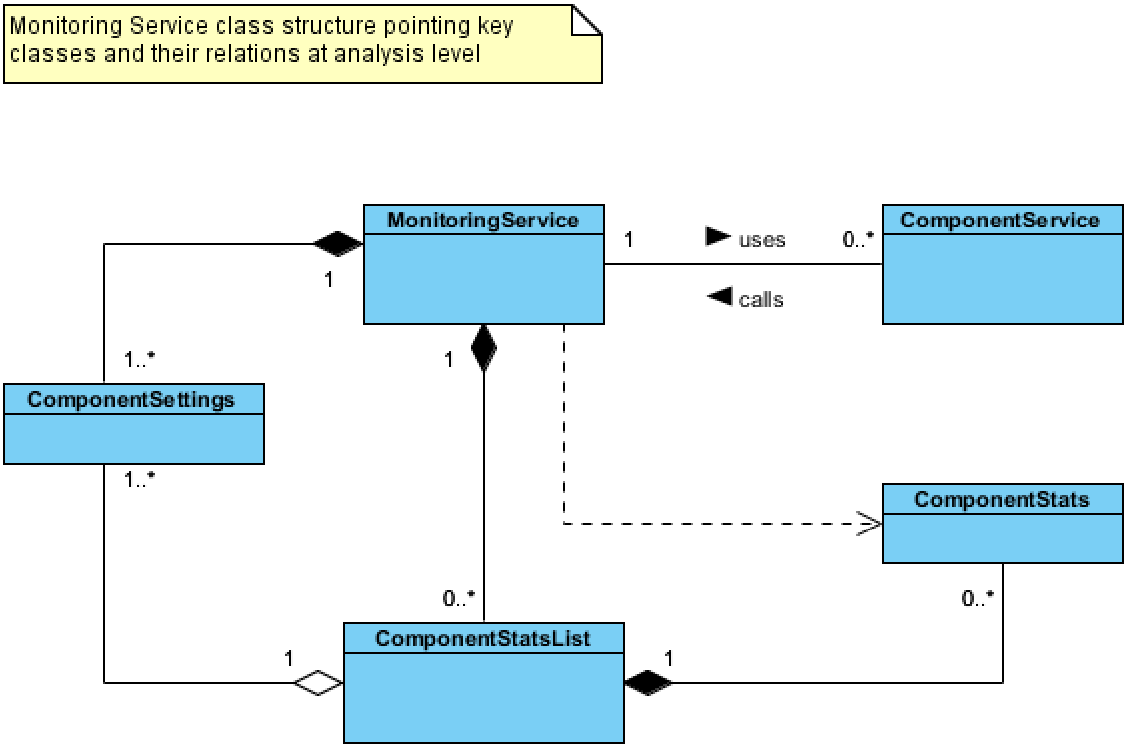


Figure 3.3 – Class structure for *MonitoringService* at analysis level.

Below you can see the brief overview of the classes.

- **MonitoringService:** This class is responsible to call selected components' ComponentServices to start/stop sending their values for monitoring.
- **ComponentService:** This class is responsible to start/stop sending values to the MonitoringService.
- **ComponentSettings:** This class is used to store components' information such as component name and how many numbers are sent at a time.
- **ComponentStats:** This class is responsible to store the values for a component.
- **ComponentStatsList:** This class is responsible to store all selected components' values.

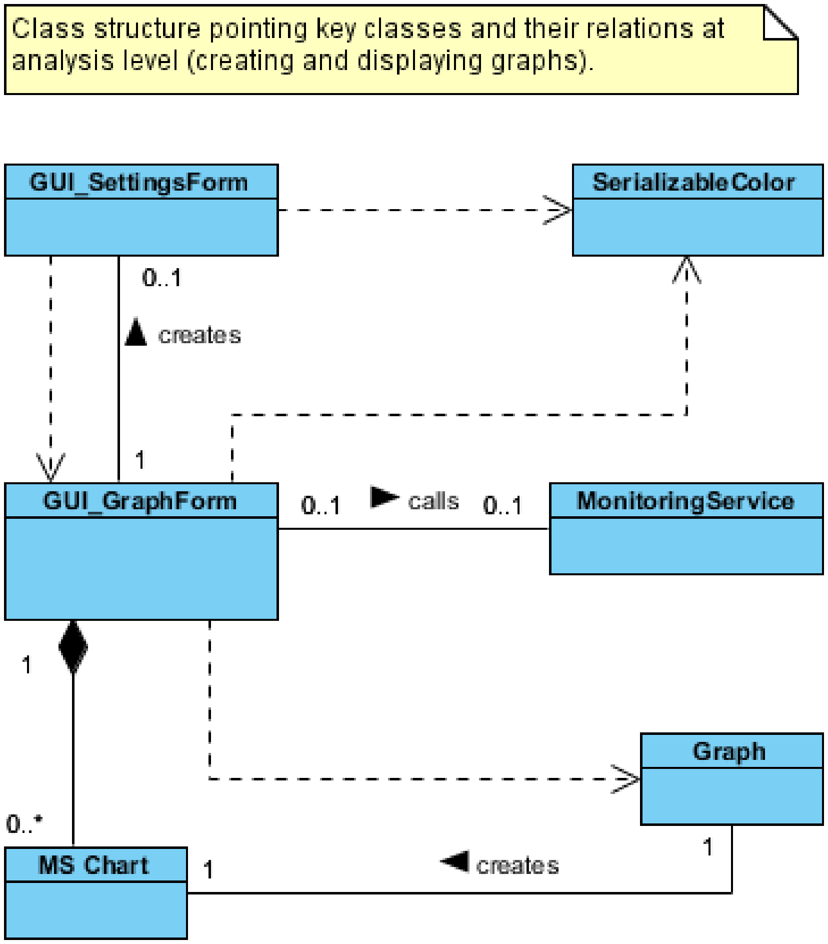


Figure 3.4 – Class structure for *GUI* at analysis level

- **GraphForm:** This class allows selecting components and displays the graphs.
- **SettingsForm:** This class allows users to change/save settings.
- **SerializableColor:** This class allows serialize and de-serialize colors user chose on SettingsForm.
- **Graph:** This class creates graphs and sets their locations on the screen dynamically.
- **MS Chart:** This is Microsoft's Chart library. It allows creating and displaying graphs.

CHAPTER 4

ARCHITECTURAL DESIGN

4.1. System Architecture

This chapter explains the architectural design of the *Charm Components Monitoring System (CCMS)*. UML diagrams provide industry standard mechanisms for visualizing, specifying, constructing, and documenting software systems. Class diagrams show the static structure of classes in the system [4]. “System design is the process of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements.”[8]. In design, the result of the analysis is expanded into a technical solution. New classes can be added to support the technical infrastructure. The domain problem classes from the analysis are embedded into this technical infrastructure [4].

Different packages have been used to make the design object oriented. Each package has classes that are related to each other or used together. In this way, the maintenance and testing became much easier. Having different packages also allowed to have low coupling among different classes and methods.

One of the interesting design decisions is that the system has been built using the Vitruvian framework. The main reason using the Vitruvian framework was that it made the distribution and logging easier. It takes care of the connection problems. Vitruvian framework is explained in chapter 6 in details.

Figures 4.1 and 4.2 show new classes and packages for CCMS. Figures 4.3 and 4.4 include the attributes and methods of these classes. The package *Vitruvian* contains classes and methods for distributing objects. It also contains classes and methods for logging.

The package *Monitoring* contains classes for both MonitoringService and ComponentService. The package *MonitoringForm* contains classes for creating and displaying graphs on the screen. Each package is explained in following sections.

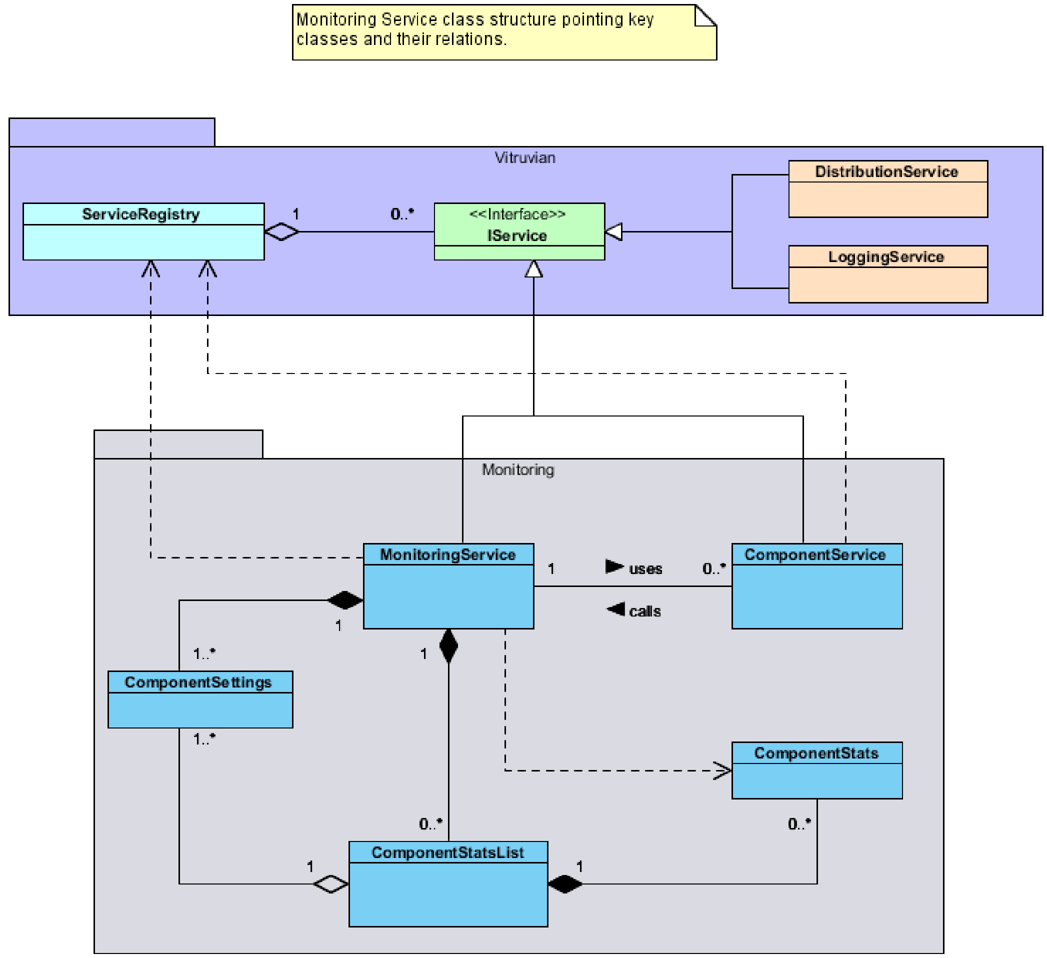


Figure 4.1 – A package diagram displaying architecture of services of the CCMS (not detailed)

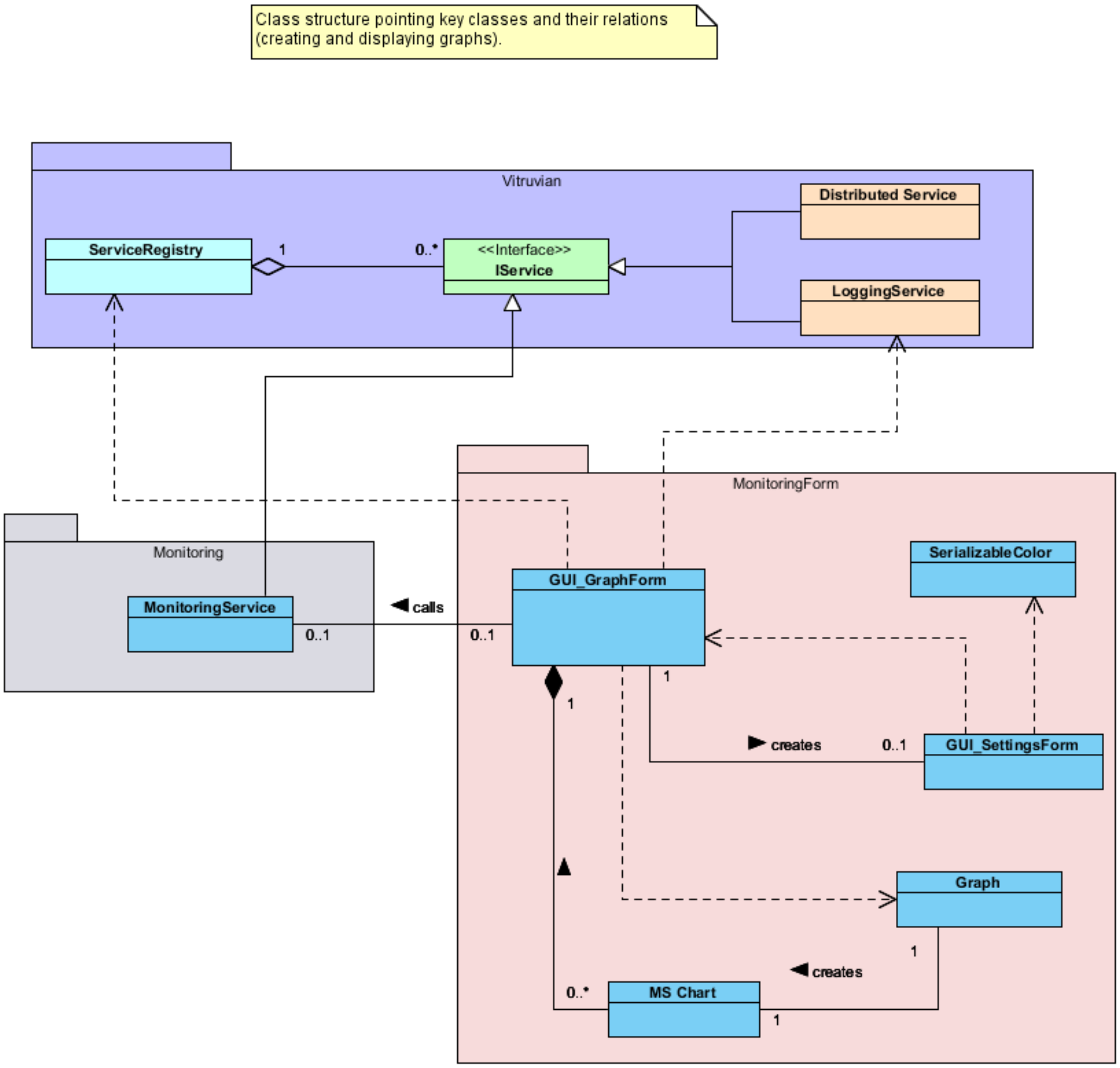


Figure 4.2 – A package diagram displaying architecture of GUI part of the CCMS (not detailed)

