Chapter 4 DoIT's Approaches, Architecture, Framework

4.1 Our Approach

First, MMOG's behavior is generally modeled as driven by events. Real players send commands to control the virtual players. The changes made by one virtual player (either Player Character(PC) or Non-Player Character(NPC)) affect what other players view on their monitors. Second, in most cases in client, many codes can be executed in parallel. For example, when sending a command to the server, client can execute an appropriate prediction algorithm to provide smoothing game experience to MMOG players rather than block and wait for the corresponding update from the server. Furthermore, message-oriented Middleware (MOM) technology has the advantage of allowing the platform to decouple the versioning relationship for both the client and the server. Moreover, in the M/M/1 model, MOM-based communication works better for long-lived transactions that require long execution times from the service Provider. Therefore, MOM technology was chosen for building an efficient platform rather than RPC-based architecture.

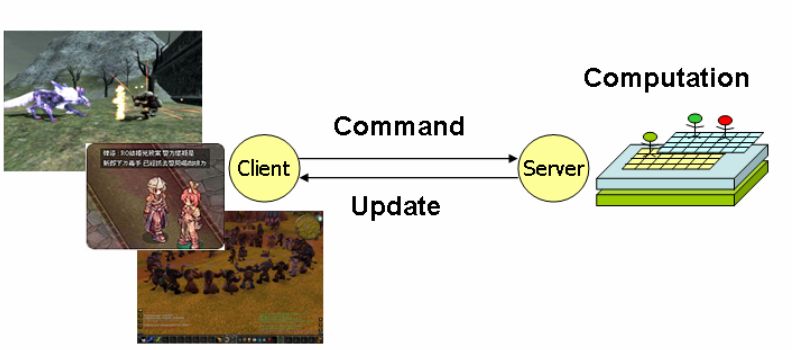


Figure 4-1. Communication model of traditional MMOG

4.2 DoIT Architecture

Based on the survey and analysis in chapter 3, we believe that the above analysis allowed us to say that appropriately designed proxy-based communication architecture has inherently better scalability for our MMOG platform and we thus chose to employ it in the way described in this section.

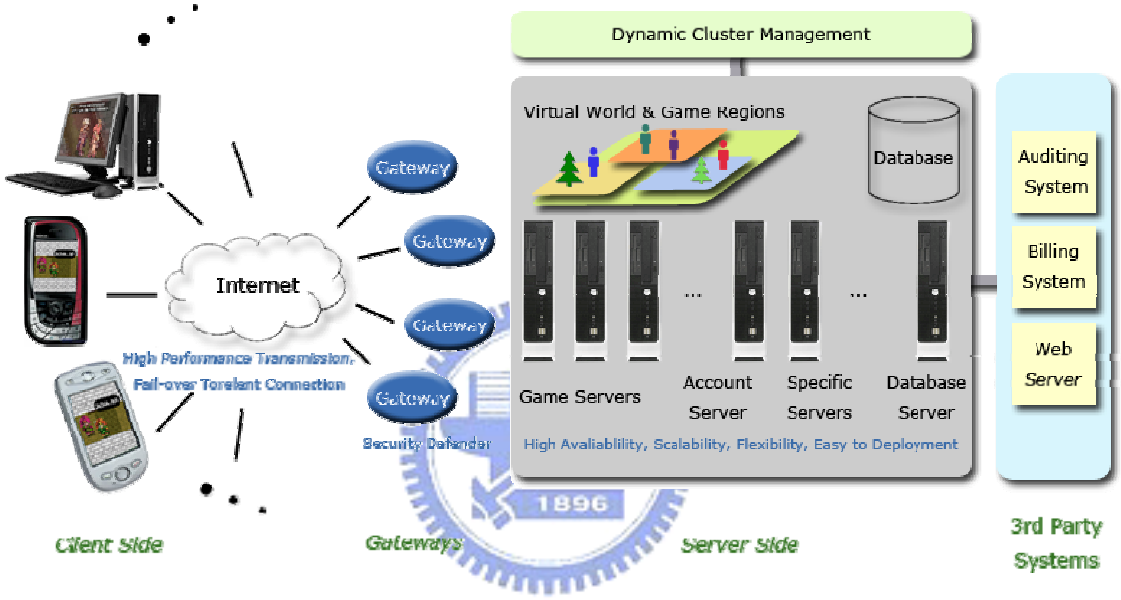


Figure 4-2. DoIT platform 4-tried architecture

We have been engaged since summer 2002 on a middleware building project, which we term the **Distributed-organized Information Terra (DoIT platform)**, to support MMOG. DoIT is implemented by pure Java programming language. Figure 4-2 shows its architecture. The platform can also be categorized as 4-tier middleware architecture. Moreover, it addresses the four ease requirements listed in the previous chapter. This section explains each component in our platform. It contains three major components: *Game Servers*, *Gateways*, and *Coordinator*.

Game Servers are the most important components in our platform. Game server

passively accepts the Gateway connection for receiving messages from client and actively connects to Coordinator for participating in region migration. Note that there is no connection between each server when the system starts up, but one server may dynamically connect to another one on the process of region migration.

A Game Server is capable of computing the game logic and keeping the game states. It receives messages from external (mostly are sent from client), renews the states of virtual world, and then returns the updates to the corresponding clients and components. The game logics are provided by game developers and deployed on Game Servers.

Virtual world initiated in DoIT is divided into multiple regions. Every Game Server may contain zero to many regions and a region can't span across more than one server. A region is the concept of a single virtual world. In the same region, the game logics share the same game context, the same game space. In addition, an avatar in a region may migrate to another region. The details will be described in chapter 4.3 Framework. For the load sharing purpose, a region in a Game Server may migrate to another Game server. The detail will be also discussed further in chapter 4.3 too.

Gateways are the inter-mediator of MMOG platform and the Internet. On the internal side, they actively connect to game servers; on the external side, they passively accept connections from clients. The major responsibility of a gateway is to forward control messages from client to the servers and forward update messages from servers to the clients. However, in DoIT platform, a gateway does not simply forward messages but assembles messages as an internal form.

To achieve high performance and flexibility, the implementation of gateway has many considerations. For example, we provide several kinds of protocol at the external

interface. If a client is not under a firewall, he or she can communicate with server in UDP/IP in order to gain more efficient communication. On the contrast, for clients under a firewall, we provide TCP/IP as underlying protocol in order to pass through the firewall. For a game demands high security, it can also expose SSL as its communication protocol. Another example is that a gateway can also provide hack protection mechanism. The most famous hack technology is known as “acceleration program.” It makes sense to add the facility at a gateway to protect against this kind of vicious programs.

Coordinator is the component that coordinates the process of region migration. It monitors the load of game servers and initiates the process of region migration. In current implementation, coordinator is only a console application. The process of region migration can only be initiated by a human.

Some other essential components for a practical MMOG platform that are not the major components in DoIT are listed and explained here:

A client is a component that player directly interact with. It provides graphics presentation of game environment and interaction interface to interact with the game environment. The client component connects to gateway to interact with a virtual world via certain protocol defined by virtual world provider.

