Bench4Q

Bench4Q-rm 1.0

Design Document

2010-8-26

Contents

1 Introduction 1

2 System Structure 1

3 Modules 2

3.1 Bench4Q Tool 2

3.1.1 Resource monitor component 2

3.1.2 UI component 5

3.2 SUT 6

3.2.1 Architecture 6

3.2.2 Design brief 7

3.3 Detailed design 11

3.4 ServerMon 16

3.4.1 Architecture Overview 17

3.4.2 Interfaces with the outside world 17

3.4.3 JAVA Classes 18

3.4.4 ServerMonitorWindows 20

3.4.5 ServerMonitorLinux 20

3.5 ClusterMon 21

3.5.1 Architecture Overview 21

3.5.2 Interfaces with the outside world 23

3.5.3 JAVA Classes 24

1 Introduction

Bench4Q is an open-source implementation of TPC-W benchmark; it also proposes and implements its own QoS-oriented benchmark scheme.

Bench4Q-rm is based on Bench4Q which implements server-side resource monitoring. It incorporates ServerMon and ClusterMon which run independently of the application server or database server and give monitoring data of one server machine or every node of a server cluster respectively. The client connects to ServerMon or ClusterMon to retrieve the monitoring data, including CPU utilization, free physical memory, disk reading and writing velocity, and network receiving and transmitting velocity. The monitoring data will be illustrated at the client's GUI. This is for OW2 Programming Contest 2010 topic "38. Implement server-side resource monitor in Bench4Q".

Bench4Q-rm also implements its SUT (System Under Test) web application with EJB3/Servlet; at the client side, users can choose whether to use SUT based on EJB3/Servlet or pure Servlet. This is for OW2 Programming Contest 2010 topic "39. Implement on-line book store in Bench4Q by Servlet/EJB3 ".

All the softwares or components mentioned above are licensed under LGPLv2.1 or later versions.

2 System Structure

The architecture of the system containing Bench4Q-rm is shown in Figure 1.

Bench4Q-rm is composed of agent, console, SUT (System Under Test), ServerMon and ClusterMon.

Agent is deployed on every load generator, and there is a machine on which console is deployed to control the benchmarking process and display results. SUT is deployed on application servers. ServerMon is used on a single server, whether application server or database server.

ClusterMon is used on clustered servers; every node of the cluster should have one ClusterMon instance, and exact one of them should be designated leader, the console of the client connects to the leader to get the monitoring data of the cluster.

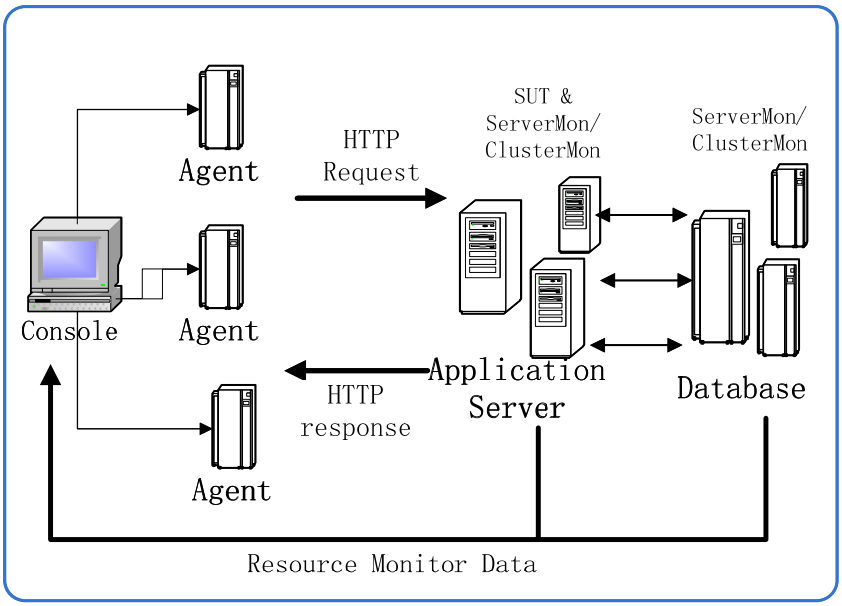

Figure 1 The architecture of the system containing Bench4Q-rm

3 Modules

3.1 Bench4Q Tool

3.1.1 Resource monitor component

3.1.1.1 Overview of component functions

The resource monitor component implements the function which can connect UI with ServerMon or ClusterMon and transfer the record of server into the presentation component.

3.1.1.2 Component classes and class relations in the graph

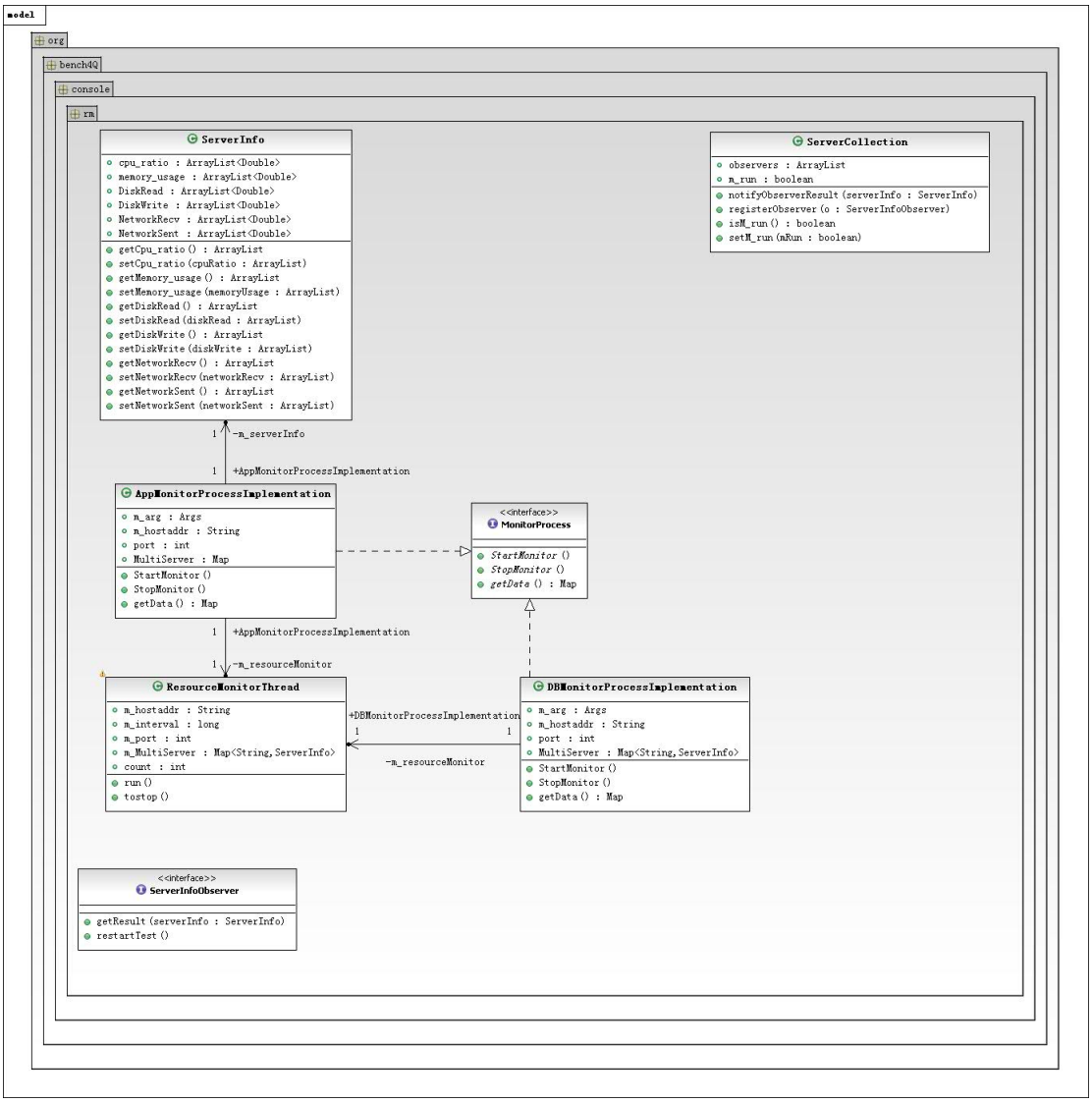

Figure 2 the class diagram of resource monitor module

- **MonitorProcess interface:**

  Function: Define the basic operation of monitoring process.

