New APIs in Java: Programming Large-Scale Distributed Systems from Scratch

Bing Li

1. Introduction

This book is the first one I write in my life. Hope it were not the last one. Writing a book about programming is out of my imagination since I have no such a plan at least six years ago. When going back to my motherland, China, in 2005, my goal was to design a popular commercial system that could make me rich. However, it is really a tough job in terms of creation, interests, knowledge and even fortune. All of the above issues are considered by me to some extent before I began the long journey except programming because I was confident with my experiences and knowledge. To my surprise, during the procedure I was deeply aware that my programming skills were far from the requirements to accomplish my goal.

You need to know that I started to design my system after I got my PhD degree from Arizona State University. Moreover, before that, I also got the master and the bachelor degrees from Beijing Institute of Technology. All of degrees belonged to the same area, Computer Science. I believe no one would worry about the problems of programming after such a long time academic experiences.

In addition to studying in schools, I had ever worked for some industrial companies. Even worse, some of them were world-famous, such as Lucent Technologies and IBM. It can claim that my programming skills were not improved during the days I worked for them. Fortunately, my working time in them was not long. Thus, it is unreasonable to say those companies could not provide me with a high quality circumstance to strengthen my programming ability.

No matter what was my point of view about my skills, the long-term struggle started nine years ago. Since I focused on the field of the Internet, the first issue raised in my mind was how to program a server. When I was doing research in schools, some mature servers, such as Tomcat [] and JBoss [], were used to achieve the goal of communications. Both of them are supported by the techniques of servlet [], through which a client could send a request remotely to a server via the protocol of HTTP []. At that time, my research aimed at the areas of enterprises and businesses. A centralized waiting server was good enough to satisfy the relevant requirements in the way I called the messaging protocol []. With the approach, two nodes can interact with each other only in the manner of requesting and responding. However, when programming an Internet application, it is possible that a node needs to communicate from one another actively and passively in an arbitrary architecture [] rather than a centralized one []. In addition, the data transferred among them might be heavyweight like streams [] instead of messages [], which are lightweight. To solve those problems, my past experiences did not fit. That means the enterprise-level [] servers I was familiar with must be abandoned if flexible communication methods were needed in my system.

Another reason that I had to give up those servers was that they were too heavy to be deployed on each node even though the communication ways they supported were acceptable in some limited cases. Those servers were designed to provide enterprises with a computing environment to accomplish the mission-critical tasks. Such an environment was different from the heterogeneous Internet in terms of the requirements of high performance, stability and security. To fulfill them, many built-in features [] were implemented as inherent characters, such as state management [], lightweight messaging [], concurrency controls [], business logic dynamic management [] and so forth. However, for me, almost all of them were unnecessary or I need to deal with specific cases in which all of the solutions should be tailored. Anyway, my goal was to design an application that could be manipulated by a user who had no any background on techniques. It was impossible for most of them to own such a high-quality circumstance. That is, many designs of those enterprise platforms were useless in my product, not only because of their built-in modules but also the fact that usually the servers could hardly be customized to meet my requirements upon the configuration interfaces they provided.

In short, it is mandatory to grasp the skills to implement a computing system which is tightly bound to a proprietary environment rather than to borrow a mature one which is tough to be customized if an extremely perfect product is the goal.

On the other hand, I need to emphasize that I was working as a teacher other than a developer and a researcher during the days. It is highly required for beginners to learn advanced programming skills with practical experiences other than concepts on techniques or sample code, which only presents the usage of APIs (application program interfaces). After being back to China, I worked as a faculty in the Software School, Peking University. From a teacher's point of view, I do not think the current textbooks on programming are appropriate to students who intend to implement a complicated system from scratch.

First of all, although sufficient concepts and theories are introduced in those books, students have no ideas how to apply them in a real world system. For example, the class of Operating Systems [] is one classic one in almost all of the departments of computer science. In the class, students learn the techniques of caching []. They must understand that the main memory is small, expensive, volatile and fast whereas the mass storage is large, cheap, persistent and slow. For those respective characters, it is required for a practical system to combine them by utilizing their advantages and minimizing the affects of their disadvantages. That is the goal a cache intends to achieve. Unfortunately, the concept cannot be practiced in the form of any programming languages. Some books might draw figures to present the concepts and algorithms. Some even provide students with pseudo code. However, all of them are not executable. Even though the real world code is given, it is isolated from an entire system. The running is only a simulation at most. Another obvious example is the technique of threading []. Almost all of the books indicate that the importance of concurrency. However, when I asked students about it, I was told that they knew

the technique but they never used it in practice. Such phenomena are frequently found in almost all of fundamental textbooks of computer science.