Persistence is also an important part for a MMOG. The reason is that there are many Object-Relation mapping solution in the real world. It is unnecessary to reinvent the wheels. To integrate the persistence mechanism, we can design it as a game logic or a server plug-in.

Update server is the component that updates the latest version of client program before clients’ logging in the server. Web or P2P server are both common ideas to

implement as update server. Update server provides a configuration file to indicate which version is the latest one and describe where the latest version client program locates at. The configuration file is put on a given URL. The client program retrieve the configuration file and compare the version with the configuration file version. If the version is old, get the latest version from the location specified in the configuration.

4.3 Framework

In this section, we illustrate the whole framework and the detail of each component within the framework.

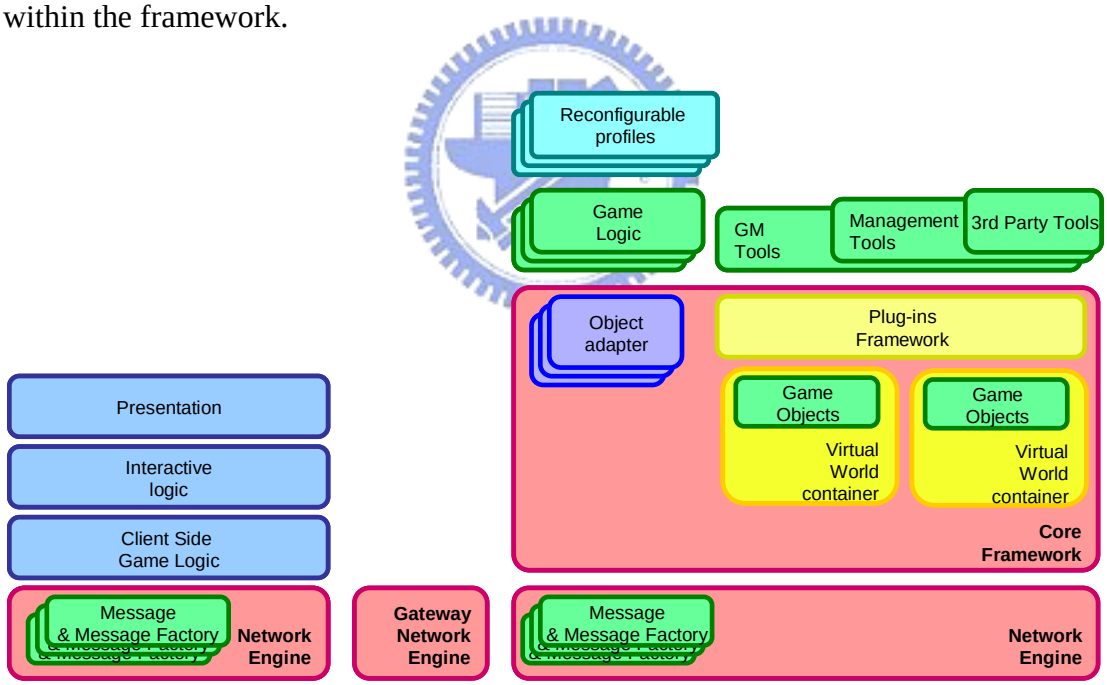


Figure 4-3. System Components of DoIT

In our design, the layered and service-oriented system architecture achieves the functionalities of the system components mentioned above. Fig. 4-3 shows the overall system architecture, which essentially comprises three components: Network Engine, Virtual World Container, and virtual world Game Logic's Object Adapters. The first

provides the DoIT platform with transparent communication; the second, which maintains the user states, provides interfaces for operating all the virtual world's states; the third manages the virtual world logic processing functions. A detailed description of the three will be given in later.

The relationship between the three is shown in Fig. 4-4. The network engine (shown in Fig. 4-5) dispatches messages according to type to registered message handlers. For example, a "move" control message handler sends a message of this type to the virtual world game logic adapter (path 1) which then finds the move-related virtual world game logic for processing the control. Virtual world game logic then obtains the related states from the virtual world container and sends an update request, such as "move player A to new location (x,y)", to the container (paths 2, 3). Finally, the container sends the update to the client through the network engine (path 3).

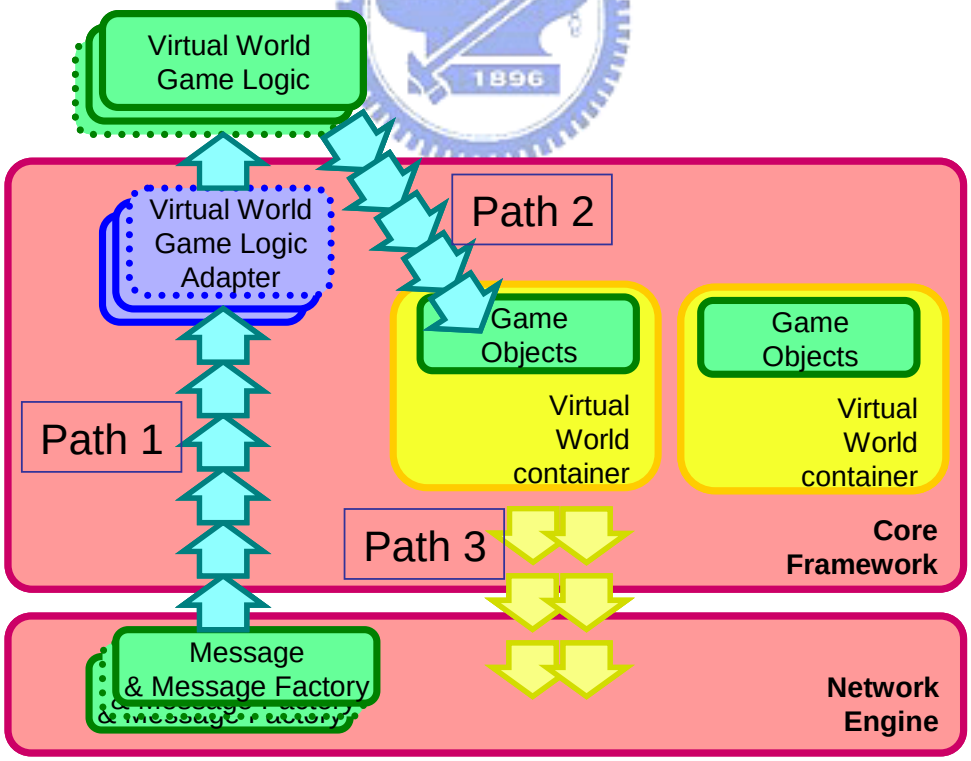


Figure 4-4. Relationship of Server Components

The remainder of section 4.3 is organized as follows: In Section 4.3.1, we discuss the design of DoIT network engine. In Section 4.3.2, we describe the game logic adapter that provide hot-swap features. In Section 4.3.3, we introduce the object container component where virtual world objects is resided. In Section 4.3.4, we illustrate the design and process of object migration in DoIT environment. In Section 4.3.5, we describe the plugins framework that make DoIT extensible. In Section 4.3.6, we set out the design, process and results of the management framework.

4.3.1 Network Engine

NetEngine is a message-oriented lightweight network service. Namely, all data are sent as a concept of message. The Figure **4-5** presents the whole picture of NetEngine library. The NetEngine class is the core façade class. We also start to introduce this library from this class.