  Main operations:

  - **StartMonitor: start the monitor**
  - **StopMonitor: stop the monitor**
  - **getData: get data from the monitor**

- **ServerInfo class:**

Function: Define the data structure of server information

Main attributes:

- **Cpu_ratio: the CPU Usage of the server during the measurement interval**
- **Memory_usage:** the free physical memory of the server during the measurement interval
- **DiskRead:** the speed of disk read during the measurement interval
- **DiskWrite:** the speed of disk write during the measurement interval
- **NetworkRecv：** the speed of server received data from the network during the measurement interval
- **NetworkSent：**the speed of server sent data to the network during the measurement interval

- **AppMonitorProcessImplementation class and DBMonitorProcessImplementation class**

Function: Implement the MonitorProcess interface and monitor application server and database server respectively.

Inheritance: implements MonitorProcess

Main Attributes:

- **m_hostaddr:** the IP address of the server
- **port:** the port of the ServerMon or ClusterMon
- **MultiServer:** the whole record of the cluster or the record of the server

ResourceMonitorThread class:

Function: connect with the ServerMon or ClusterMon and record the data

Inheritance: extends Thread

ServerInfoObserver interface:

Function: define the operations from which can get server data

Main operations:

- **getResult:** provide the server data
- **restartTest:** restart the test

3.1.1.3 Relationship with other components

The resource monitor component provide server data to other components and control the whole monitoring process. The diagram component implements the ServerInfoObserver interface and gets the server information. The UI component instantiates App or DB monitor process and monitor the server.

3.1.2 UI component

3.1.2.1 The overview of component functions

User interface component provides several diagrams of the server data.

3.1.2.2 Component classes and class relations in the graph

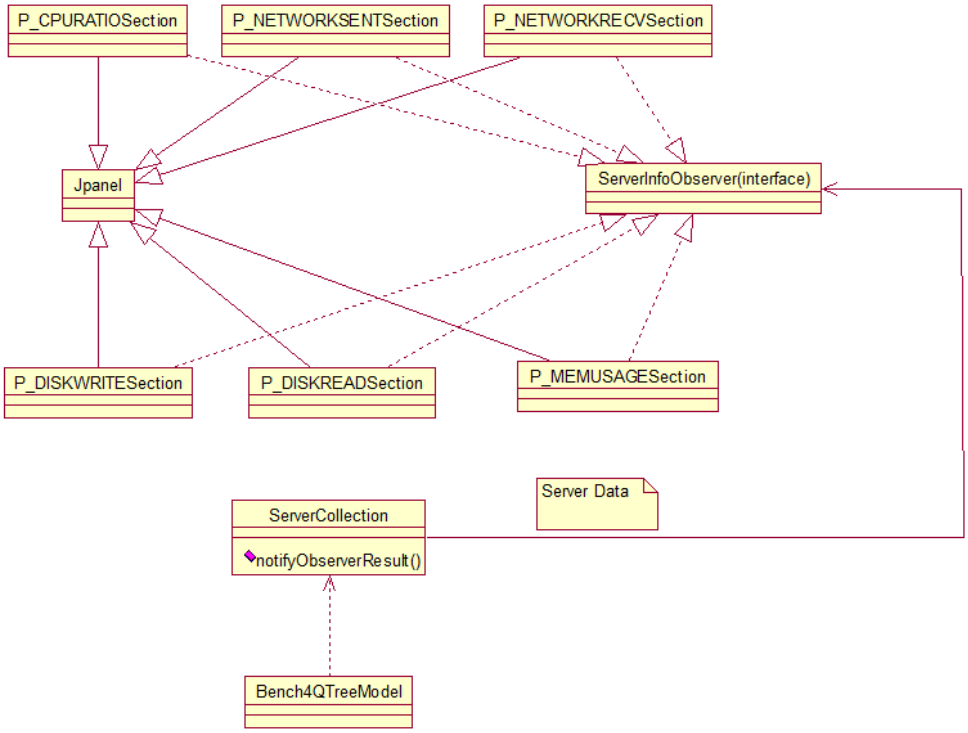

Figure 3 the class diagram of UI module

- **Bench4QTreeModel class:**

  Function: initialize the Bench4Q tree Model and add new listener to receive the result

  Main operations(new):

  - **m_processControl.addProcessResultListener:** register a new listener and when the user presses the collecting button, this new listener will be triggered and create server nodes as many as the cluster has.
  - **removeNode:** remove the web and database server nodes before user starts the test.

3.2 SUT

Bench4Q contains an on-line book store web application. We call it System Under Test or SUT for short. Currently, this web application is implemented by Java Servlet and JDBC. Our goal is to re-implement it by Servlet and EJB3 while keep the original JDBC approach to access the database.

3.2.1 Architecture

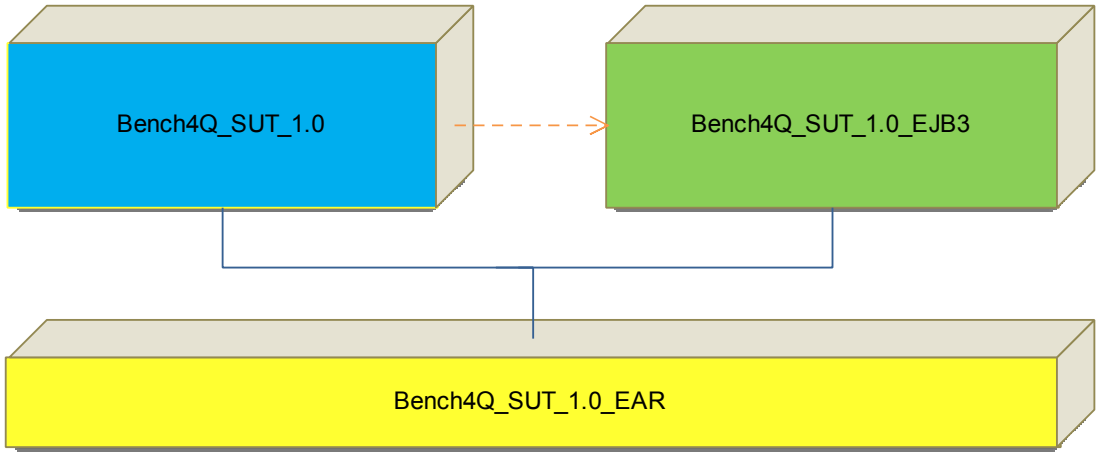

Figure 4 Projects that make up the new SUT

Originally, SUT contains only one Eclipse project **Bench4Q_SUT_1.0**，which is a java web project. We add two projects,

- **Bench4Q_1.0_EJB3**: an EJB project to deal with the interaction with the database by

EJB3.

- **Bench4Q_SUT_1.0_EAR**: an Enterprise Application Project to bundle the web and ejb part together.

3.2.2 Design brief

3.2.2.1 Original interaction pattern

SUT implements an online book store web application based on TPC-W specification. It uses DB2 to store data. To access the data, it adopts the JDBC/datasource approach. It also adopts transaction mechanism to guarantee data consistency.

In SUT, when a servlet receives a request, typically, it invokes a (some) static method(s) of ***org.bench4Q.servlet.Database*** class to update or retrieve data, then it construct correct response according to the result returned from the ***Database*** class. ***Database*** class contains the main business logic and is the only part to access database directly. This has improved the maintainability of the code.

Figure 5 shows how a servlet access data through Database class.