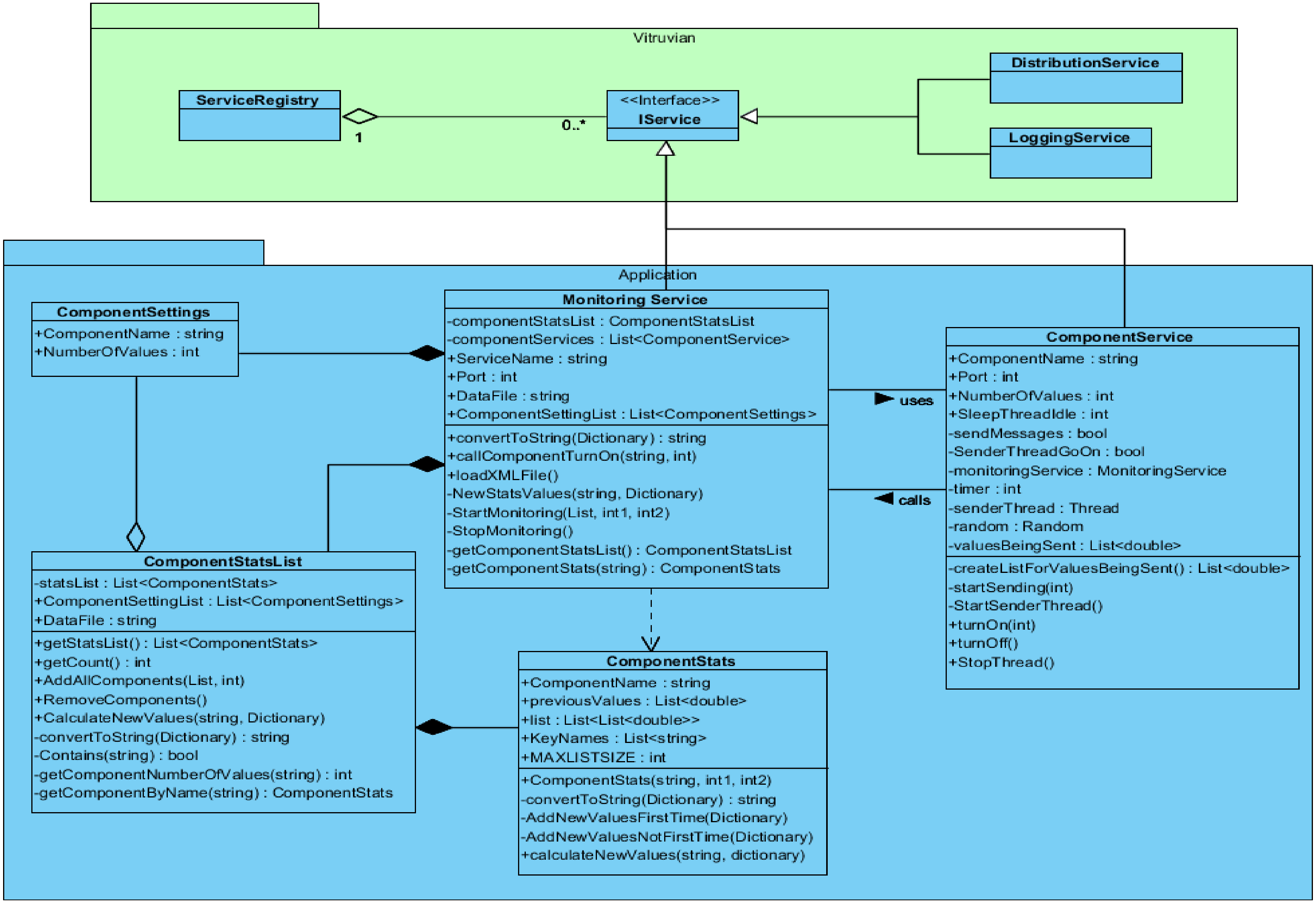

Vitruvian
ServiceRegistry
1
0..*
<<Interface>>
IService
DistributionService
LoggingService
Application
ComponentSettings
+ComponentName : string
+NumberOfValues : int
Monitoring Service
-componentStatsList : ComponentStatsList
-componentServices : List<ComponentService>
+ServiceName : string
+Port : int
+DataFile : string
+ComponentSettingList : List<ComponentSettings>
+convertToString(Dictionary) : string
+callComponentTurnOn(string, int)
+loadXMLFile()
-NewStatsValues(string, Dictionary)
-StartMonitoring(List, int1, int2)
-StopMonitoring()
-getComponentStatsList() : ComponentStatsList
-getComponentStats(string) : ComponentStats
uses
calls
ComponentService
+ComponentName : string
+Port : int
+NumberOfValues : int
+SleepThreadIdle : int
-sendMessages : bool
-SenderThreadGoOn : bool
-monitoringService : MonitoringService
-timer : int
-senderThread : Thread
-random : Random
-valuesBeingSent : List<double>
-createListForValuesBeingSent() : List<double>
-startSending(int)
-StartSenderThread()
+turnOn(int)
+turnOff()
+StopThread()
ComponentStatsList
-statsList : List<ComponentStats>
+ComponentSettingList : List<ComponentSettings>
+DataFile : string
+getStatsList() : List<ComponentStats>
+getCount() : int
+AddAllComponents(List, int)
+RemoveComponents()
+CalculateNewValues(string, Dictionary)
-convertToString(Dictionary) : string
-Contains(string) : bool
-getComponentNumberOfValues(string) : int
-getComponentByName(string) : ComponentStats
ComponentStats
+ComponentName : string
+previousValues : List<double>
+list : List<List<double>>
+KeyNames : List<string>
+MAXLISTSIZE : int
+ComponentStats(string, int1, int2)
-convertToString(Dictionary) : string
-AddNewValuesFirstTime(Dictionary)
-AddNewValuesNotFirstTime(Dictionary)
+calculateNewValues(string, dictionary)

Figure 4.3 – A package diagram displaying architecture of services of the CCMS (detailed)

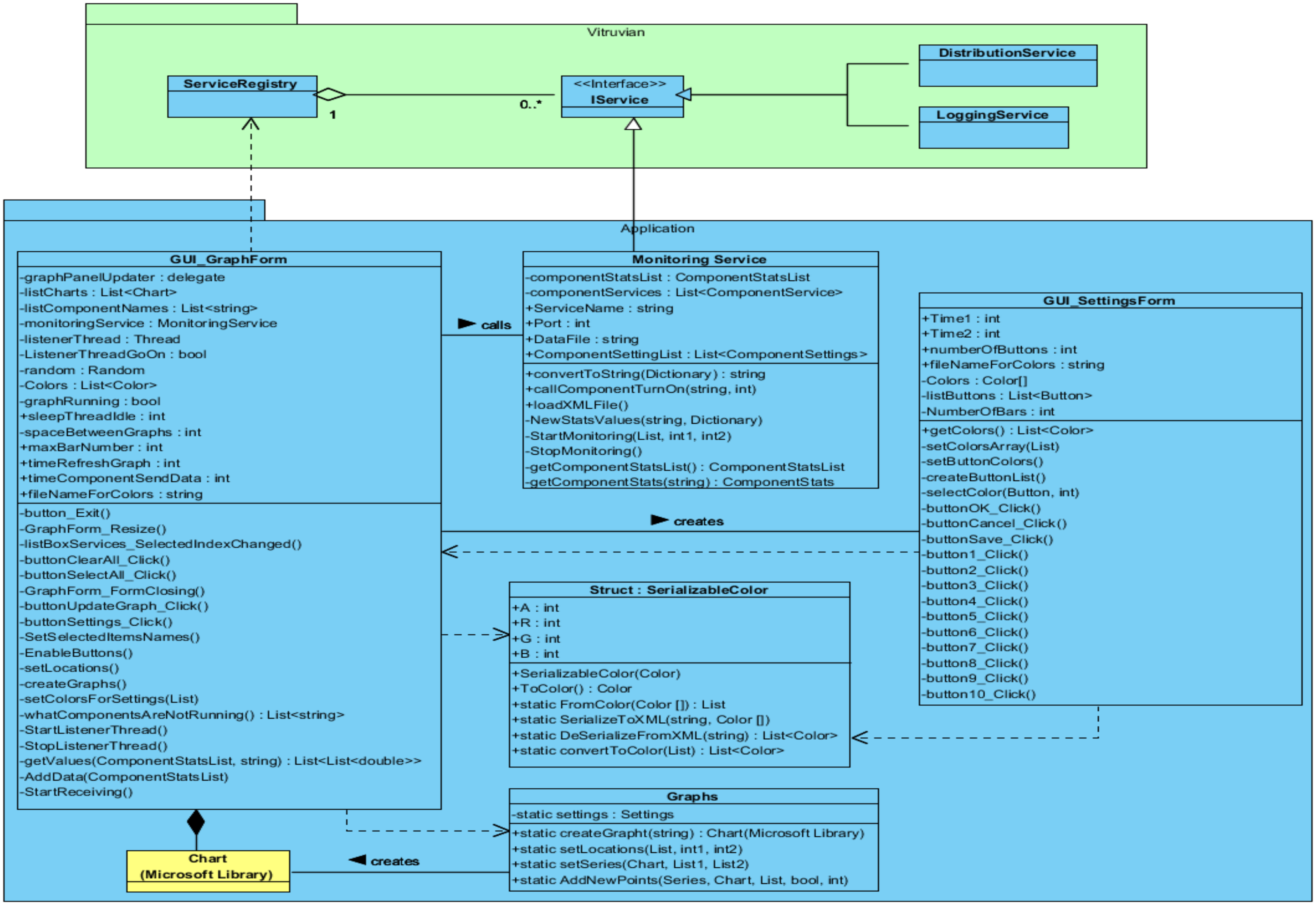

Vitruvian
ServiceRegistry
1
0..*
<<Interface>>
IService
DistributionService
LoggingService
Application
GUI_GraphForm
-graphPanelUpdater : delegate
-listCharts : List<Chart>
-listComponentNames : List<string>
-monitoringService : MonitoringService
-listenerThread : Thread
-ListenerThreadGoOn : bool
-random : Random
-Colors : List<Color>
-graphRunning : bool
+sleepThreadIdle : int
-spaceBetweenGraphs : int
+maxBarNumber : int
+timeRefreshGraph : int
+timeComponentSendData : int
+fileNameForColors : string
-button_Exit()
-GraphForm_Resize()
-listBoxServices_SelectedIndexChanged()
-buttonClearAll_Click()
-buttonSelectAll_Click()
-GraphForm_FormClosing()
-buttonUpdateGraph_Click()
-buttonSettings_Click()
-SetSelectedItemsNames()
-EnableButtons()
-setLocations()
-createGraphs()
-setColorsForSettings(List)
-whatComponentsAreNotRunning() : List<string>
-StartListenerThread()
-StopListenerThread()
-getValues(ComponentStatsList, string) : List<List<double>>
-AddData(ComponentStatsList)
-StartReceiving()
calls
Monitoring Service
-componentStatsList : ComponentStatsList
-componentServices : List<ComponentService>
+ServiceName : string
+Port : int
+DataFile : string
+ComponentSettingList : List<ComponentSettings>
+convertToString(Dictionary) : string
+callComponentTurnOn(string, int)
+loadXMLFile()
-NewStatsValues(string, Dictionary)
-StartMonitoring(List, int1, int2)
-StopMonitoring()
-getComponentStatsList() : ComponentStatsList
-getComponentStats(string) : ComponentStats
GUI_SettingsForm
+Time1 : int
+Time2 : int
+numberOfButtons : int
+fileNameForColors : string
-Colors : Color[]
-listButtons : List<Button>
-NumberOfBars : int
+getColors() : List<Color>
-setColorsArray(List)
-setButtonColors()
-createButtonList()
-selectColor(Button, int)
-buttonOK_Click()
-buttonCancel_Click()
-buttonSave_Click()
-button1_Click()
-button2_Click()
-button3_Click()
-button4_Click()
-button5_Click()
-button6_Click()
-button7_Click()
-button8_Click()
-button9_Click()
-button10_Click()
creates
Struct : SerializableColor
+A : int
+R : int
+G : int
+B : int
+SerializableColor(Color)
+ToColor() : Color
+static FromColor(Color []) : List
+static SerializeToXML(string, Color [])
+static DeSerializeFromXML(string) : List<Color>
+static convertToColor(List) : List<Color>
Graphs
-static settings : Settings
+static createGrapht(string) : Chart(Microsoft Library)
+static setLocations(List, int1, int2)
+static setSeries(Chart, List1, List2)
+static AddNewPoints(Series, Chart, List, bool, int)
Chart
(Microsoft Library)
creates

Figure 4.4 – A package diagram displaying architecture of GUI part of the CCMS (detailed)

4.2. Monitoring Package (Application Layer)

The Monitoring package contains classes for services. The *MonitoringService* and the *ComponentService* are the names of the services. This package is a library for services. Following sections provide details about services.

4.2.1. MonitoringService

MonitoringService is a distributed service. It uses the *Vitruvian* framework to distribute objects. It inherits Vitruvian's IService interface and it implements IService's methods. Since it is distributed, GUI and other components communicate with the *MonitoringService* via *ServiceRegistry* (ServiceRegistry is another component of the Vitruvian). *MonitoringService* is the core class of CCMS. Using the GUI, a user communicates with the MonitoringService to start/stop monitoring the selected components. Selected components also communicate with the *MonitoringService* to send their values. These values are stored in the *MonitoringService*. Then another thread, in the GUI, calls the *MonitoringService* to receive these values. The GUI and selected components call some methods inside the *MonitoringService*. These methods have *sync patterns* and they are declared as *virtual* (Vitruvian requires this).

4.2.2. ComponentService

Like MonitoringService, *ComponentService* is a distributed service too. Some methods are declared as *virtual* and have *sync pattern* since the *MonitoringService* is calling some of its methods. The *ComponentService* is responsible to send the component's values to the *MonitoringService*. The MonitoringService communicates with components via their *ComponentServices*. Components send their values to the MonitoringService via their ComponentServices.

4.2.3. ComponentStats

The *ComponentStats* class is responsible to add received values to the related component's graph. It also stores the key names for each received values and these names are also displayed on the graphs. Key names are the names that the numbers represent for. After displaying the numbers on the graphs, users need to know what those displayed values for. For example, the component called *QueryManager* sends three sets of values (total # of request, # of requests sent to the agents, and # of replies received from agents). For each time interval, the QueryManager sends new 3 sets of values to the *MonitoringService*. Then the MonitoringService calls *ComponentStats* and this object makes some calculations and adds new values to the *QueryManager's* value list and this list is used by the graph. *ComponentStats* is storing the key names dynamically so that if a component changes the order of these values, the *CCMS* can still display the values in the right order because the *MonitoringService* is not dependent on the order of the values. *ComponentStats* class provides a convenient mapping between a key name and a set of numbers.

4.2.4. ComponentStatsList

This class is used by the MonitoringService. The ComponentStatsList class contains ComponentStats objects. The ComponentStatsList and the ComponentStats classes are created to make the design more OO. First design did not have these two classes and all methods were in the MonitoringService class. Then the MonitoringService class became very bulky and too big. To prevent high coupling, the ComponentStats and the ComponentStatsList Classes were created and they reduced high coupling among methods in the MonitoringService. In current design, classes and their methods became easier to maintain and test. This class allows abstractions and some implementations are being hidden from a user.

4.2.5. ComponentSettings

The ComponentSettings class is used by the MonitoringService too. The MonitoringService class reads a file and loads the components' information using this class. This is a very small class and it has only two attributes (ComponentName and NumberOfValues respectively). The *ComponentStatsList* uses this class to create the # of containers for the selected components so that The ComponentStats can put the new values in the correct containers.

4.3. MonitoringForm Package (User Interface Layer)

The *MonitoringForm* package contains classes for GUI. These classes provide the communication with the *MonitoringService*. Users can select the components that will be monitored and send the list of the component names to the MonitoringService to start monitoring them. Also, this package is responsible to get the selected components' values from the MonitoringService and display them on each selected component's graph.

4.3.1. GraphForm

This is the main form to see once the application is run. Using this form, users can select components to monitor. Users can also change the settings for the graphs. This form sends selected components name and time intervals to the *MonitoringService*. Then the MonitoringService calls each selected component and components start sending their values to the MonitoringService. The GraphForm is also responsible to get these values from the MonitoringService. Then the GraphForm creates a graph for each selected component. Based on the # of selected components, the GraphForm dynamically calculates the locations of each graph on the screen (if there is only one selected component, the graph of this component would be bigger, however; if there are five selected components, their graphs would be smaller to fit all graphs on a screen). After creating and calculating the locations of graphs, the GraphForm parses the values received from the MonitoringService and inserts these values to the related components' graphs. It does these in a "t" time period. This period is defined by the user.

4.3.2. SettingsForm

This form is used for changing settings for graphs. A user can change settings about timing, bars' color on the graphs, and the max # of bars that will be displayed on the graphs. Users have an option to save these settings on the HDD. After making the changes, these changes are applied without restarting the application. If the user saves these settings in the HDD, the system will use new settings instead of default one. The user can change the timing settings in the *Service_01.xml* file (see Figure A-1 in the Appendices). Figure A-3 in Appendices shows a sample saved settings file.

4.3.3. SerializableColor

This is a *Struct* class with some *Static* methods. This class is used to load/save the color settings from/in the HDD. The class has some static methods because there is no need to create an object to use those methods such as serialize and de-serialize objects etc. This class allows abstraction so that color transactions are separated from GUI forms.

4.3.4. Graphs

The *Graphs* class is an abstract class. It is used by the *GraphForm* to create graphs, set graphs' locations, and add new points (values) to the graphs. While doing these things, this class is using Microsoft's *Chart* objects. This class is created because it provides abstraction and it isolates graph related transactions from the GraphForm. Methods were created to do one thing instead of doing several things. In this way, high coupling was prevented; thus the maintenance and testing became much easier.