In addition, although a lot of programming books are available, almost of all of them have no particular focus such that students cannot develop a real world system with the techniques directly. With respect to my experiences, the samples in those books are far from mature ones. In a practical implementation, it is necessary for me to spend much additional effort to transform them to the ones that meet the requirements of a real computing environment. In brief, because of the isolation, the samples look too basic to deal with any specific applications. For example, the technique of threading is critical to the performance of any systems. When introducing programming with threads, the contents mainly include how to invoke, how to transfer parameters, how to synchronize and the approaches to use them sufficiently by pooling. However, since those techniques are not involved with a practical scenario, the sample code for them look like toys to play a game rather than specific tools to solve problems.

Moreover, among those programming tutorials, some of them teach students to program over high level frameworks, such as JEE (Java Edition Enterprise) [] and WCF (Windows Communications Framework). To lower developers' workloads, a large amount of general details are already implemented inside the underlying architecture. For an enterprise system, the generic components consist of mature communication interfaces, state management, transaction management, business logic containers, concurrency management and even security subsystems. Upon the mature frameworks, developers are not required to care about the above details. Instead, they are concentrated on the description of a particular enterprise's requirements. Even though they are not familiar with some important techniques, it is convenient for them to implement a complicated enterprise system. This is the success of software engineering. But, it is a nightmare of a beginner who would like to learn deeply about programming skills. Because of the popularity of the frameworks, some students misunderstand that an enterprise system is even easier to be implemented than an embedded one. In the real world, there are still a lot of systems that do not fit the requirements of an enterprise computing environment, such as the heterogeneous Internet. If grasping programming skills on the stable one, it is tough to solve any issues when the environment is updated.

Finally, design patterns [] seem to be more advanced than low level APIs in JSE. However, I have to indicate that they do not accompany with the real computing environment deeply enough. Therefore, they are far from the practical usage, especially in the heterogeneous large-scale computing environment. At least to me, I even taught the relevant class to students, I could hardly think about them when programming in practice, except some basic ones that binds tightly with the object-oriented concepts. Moreover, for most programmers, the most useful stuffs to them are mature APIs rather than any conceptual models, including design patterns. Unfortunately, although most developers read the books about design patterns, they do not program with them directly. They might have the idea in their minds, but they cannot transform the idea to the real code intuitively. At least, after getting the code patterns in the procedure of

developing my system, the efficiency of implementing such a system becomes much faster than doing that with almost naked JSE APIs and following conceptual patterns.

In short, the changing computing world needs to create new APIs for Java programmers. Now the Internet is pervasive to any corners of the world. Each device should be connected in an efficient form. Although it can be done with the existing techniques in JSE, it takes a lot of effort. It can also be dealt with mature free software, but it is hard to customize them exactly to meet the specific environment. Usually they are fat and heavy in most cases. As a computing environment should be a cluster, i.e., an efficiently connected world, why do not we put them into the APIs which developers are willing to handle?

2. Underlying

2.1 Remote

- **IPPort**

```
package com.greatfree.remote;

import com.greatfree.util.FreeObject;
import com.greatfree.util.Tools;

/*
 * The class consists of all of the values to create an instance of FreeClient. For example, it is the source that is used in RetrievablePool. 09/17/2014, Bing Li
 */

// Created: 09/17/2014, Bing Li
public class IPPort extends FreeObject
{
	// The IP address. 09/17/2014, Bing Li
	private String ip;
	// The port number. 09/17/2014, Bing Li
	private int port;

	/*
	 * Initialize the instance of the class. 09/17/2014, Bing Li
	 */
	public IPPort(String ip, int port)
	{
		// Create and set the key of the class. The key represents all of the FreeClients that connect to the same remote end. 09/17/2014, Bing Li
		super(Tools.getKeyOfFreeClient(ip, port));
		this.ip = ip;
		this.port = port;
	}

	/*
	 * Expose the IP address. 09/17/2014, Bing Li
	 */
	public String getIP()
	{
		return this.ip;
	}

	/*
	 * Expose the port number. 09/17/2014, Bing Li
	 */
	public int getPort()
	{
		return this.port;
	}
}
```