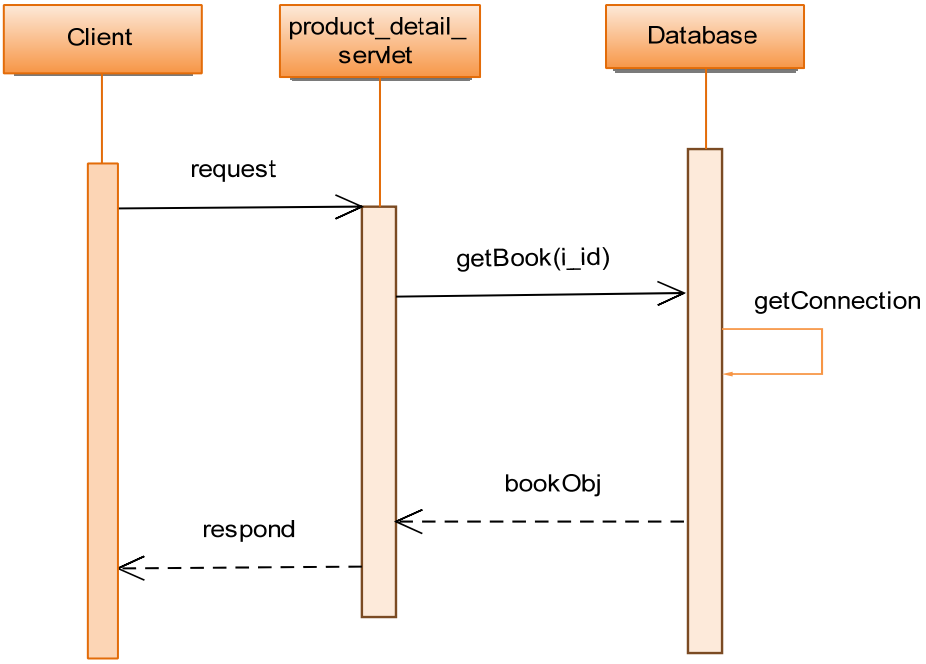

Figure 5 Original interaction pattern of accessing data

3.2.2.2 New interaction patterns

We need to add EJB3 approach as a new available option. To minimize influence to original code, we convert *Database* class as a common façade to both database access approaches, JDBC and EJB3.

We achieve this by doing following three steps,

1. Extract an interface ***org.bench4Q.servlet.IBench4QDao*** containing methods definitions needed by the servlets, this work can be done based on the public methods in ***Database*** class.
2. Add two new classes ***org.bench4Q.servlet.JdbcDao*** and ***org.bench4Q.servlet.EJBDao***, both of which implement ***org.bench4Q.servlet.IBench4QDao***. ***JdbcDao*** implements the JDBC approach to access the database and most of its code immigrate from *Database* class. Comparatively, ***EJBDao*** implements the EJB3 approach. It uses stateless session beans(with entity beans behind them) to interact with the database.
3. For ***Database*** class, keep those methods called by the servlets and remove others. Change the body of the methods so that they call the corresponding methods of ***JdbcDao*** or ***EJBDao*** according to the value of a flag variable.

Flag variable

We add a static boolean field *useEJB3* to class ***Database.*** If it has value true, *Database* delegates requests to data to an ***EJBDao*** instance, or else delegates requests to a ***JdbcDao*** instance.

The flag variable needs to be configurable from client.

In the new SUT, there are three patterns of how servlets' needs to access data are fulfilled by the backend.

1. **JDBC Pattern**

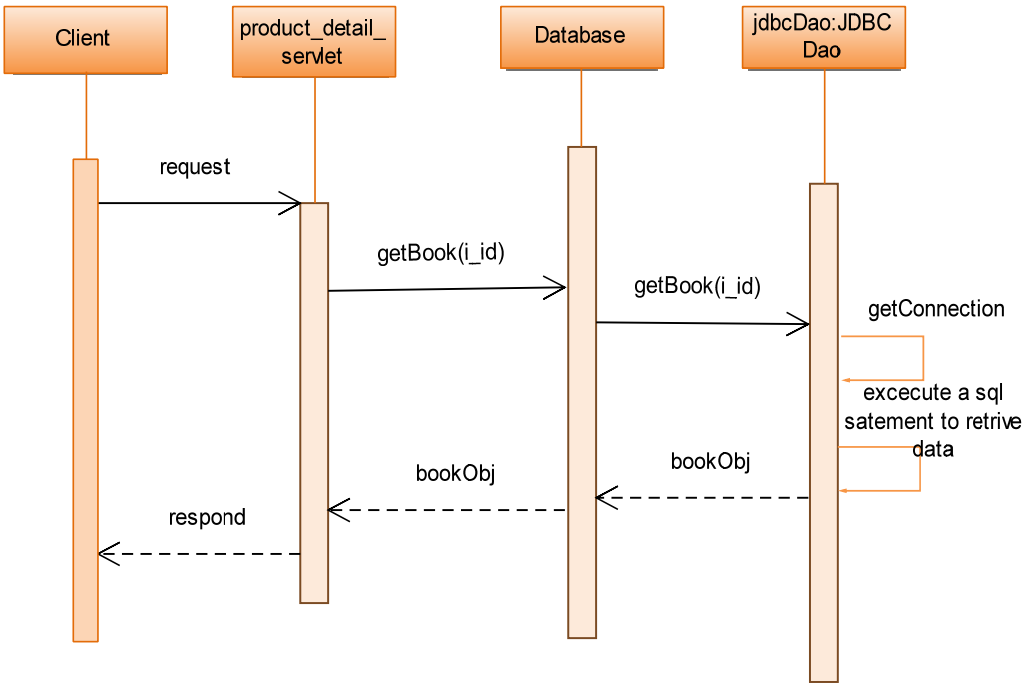

Figure 6 "JDBC Pattern" in new SUT

Figure 6 shows how a request to *product_detail_servlet* is fulfilled when the flag variable *Database.useEJB3* has value false. Note that both Database and JdbcDao has the same method signature "getBook(i_id)". The main logic is done in ***JdbcDao*** class, where a sql statement is executed to retrieve data. For other servlets, there are similar interactions, except that for some requests, ***JDBCDao*** will execute several statements(*select/update/insert*).

2. EJB Simple Pattern

In this pattern, the request is relatively simple, so that we don't need to use cross-table queries. We invoke methods in stateless session beans corresponding to entity beans to retrieve or persist data. Figure 7 is an example of this pattern. *fItem* and *fAuthor* are objects of stateless session beans corresponding to entity beans respectively.