4.3.5. Microsoft Chart Library

This is *Microsoft's Chart* library. This library is used to create and display graphs. Before *Microsoft's Chart* library, *ZedGraph* library was used, but *ZedGraph* library did not allow creating the graphs dynamically. If the graphs were created just once and displayed, then this library could be used perfectly. But in this project, the system needs to update graphs' values every *"t" time* and then redraw the graphs. Unfortunately the ZedGraph did not allow doing it. For this reason, *Microsoft's Chart library* is used. It allows adding new points to the existing graphs and then displays them dynamically without any problem.

CHAPTER 5

IMPLEMENTATION

5.1. Overview

This chapter is about the implementation of CCMS. UML diagrams, pseudocode, and actual code are used to explain the implementation. The communication among services, components, and GUI is explained in details.

5.2. Overview of How CCMS Works

After running the application (CCMS), a user can select components and then start monitoring. GUI stores selected component names and makes RPC to *MonitoringService* (the selected component names are passed with RPC). Then the *MonitoringService* makes RPC to each selected component's *ComponentService* (Figure 5.1).

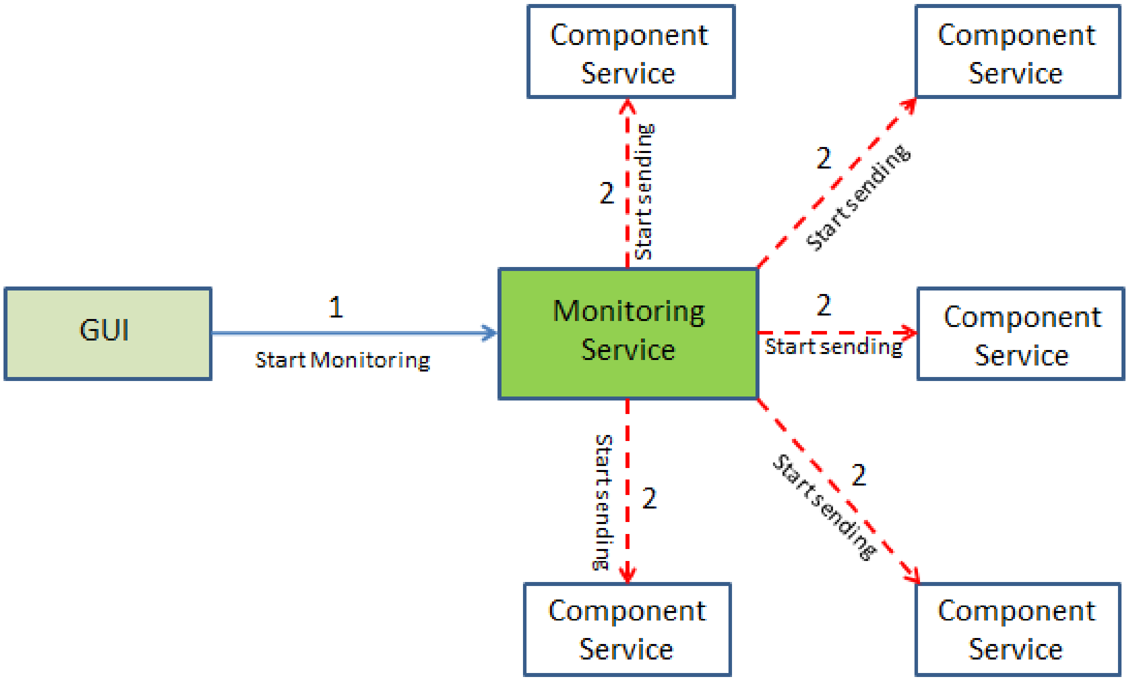


Figure 5.1 - GUI calls the *MonitoringService* and the *MonitoringService* calls each selected component's *ComponentService*.

After being called, *ComponentServices* start sending their values to the *MonitoringService* (they make RPC too) and the *MonitoringService* stores these values. GUI has another thread to make RPC to get these values from the *MonitoringService* and displays them in the graphs at certain time intervals (Figure 5.2).

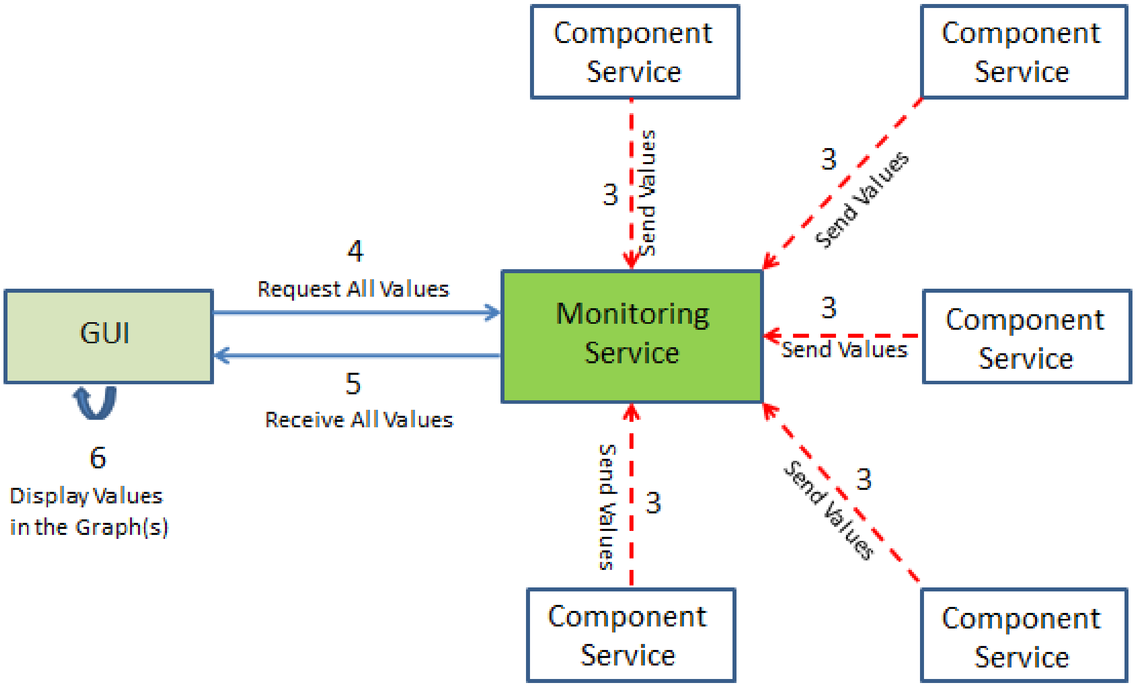


Figure 5.2 - Each *ComponentService* sends its values to the *MonitoringService* (RPC) and GUI (using another thread) makes RPC to get these values and displays them in the graph(s).

5.3. Start Monitoring

Figure 5.3 shows the sequence diagram of the communication of *GUI* and the *MonitoringService*. Since it is not always possible to show everything on sequence diagrams, some pseudocode and actual code are used to give these details.

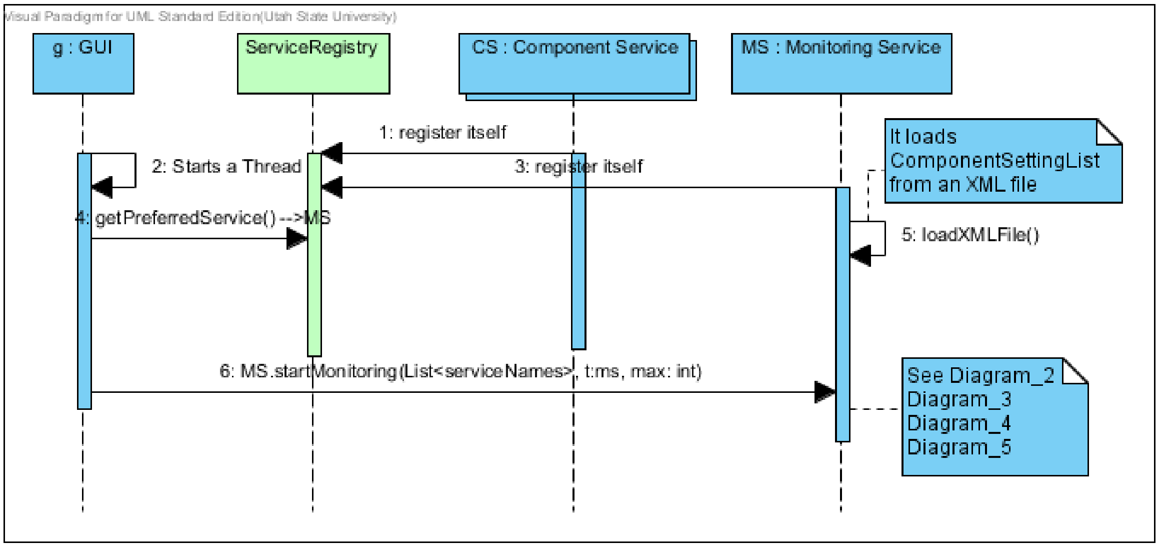

Visual Paradigm for UML Standard Edition(Utah State University)
g : GUI
ServiceRegistry
CS : Component Service
MS : Monitoring Service
1: register itself
2: Starts a Thread
3: register itself
It loads ComponentSettingList from an XML file
4: getPreferredService()-->MS
5: loadXMLFile()
6: MS.startMonitoring(List<serviceNames>, t:ms, max: int)
See Diagram_2 Diagram_3 Diagram_4 Diagram_5

Figure 5.3 - GUI Diagram.

After selecting the components, the user clicks on a button to see the selected components' graphs. Figure 5.4 shows the *pseudocode* of what is happening after clicking on the monitor button.

```
Select desired components to monitor.
Click on the monitor button.
Create a compList (List<string>) and put selected components' names in it.
Get MonitoringService from the registry (GetPreferredService).
If (any selected component is not running)
  Error message
Else
{
  If (colors for settings is not loaded)
            Load colors (from XML or use defaults values)
  Stop worker thread
  RPC call to MonitoringService(compList) (it calls selected components' ComponentServices to send values to the MonitoringService)
  Create graphs
  Set locations of graphs // locations are calculated dynamically
  Start worker thread (it makes RPC call to the MonitoringService and gets the values for graph(s))
}
```

Figure 5.4 - ***Pseudocode*** **of what happens in** ***GraphForm*** **after selecting components.**

CCMS uses Microsoft's chart library to create and display graphs. The library is called *Microsoft Chart Control for .Net Framework*. This library allows creating, displaying, and refreshing graphs dynamically. In *GraphForm*, another thread is used to get the values from the *MonitoringService*. After getting the values, these values are parsed and displayed on the graphs. Figures 5.5, 5.6, and 5.7 show this process.

The worker thread runs at the background and gets *componentStatsList* (stores all values that ComponentServices sent) from the *MonitoringService*. If the result is not null or the result is not empty, then it calls AddData method with passing the *componentStatsList* (Figure 5.5). Then the thread sleeps for a given time period and gets *componentStatsList* again and it goes on like this.

Worker Thread:

```
while (ListenerThreadGoOn)
{
        if (listCharts.Count > 0)
   {
     ComponentStatsList componentStatsList = monitoringService.
                                             getComponentStatsList();
     if (componentStatsList.getStatsList.Count > 0 &&
          componentStatsList.getStatsList[0].KeyNames != null)
     {
        AddData(componentStatsList);
     }
     Thread.Sleep(timeRefreshGraph);
   }
   else
        Thread.Sleep(sleepThreadIdle);
}
```

Figure 5.5 - Worker thread in MonitoringForm.cs.

AddData method parses the *componentStatsList* and if there are any values, adds these values to the related graphs as new points (Figure 5.6). The method called *getValues* returns the values of the given component name (Figure 5.7).

```
private void AddData(ComponentStatsList list)
{
   foreach (Chart ch in listCharts)
   {
     List<List<double>> values = getValues(list, ch.Titles[0].Text);
     if (values != null)
      {
            int size = ch.Series.Count;
            for (int i = 0; i < size; i++)
              Graphs.AddNewPoints(ch.Series[i], ch, values[i],
                                   checkBoxPointLabel.Checked,
                                   maxBarNumber);
        }
    }
}
```

Figure 5.6 - ***AddData*** **method in** ***GraphForm*****.cs.**

```
private List<List<double>> getValues(ComponentStatsList list,
                                           string componentName)
{
    int size = list.getStatsList.Count;
    for (int i = 0; i < size; i++)
        if (list.getStatsList[i].ComponentName.Equals(componentName))
            return list.getStatsList[i].list;
    return null;
}
```

Figure 5.7 - ***getValues*** **method in** ***GraphForm.cs*****.**

List<List<double>> (list of list of double) is used because CCMS needs to store different sets of values for different components. For example;

- *AuditManager* displays one set of values
- *CoreAgent* displays two sets of values
- *QueryManager* displays three sets of values on the graph.

Different components display different sets of values on their graphs. A List is used because the # of selected components is not known in advance.

Second list (*List<double>*) is used to store the values that the components send. The user can change how many bars will be displayed on the graph. That is why the second List is used. Values sent by components are stores as *double* because *MSChart* library requires doubles for graphs.

5.4. MonitoringService

GUI calls MonitoringService's *StartMonitoring* method and passes;

- selected components' names,
- time interval that will be used by components to send their values to the *MonitoringService*,
- Maximum bar number. The maximum bar number will tell the system maximum how many set of numbers will be stores for each graph. In other words, it represents maximum how many bars will be displayed on each graph. Once the maximum number has been reached, the first set of values in the list will be removed and the new set of values will be added.

Figure 5.8 shows the *MonitoringService*'s communications with other components. The GUI makes RPC to the *MonitoringService* and passes three parameters that were mentioned above. *StartMonitoring* triggers the monitoring process.

RPC syncpattern technique is used for some of the methods in the MonitoringService. These methods are;

- *StartMonitoring* – Starts monitoring process
- *StopMonitoring* – Stops monitoring process
- *NewStatsValues* – Components' ComponentServices call this method to pass their new values
- *getComponentStatsList* – Returns the private ComponentStatsList (it holds selected components' set of values received from the ComponentServices)
- *getComponentStats* – Returns ComponentStats based on the component name.

Since these methods have RPC syncpattern, they are declared as *virtual* (Vitruvian requires it). These methods are defined as RPC because *MonitoringService* is a distributed service.

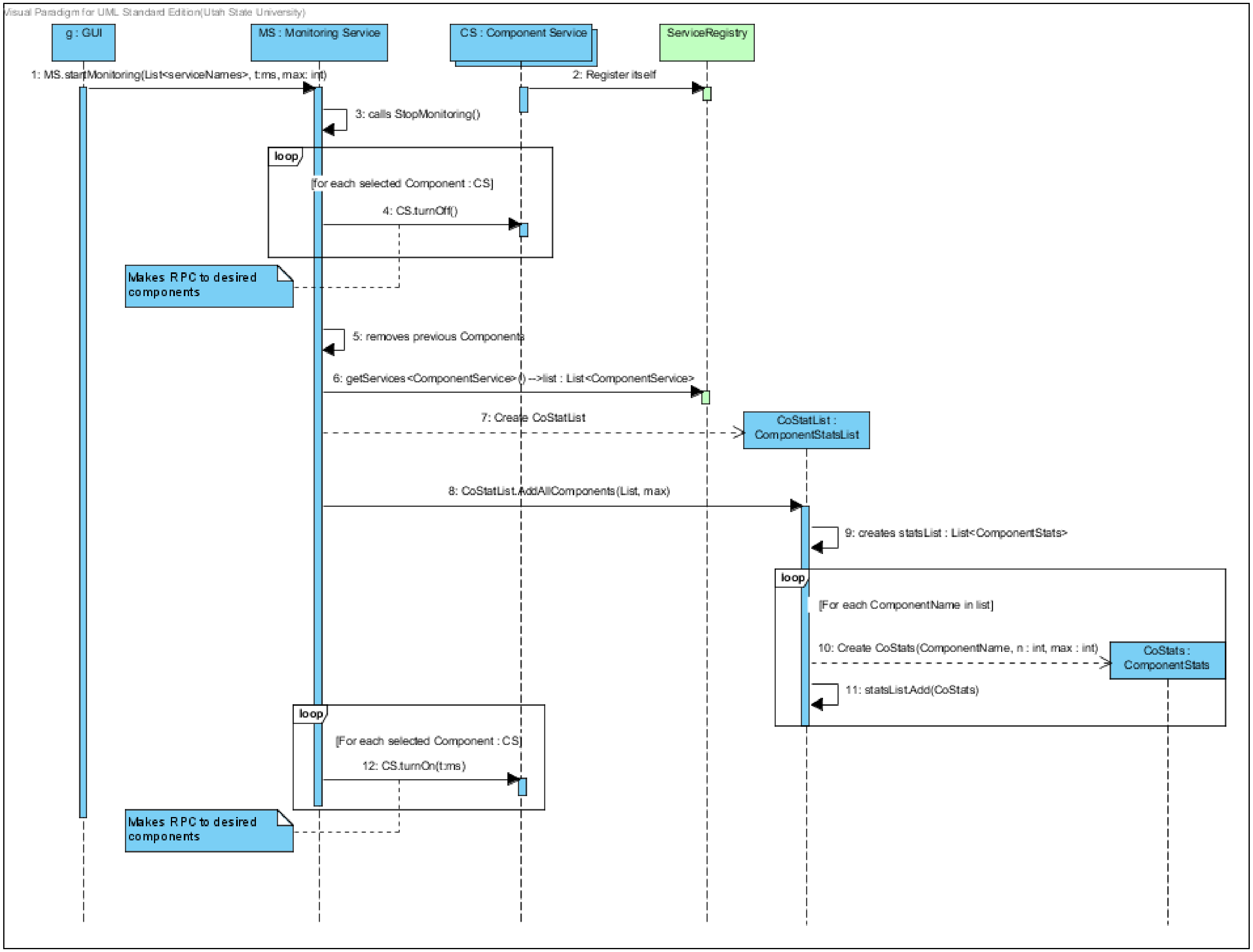

Visual Paradigm for UML Standard Edition(Utah State University)
g : GUI
MS : Monitoring Service
CS : Component Service
ServiceRegistry
1: MS.startMonitoring(List<serviceNames>, t:ms, max: int)
2: Register itself
3: calls StopMonitoring()
loop
[for each selected Component : CS]
4: CS.turnOff()
Makes RPC to desired components
5: removes previous Components
6: getServices<ComponentService>() -->list : List<ComponentService>
7: Create CoStatList
CoStatList : ComponentStatsList
8: CoStatList.AddAllComponents(List, max)
9: creates statsList : List<ComponentStats>
loop
[For each ComponentName in list]
10: Create CoStats(ComponentName, n : int, max : int)
CoStats : ComponentStats
11: statsList.Add(CoStats)
loop
[For each selected Component : CS]
12: CS.turnOn(t:ms)
Makes RPC to desired components

Figure 5.8 – MonitoringService (Diagram_02)

5.5. ComponentService

The *ComponentService* is a distributed service like *MonitoringService*. This service is using Vitruvian for distribution. Chapter 6 is about the Vitruvian framework and how this framework was used in this project.