NetEngine can be used as client or server. To be a client, we call the connect() method with the specified address to open a connection. This method will return a Channel object. To be a server, we call the listen() method with the specified address. The NetEngine will listen to a specified port. It is necessary to register a listener implementing the interface ChannelListener. When a channel connects in, the NetEngine will call back the channelConnected() of ChannelListener. In the same way, when a channel disconnects, the system will call back the channelDisconnected() of ChannelListener

To send a message, we make use of send() method of Channel. This method should pass in an object implementing Message class. For each type of message, developers should define a class extending the Message class. It must override the encode() and

decode() abstract methods. The system will call back these two methods in order to encode/decode the message data to/from byte buffers. In addition, we recommend that these message classes adhere the JavaBeans [43] convention, that is, if a class has a property named id, it should provides the setId() and getId() methods in this class. It will make it easier to get use of the message objects.

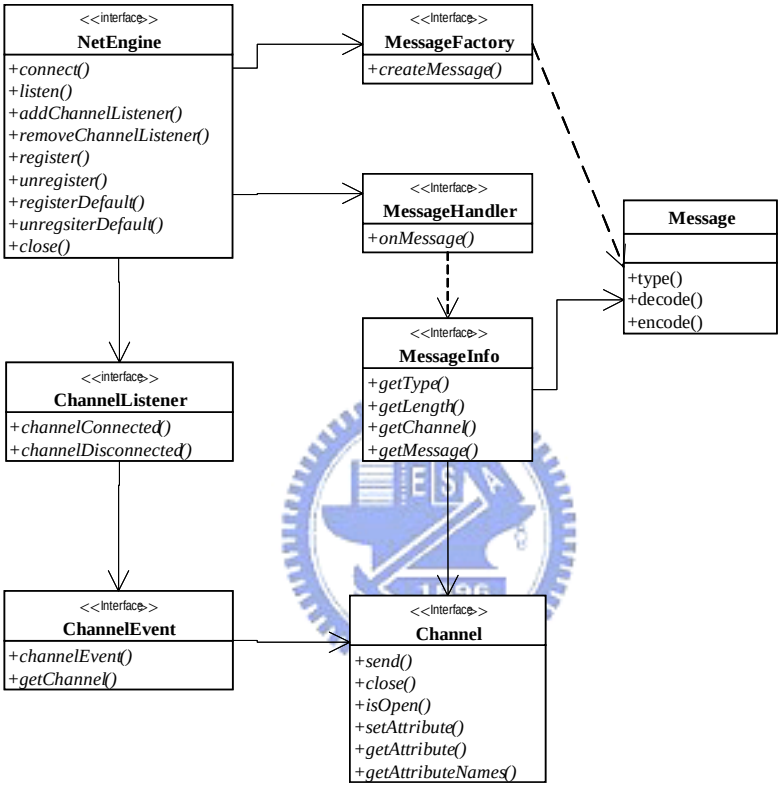


Figure 4-5. Class diagram of NetEngine library

To receive a message, we should first register a message handler and a message factory for a given type of message. The factory should implement the MessageFactory class and overrides the createMessage() method. The handler should implement the MessageHandler class and overrides the onMessage() method. When a message comes, the NetEngine will first analyze the type of message from its header. Then it creates the message from the corresponding message factory and call the decode() method such that it can read the message from binary stream. Subsequently,

it dispatches the message to the corresponding message handler.

Channel is abstraction of connection. We can send a message by send() method. In addition, the channel can be seen as a session to the remote peer. We can store some attributes associating to this session. NetEngine supports two channel establishment methods: active connecting by connect method and passively listening by listen. And when a connection is established, we can get the connection handle Channel by ChannelListener. The following is some example to use NetEngine.

Server-side Example:

```
import java.net.InetSocketAddress;
import mmog.net.*;
import mmog.net.tcp.TCPNetEngine;
//...
NetEngine engine = new TCPNetEngine();
InetSocketAddress sockaddr = new InetSocketAddress(13579);
engine.register(MyMessage.TYPE,
                  new MyMessageFactory(),
                  new MyMessageHandler());
//...
engine.listen(sockaddr);
```

Client-side Example:

```
import java.net.InetSocketAddress;
import mmog.net.*;
import mmog.net.tcp.TCPNetEngine;
//...
NetEngine engine = new TCPNetEngine();
InetSocketAddress sockaddr = new InetSocketAddress("localhost",
13579);
engine.register(MyMessage.TYPE,
                new MyMessageFactory(),
                new MyMessageHandler());
Channel channel = engine.connect(sockaddr);
MyMessage msg = new MyMessage();
channel.send(msg);
```

4.3.2 Game Logic Adapter

Fig. 4-6 shows the design of the virtual world game logic adapter. It behaves like a lightweight real-time CORBA Portable Object Adapte [39]. A control message received by the network engine is asynchronously put into the game logic adapter. According to the control type specified, the adapter searches and dispatches the message to the corresponding virtual world game logic, which then processes the control data in the message and places the update request into the virtual world container. The content developers create the game logic, which can be plugged into or removed from the VW logic adapter at runtime, and this makes any changes in them easy. The appropriate game logics are held by the servant managers of the game logic adapters. Adapters are hierarchically organized with indexing mechanisms to reduce plugged virtual world logic search time. A configurable thread model helps optimization for the concurrent execution of plugged game logic.

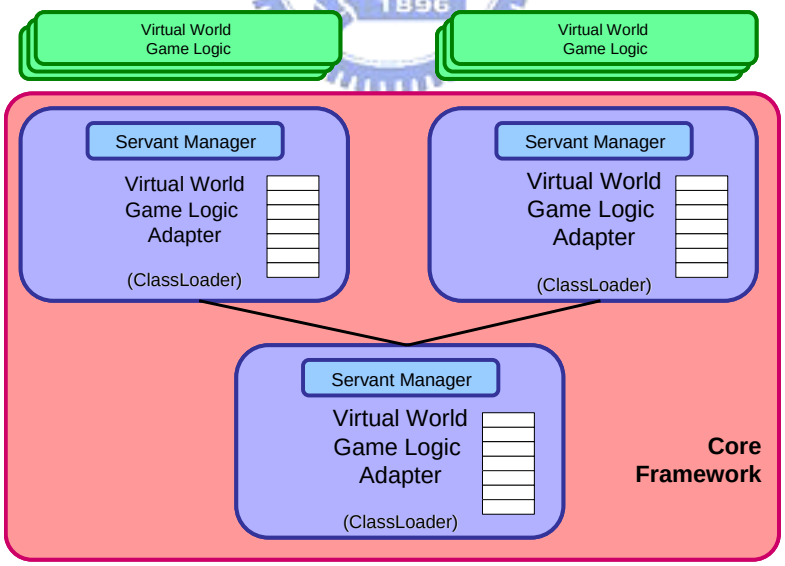


Figure 4-6. Design of Virtual World Game Logic Adapter

Fig. 4-7 shows the class diagram for the virtual world game logic adapter. The ObjectAdapterManager maintains the hierarchically organized Object Adapters and

their states. When control messages are asynchronously put to the OAQueue of the ObjectAdapter, it is the ObjectAdapterManager that finds the game logic adapter with the game logic that corresponds with the control message. The ServantManager provides hot-swap functionalities for the game logics. It is currently implemented by monitoring a specified directory to automatically load and unload the game logics in it. A hash function is used on the dispatch matching to improve performance.