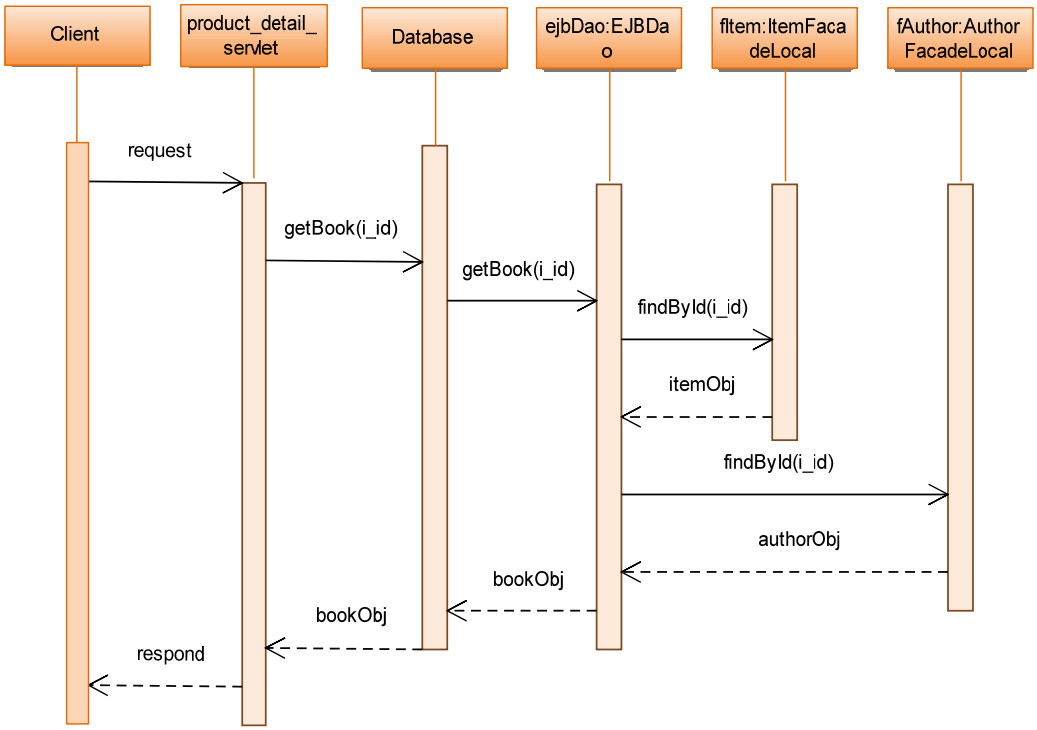

Figure 7 "EJB3 Simple Pattern" in new SUT

3. EJB Complex Pattern

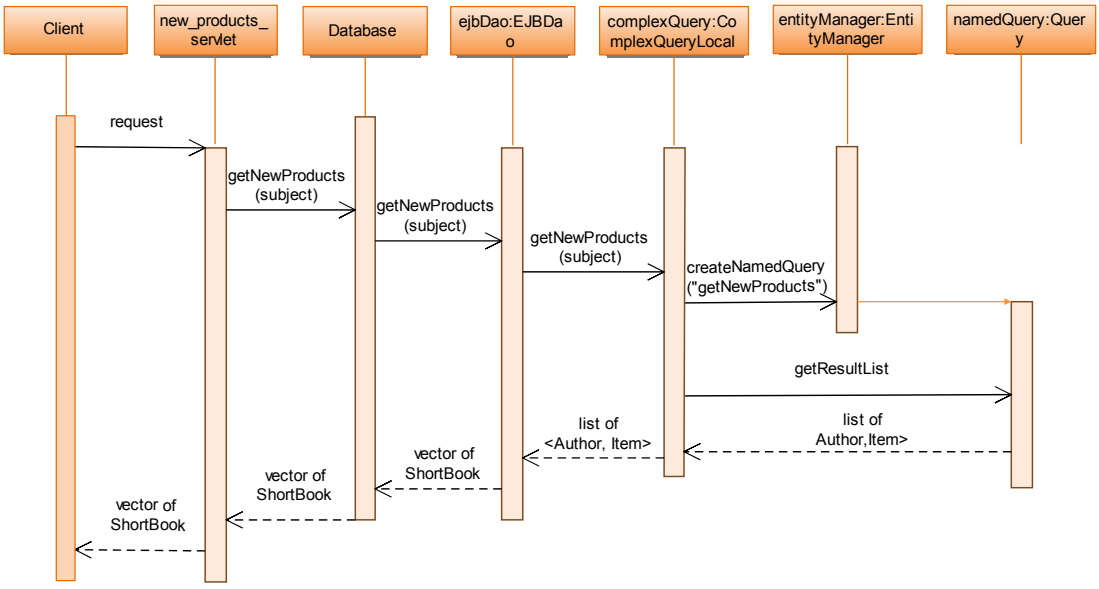

Figure 8 "EJB Complex Pattern" in new SUT

In this pattern, the request is relatively complex so that it's necessary to use cross-table queries. We design a stateless bean ***org.bench4Q.ejb3.ComplexQuery***(it implements interface

org.bench4Q.ejb3.ComplexQueryLocal) to handle all the cross-table queries. In contrast to other stateless session beans, it doesn't correspond to a specific entity bean or a specific table in the Database.

3.3 Detailed design

3.3.1.1 Bench4Q_SUT_1.0_EJB3

Figure 9 show the database schema of Bench4Q SUT. In Bench4Q_SUT_1.0_EJB3, we use MyEclipse to generate an entity class and a stateless session bean class(and its local interface) for each table automatically. The generated entity classes don't need to be edited. The generated stateless session bean classes already contain some useful methods(*findById, save, update, …*).

As is shown in Figure 8, we create a special stateless session bean ***org.bench4Q.ejb3.ComplexQuery***(it implements interface ***org.bench4Q.ejb3.ComplexQueryLocal***) to handle cross-table queries. When we need to add a method to interact with the database, we add it to corresponding session bean class if it relates to just one table. If it's not the case, we add it to ***org.bench4Q.ejb3.ComplexQueryLocal***.

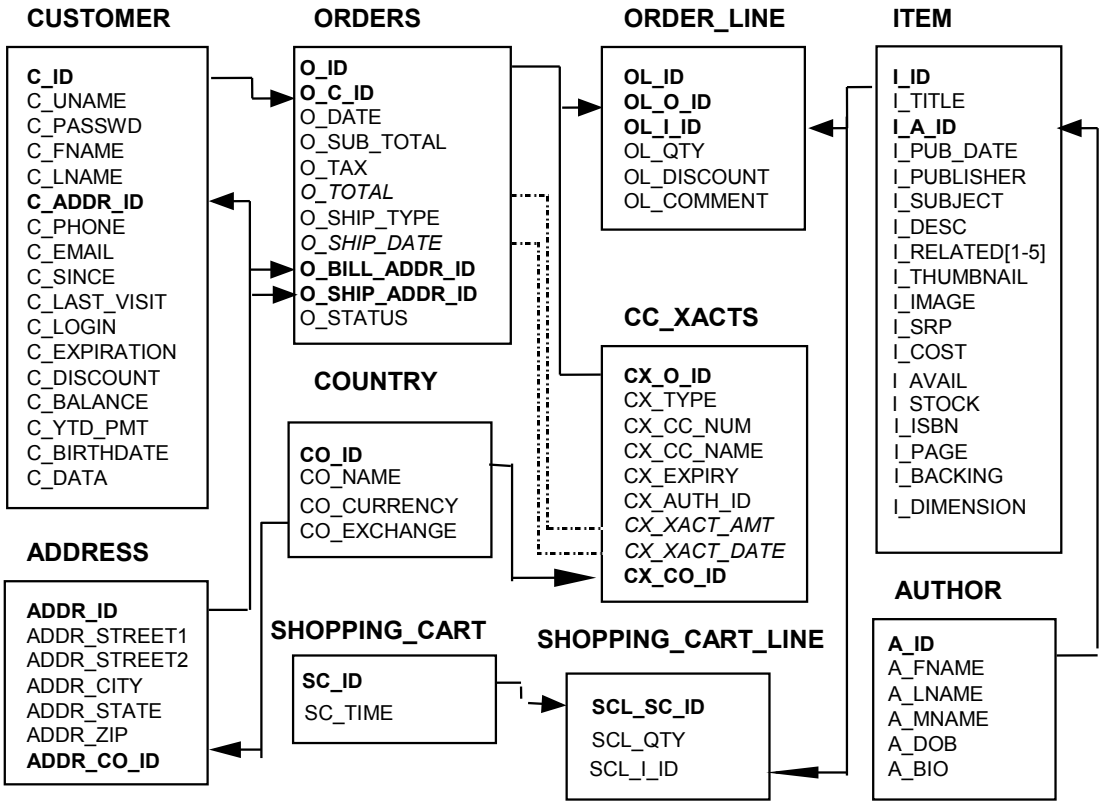

Figure 9 Database Schema of Bench4Q SUT

Figure 10 shows the class diagram of ***org.bench4Q.ejb3. ComplexQueryLocal***.

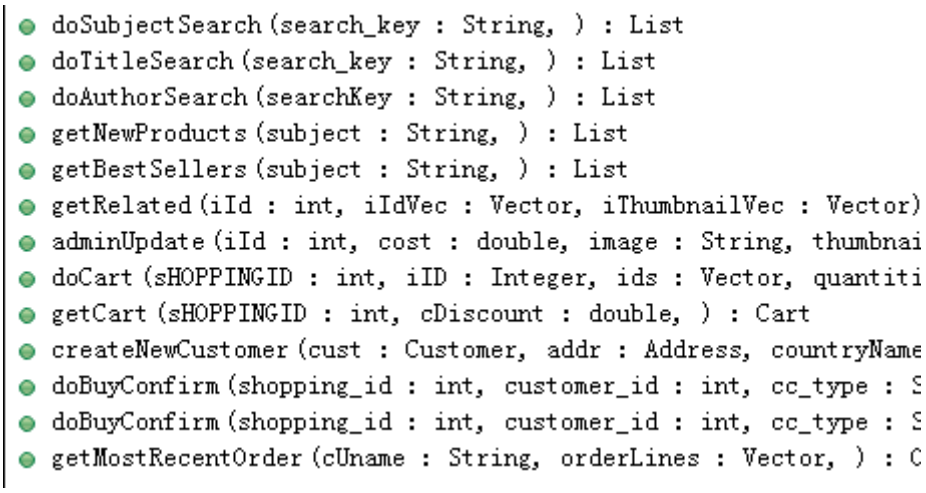

Figure 10 Class diagram of ***org.bench4Q.ejb3.ComplexQueryLocal***

To assure that **Bench4Q_1.0_EJB3** doesn't depend on **Bench4Q_SUT_1.0**, in **Bench4Q_1.0_EJB3** we add ***mirror classes*** to some of the pojo classes(call them ***source classes***) in **Bench4Q_SUT_1.0**. **Mirror classes** have similar content compared with their ***source classes***.