Since the ComponentService is a distributed service, it uses syncpattern too. This class has three methods that have syncpattern *RPC*. These are;

- *turnOn* – The MonitoringService calls it so that the ComponentService starts sending values
- *turnOff* – The MonitoringService calls it so that the ComponentService stops sending values
- *StopThread* – It kills the sender thread (The component calls it before it dies)

Figure 5.9 shows the Pseudocode of what happens once the MonitoringService calls ComponentService's *turnOn* method.

```
startSenderThread (if it is not running) // This thread sends the values to the
                                            MonitoringService
create a List to put values in it
set the new time interval to send values to the MonitoringService
start sending values to the MonitoringService
```

Figure 5.9 - *Pseudocode* of what happens in *ComponentService* after its *turnOn* method has been called.

Since real CHARM components could not implement the Vitruvian framework yet, dummy components are used to test the CCMS and these dummy components are sending random numbers to the MonitoringService for graphs. Figure 5.10 shows the ComponentService's diagram.

ComponentService is sending new values using a *Dictionary* (Dictionary< string, double>). A dictionary is used because it provides a convenient mapping between a key name and a set of numbers. A list is not used because in this way, the order of the values would be very crucial for CCMS and any changes in the order would cause displaying the wrong information on the graphs. This problem is solved by using a dictionary. Even if a ComponentService changed the order, it would never create any problems because whenever the first set of values is received by the MonitoringService, the key names are set first. For this reason, values' orders do not matter.

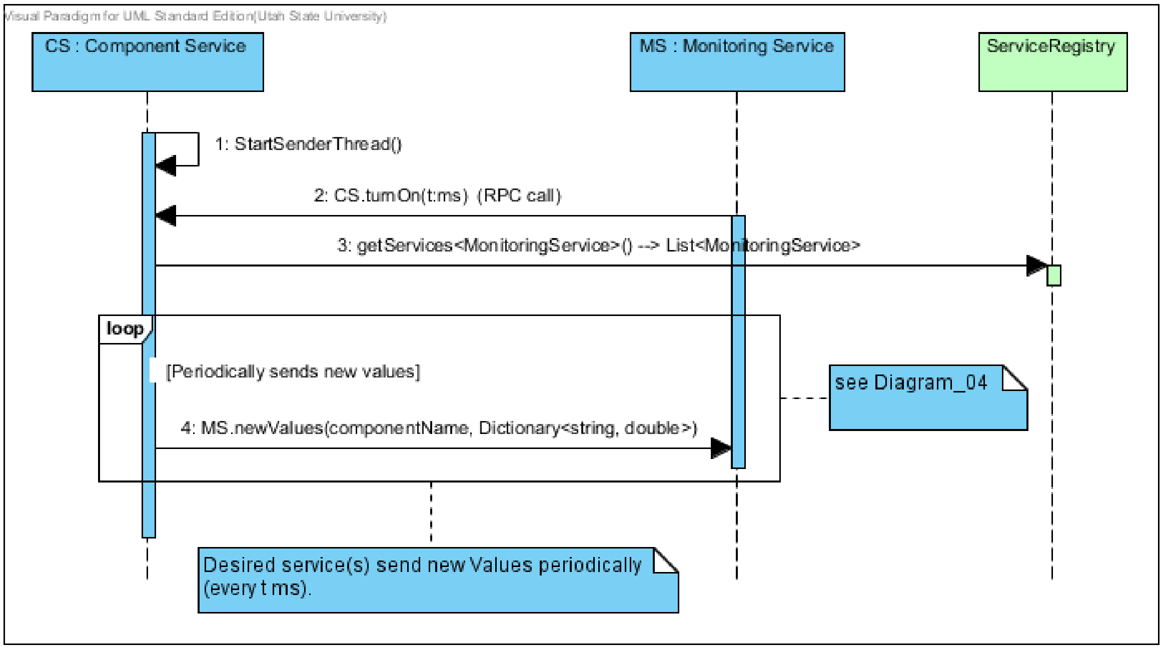

Visual Paradigm for UML Standard Edition(Utah State University)
CS : Component Service
MS : Monitoring Service
ServiceRegistry
1: StartSenderThread()
2: CS.turnOn(t:ms) (RPC call)
3: getServices<MonitoringService>() --> List<MonitoringService>
loop
[Periodically sends new values]
4: MS.newValues(componentName, Dictionary<string, double>)
see Diagram_04
Desired service(s) send new Values periodically (every t ms).

Figure 5.10 – ComponentService (Diagram_03)

5.6. ComponentStats

ComponentStats is used to store the values the *ComponentService* send. This class is used by *ComponentStatsList*. Basically, the *ComponentStatsList* class has a list of *ComponentStats*. *MonitoringService* communicates with *ComponentStatsList* and *ComponentStatsList* communicates with *ComponentStats* for each component.

ComponentStats holds the values *ComponentServices* send to the *MonitoringService*. *ComponentStats* class has three important methods;

- *calculateNewValues* – It calls one of the methods below.
- *AddNewValuesFirstTime* – It adds first set of values the ComponentService send. (Figure 5.11)
- *AddNewValuesNotFirstTime* – It adds second and later sets of values the ComponentService send (Figure 5.12).

For adding new values, two methods are used because keeping all logic in one method would make that one method very bulky and testing and maintenance of the bulky methods are always more difficult. Bulky methods also increase coupling. First method is adding the new values to the list directly. But the second method needs to do some math before adding them in the list. Since components send their total numbers, second method needs to calculate the *delta values*. Calculation algorithm is explained in the following section.

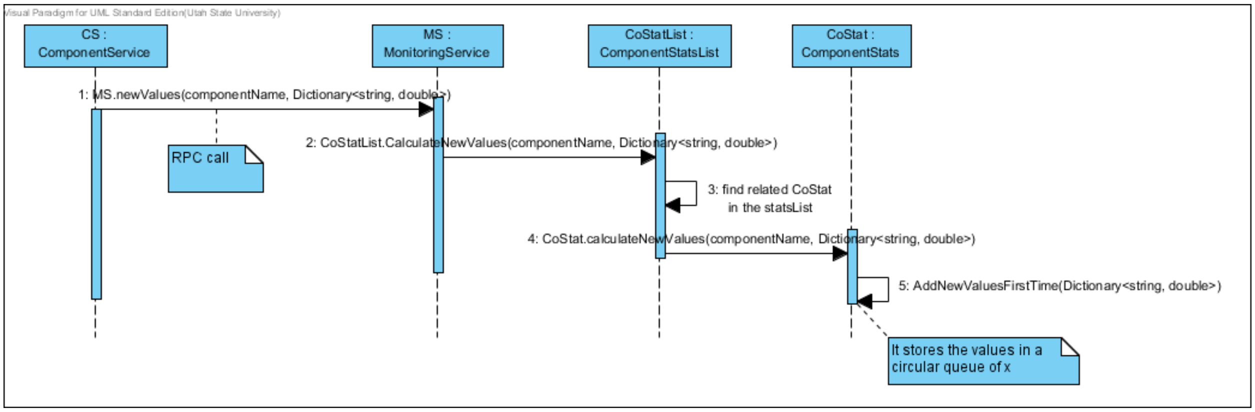

Visual Paradigm for UML Standard Edition(Utah State University)
CS : ComponentService
MS : MonitoringService
CoStatList : ComponentStatsList
CoStat : ComponentStats
1: MS.newValues(componentName, Dictionary<string, double>)
RPC call
2: CoStatList.CalculateNewValues(componentName, Dictionary<string, double>)
3: find related CoStat in the statsList
4: CoStat.calculateNewValues(componentName, Dictionary<string, double>)
5: AddNewValuesFirstTime(Dictionary<string, double>)
It stores the values in a circular queue of x

Figure 5.11 – Add New Values First Time (Diagram_04)

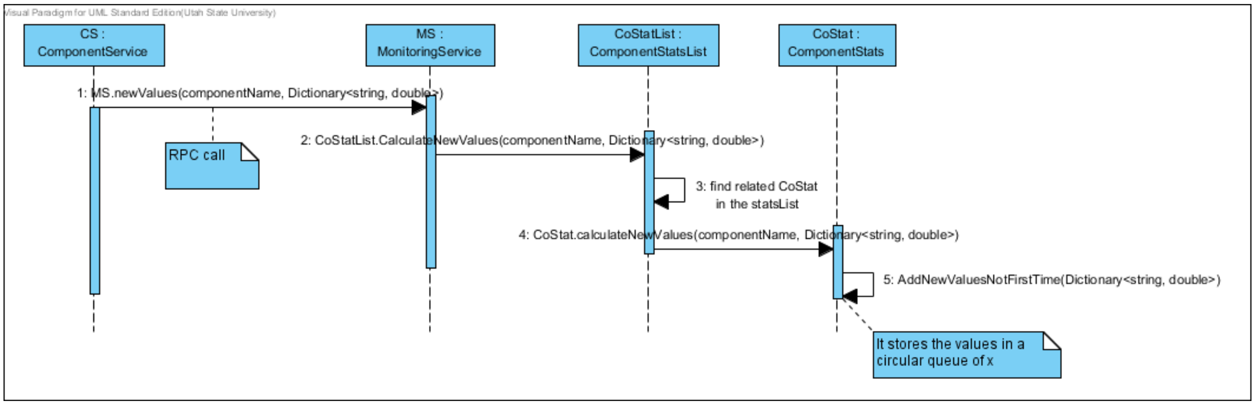

Visual Paradigm for UML Standard Edition(Utah State University)
CS : ComponentService
MS : MonitoringService
CoStatList : ComponentStatsList
CoStat : ComponentStats
1: MS.newValues(componentName, Dictionary<string, double>)
RPC call
2: CoStatList.CalculateNewValues(componentName, Dictionary<string, double>)
3: find related CoStat in the statsList
4: CoStat.calculateNewValues(componentName, Dictionary<string, double>)
5: AddNewValuesNotFirstTime(Dictionary<string, double>)
It stores the values in a circular queue of x

Figure 5.12 – Add New Values Not First time (Diagram_05)

Calculation Algorithm: Whenever a first set of values is received from a ComponentService, this set of values is stored in the list and the attribute previousValues is set to these values. After the first set of values, whenever a new set of values are received, the differences between the new values and the previous values are stored in the list for the graphs. Then the attribute previousValues are set to the new set of values.

5.7. StopMonitoring

GUI makes RPC to the MonitoringService to StopMonitoring. Then the MonitoringService makes RPC to each ComponentService to stop sending values (Figure 5.13). After that point, ComponentServices stop sending values.

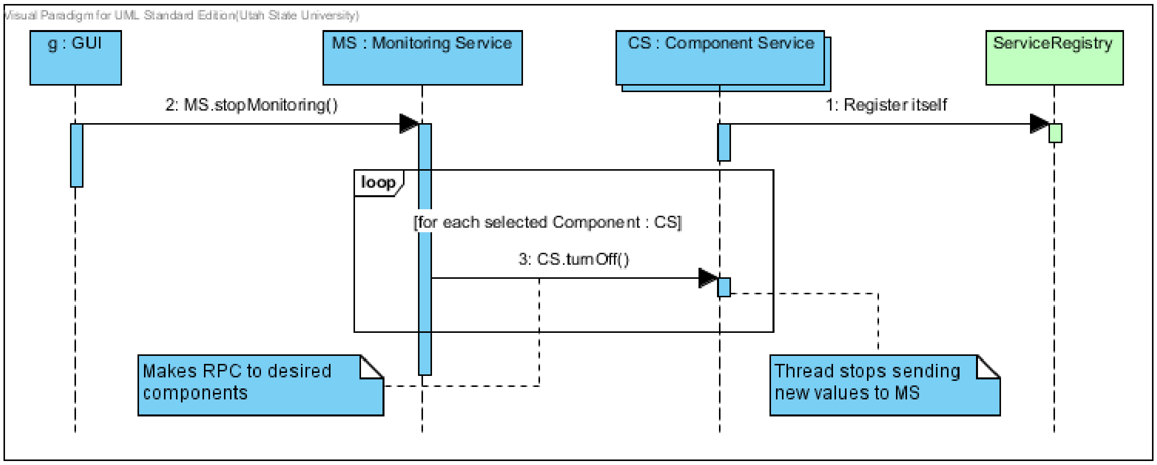

isual Paradigm for UML Standard Edition(Utah State University)
g : GUI
MS : Monitoring Service
CS : Component Service
ServiceRegistry
2: MS.stopMonitoring()
1: Register itself
loop
[for each selected Component : CS]
3: CS.turnOff()
Makes RPC to desired components
Thread stops sending new values to MS

Figure 5.13 - Stop Monitoring Diagram_6

5.8. Display Graphs

Second thread in GraphForm (GUI) wakes up, every *t* time, and makes RPC to the MonitoringService to receive the values for each component. After that the GraphForm parses these values and adds new values to the related graphs' list. Then the graphs are refreshed and new values are displayed on the graphs (Figure 5.14).