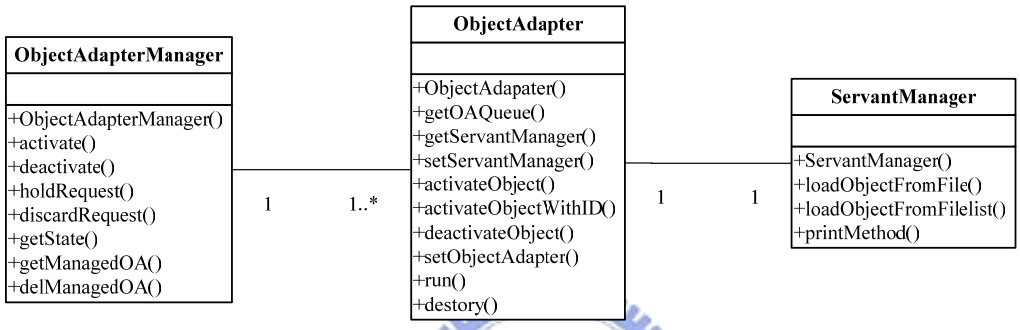


Figure 4-7. Virtual World Logic Adapter Class Diagram

Game logic provides the interface for developing its logic. The ServantManager extends the java.lang.Classloader to dynamically load the game logic classes. This feature allows developers to change the class, for instance by replacing bugged a game logic class by a bug-free class, even when the MMOG application is running. This ‘Hot swap’ feature helps to make our platform more maintainable.

4.3.3 Object Container

In GameMessageHandler, we can receive a message for a given type, process message, and send updates back to the game object channel. But it is only possible to send updates to original game objects. We need more functionality to deal with game objects. In this subsection, we introduce the two classes, GameContext and

GameSpace.

Figure 4-8. Interface of GameContext

GameContext represents the context of the game. The concept is similar to ServletContext in the Servet API [44]. But the game context only associates with the current region. We can store and retrieve attributes to and form the context. Besides, we can also obtain some information of the game. The GameContext instance can be obtained from the init() method of MessageHandler, AvatarChannelListner, and AvatarMigrationListener. The Fig 4-8 is the class information of GameContext.

GameSpace can be seen as a data structure of the virtual worlds. We can *add*, *remove*, *move*, and *find* game objects from the GameSpace. The element type of GameSpace is GameObjChannel. Before adding a GameObjChannel to a GameSpace, we should first associate it with a game object. It is because GameSpace set and retrieve the location information from setX()/getX() and setY()/getY() methods of GameObjChannel. These methods delegate the implementation to the same methods of GameObject associated by the GameObjChannel. Therefore, the GameSpace

prohibit a GameObjChannel without GameObject from added to it. The size of GameSpace is specified in the game descriptor. The GameSpace instance can be obtained from the GameContext.

Figure 4-9. Interface of GameSpace

GameSpaceUtil is a class let us use GameSpace more conveniently. We usually need to send a message to surroundings of a game object. We can call the sendMsgToSurroundings() method. It will get the GameObjChannels from the given range and send the message respectively. Likewise, if we need to receive updates from surroundings, we can call the recvUpdtFromSurroundings() method. It will get the GameObjChannels from the given range, get the updates from status()method of GameObjChannels, and send them to the given channel.

4.3.4 Object Migration

(1) Avatar Migration

Due to the virtual world is divided into multiple regions. An avatar may migrate from one region to another region dynamically. We call it *Avatar Migration*. To implement this feature, the following should be considered. First, what data should be moved to

destination? Secondly, the gateway should be aware of this migration in order to let gateway dispatch messages to new region for the successive messages. Thirdly, we should consider the two conditions that source and target region in the same server and they are in the different servers.

We use *AVAMIG* sent between gateways and servers. *AVAMIG* consists of avatarid indicating which avatar to migrate, source region id, target region id, and avatar data. We must note that the data are provided by developers himself through the API in the form of byte array. Fig 4-10 explains the condition that source and destination regions are in the same game server. The game server migrate the avatar data to destination directly. In the meantime, it sends an *AVAMIG* to notify the gateway the update of avatar location. Fig 4-11 explains the condition that source and destination regions are in different server. The source game server sends an *AVAMIG* to the gateway, and then the gateway forwards this message to the game server where the destination region resides.

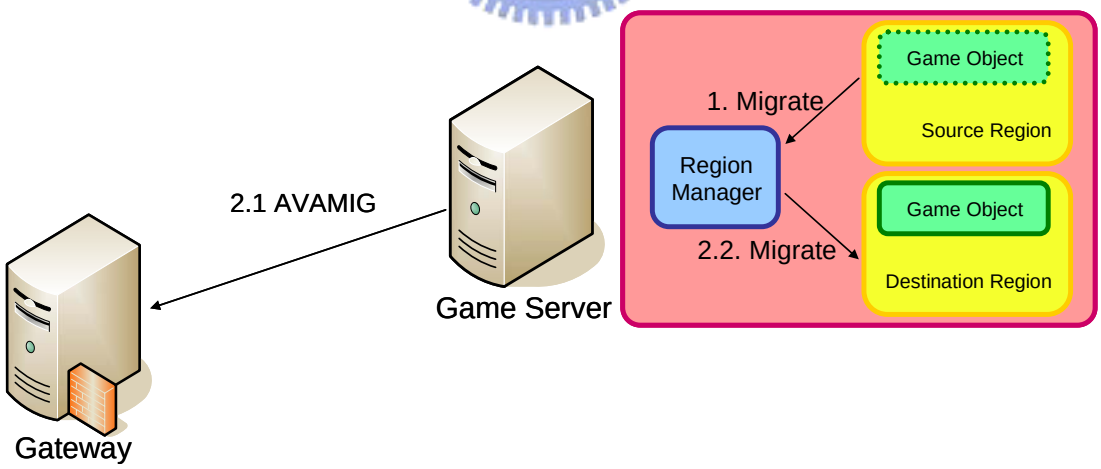


Figure 4-10. Avatar Migration, source region and destination region are in the same server.