Table 1 **Mirror classes** and their corresponding **source classes**

mirror classes in Bench4Q_1.0_EJB3	*source classes* in Bench4Q_SUT_1.0
org.bench4q.EJB3.BuyConfirmResult	org.bench4q.servlet.BuyConfirmResult
org.bench4Q.EJB3.Cart	org.bench4Q.servlet.Cart
org.bench4Q.EJB3.CartLine	org.bench4Q.servlet.CartLine
org.bench4Q.EJB3.Order	org.bench4Q.servlet.Order
org.bench4Q.EJB3.OrderLineTemp	org.bench4Q.servlet.OrderLine

The introduction of mirror classes increases some coding work, but decreases confusion. The conversion between a mirror class object and its corresponding source class object is carried out in **Bench4Q_SUT_1.0**.

3.3.1.2 Bench4Q_SUT_1.0

1. Flag variable

Flag variable indicates which db access approach should be used. When its value is false, JDBC approach should be used, and when the value is true, EJB3 approach is used. The default value is false.

To make the flag variable configurable from client, we introduce a servlet to deal with its configuration. The servlet is ***db_access_approach_servlet***.

Since the SUT may run on a application server cluster, we need to make sure that the flag variables in all the nodes have the same value. We adopt a notification mechanism to achieve this. When a node receives a request to update the flag variables, it first updates the one in it. Then it uses ***java.net.HttpURLConnection*** to send a notification request to each other nodes in the same cluster. When a node receives notification request from another node, it just update the flag variable in it. Figure 11 shows this process,

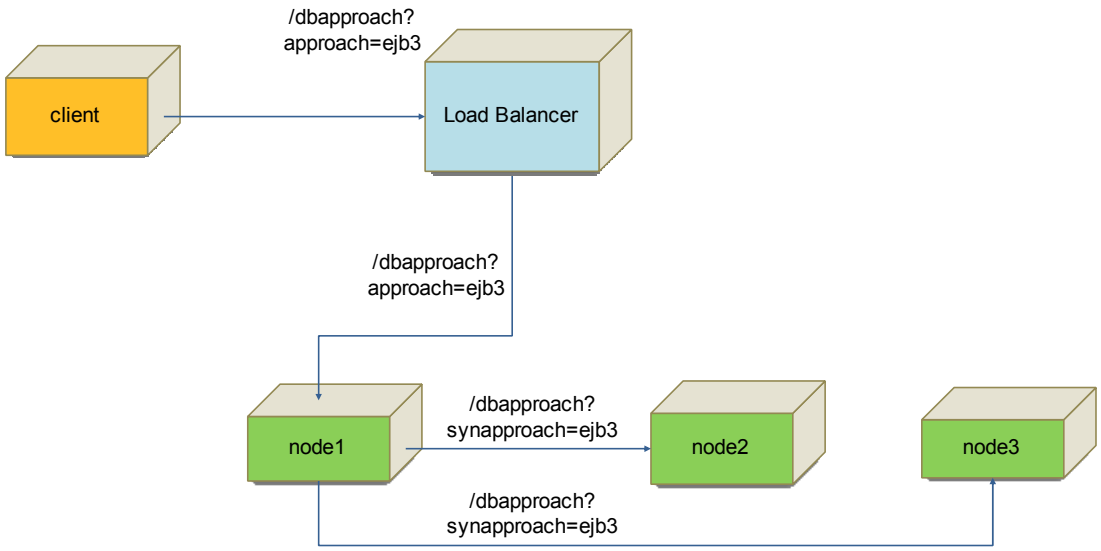

Figure 11 Configuration of the flag variable in an application server cluster

We incorporate a class ***ClusterHelper*** and a configuration file *cluster.properties* to help decide which ips should the notification request be sent to. The file *cluster.properties* define just one property, following is a sample file:

```
#if benc4q_sut is deployed on a application server cluster, set nodes
property to all the node ips
nodes = 133.133.133.156, 133.133.133.158
```

ClusterHelper is responsible for reading file *cluster.properties,* and generating a list of node ips to notify.

2. Main classes

1) org.bench4Q.servlet.IBench4QDao

The class ***IBench4QDao*** defines the business api of the book store. Its methods are extracted from original Database class. Figure 12 shows its class diagram.

2) org.bench4Q.servlet.JdbcDao

JdbcDao implements the JDBC approach to access data from the database. Its code is imigrated from original ***org.bench4Q.servlet.Database*** class.

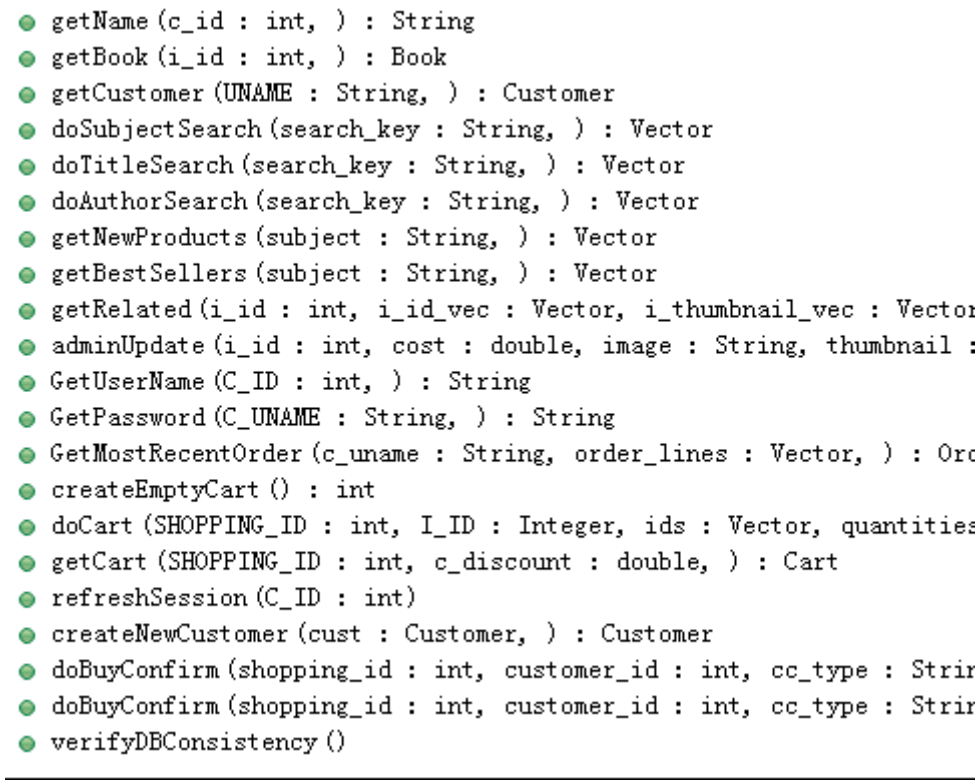

Figure 12 Class diagram of org.bench4Q.,servlet.*IBench4QDao*

3) org.bench4Q.servlet.EJBDao

EJBDao implements the EJB3 approach. Internally it invokes methods of stateless session beans defined in **Bench4Q_SUT_1.0_EJB3** to retrieve and save data.