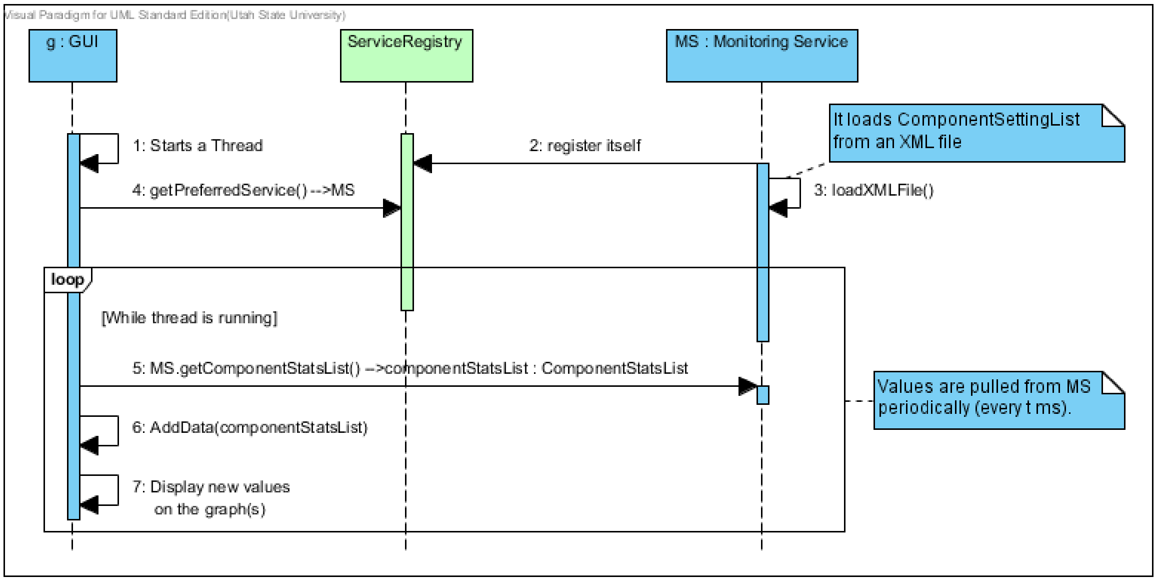

isual Paradigm for UML Standard Edition(Utah State University)
g : GUI
ServiceRegistry
MS : Monitoring Service
It loads ComponentSettingList from an XML file
1: Starts a Thread
2: register itself
4: getPreferredService() -->MS
3: loadXMLFile()
loop
[While thread is running]
5: MS.getComponentStatsList() -->componentStatsList : ComponentStatsList
Values are pulled from MS periodically (every t ms).
6: AddData(componentStatsList)
7: Display new values on the graph(s)

Figure 5.14 - Display Graph(s)

CHAPTER 6

VITRUVIAN

6.1. Overview

CCMS uses the Vitruvian framework to distribute objects and handle all of the underlying communications, which includes taking care of the socket connections and ensuring their aliveness. The Vitruvian framework also has logging feature that helps for debugging. The Vitruvian framework also handles serializing and de-serializing objects. The parameters for serializing and de-serializing objects can be set in an XML file which gives us great flexibility and prevents hardcoding parameters in the code so that the parameters can be changed without recompiling the overall code.

How the Vitruvian framework handles serializing/de-serializing objects and socket connections are explained in this chapter.

6.2. References to Be Added

The CCMS uses the Vitruvian framework to distribute services (MonitoringService, ComponentService). For this reason, first thing we need to do is to add references. These references **must be added** in order to work with Vitruvian. The list of references is;

- Castle.Core
- Vitruvian.Communications
- Vitruvian.Core
- Vitruvian.Distribution
- Vitruvian.Logging
- Vitruvian.Serialization

- Vitruvian.Services
- Vitruvian.Windows

Usage of the Vitruvian framework will be explained in five separate sections. These are;

- GraphFrom
- MonitoringService
- ComponentService
- ComponentSettings
- ComponentForm

6.3. GraphForm

6.3.1. Program.cs

After adding the references, changes in the “Program.cs” must be done as described below.

```
[STAThread]
static void Main(string[] args)
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);

    string configFile = @"Config/Service_01.xml";
    if (args.Length > 0)
        configFile = args[0];

    ServiceRegistry.Load(configFile);
    ServiceRegistry.Init();
    ServiceRegistry.Run();
    ServiceRegistry.Cleanup();
}
```

Figure 6.1 - The *Main* method of *Program.cs*.

Vitruvian requires four methods and these methods have to be in the *Program.cs* file. These are;

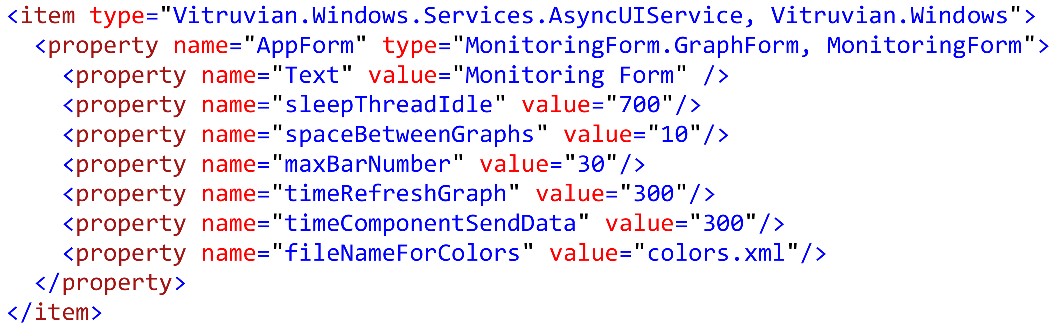
- Load
- Init

- Run
- Cleanup

If any of these methods does not exist in the *Program.cs*, CCMS will get a compile error. For this reason, it is essential to have these methods even though some of them have no code in it (such as Init, Run, and Cleanup in our example).

Once the program is run, it reads and loads information in *Service_01.xml*. The file has the necessary information about *GraphForm*, *MonitoringService*, *SyncPatterns*, and Distri*buted Services*. Figure A-1 in Appendices contains the complete content of *Service_01.xml*.

6.3.1.1. Graph Form Information



```
<item type="Vitruvian.Windows.Services.AsyncUIService, Vitruvian.Windows">
  <property name="AppForm" type="MonitoringForm.GraphForm, MonitoringForm">
    <property name="Text" value="Monitoring Form" />
    <property name="sleepThreadIdle" value="700"/>
    <property name="spaceBetweenGraphs" value="10"/>
    <property name="maxBarNumber" value="30"/>
    <property name="timeRefreshGraph" value="300"/>
    <property name="timeComponentSendData" value="300"/>
    <property name="fileNameForColors" value="colors.xml"/>
  </property>
</item>
```

Figure 6.2 - Graph Form information in the XML file.

Figure 6.2 shows the GUI form information in the xml file. Some values are set in this file.

- *Text*: The value is displayed as the title of the Graph Form.
- *sleepThreadIdle*: This is a sleeping time for the thread in the GUI. The thread is checking if a user has selected any components to display the values on the graph. If the user has selected a component, thread gets that component's

values from the *MonitoringService* and displays the values on the graph. It the user has not selected any components, and then the thread sleeps. The sleep time period is defined in the attribute *sleepThreadIdle*.

- *spaceBetweenGraphs*: This value tells how much space will be given between graphs.
- *maxBarNumber*: This attribute is used to define maximum how many bars will be displayed on each graph.
- *timeRefreshGraph*: This attribute is used to define the refresh time of GUI for graphs. Each selected component's values are received from *MonitoringService* and these values are set for the graphs. Then the screen is refreshed every *timeRefreshGraph* time period (millisecond).
- *timeComponentSendData*: Selected components are sending their values to the *MonitoringService*. Time interval that components send their values to the *MonitoringService* is defined in *timeComponentSendData*.
- *fileNameForColor*: Users can save their settings in a file. This is the name of the file to save settings. This is an XML file and if it exists, this file is loaded during run time. Otherwise default values are loaded.

6.3.1.2. Monitoring Service Information

```
  <!-- MonitoringService Service -->
  <item type="Monitoring.MonitoringService, Monitoring" id="localService" >
<property name="DataFile" value="componentSettingsList.xml"/>
    <property name="ServiceName" value="MonitoringService"/>
    <property name="Port" value="19301"/>
  </item>
```

Figure 6.3 - MonitoringService information in the XML file.

Figure 6.3 contains the *MonitoringService's* information.

- *Id*: "localService" is being used in distribution service. This can be any word. But this value and the value in the distribution service must be the same (it will be explained in section 6.2.1.4.).
- *DataFile*: This is an XML file that contains the information about components (ComponentName and NumberOfValues). *NumberOfValues* defines how many different values will be displayed on each bar on the graph. For example, *queryManager* sends 3 numbers each time to the *MonitoringService* while *auditManager* sends 1 and *coreAgent* sends 2. Figure A-2 in Appendices shows the content of this file.

  If new components are added to the CHARM, those components' information must be added in this file.
- *ServiceName*: Name of the service.
- *Port*: Port number.

6.3.1.3. SyscPatterns Information

```
  <!-- Sync Patterns Service -->
  <item
type="Vitruvian.Distribution.SyncPatterns.EmbeddedSyncPatternsService,
Vitruvian.Distribution">
    <property name="BaseType" value="MonitoringForm.Program,
MonitoringForm"/>
    <property name="ResourcePath" value="Config.SyncPatterns.xml"/>
  </item>
```

Figure 6.4 - Sync Pattern information in the XML file.

Figure 6.4 contains the Sync Pattern's information.

- *ResourcePath*: The value is a file name that contains Sync Pattern information. These patterns are used when the objects are distributed. Figure A-4 in Appendices contains the complete content of *SyncPatterns.xml*.

6.3.1.4. Distribution Service's Information

```
  <item type="Vitruvian.Distribution.DistributionService,
Vitruvian.Distribution">
    <property name="Encoder"
type="Vitruvian.Distribution.Encoders.XmlObjectEncoder,
Vitruvian.Distribution" />
    <property name="MessageTimeout" value="15000"/>
    <property name="HeartbeatFrequency" value="10000"/>
    <property name="HeartbeatTimeout" value="120000"/>
    <property name="UseHeartbeats" value="true"/>
    <property name="Services">
      <item ref="localService"/>
    </property>
    <property name="Providers">
      <item type="Vitruvian.Communications.Sockets.TcpServerProvider,
Vitruvian.Communications">
        <property name="LocalEndPoint" value="127.0.0.1:19301"/>
      </item>
    </property>
  </item>
```

Figure 6.5 - Distribution Service's information in the XML file.

Figure 6.5 contains the distribution service's information. In section 6.2.1.2, we talked about *id* for MonitoringService. The value of *id* in the distribution service is used as *ref* here and the values must be the same. Otherwise, distribution will not work. If other services are desired to be distributed, these services' ids need to be added under *Services* just like *localService*.

There is no need to make any changes on;

- MessageTimeout
- HeartbeatFrequency
- HeartbeatTimeout

- UseHeartbeats

These values are used when a connection is made with another computer to check the aliveness of the connection. There are three important points need to be remembered;

- Make sure to add all services that need to be distributed
- Add *IP* and *Port* numbers. In this setting, our value is *127.0.0.1:19301*
- Decide whether you use this service as a *LocalEndPoint* or a *RemoteEndPoint*.

Here the MonitoringService is used as a *LocalEndPoint*. In other words, this listens to the communications using the given IP and Port numbers.

The syntax in the xml file is very important. If the correct syntax is not entered, the system will not give an error and the application will not work correctly either.

6.3.2. GraphForm.cs

```
[OptimisticSerialization]
public partial class GraphForm : Form
{
    public int sleepThreadIdle { get; set; }
    public int maxBarNumber { get; set; }
    public int timeRefreshGraph { get; set; }
    public int timeComponentSendData { get; set; }
    public string fileNameForColors { get; set; }
      ...
```

Figure 6.6 - GraphForm.cs.

Since *Service_01.xml* has values for *GraphForm.cs*, *[OptimisticSerialization*] must be added just before the class declaration for the form. This is extremely important because if it is not added, the application will not set the values entered in the *Service_01.xml*. As a result of this, the application will not work.

The Vitruvian references that we mentioned at the beginning of this chapter must be added.

Another important part is that the attributes that are set in the *Service_01.xml* must be declared as *public*. Otherwise, the xml file cannot set the values for the attributes and the application will not work.

If an attribute's value is set in the xml file, that attribute;

- ***must be declared as public in the class***
- ***must have a setter and a getter methods***

6.3.3. app.config

The file *app.config* is used for logging purpose (Figure 6.7). While changing some configurations in this file, it can be ruled what to log and what to ignore.

```
<?xml version="1.0"?>
<configuration>
  <configSections>
    <section name="log4net"
type="log4net.Config.Log4NetConfigurationSectionHandler, log4net"/>
  </configSections>

  <log4net>
    <appender name="Console" type="log4net.Appender.ConsoleAppender">
      <layout type="log4net.Layout.PatternLayout">
        <conversionPattern value="%logger %message%newline"/>
      </layout>
    </appender>

    <appender name="File" type="Vitruvian.Logging.FileAppender">
      <file value="./Logs/%appname [(%date) (%ticks)].txt"/>
      <appendToFile value="false"/>
      <layout type="log4net.Layout.PatternLayout">
        <header value=""/>
        <footer value=""/>
        <conversionPattern value="%date [%thread] %-5level %logger -
%message%newline"/>
      </layout>
    </appender>
    <root>
      <level value="INFO"/>
      <appender-ref ref="File"/>
      <appender-ref ref="Console"/>
    </root>

    <logger name="Vitruvian">
      <level value="INFO"/>
    </logger>

  </log4net>

  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/>
  </startup>
</configuration>
```

Figure 6.7 - The content of *app.config*.

6.4. MonitoringService

```
[OptimisticSerialization]
[DistributionInfo(Migratable = false)]
public class MonitoringService : IService
{
    public string ServiceName { get; set; }
    public int Port { get; set; }
    public string DataFile { get; set; }
         ...
```

Figure 6.8 - *MonitoringService.cs* class declaration.

Figure 6.8 shows the some of the MonitoringService.cs class declaration. This class has;

- [OptimisticSerialization] just before the class declaration
- attributes declared as *public* because these attributes are set in the xml file
- [DistributionInfo(Migratable = false)] (before the class declaration). This syntax is used to distribute the objects. This syntax is also crucial. If this line is not added before the class declaration, the distribution will not happen.

- The last thing that needs to be done is that the class inherits *IService*. *IService* is also necessary to distribute the objects.

MonitoringService also has *IService* members. The class must have these members as well. Without *IService* members, objects cannot be distributed. Figure 6.9 shows the *IService* members and their usage in the MonitoringService.cs.

```
public void Cleanup()
{ }

private Guid serviceId = Guid.NewGuid();
public Guid Id
{
    get { return serviceId; }
    set { serviceId = value; }
}

public void Init()
{
    loadXMLFile();
}

private string serviceName = "MonitoringService";
public string Name
{
    get { return serviceName; }
    set { serviceName = value; }
}
```

Figure 6.9 - *IService members* in the *MonitoringService.cs*.

The *Cleanup* method runs before the object dies.

The *Id* method is used as a getter and setter. This method sets or returns the serviceId (which is GUID) of the object.

The *Init* method runs before the object of this class is created.

The *Name* method is used as a getter and setter. This method sets or returns the name of the object.

```
[SyncPattern("RPC")]
```

```
        public virtual void NewStatsValues(string component,
Dictionary<string, double> values)

        [SyncPattern("RPC")]
        public virtual void StartMonitoring(List<string> list, int t, int MAX)

        [SyncPattern("RPC")]
        public virtual void StopMonitoring()

        [SyncPattern("RPC")]
        public virtual ComponentStatsList getComponentStatsList()

        [SyncPattern("RPC")]
        public virtual ComponentStats getComponentStats(string componentName)
```

Figure 6.10 - ***SyncPatterns*** **in the** ***MonitoringService.cs*****.**

After distributing the objects, in order to call the distributed object's methods;

- Methods of the distributed object that will be called from other objects must be declared as virtual
- Methods must have SyncPattern such as [SyncPattern("RPC")]. In MonitoringService, RPC is used as SyncPattern. Different SyncPatterns can be used for different purposes. In this design, RPC SyncPattern is used since data are passing/getting to/from these methods.

The class that will be distributed must;

- *have [DistributionInfo(Migratable = false)] before the class declaration*
- *inherit from IService. In other words, IService must be the parent class*
- *have IService member methods (Cleanup, serviceId, Init, and Name)*
- *have [SyncPattern("RPC")] just before the method declaration for RPC*
- *declare RPC methods as virtual*

6.5. ComponentService

First of all, *ComponentService* needs to have the references mentioned at the beginning of this chapter.

```
[OptimisticSerialization]
[DistributionInfo(Migratable = false)]
public class ComponentService : IService
{
    public string ComponentName { get; set; }
    public int Port { get; set; }
    public int NumberOfValues { get; set; }
    public int SleepThreadIdle { get; set; }
```

Figure 6.11 - ComponentService.cs.

Since *ComponentService* is a distributed service, the Vitruvian structure is the same as *MonitoringService*. Like *MonitoringService*, *ComponentService*;

- has [OptimisticSerialization] and [DistributionInfo(Migratable = false)] just before the class declaration
- has attributes declared as *public* because their values are set in the xml
- inherits from *IService* and because of this it has *IService methods* (Figure 6.12)

```
public void Cleanup()
{ }

private Guid serviceId = Guid.NewGuid();
public Guid Id
{
    get { return serviceId; }
    set { serviceId = value; }
}

public void Init()
{ }

private string serviceName = "ComponentService";
public string Name
{
    get { return serviceName; }
    set { serviceName = value; }
}
```

Figure 6.12 - IService members in the ComponentService.cs.