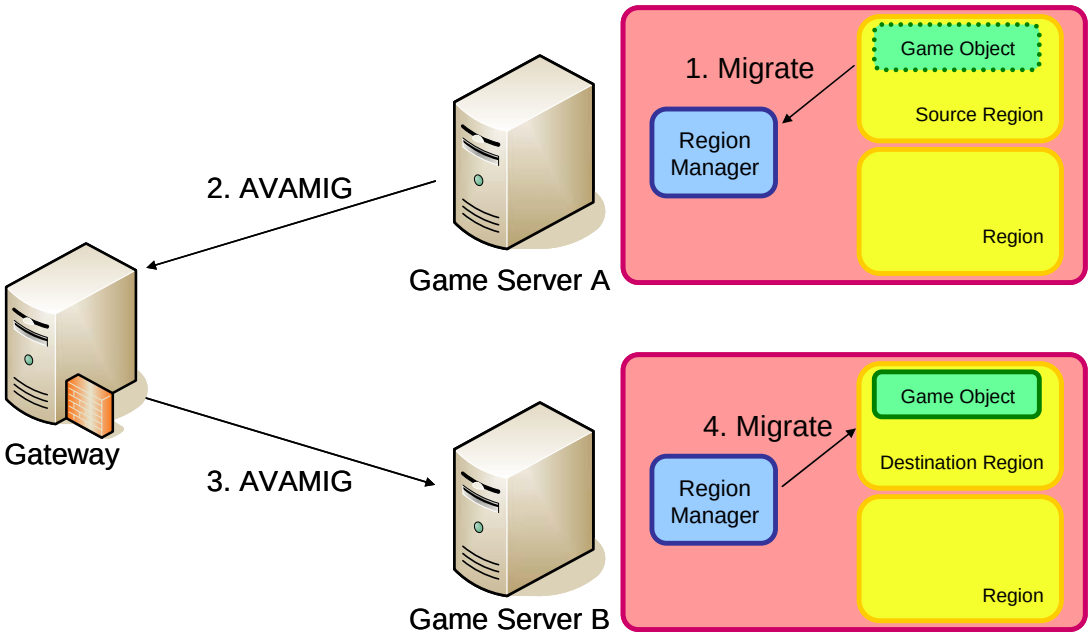


Figure 4-11. Avatar Migration, source region and destination are in different servers.

(2) Region Migration

In order to share the load between game server clusters, a region can migrate from one server to another dynamically. We call it *Region Migration*. Here come the problems: the way to collaborate between several components, the way to send migrating data, the data to migrate, the way to migrate game-specific objects, the way to migrate region scope plugin, and the way for region scope components to bind server scope services.

We use the component coordinator to coordinate the process of region migration. The migrating flow is explained in the Fig 4-12. When a coordinator broadcasts an *RMINIT* message to initiate a region migration, the destination sever opens a port and passively waits for the connection from source region. When the destination server is ready for accepting the socket from the source server, it sends *RMREADY* message to the coordinator. The coordinator forwards this message to source server. As source

server receives this message, it connects to the destination and begins to send migration data as stream. As all data are sent successfully, the destination server sends the *RMCOMPLETE* to notify the completion of migration. Then the coordinator broadcasts this result to all participants. Otherwise, the *RMFAIL* message is sent to notify that some failure occurred when migrating.

However, it is important to know what data are sent when migrating. First, the system data of region should be migrated. Secondly, the game specific data which is created and managed by developers' code should be migrated. Thirdly, the region scope plugin should be migrated. The dependencies are crosscut between these objects. It will be a big problem to recover all the data and dependency in the new server. Fortunately, java serialization mechanism solves all these problems. Region in our system is designed as a *Serializable capsule*. The complex dependency can be serialized and restored to the origin form by mean of java serialization.

The other issue is that the region scope components and plugin may depend on server scope components resided on server. Therefore, before a region's migrating from a server, these region components should be unbind from server, as well as after a region's migrating to a new server, they should bind to the new server. Thus, the base class of region scope component provide bind() and unbind() abstract method. Through these callback methods, the component itself can handle the logic of bind and unbind logic.

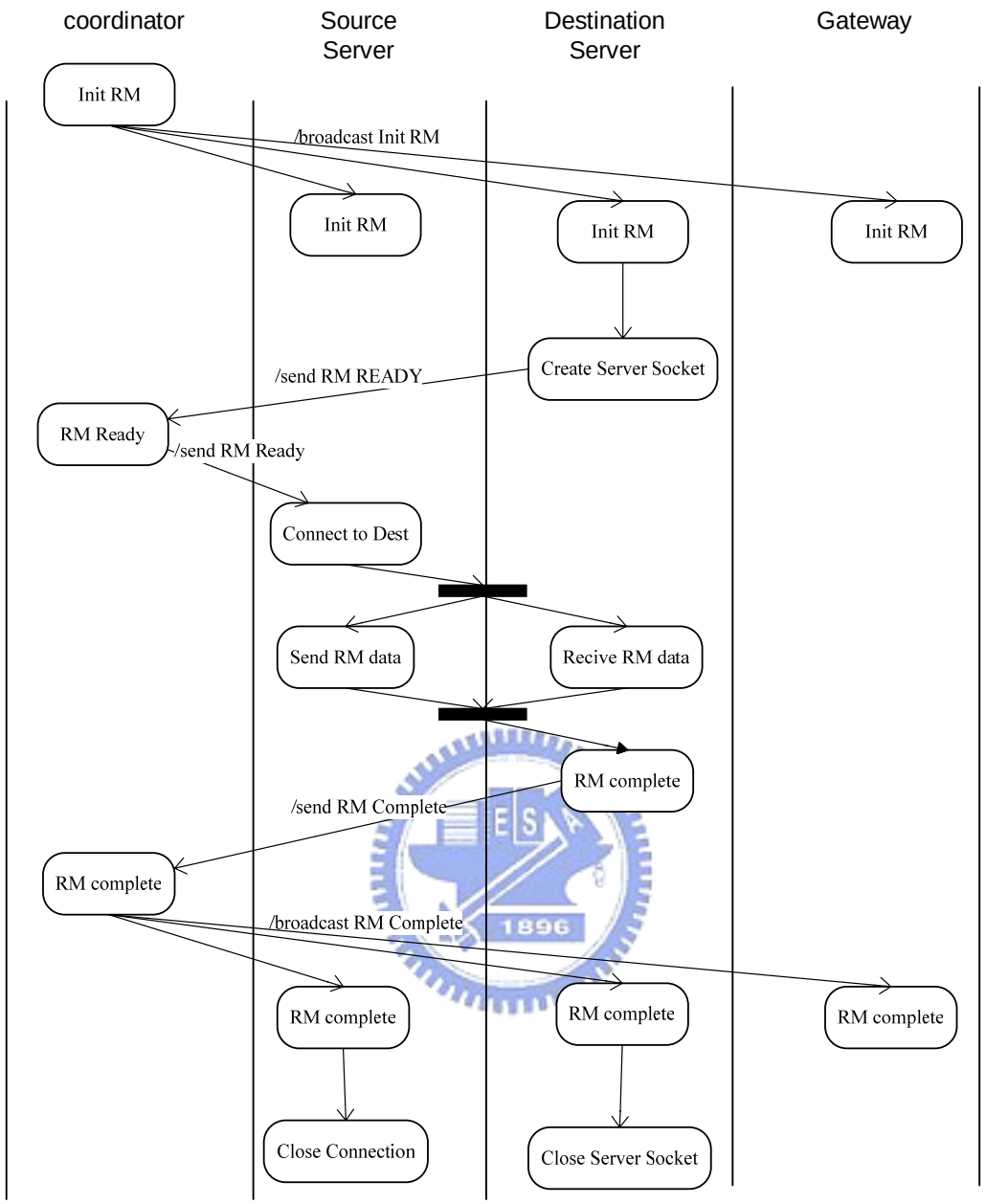


Figure 4-12. Region Migration

4.3.5 Plug-in Framework

MMOG middleware should serve the varying needs of game developers. For example, some middleware provides PC client libraries to interact with servers, while other middleware is aimed at game console or mobile device clients. The point here is that

even where client libraries are provided, they are never enough to meet demands from developers. Ideally, the middleware should provide developers with a variety of layers of API, for example, a server-scope API for using with a system timer, or a thread pool, or a region-scope-level API for creating game objects in the virtual world. It must provide a good plug-ins framework.