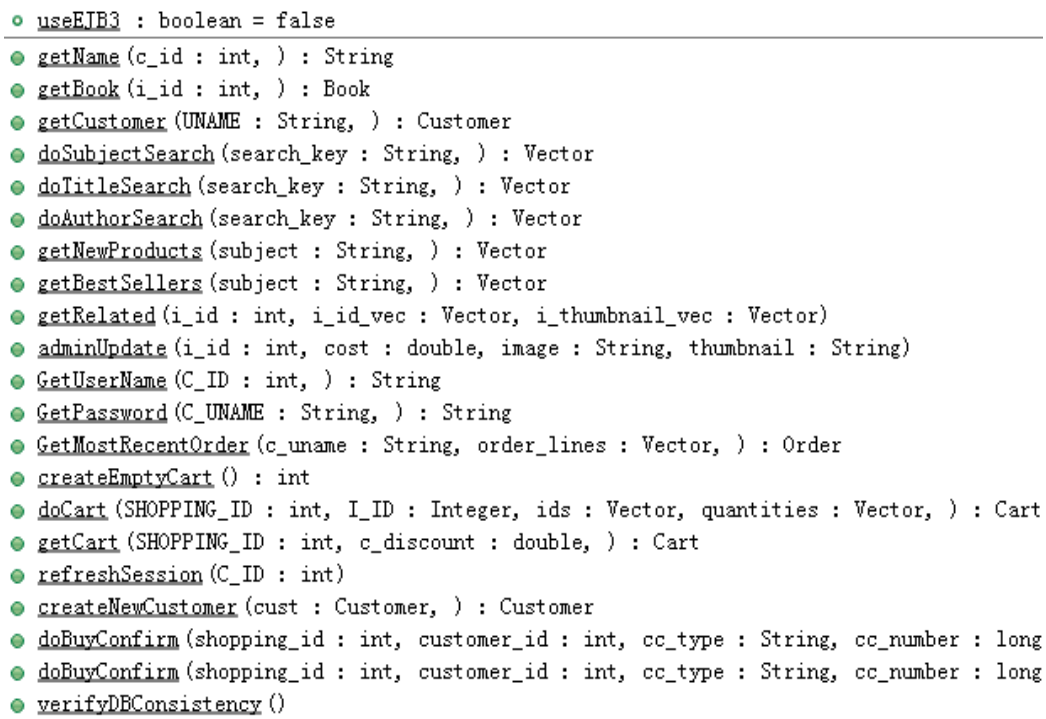

Figure 13 Class diagram of the new ***org.bench4Q.servlet DataBase*** class

4) org.bench4Q.servlet.DataBase

DataBase class acts as a unique entry to the data. Based on the value of the flag variable, it delegates requests to a ***JdbcDao*** instance or an ***EJBDao*** instance. Figure 13 shows the class diagram of new ***Database*** Class.

3.3.1.3 Bench4Q_SUT_1.0_EAR

Bench4Q_SUT_1.0_EAR is an Enterprise Application Project to bundle the web and ejb part together. It contains following application.xml as its main content:

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="5"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/application_5.xsd">
    <display-name>Bench4Q_SUT_1.0_EAR</display-name>
    <module id="ejb">
        <ejb>Bench4Q_SUT_1.0_EJB3</ejb>