The *Cleanup* method runs before the object dies.

The *Id* method is used as a getter and setter. This method sets or returns the serviceId (which is GUID) of the object.

The *Init* method runs before the object of this class is created.

The *Name* method is used as a getter and setter. This method sets or returns the name of the object.

```
[SyncPattern("RPC")]
public virtual void turnOn(int t)

[SyncPattern("RPC")]
public virtual void turnOff()

[SyncPattern("RPC")]
public virtual void StopThread()
```

Figure 6.13 - SyncPatterns in the ComponentService.cs.

As it is explained in section 6.2, after distributing the objects, in order to call the distributed object's methods;

- Methods of the distributed object that will be called from other object must be declared as virtual
- Methods must have SyncPattern such as [SyncPattern("RPC")]. In MonitoringService, RPC is used as SyncPattern. Different SyncPatterns can be used for different purposes. In this design, RPC SyncPattern is used since data are passing/getting to/from these methods.

6.6. ComponentSettings.cs

```
[OptimisticSerialization]
public class ComponentSettings
{
    public string ComponentName { get; set; }
    public int NumberOfValues { get; set; }
```

```
...
```

Figure 6.14 - ComponentSettings.cs.

As mentioned in section 6.2.1.2, *componentSettingsList.xml* contains necessary information about components. Because values of the *ComponentName* and the *NumberOfValues* are loaded from an xml file;

- these attributes must be declared as *public*
- [OptimisticSerialization] must be added just before the class declaration.

componentSettingsList.xml must be updated whenever a new component has been added to the CHARM.

6.7. ComponentForm

The references, mentioned at the beginning of this chapter, must be added for this form as well. Without having these references, the application will not work properly.

6.7.1. Program.cs

```
[STAThread]
static void Main(string[] args)
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);

    string configFile = @"Config/Service_02.xml";
    if (args.Length > 0)
        configFile = args[0];

    ServiceRegistry.Load(configFile);
    ServiceRegistry.Init();
    ServiceRegistry.Run();
    ServiceRegistry.Cleanup();
}
```

Figure 6.15 - The Main method of *Program.cs*.

This section is the same as section 6.1.1. The only difference *Service_02.xml* is used this time. But the content of this file is very similar to *Service_01.xml*. Figure A-5 in Appendices has the complete content of *Service_02.xml*.

```
namespace ComponentForm
{
    [OptimisticSerialization]
    public partial class MainForm : Form
    {
```

Figure 6.16 - The ComponentForm.cs.

Figure 6.16 shows that [OptimisticSerialization] is placed just before the class declaration. This syntax is used to set the values of attributes from the xml file.

CHAPTER 7

SOFTWARE TESTING

7.1. Introduction

Software testing is a key component of the software development life-cycle. Software testing is a very broad term including a wide spectrum of different activities, from the testing of a small piece of code (unit testing), to the customer validation of a large information system (acceptance testing), to the monitoring at run-time of a network-centric service-oriented application [1].

Software testing is a process of improving quality and validating and verifying of a software product that;

- meets the requirements,
- works as expected,
- is implemented with the same characteristics [6].

Verification: Have we built the software right? (*Does it match the specification?*)

Validation: Have we built the right software? (*Is this what the customer wants?*)

“The test cases should aim at objectives, such as exposing deviations from user’s requirements, assessing the conformance to a standard specification, evaluating robustness to stressful load conditions to malicious inputs, measuring given attributes, such as performance or usability, estimating the operational reliability, etc.” [3].

In order to test the performance and functioning of the *CCMS*, unit testing, integration testing, system testing, and acceptance testing are performed.

7.2. Unit Testing

The main purpose of unit testing is to take the smallest piece of testable software in the application, isolate it from the rest of the code, and determine if it behaves exactly as we expect. Each unit is tested separately before we integrate them each other. Unit testing is essential because it is proven that a large percentage of defects are identified during unit test's usage. If unit testing is done properly, later testing phases will be more successful [10].

In CCMS, each method in each class has been tested and test cases can be found in each class file. During unit testing, our goal was to test each method to see if it produced expected results. Valid inputs as well as invalid ones were used to make sure that methods were functioning properly. Algorithms in the methods were also tested.

7.3. Integration Testing

During integration testing individual software modules are combined and tested as a group. It comes after unit testing and before system testing. In integration testing, two or more units that have already been tested are combined into a component and the interfaces among these units are tested [10]. Integration testing identifies problems that occur when units are combined. The idea is to test combinations of units and eventually expand the process to test the modules with those of other groups. Eventually all the modules making up a process are tested together [9]. During integration testing, different units were tested together to see if they function and communicate properly.

7.4. System Testing

This technique is used to test a system when it is integrated with other systems. The purpose of integration testing is to detect any inconsistencies among the software units that are integrated together. During system testing both functional requirements and non-functional quality attributes, such as reliability, security, and maintainability are also tested [1, 6].

System testing of the *CCMS* was done by testing the all components and services. As a result of the system testing, we were able to test the whole system and verify that the whole system functions and communicates properly and produces the desired and expected results. We were not able to test the Monitoring System with real CHARM components because they are not ready for testing yet. For this reason, we used dummy components for testing.

7.5. Acceptance Testing

Acceptance test is a test to determine if the requirements are met. The acceptance testing does not focus on finding errors or system problems rather it gives confidence to the clients that the system is working as expected and how the system will perform in production. “The purpose of this process is to ensure that the software system meets the previously defined system external specifications, acceptance criteria and system requirements definition before it is installed, integrated and checked out in the operational environment.”[11].

In this testing, the user tested the *CCMS* to see if it satisfied the both functional and non-functional requirements.

CHAPTER 8

USER GUIDE

8.1. The usage of CCMS

First of all, components and *MonitoringService* need to be running. Otherwise, the *CCMS* will not work. For the convenience, whenever we run the CCMS, MonitoringService also runs.

After running the application (CCMS), a user can select components and then start monitoring. After receiving the values from the *MonitoringService*, the GUI parses these values and put related values into the related graphs and displays these graphs (Figures from 8.1 to 8.10).

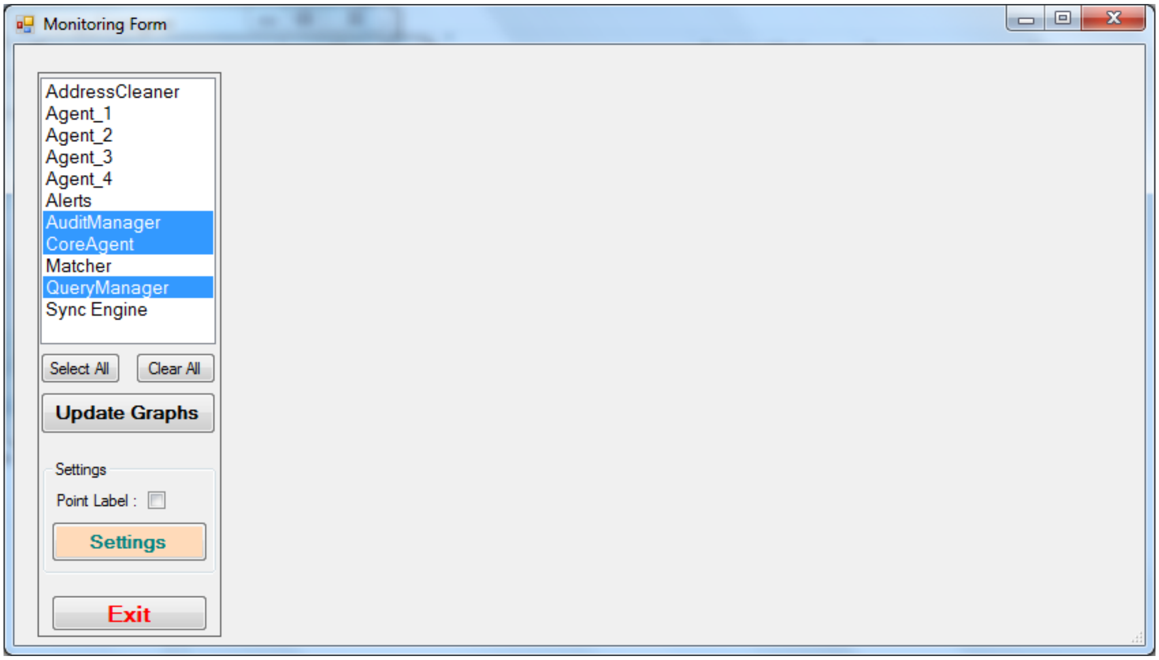


Figure 8.1 - The user selects the components to monitor.

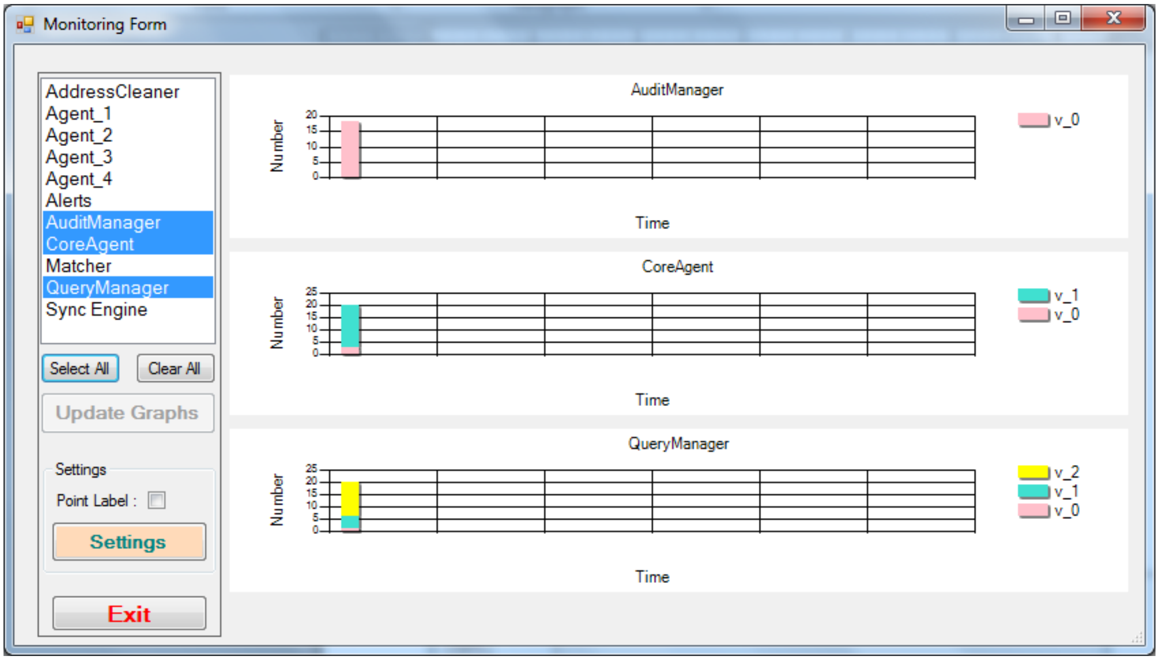


Figure 8.2 - Graphs of selected components' first set of values received from the *MonitoringService*.

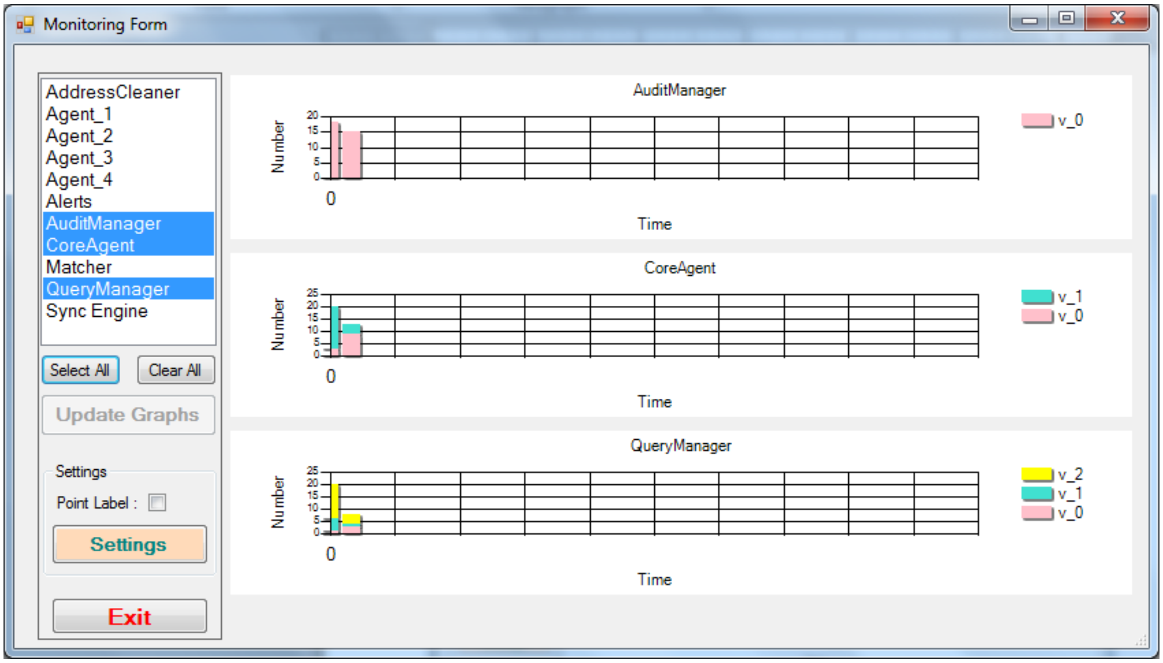


Figure 8.3 - Graphs of selected components' first two sets of values received from the *MonitoringService*.

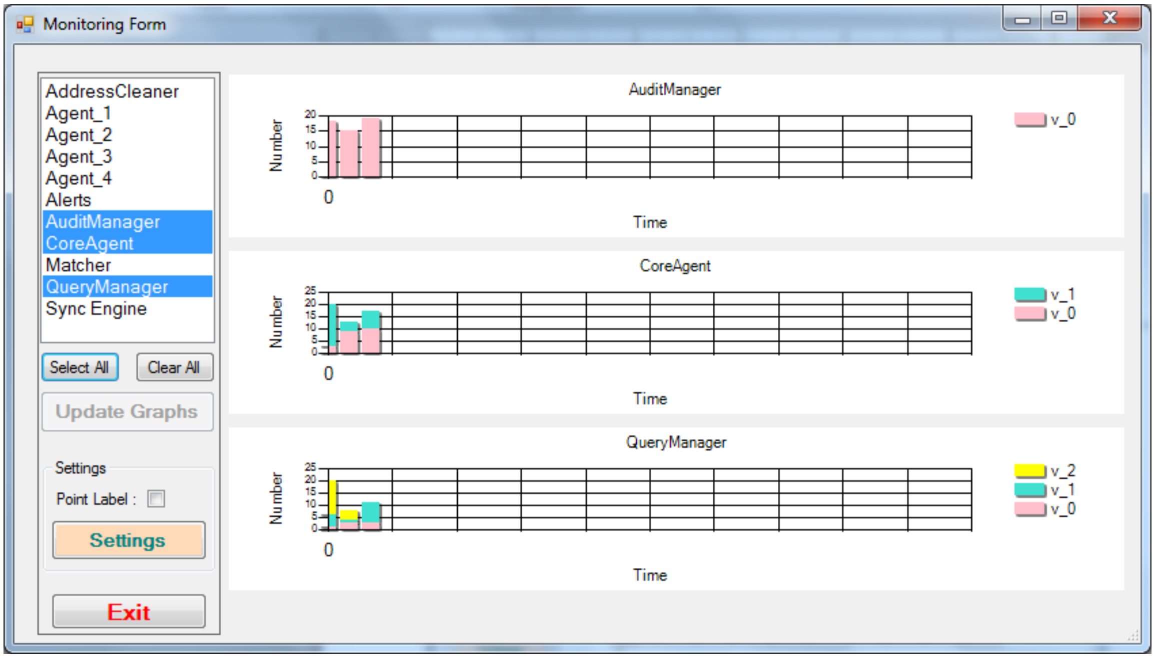


Figure 8.4 - Graphs of selected components’ first three sets of values received from the *MonitoringService*.