In our sever implementation, we build up the server in a component-based architecture. We separate components into server scope components and region components scope. Besides, For the sake of flexibility, we allow developers to provide their own plugin. The Fig 4-13 depicts the architecture. A server scope component, as the name implies, lives with the game server. It can be seen as a service on a server, such as timer, thread pool, and so on. A region scope component lives with the region. A region scope component always tightly couple with a region, such as NPC engine. There should be some dependencies between components. A region scope component can depend on a server scope component, but a server scope component can not depend on a region scope component.

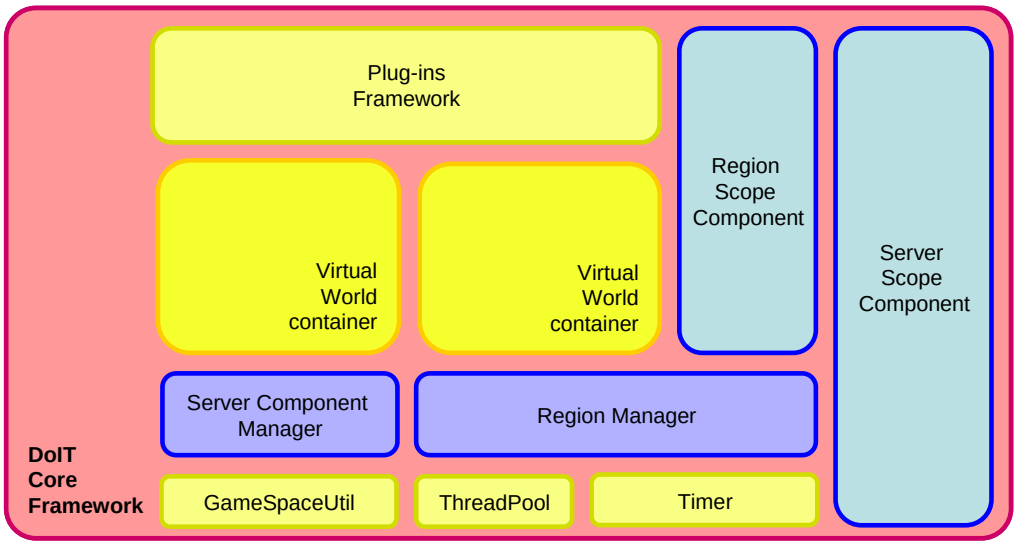


Figure 4-13. Server Component-based Framework

In DOIT platform, we can plug service components to extend the functionality. Plugins are divided server scope plugins and region scope plugins. To write a server scope plugin, we should write a class extending mmog.server.ServerScopePlugin. When the server starts up, the server will call back the init() method of ServerScopePlugin. The system will pass in the ServerContext and plugin initial parameters to this method. In ServerContext, we can get the information of server and look up other server components in this server.

As for designing a region scope plugin, we should provide a class extending mmog.server.RegionScopePlugin. Similarly, when a region is initiated, the server will call back the init() method of RegionScopePlugin. The difference is that there is no RegionContext parameter passed in the init() method of RegionScopePlugin. It will be postponed to bind() method. Region scope plugins live with a region. When a region is bound to a server, all the region scope plugins in this region will have the bind() called. Relatively, when a region is unbound from a server, all the region scope plugins in this region will have the unbound called. It must be noted that init() is only called once throughout the lifecycle of game platform, and however, bind()/unbind() is called once whenever a region is migrated in/out. Moreover, region scope plugins should implement java.io.Serializable. Because region scope plugins migrate accompany with regions, the plugin should be the serialization form. Note to add the transient keyword to the field which is not able to serialize. We can restore this field in the bind() method.

A plugin should be pack as a jar file and put in the “plugins” directory of DOIT platform. For each plugin, we should provide a plugin definition file. In this file, we can define the type classname of plugin, plugin type, plugin initial parameters. The Figure 4-14 is an example of plugin definition file.

```
# MyPlugin.properties
# put plugins config here
mmog.plugin.server.test = hello.MyPlugin
mmog.plugin.server.test.foo = bar
mmog.plugin.server.test.foo2 = bar2
```

Figure 4-14. description of MyPlugin.properties

4.3.6 Management Framework

For flexibility and manageability, an n-tiered level architecture that is similar to JMX architecture [45] is presented. The device has three basic layers: manager, delegation, and agent, the relationship between which is illustrated in Figure 4-15. At the manager layer, a number of remote management APIs is defined. In this way, a remote management application is employed to retrieve/manipulate the information of managed objects by calling the APIs. The delegation layer provides a management component to help the manager deal with all the distributed nodes, such as create, register, and startup, corresponding to the management beans. The ServerFactory records the reference of the different service nodes. The delegation layer also provides a remote connection port that allows remote control. At the agent level, an individual service node packages the resource into the manageable object, and registers the local server information to the delegation server.

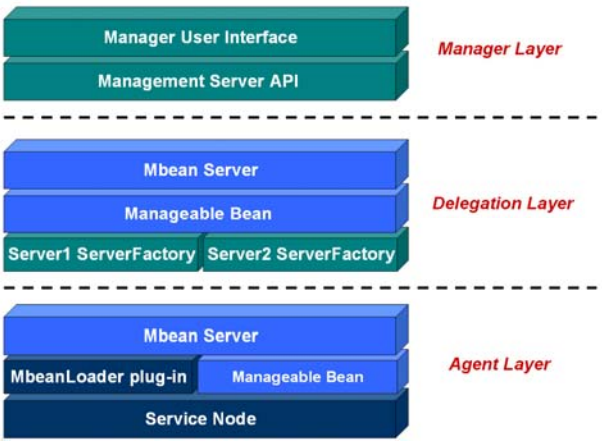


Figure 4-15. n-tiers management architecture

The focus of the development environment is on managing the distributed computing environment, such as the J2EE application server clusters, the MMOG platform, and the grid service node. Their main characteristic is that most of the services have the same functionalities, and are organized by many computers. For example, in a MMOG service, each node handles different geographical coordinates of the virtual world, but each node has the same individual object to be managed, such as, player characters and auction lists. Therefore, to manage the identical object state of each node with the same service component, we use a code generation model that provides a rapid development kit for the purpose.

Here, the implementation of a real management system based on the framework of the MMOG Platform: DoIT platform is described. The MMOG environment has a combination of many servers. Each service node handles different requests from different clients. Because the DoIT platform is built with pure Java language, the notion is expressed as an extension of the JMX framework. The detailed relationship between the layers and components is shown in Fig 4-16. The device is divided into three layers: agent, delegation, and manager. The entire management system work flow and design detail is discussed in the follow subsection.