    </module>
    <module id="web">
        <web>
            <web-uri>Bench4Q_SUT_1.0.war</web-uri>
            <context-root>/bench4Q</context-root>
        </web>
    </module>
</application>
```

3.4 ServerMon

ServerMon is a server-side resource monitoring component. It contains both JAVA and C++ code. The JAVA code uses JNI to invoke native methods written in C++ to collect the resource monitoring data, and it exposes an RMI interface for the console to invoke so as to retrieve the monitoring data.

3.4.1 Architecture Overview

ServerMon contains both JAVA and C++ code. The JAVA code acts as the bridge between C++ code, which gets the resource monitoring data from the operating system, and the client, which connects to it to retrieve the data. The class diagram of ServerMon's JAVA code is shown in Figure 14.

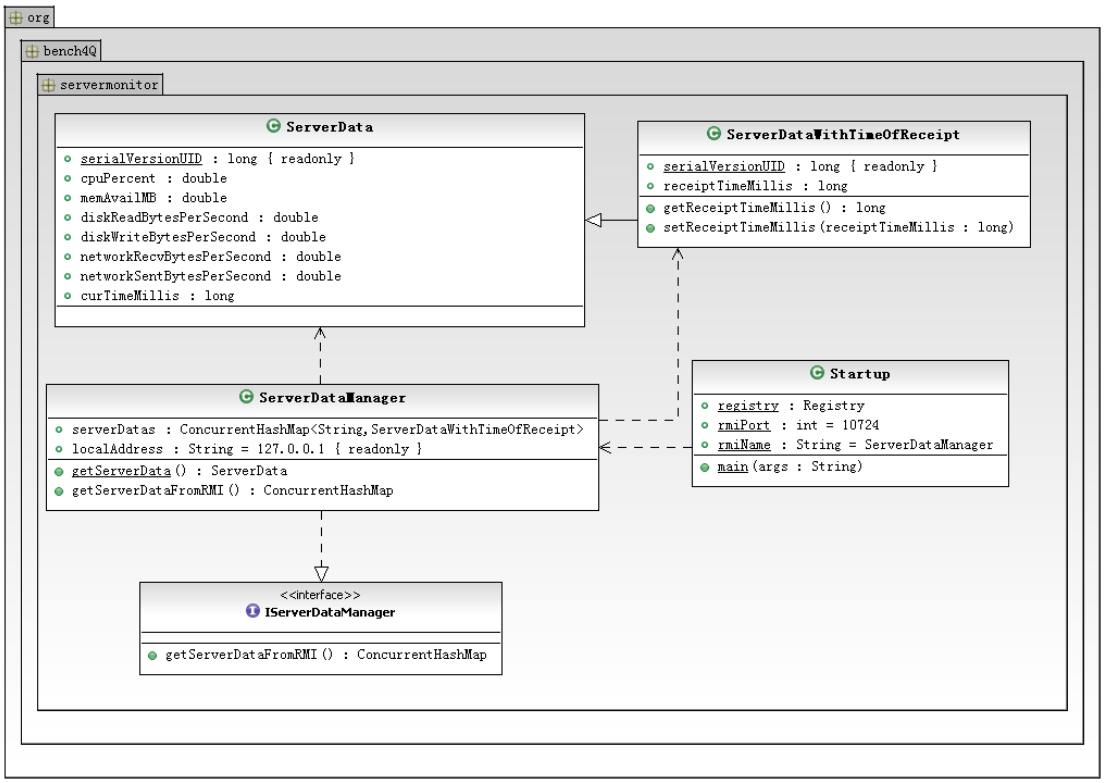

Figure 14 The class diagram of **ServerMon**

The C++ code are compiled into *ServerDataProvider.dll* and *libServerDataProvider.so*, which are used through JNI under Windows and Linux respectively. The C++ code design will be covered by 3.4.4 and 3.4.5.

3.4.2 Interfaces with the outside world

ServerMon uses RMI to interact with the client, which connects to it to retrieve monitoring data. It will create or use an RMI registry on local machine, with the default port being *10724*. Users can change this port by setting the system property *"servermonregport"*, for example, users

might add *"-Dservermonregport=10725"* to the command line to change the port to *10725*. The name of the RMI service is *"ServerDataManager"*. The first invocation of the service may return incorrect values and should be ignored; a field of the return value will be *-1.0* if the very field cannot be obtained from the operating system.

3.4.3 JAVA Classes

3.4.3.1 Package org.bench4Q.servermonitor

- **Interface IServerDataManager:**

This is the interface of the RMI service exported to the client. It has the following method:

- **public ConcurrentHashMap<String, ServerDataWithTimeOfReceipt> getServerDataFromRMI() throws RemoteException**

Retrieve the server's monitoring data. The data is enclosed in a ***ConcurrentHashMap***, which should have only one entry, with the key being *"127.0.0.1"*. The first invocation of this method may return incorrect values. A field of a value of the ***ConcurrentHashMap*** will be -1.0 when it cannot be obtained from the operating system.

- **Class ServerData**

This is a bean class to represent the server resource monitoring data. Its fields are as follows:

- **private double cpuPercent**

Current CPU utilization, in percentage.

- **private double memAvailMB**

Current free physical memory, in mega bytes.

- **private double diskReadBytesPerSecond**

Current disk reading velocity, in bytes per second.

- **private double diskWriteBytesPerSecond**

Current disk reading velocity, in bytes per second.

- **private double networkRecvBytesPerSecond**

Current network receiving velocity. It is the summation of receiving velocity of all IPv4

network interfaces, except the loopback interface, in bytes per second.

- **private double networkSentBytesPerSecond**

Current network transmitting velocity. It is the summation of transmitting velocity of all IPv4 network interfaces, except the loopback interface, in bytes per second.

- **private long curTimeMillis**

Current timestamp of the monitored machine, in milliseconds. The absolute value of it is operating-system-specific, so client should not use it to determine the server's time.

Class ServerDataWithTimeOfReceipt

It is a subclass of ***ServerData,*** also a bean class. It mainly has the following fields:

- **private long receiptTimeMillis**

It is used to be compatible with **ClusterMon**, so of no use in **ServerMon**.

Class ServerDataManager

This class extends ***UnicastRemoteObject*** and is an implementation of the ***IServerDataManager*** interface. It mainly has the following methods:

- **public static native ServerData getServerData()**

It is a native method to retrieve server monitoring data. This method is implemented in *ServerDataProvider.dll* and *libServerDataProvider.so.*

- **public ConcurrentHashMap<String, ServerDataWithTimeOfReceipt> getServerDataFromRMI() throws RemoteException**

Invokes *getServerData* and get the returned a ***ServerData*** object, wrap this object in a ***ServerDataWithTimeOfReceipt*** object, put the ***ServerDataWithTimeOfReceipt*** object into a ***ConcurrentHashMap*** object and returns it.

Class Startup

This class is responsible for the starting and stopping of **ServerMon**. It mainly has one method:

- **public static void main(String[] args)**

Create the RMI registry at the specified port.

Create a ***ServerDataManager*** object and bind it with RMI registry.

Wait for user to type 'exit', and then exits the program.

3.4.4 ServerMonitorWindows

ServerMonitorWindows is written in C++, incorporating MFC library. It is compiled into *ServerDataProvider.dll* and exports a method for JAVA code to invoke through JNI to retrieve the monitoring data. It uses Windows' PDH api to get the monitoring data from Windows system. The class diagram of **ServerMonitorWindows** is shown in Figure 15.

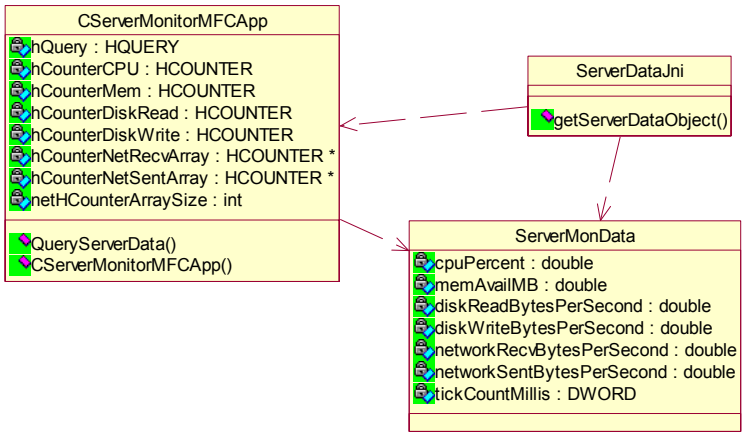

Figure 15 The class diagram of **ServerMonitorWindows**

3.4.5 ServerMonitorLinux

ServerMonitorLinux is written in C++. It is compiled into *libServerDataProvider.so* and exports a method for JAVA code to invoke through JNI to retrieve the monitoring data. It uses proc file system to get the monitoring data from Linux system:

- /proc/stat for CPU utilization
- /proc/diskstats for disk reading and writing velocity
- /proc/meminfo for free physical memory

- /proc/net/dev for network sending and receiving velocity

The class diagram of **ServerMonitorLinux** is shown in Figure 16.

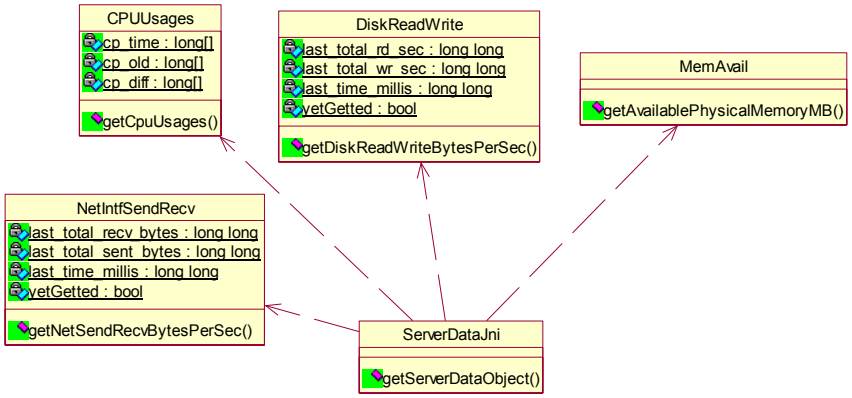

Figure 16 The class diagram of **ServerMonitorLinux**

3.5 ClusterMon

The cluster should be on a network supporting IP multicasting; a LAN-based cluster will certainly be OK. When running, every node of the cluster should have a **ClusterMon** instance, and only one of them be designated as leader (by the user manually), with others being subordinates. The leader uses IP multicasting to inform the subordinates of its address, and the subordinates send their resource monitoring data to the leader at intervals (every 0.5 second) via UDP datagrams. The client connects to the leader to retrieve the data of the cluster through RMI, which is a map with keys being the nodes' IP address and an exception of the leader's address being "127.0.0.1".

The leader/subordinate scheme let the client only need to reach the leader machine, and thus support the server cluster in an internal network but the client not.

3.5.1 Architecture Overview

ClusterMon contains JAVA and C++ code. The C++ code are exactly the same as ClusterMon's, which is covered in 3.4.4 and 3.4.5. The JAVA code realizes the cluster communication and acts as the bridge between C++ code, which gets the resource monitoring data

from the operating system, and the client, which connects to it to retrieve the data. The class diagram of **ClusterMon**'s JAVA code is shown in Figure 17.