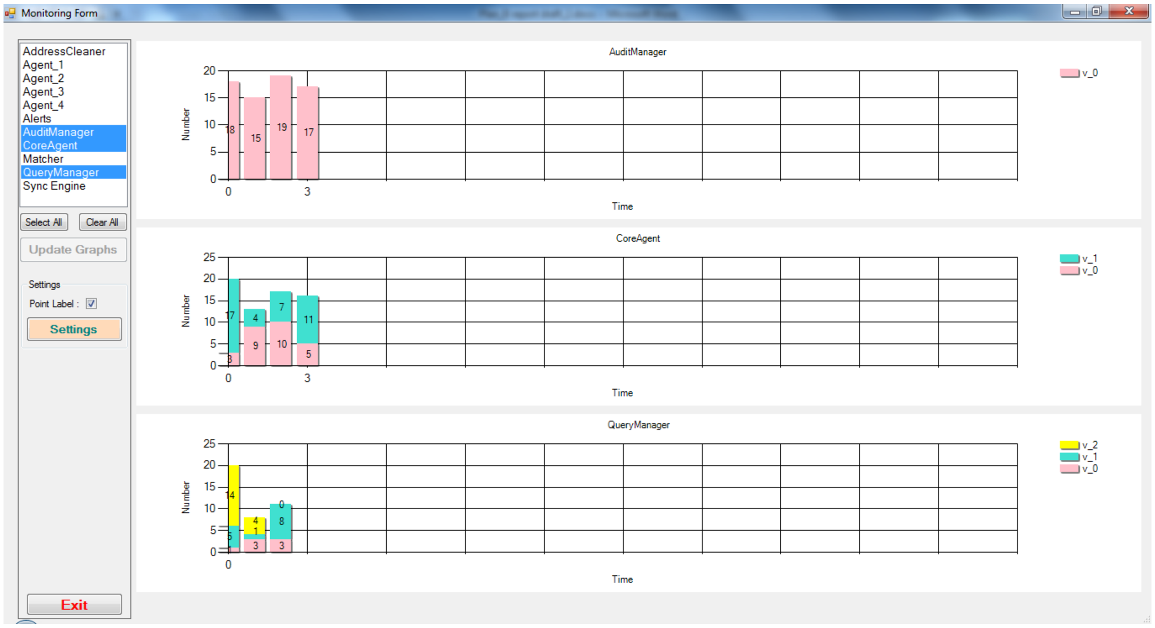
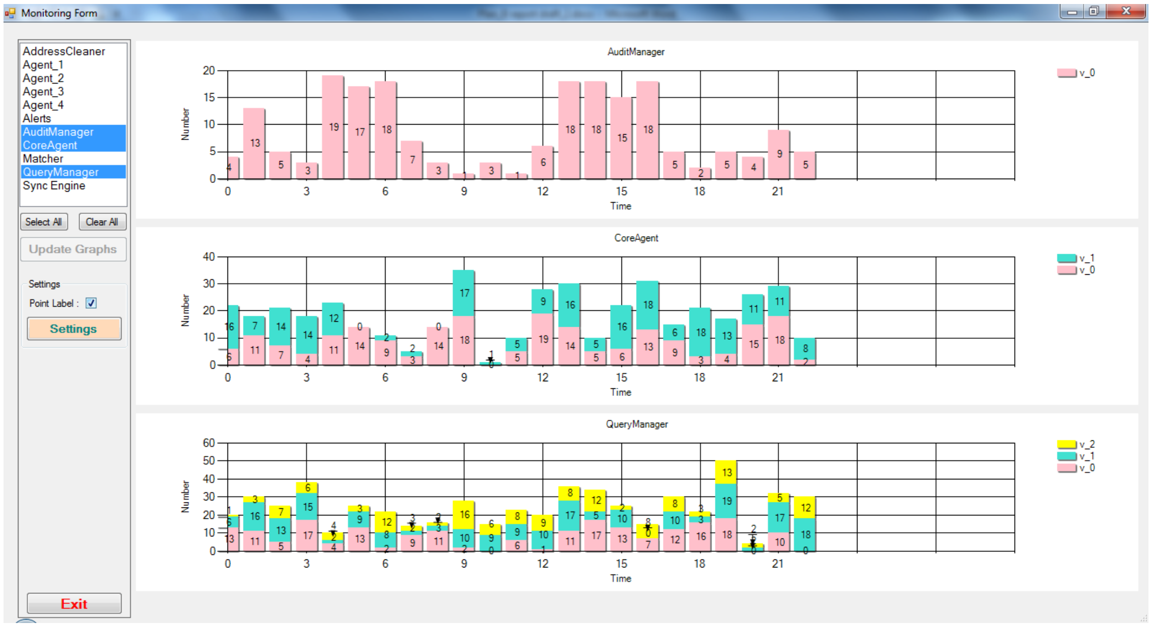
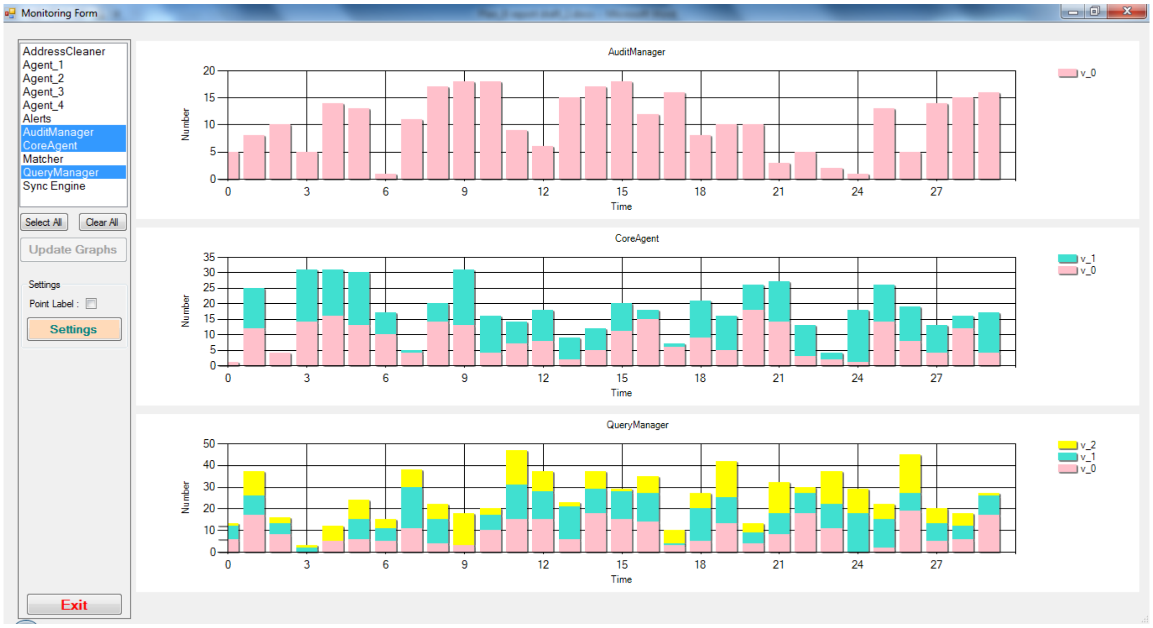


Figure 8.5 - Graphs of selected components’ first four sets of values received from the *MonitoringService*.



Point Label : ☑

Figure 8.6 - Graphs of selected components' 23 sets of values received from the *MonitoringService*.



Point Label : ☐

Figure 8.7 - Graphs of selected components' 30 sets of values received from the *MonitoringService*.

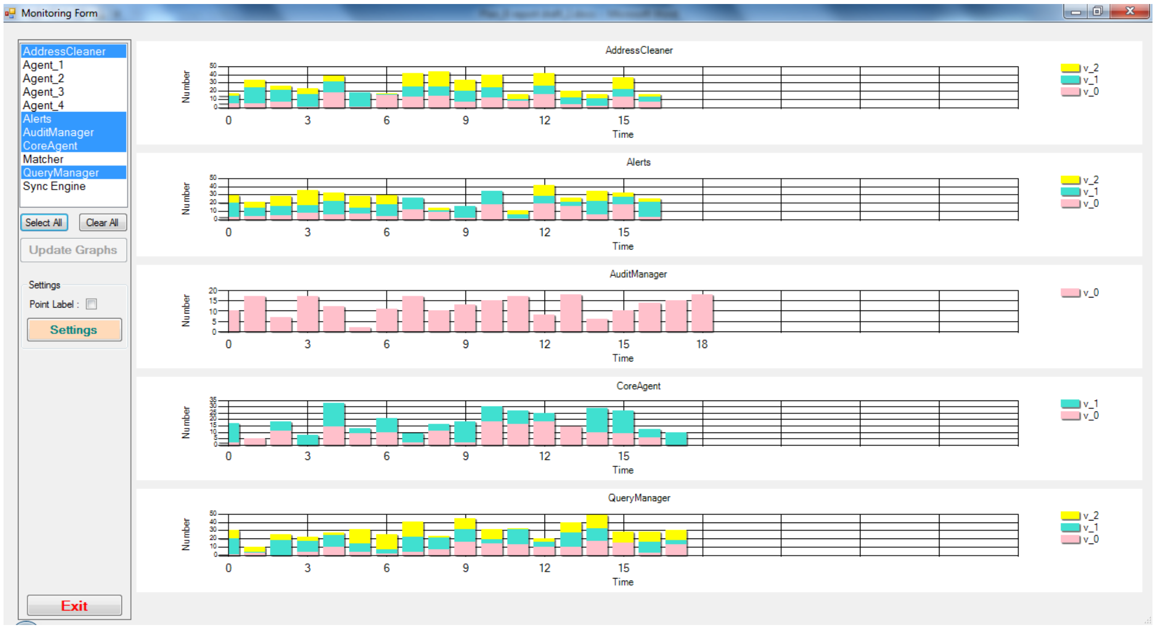


Figure 8.8 - GUI displays received values. Based on the # of selected components, the GraphForm calculates the graph sizes dynamically to display them on the screen.

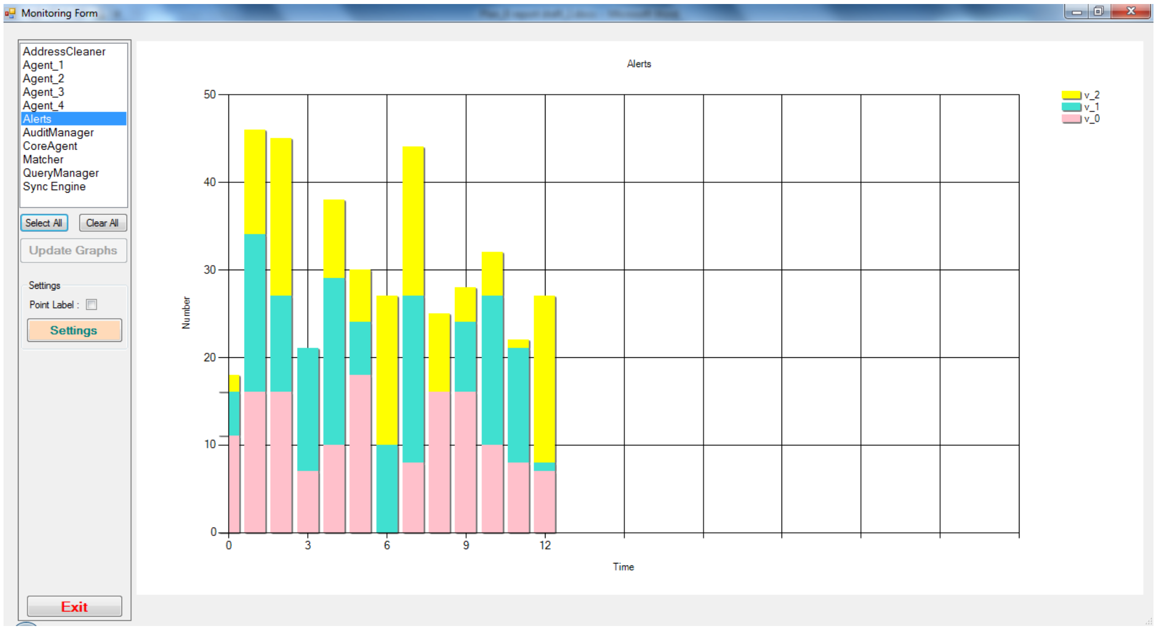


Figure 8.9 - Graph of one component. Since there is only one selected component, the graph size has been changed dynamically to display one graph.

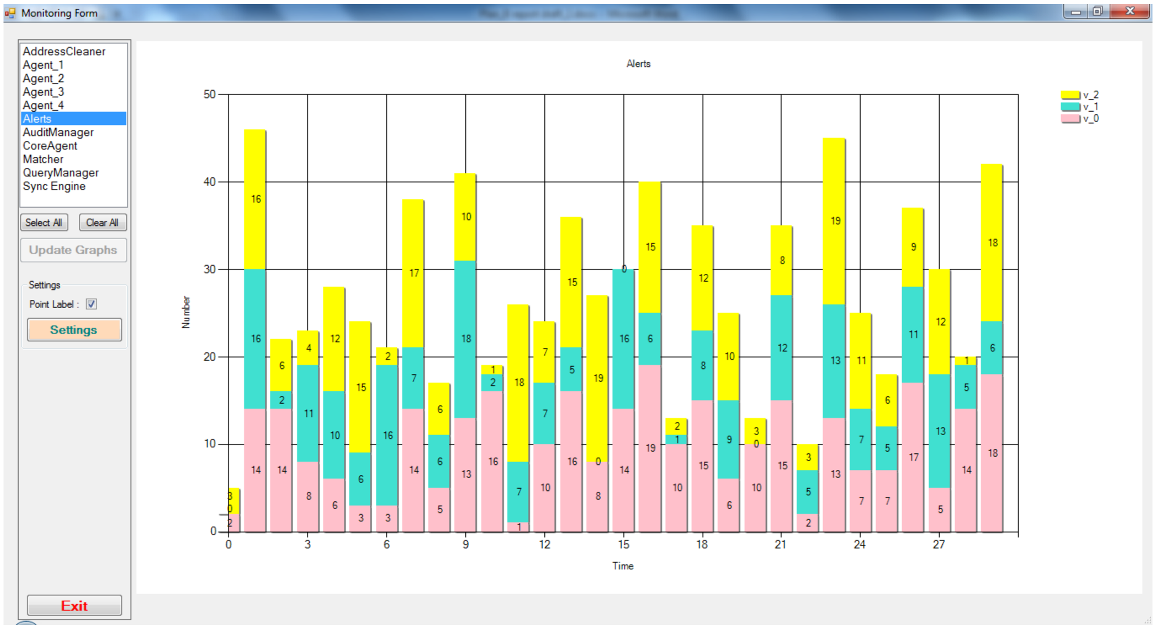


Figure 8.10 - GUI displays numbers related to each color of a bar.

8.2. Settings

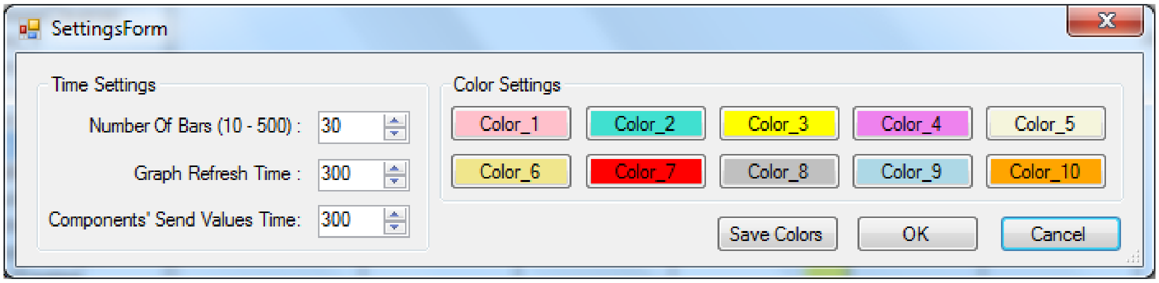
The user can change the settings of the graphs using the “Settings" button (Figure 8.11). *Settings* form has two sections;

- Time settings
- Color settings

Using these two sections, the user is able to change;

- The value of *Number Of Bars*
- The value of *Graph Refresh Time*
- The value of *Components’ Send Values Time*
- The *color of the bars* in the graph(s).

The user can also save the color settings after making changes. Time settings default values can be changed in the file called Service_01.xml (Figure A-1 in Appendices).