The purpose is to build a basic management function that reveals how many regions, player objects, non-player objects there are in a single game server. First, the programmer has to declare what information is to be retrieved or manipulated. Figure 4-17 illustrates the partial XML files that the programmer created. The manageable objects are named ServerManager with interfaces: regionnumber, regioncount, AVAcount, NPCcount. For example, the regionnumber interface is intended for setting a region number with an integer type (therefore, authority is setting to write permission) The NPCcount interface is intended to help the manager quantify how

many Non-Player-Character Objects are in the region given by the regionnumber. Figure 4-19 is the interface code of ServerManagerMBean that is generated by the code generation engine.

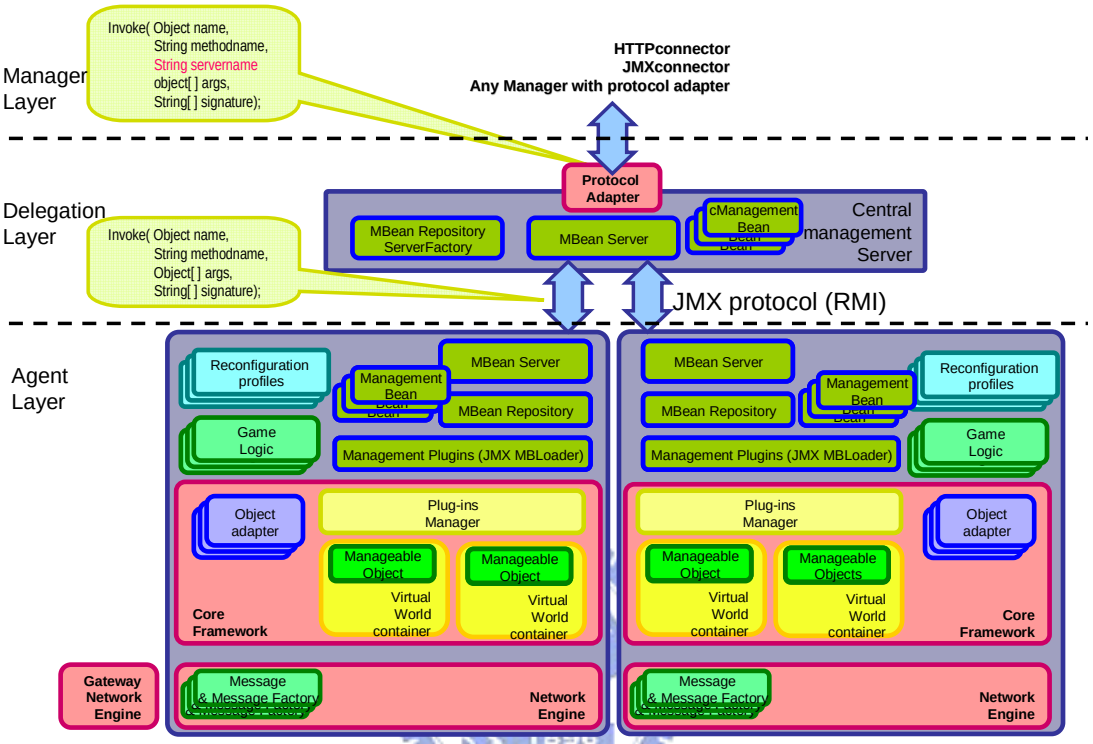
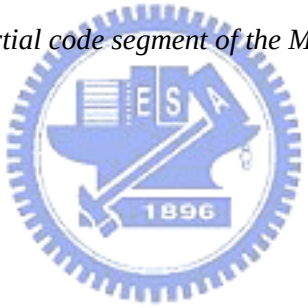


Figure 4-16. The Management System on the DoIT platform

```
  <Classpath>C://DEMO/MMOG_development_system/</Classpath>
- <MBeans>
  - <MBean>
      <MBeanName>ServerManager</MBeanName>
    - <MBeanParams>
      - <MBeanParam>
          <MBeanParamName>regionnumber</MBeanParamName>
          <MBeanParamType>Integer</MBeanParamType>
          <Authority>w</Authority>
        </MBeanParam>
      - <MBeanParam>
          <MBeanParamName>regioncount</MBeanParamName>
          <MBeanParamType>Integer</MBeanParamType>
          <Authority>r</Authority>
        </MBeanParam>
      + <MBeanParam>
      - <MBeanParam>
          <MBeanParamName>AVAcount</MBeanParamName>
          <MBeanParamType>Integer</MBeanParamType>
          <Authority>r</Authority>
        </MBeanParam>
      - <MBeanParam>
          <MBeanParamName>NPCcount</MBeanParamName>
          <MBeanParamType>Integer</MBeanParamType>
          <Authority>r</Authority>
        </MBeanParam>
```

Figure 4-17. Partial code segment of the MBean declaration.



4.3.6.1 Agent Layer

Figure 4-18 shows the architecture and work flow. The management plug-in program, termed MbeanLoader, follows the design of the DoIT platform plug-in module and is operated when the server is startup. With startup, the MBeanLoader creates the Mbeans individually and registers them to the MBserver on the same node and the delegation management server. Figure 4-19 shows an example of a common management API of a ServerManagerMbean. Programmer has to implements the interfaces.