At start, the leader will create two threads. One of them is called *MulticastNotifier*; it sends a multicast data packet every second to inform subordinates of its address. Another is called *SubordinateDataReceiver,* which is waiting at a specified port to receive UDP datagrams containing subordinate's monitoring data, the data will be recorded together with sender's IP address and time of receipt.

A subordinate will also create two threads at start. One of them is called *MulticastReceiver,* which is waiting at a specified port of a specified multicast address, and when received a packet, it updates the leader's address to packet sender's address. Another is called *MonitoringDataSender*; it gets the monitoring data through JNI and sends the data to the leader's address via UDP datagram at intervals (every 0.5 second).

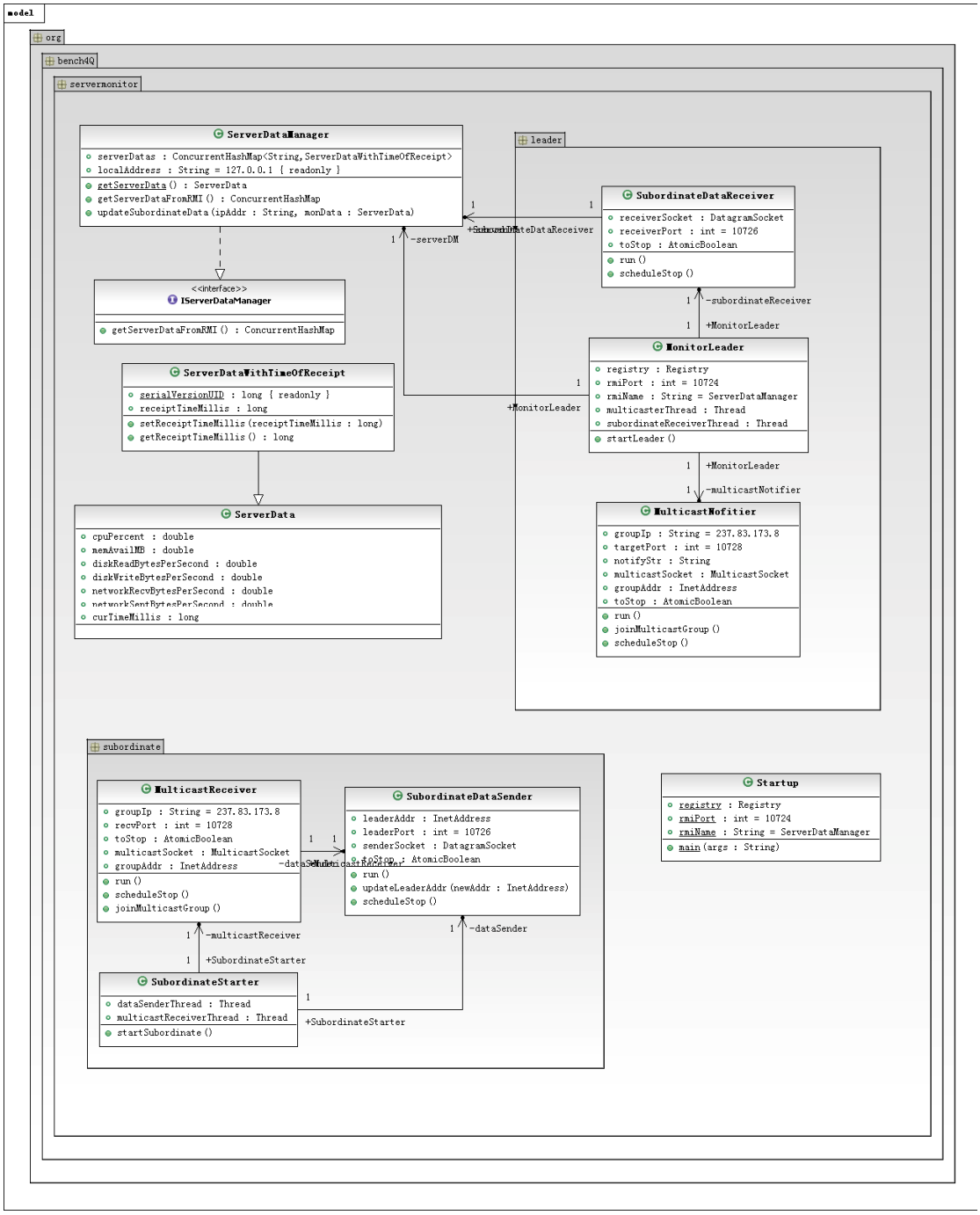

Figure 17 The class diagram of **ClusterMon**

3.5.2 Interfaces with the outside world

ClusterMon leader uses RMI to interact with the client, which connects to it to retrieve monitoring data. It will create or use an RMI registry on local machine, with the default port being 10724. Users can change this port by setting the system property *"servermonregport"*, for example, users might add *"-Dservermonregport=10725"* to the command line to change the port to *10725*.

The name of the RMI service is *"ServerDataManager"*. The first invocation of the service may return incorrect values and should be ignored; a specified field of the return value will be -1.0 if the very field cannot be obtained from the operating system.

Besides the system property *"servermonregport",* there are some other system properties acting as parameters:

- "subordinatereceiverport" specifies which port the leader gets subordinates' monitoring data from and a subordinate sends monitoring data to, the default value being *10726*.
- "multicasttargetport" specifies which port the leader sends the multicast packet to and a subordinate gets multicast packet from, the default value being *10728*.
- "multicastgroupip" specifies the multicast address the leader and subordinates will use, the default value being *"237.83.173.8"*.

3.5.3 JAVA Classes

3.5.3.1 package org.bench4Q.servermonitor

- **Interface IServerDataManager:**

This is the interface of the RMI service exported to the client. It has the following methods:

- **public ConcurrentHashMap<String, ServerDataWithTimeOfReceipt> getServerDataFromRMI() throws RemoteException**

Retrieve the cluster's monitoring data. The data is enclosed in a ***ConcurrentHashMap***, with the key being node's IP address and an exception of the leader's address being *"127.0.0.1"*. The first invocation of this method may return incorrect values. A field of a value of the ***ConcurrentHashMap*** will be *-1.0* when it cannot be obtained from the operating system.

- **Class ServerData**

The same a**s ServerMon**'s.

- **Class ServerDataWithTimeOfReceipt**

It is a subclass of ***ServerData***, also a bean class. It mainly has the following field:

- **private long receiptTimeMillis**

The time (on leader's machine) when the leader receives the data contained in this class's super class ServerData, in milliseconds between the current time and *midnight, January 1, 1970 UTC*.

Class ServerDataManager

This class extends ***UnicastRemoteObject*** and is an implementation of the ***IServerDataManager*** interface.

It mainly has the following fields:

- **private ConcurrentHashMap<String, ServerDataWithTimeOfReceipt> serverDatas**

The map of leader's received monitoring data and the data of its own machine. When the **ClusterMon** acts as a subordinate, this field is not used.

It mainly has the following methods:

- **public static native ServerData getServerData()**

It is a native method to retrieve this node's monitoring data. This method is implemented in *ServerDataProvider.dll* and *libServerDataProvider.so*.

- **public ConcurrentHashMap<String, ServerDataWithTimeOfReceipt> getServerDataFromRMI() throws RemoteException**

Invokes *getServerData* and get the returned a ***ServerData*** object, wrap this object in a ***ServerDataWithTimeOfReceipt*** object, put the ***ServerDataWithTimeOfReceipt*** object into the map *serverDatas* and returns the map *serverDatas.*

- **public void updateSubordinateData(String ipAddr, ServerData monData)**

This method is invoked in the *SubordinateDataReceiver* thread to save or update subordinate's data in map *serverDatas*.

Class Startup

This class is responsible for the starting and stopping of **ClusterMon**. It mainly has one method:

- **public static void main(String[] args)**

  Determine whether this instance should be a leader or a subordinate.

  If it should be a leader, it creates a ***MonitorLeader*** object and calls the *startLeader()* method; if it should be a subordinate, it creates a ***SubordinateStarter*** object and calls the *startSubordinate ()* method.

3.5.3.2 Package org.bench4Q.servermonitor.leader

- **Class MonitorLeader**

This class is responsible for the starting and stopping of **ClusterMon** leader. It mainly has one method:

- **public void startLeader()**

  Create the RMI registry at the specified port.

  Create a ***ServerDataManager*** object and bind it with RMI registry.

  Start two threads: *"MulticastNotifier"* and *"SubordinateDataReceiver"*.

  Wait for user to type 'exit', and then exits the program.

- **Class MulticastNofitier**

This class implements ***Runnable***, and will be incorporated into the thread *MulticastNotifier*. It mainly has one method:

- **public void run()**

  Sends a multicast data packet every second to inform subordinates of its address.

- **Class SubordinateDataReceiver**

This class implements ***Runnable***, and will be incorporated into the thread *SubordinateDataReceiver*. It mainly has one method:

- **public void run()**

Continuously wait at a specified port to receive UDP datagrams containing subordinate's monitoring data; after receiving a valid data it will call the *updateSubordinateData* method of the ***ServerDataManager*** instance.

3.5.3.3 Package org.bench4Q.servermonitor.subordinate

- **Class MulticastReceiver**

This class implements ***Runnable***, and will be incorporated into the thread *MulticastReceiver*. It mainly has one method:

- **public void run()**

Continuously wait at a specified port of a specified multicast address, and when received a packet, it updates the leader's address to packet sender's address.

- **Class SubordinateDataSender**

This class implements ***Runnable***, and will be incorporated into the thread *MonitoringDataSender*. It mainly has one method:

- **public void run()**

Gets the monitoring data through JNI and sends the data to the leader's address via UDP datagram at intervals (every 0.5 second).

- **Class SubordinateStarter**

This class is responsible for the starting and stopping of **ClusterMon** subordinate. It mainly has one method:

- **public void startSubordinate()**

Start two threads: *"MulticastReceiver"* and *"MonitoringDataSender"*.

Wait for user to type 'exit', and then exits the program.