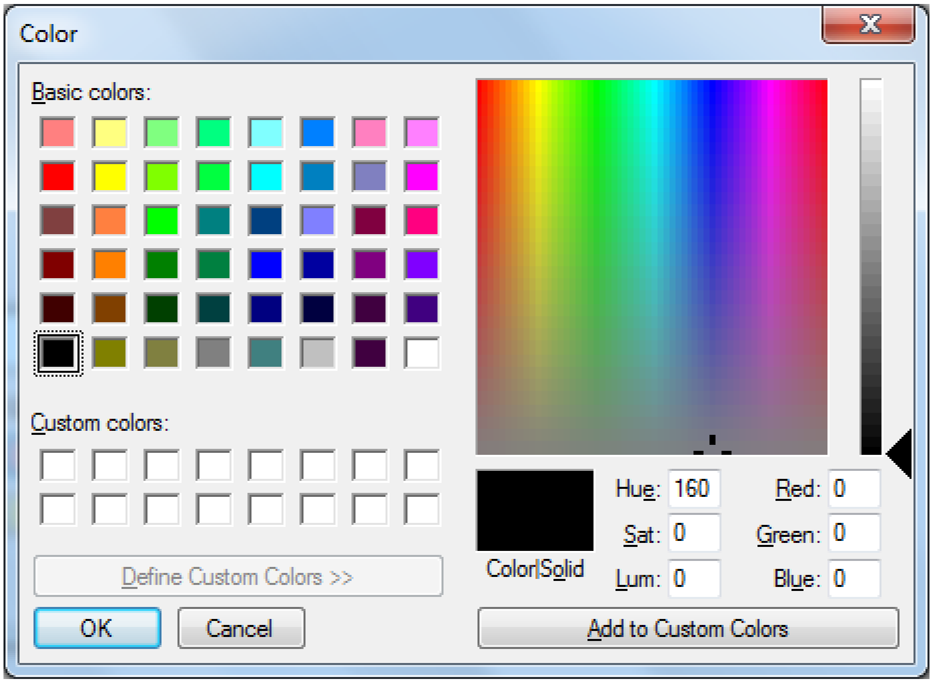
Number Of Bars (10 - 500) : 30

Graph Refresh Time : 300

Components' Send Values Time: 300

Figure 8.11 - GUI has settings option.

When the user clicks on any button in the “Color Settings” section, the color selection panel appears for a selection of a new color (Figure 8.12).



Hue: 160

Sat: 0

Lum: 0

Red: 0

Green: 0

Blue: 0

Figure 8.12 - Once a user clicks on a button in the “Color Settings” Color selection panel appears for the user to select a new color.

After changing the colors of the bars, users can save these colors for the future use. If users save the settings, these settings will be loaded next time when the application is run. If users do not save the settings, the default values will be used after re-starting the application.

If users make some changes and do not save the settings, new settings will be used by current running application, but next time when the application is run, the default or previously saved values will be used by the application

CHAPTER 9

CONCLUSION AND FUTURE WORK

The *CCMS* is a tool to monitor the performance of the selected components. It provides users a very easy to use user interface to monitor the components. It helps operators quickly identify the performance problems. The system also provides great statistical data about components.

The CCMS uses *Vitruvian* for distributing objects. Distributed objects also communicate via Vitruvian. The CCMS also uses Microsoft's Chart library to create and locate graphs, add new points (values) to them and display these graphs on the screen. Since the real CHARM components are not ready to use in the system, dummy components are used to simulate the monitoring. For this reason, ComponentServices are passing random numbers to the MonitoringService for graphs. Once the real CHARM components are ready, the algorithm in the ComponentService must be changed and the real components should send values to the ComponentService and the ComponentService will send these values (not the random numbers) to the MonitoringService.

In addition to this, each component needs to have a listener thread to listen to its ComponentService. The MonitoringService communicates with ComponentServices to receive ComponentServices' values. Once the ComponentServices have been asked to send their values, real CHARM components would send the values to their related ComponentServices and the ComponentServices would send these values to the MonitoringService.

REFERENCES

1. Bertolino, A. (2007). Software Testing Research: Achievements, Challenges, Dreams. IEEE Computer Society, 1.
2. Class Diagrams, Wikipedia, http://en.wikipedia.org/wiki/Class_diagram
3. Dustin, E. (2002). Effective Software Testing. Boston, MA: Pearson Education.
4. Eriksson, Hans-Eric; Penker, Magnus; Lyons, Brian; Fado, David. (2004). UML 2 Toolkit. Indianapolis, Indiana: Wiley Publishing Inc.
5. CHARM, http://charm.health.utah.gov/index.html
6. Software Testing, Wikipedia, http://en.wikipedia.org/wiki/Software_testing
7. System Analysis and Design, Google.com, http://knol.google.com/k/ali-abbas/system-analysis-and-design/20lv4waafc4io/2#
8. Systems Design, Wikipedia, http://en.wikipedia.org/wiki/Systems_design
9. Integration Testing, Microsoft, http://msdn.microsoft.com/en-us/library/aa292128(v=vs.71).aspx
10. Unit Testing, Microsoft, http://msdn.microsoft.com/en-us/library/aa292197(v=vs.71).aspx
11. Li, E. Y. (1990). Software Testing In A System Development. Journal of Systems Management, 23-31.
12. Use Case Diagrams, Wikipedia, http://en.wikipedia.org/wiki/Use_case_diagram

APPENDICES

```xml
 <?xml version="1.0" encoding="utf-8" ?>
 <object>
   <item type="Vitruvian.Logging.LoggingService, Vitruvian.Logging" />
   <item type="Vitruvian.Distribution.Time.SystemTime,
 Vitruvian.Distribution" />

   <item type="Vitruvian.Windows.Services.AsyncUIService,
 Vitruvian.Windows">
     <property name="AppForm" type="MonitoringForm.GraphForm,
 MonitoringForm">
       <property name="Text" value="Monitoring Form" />
       <property name="sleepThreadIdle" value="700"/>
       <property name="maxBarNumber" value="30"/>
       <property name="timeRefreshGraph" value="300"/>
       <property name="timeComponentSendData" value="300"/>
       <property name="fileNameForColors" value="colors.xml"/>
     </property>
   </item>

   <!-- MonitoringService Service -->
   <item type="Monitoring.MonitoringService, Monitoring" id="localService"
 >
<property name="DataFile" value=" componentSettingsList.xml"/>
     <property name="ServiceName" value="MonitoringService"/>
     <property name="Port" value="19301"/>
     <!--<property name="MAXLISTSIZE" value="80"/>-->
   </item>

   <!-- Sync Patterns Service -->
   <item
 type="Vitruvian.Distribution.SyncPatterns.EmbeddedSyncPatternsService,
 Vitruvian.Distribution">
     <property name="BaseType" value="MonitoringForm.Program,
 MonitoringForm"/>
     <property name="ResourcePath" value="Config.SyncPatterns.xml"/>
   </item>

   <item type="Vitruvian.Distribution.DistributionService,
 Vitruvian.Distribution">
     <property name="Encoder"
 type="Vitruvian.Distribution.Encoders.XmlObjectEncoder,
 Vitruvian.Distribution" />
     <property name="MessageTimeout" value="15000"/>
     <property name="HeartbeatFrequency" value="10000"/>
     <property name="HeartbeatTimeout" value="120000"/>
     <property name="UseHeartbeats" value="true"/>
     <property name="Services">
       <item ref="localService"/>
     </property>
```

```xml
    <property name="Providers">
      <item type="Vitruvian.Communications.Sockets.TcpServerProvider,
Vitruvian.Communications">
        <property name="LocalEndPoint" value="127.0.0.1:19301"/>
      </item>
    </property>
  </item>
</object>
```

Figure A-1 Content of the file *Service_01.xml*.

```xml
<object type="Monitoring.MonitoringService, Monitoring, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null">
  <property name="DataFile" encoding="W3C" value="xmlFile.xml" />
  <property name="Id" value="bf398723-face-4518-89d4-3b7da98ddd70" />
  <property name="Name" encoding="W3C" value="MonitoringService" />
  <field name="ComponentSettingList">
    <item>
      <property name="ComponentName" encoding="W3C"
value="AddressCleaner" />
      <property name="NumberOfValues" encoding="W3C" value="3" />
    </item>
    <item>
      <property name="ComponentName" encoding="W3C" value="Agent_1" />
      <property name="NumberOfValues" encoding="W3C" value="2" />
    </item>
    <item>
      <property name="ComponentName" encoding="W3C" value="Agent_2" />
      <property name="NumberOfValues" encoding="W3C" value="2" />
    </item>
    <item>
      <property name="ComponentName" encoding="W3C" value="Agent_3" />
      <property name="NumberOfValues" encoding="W3C" value="2" />
    </item>
    <item>
      <property name="ComponentName" encoding="W3C" value="Agent_4" />
      <property name="NumberOfValues" encoding="W3C" value="2" />
    </item>
    <item>
      <property name="ComponentName" encoding="W3C" value="Alerts" />
      <property name="NumberOfValues" encoding="W3C" value="3" />
    </item>
    <item>
      <property name="ComponentName" encoding="W3C" value="AuditManager"
/>
      <property name="NumberOfValues" encoding="W3C" value="1" />
    </item>
    <item>
      <property name="ComponentName" encoding="W3C" value="CoreAgent" />
      <property name="NumberOfValues" encoding="W3C" value="2" />
    </item>
    <item>
      <property name="ComponentName" encoding="W3C" value="Matcher" />
      <property name="NumberOfValues" encoding="W3C" value="2" />
    </item>
    <item>
      <property name="ComponentName" encoding="W3C" value="QueryManager"
/>
      <property name="NumberOfValues" encoding="W3C" value="3" />
    </item>
    <item>
      <property name="ComponentName" encoding="W3C" value="SyncEngine" />
      <property name="NumberOfValues" encoding="W3C" value="2" />
    </item>
  </field>
</object>
```

Figure A-2 Content of the file *componentSettingsList.xml*.

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfSerializableColor xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SerializableColor>
    <A>255</A>
    <R>255</R>
    <G>192</G>
    <B>203</B>
  </SerializableColor>
  <SerializableColor>
    <A>255</A>
    <R>64</R>
    <G>224</G>
    <B>208</B>
  </SerializableColor>
  <SerializableColor>
    <A>255</A>
    <R>255</R>
    <G>255</G>
    <B>0</B>
  </SerializableColor>
  <SerializableColor>
    <A>255</A>
    <R>238</R>
    <G>130</G>
    <B>238</B>
  </SerializableColor>
  <SerializableColor>
    <A>255</A>
    <R>245</R>
    <G>245</G>
    <B>220</B>
```

```xml
  </SerializableColor>
  <SerializableColor>
    <A>255</A>
    <R>240</R>
    <G>230</G>
    <B>140</B>
  </SerializableColor>
  <SerializableColor>
    <A>255</A>
    <R>255</R>
    <G>0</G>
    <B>0</B>
  </SerializableColor>
  <SerializableColor>
    <A>255</A>
    <R>192</R>
    <G>192</G>
    <B>192</B>
  </SerializableColor>
  <SerializableColor>
    <A>255</A>
    <R>173</R>
    <G>216</G>
    <B>230</B>
  </SerializableColor>
  <SerializableColor>
    <A>255</A>
    <R>255</R>
    <G>165</G>
    <B>0</B>
  </SerializableColor>
</ArrayOfSerializableColor>
```

Figure A-3 Content of the file *colors.xml*.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<object>
  <property name="LocalPatterns">
    <entry>
      <entry-key value="Gossip"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Local.GossipSyncPatter
n, Vitruvian.Distribution"/>
    </entry>
    <entry>
      <entry-key value="MostRecent"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Local.MostRecentSyncPa
ttern, Vitruvian.Distribution"/>
    </entry>
    <entry>
```

```
      <entry-key value="Push"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Local.PushSyncPattern,
Vitruvian.Distribution"/>
    </entry>
    <entry>
      <entry-key value="Pull"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Local.PullSyncPattern,
Vitruvian.Distribution"/>
    </entry>
    <entry>
      <entry-key value="Constant"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Local.ConstantSyncPatt
ern, Vitruvian.Distribution"/>
    </entry>
    <entry>
      <entry-key value="RPC"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Local.RPCSyncPattern,
Vitruvian.Distribution"/>
    </entry>
    <entry>
      <entry-key value="ARPC"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Local.ARPCSyncPattern,
Vitruvian.Distribution"/>
    </entry>
    <entry>
      <entry-key value="Stub"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Local.StubSyncPattern,
Vitruvian.Distribution"/>
    </entry>
    <entry>
      <entry-key value="SmartList"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Local.SmartListSyncPat
tern, Vitruvian.Distribution"/>
    </entry>
    <entry>
      <entry-key value="MostRecent - Migratable"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Fragments.Local.MostRecentSync
Pattern, Vitruvian.Distribution"/>
    </entry>
  </property>
  <property name="RemotePatterns">
    <entry>
      <entry-key value="Gossip"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Remote.GossipSyncPatte
rn, Vitruvian.Distribution"/>
    </entry>
    <entry>
      <entry-key value="MostRecent"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Remote.MostRecentSyncP
attern, Vitruvian.Distribution"/>
    </entry>
    <entry>
      <entry-key value="Push"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Remote.PushSyncPattern
, Vitruvian.Distribution"/>
    </entry>
    <entry>
      <entry-key value="Pull"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Remote.PullSyncPattern
, Vitruvian.Distribution"/>
    </entry>
    <entry>
      <entry-key value="Constant"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Remote.ConstantSyncPat
tern, Vitruvian.Distribution"/>
    </entry>
    <entry>
      <entry-key value="RPC"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Remote.RPCSyncPattern,
Vitruvian.Distribution"/>
    </entry>
    <entry>
      <entry-key value="ARPC"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Remote.ARPCSyncPattern
, Vitruvian.Distribution"/>
    </entry>
    <entry>
      <entry-key value="Stub"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Remote.StubSyncPattern
, Vitruvian.Distribution"/>
    </entry>
    <entry>
      <entry-key value="SmartList"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Mirrors.Remote.SmartListSyncPa
ttern, Vitruvian.Distribution"/>
    </entry>
    <entry>
      <entry-key value="MostRecent - Migratable"/>
      <entry-value
value="Vitruvian.Distribution.SyncPatterns.Fragments.Remote.MostRecentSyn
cPattern, Vitruvian.Distribution"/>
    </entry>
  </property>
</object>
```

Figure A-4 Content of the file *SyncPatterns.xml*.

```
<?xml version="1.0" encoding="utf-8" ?>
<object>
  <item type="Vitruvian.Logging.LoggingService, Vitruvian.Logging" />
  <item type="Vitruvian.Distribution.Time.SystemTime,
Vitruvian.Distribution" />

  <item type="Vitruvian.Windows.Services.AsyncUIService,
Vitruvian.Windows">
    <property name="AppForm" type="ComponentForm.MainForm,
ComponentForm">
      <property name="Text" value="Comp_QueryManager" />
    </property>
  </item>

  <item type="Monitoring.ComponentService, Monitoring" id="localService"
>
    <property name="ComponentName" value="QueryManager"/>
    <property name="Port" value="19302"/>
    <property name="NumberOfValues" value="3"/>
    <property name="SleepThreadIdle" value="700"/>
  </item>

  <!-- Sync Patterns Service -->
  <item
type="Vitruvian.Distribution.SyncPatterns.EmbeddedSyncPatternsService,
Vitruvian.Distribution">
    <property name="BaseType" value="ComponentForm.Program,
ComponentForm"/>
    <property name="ResourcePath" value="Config.SyncPatterns.xml"/>
  </item>

  <item type="Vitruvian.Distribution.DistributionService,
Vitruvian.Distribution">
    <property name="Encoder"
type="Vitruvian.Distribution.Encoders.XmlObjectEncoder,
Vitruvian.Distribution" />
    <property name="MessageTimeout" value="15000"/>
    <property name="HeartbeatFrequency" value="10000"/>
    <property name="HeartbeatTimeout" value="120000"/>
    <property name="UseHeartbeats" value="true"/>
    <property name="Services">
      <item ref="localService"/>
    </property>
    <property name="Providers">
      <item type="Vitruvian.Communications.Sockets.TcpServerProvider,
Vitruvian.Communications">
        <property name="LocalEndPoint" value="127.0.0.1:19302"/>
      </item>
      <item type="Vitruvian.Communications.Sockets.TcpClientProvider,
Vitruvian.Communications">
        <property name="RemoteEndPoint" value="127.0.0.1:19301"/>
      </item>
    </property>
  </item>
</object>
```

Figure A-5 Content of the file *Service_2.xml*.