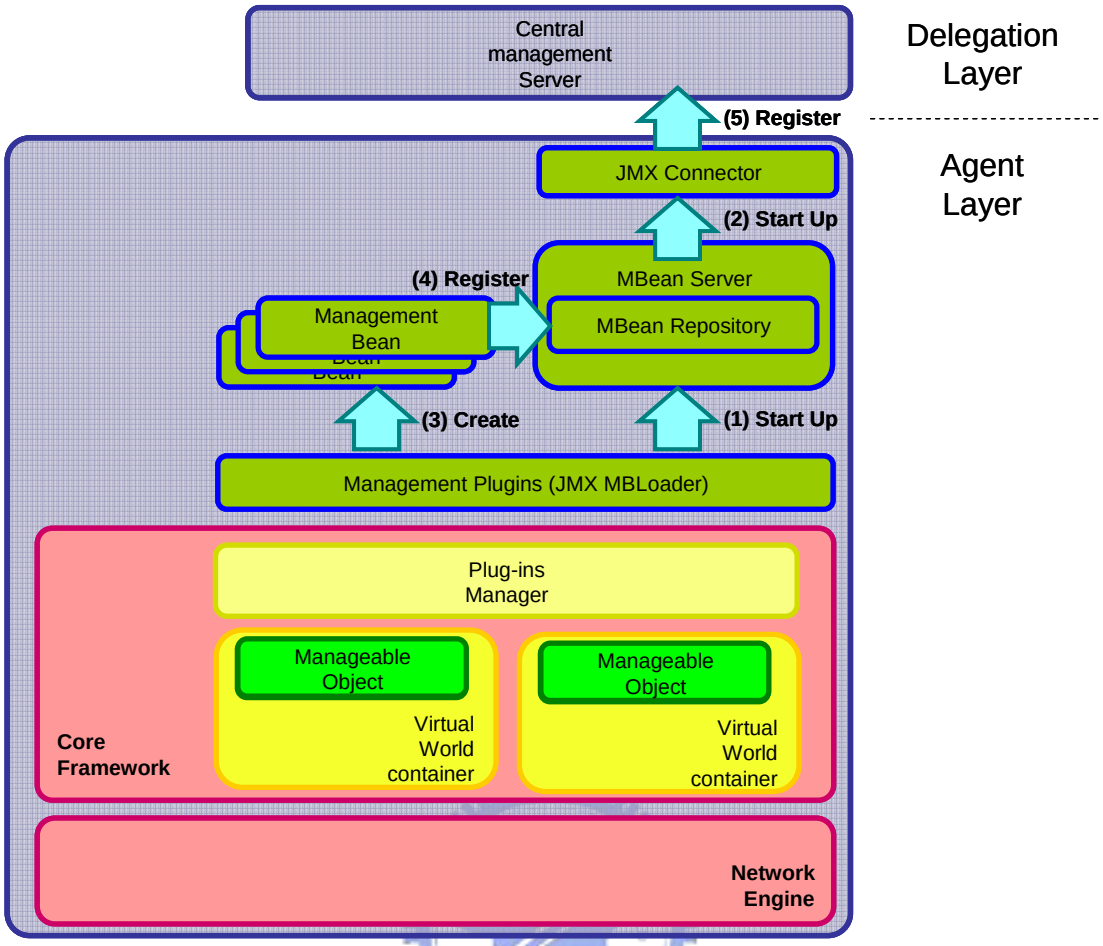


Figure 4-18. Process flow of MBeanLoader in single game server.

```
public interface ServerManagerMBean {
    public Integer getRegionCount(); //get the total number of Regions in single server
    public void setRegionNumber(Integer regionnumber); // set region number
    public String getRegionName(); // get name of region set by manager.
    public void setdebug(Integer debug);
    public void Debug();
    public Integer getAVAcount(); // get current number of Avatar in this region
    public Integer getNPCcount(); // get current number of NPC in this region.
    public Integer getRegionID(); // get current ID of Region
    public void resetRegion(); // reset Region info
    public void register() // register to Management Server
}
```

Figure 4-19. Interface of ServerManagerMBean

4.3.6.2 Delegation Layer

Figure 4-20 shows the process flow of the delegation layer. Here, ManagementServer begins with startup an Mbean server, creates a ServersManagementMbean, and registers it to the Mbean server, which then waits for the remote connection. Two kinds of service are provided, one for the game server and the other for the manager.

The game server can connect to the management server and register a CustomMBean. When this request is received, the manager creates a ServerFactory object according to the register information. When the management server receives the manager's request, it can ascertain the responsible server factory, which can then be activated to execute the manager's request. Depending on the individual server name, the individual ServerFactory object reference can be ascertained. The individual ServerFactory object can select the method for connecting to the game server and invoke it in the ServerManagerMBean. It has a one-to-one relationship to the ServerManager Mbean. The ServerManagementFactory individually shares same-named methods with those of the ServerManager Mbean. Because the ServerManagementFactory plays an intermediary role, it is responsible for connecting to the game server and for invoking the Mbean's method. It therefore shares the same methods with the ServerManager Mbean.

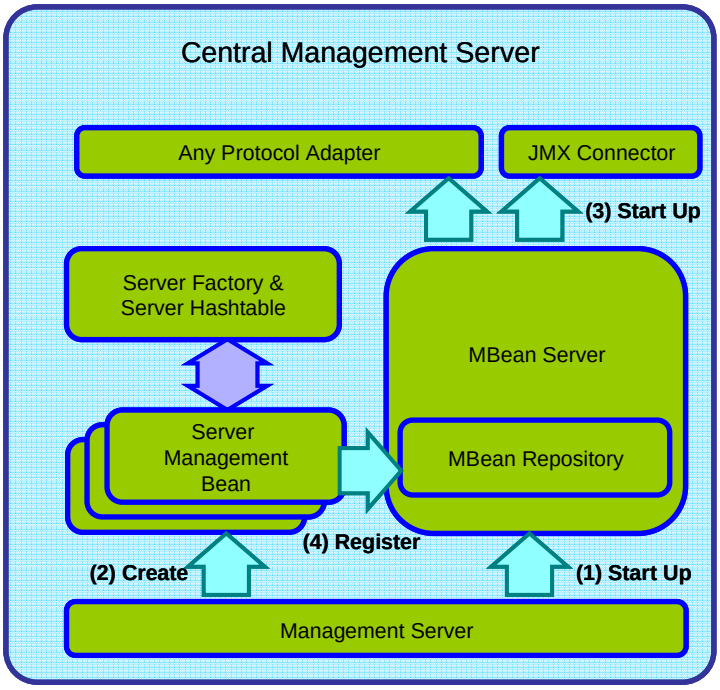


Figure 4-20. Central Management Server Work Flow

4.3.6.3 Manager Layer

At the manager layer, managers accomplish their tasks by writing their own management application. JMX technologies are used to connect to Management Server and throw the JMXConnector or different protocol adapter (such as the HTTP Connector). However, at this layer, a set of the management APIs of the server used for connecting the programmer to the management server is still defined. The ServersManagementFactory class provides some methods for use with the management application. For example, the manager can initialize a ServersManagementFactory object by a given central management server's IP address and the server port number of a management server. This object's method can then be used to connect to the management server, retrieve MBean lists, or get a specified game server's state.

Figure 4-21 shows a simple Management Client that calls a ServerManagerMBean

interface. The manager chooses to view the states of region2 of server1, and the console reports that there are 325 avatars and 254 NPC in the current region.

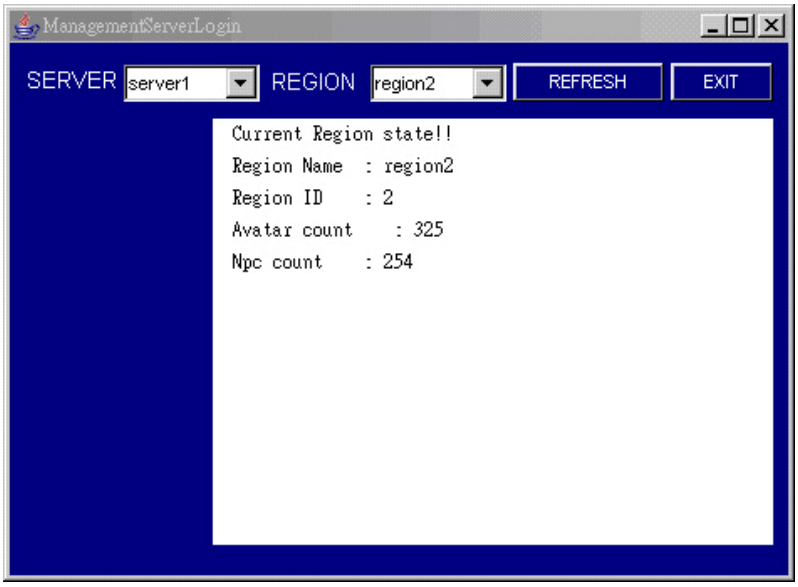


Figure 4-21. A simple management client with access to ServerManagerMBean.