- **FreeClient**

```
package com.greatfree.remote;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.concurrent.locks.ReentrantLock;

import com.greatfree.multicast.ServerMessage;
import com.greatfree.testing.message.InitReadNotification;
import com.greatfree.util.FreeObject;
import com.greatfree.util.Tools;

/*
 * This is a TCP client that encloses some details of TCP APIs such that it is convenient for developers to interact with remote servers. Moreover, the client is upgraded to fit the caching management. 08/10/2014, Bing Li
 */

// Created: 08/10/2014, Bing Li
public class FreeClient extends FreeObject
{
	// A client socket that connects to the remote server socket at a remote end. In the system, the remote end is a node in a distributed cluster. 08/24/2014, Bing Li
	private Socket socket;
	// The IP address of the remote end. 08/24/2014, Bing Li
	private String serverAddress;
	// The port number of the remote end. 08/24/2014, Bing Li
	private int serverPort;
	// The output stream that sends data to the remote end. 08/24/2014, Bing Li
	private ObjectOutputStream out;
	// The input stream that receives data from the remote end. 08/24/2014, Bing Li
	private ObjectInputStream in;
	// The lock that keeps sending and receiving operations through the client atomic. The lock is used by RemoteReader. 08/24/2014, Bing Li
	private ReentrantLock lock;

	/*
	 * Initialize the client. 08/24/2014, Bing Li
	 */
	public FreeClient(String serverAddress, int serverPort) throws IOException
	{
		// The key of the FreeClient is created upon the IP and the port of the remote end. 08/24/2014, Bing Li
		super(Tools.getKeyOfFreeClient(serverAddress, serverPort));
		this.serverAddress = serverAddress;
		this.serverPort = serverPort;
		this.socket = new Socket(this.serverAddress, this.serverPort);
		this.lock = new ReentrantLock();

		this.out = new ObjectOutputStream(this.socket.getOutputStream());
	}

	/*
	 * Dispose the associated resources. 08/24/2014, Bing Li
	 */
	public void dispose()
	{
		try
		{
			if (this.out != null)
			{
				this.out.close();
			}
```

```
        if (this.in != null)
        {
          this.in.close();
        }
        if (this.socket != null)
        {
          this.socket.close();
        }
      }
      catch (IOException e)
      {
        e.printStackTrace();
      }
    }

    /*
     * Expose the lock for outside invocation to keep the sending and receiving operations atomic. The
lock is used by RemoteReader. 08/24/2014, Bing Li
     */
    public ReentrantLock getLock()
    {
      return this.lock;
    }

    /*
     * Expose the connected server IP address. 11/04/2014, Bing Li
     */
    public String getServerAddress()
    {
      return this.serverAddress;
    }

    /*
     * Expose the connected server port number. 11/04/2014, Bing Li
     */
    public int getServerPort()
    {
      return this.serverPort;
    }

    /*
     * Initialize the ObjectInputStream by sending a notification to the remote server. 11/04/2014, Bing Li
     */
    public void initRead(String nodeKey) throws IOException
    {
      this.send(new InitReadNotification(nodeKey));
    }

    /*
     * Initialize the ObjectInputStream after getting a feedback from the server. That means the server has
already initialized the corresponding ObjectOutputStream. 11/04/2014, Bing Li
     */
    public void setInputStream() throws IOException
    {
      this.in = new ObjectInputStream(this.socket.getInputStream());
    }

    /*
     * Send a message to the remote end. The method is required to be synchronized to avoid potential
memory leak in OutputObjectStream. 08/24/2014, Bing Li
     */
    public synchronized void send(ServerMessage event) throws IOException
    {
      // Send the message to the remote end. 09/17/2014, Bing Li
      this.out.writeObject(event);
      // It is required to invoke the below methods to avoid the memory leak. 09/17/2014, Bing Li
      this.out.flush();
      this.out.reset();
```

```
  }

  /*
   * Send a request to the remote end and wait until a response is received. The same as the method of
Send(), the synchronized descriptor intends to avoid potential memory leak in OutputObjectStream.
08/24/2014, Bing Li
   */
  public synchronized ServerMessage request(ServerMessage request) throws IOException,
ClassNotFoundException
  {
    // Send the message to the remote end. 09/17/2014, Bing Li
    this.out.writeObject(request);
    // It is required to invoke the below methods to avoid the memory leak. 09/17/2014, Bing Li
    this.out.flush();
    this.out.reset();
    // Wait for the response from the remote end. 09/17/2014, Bing Li
    return (ServerMessage)this.in.readObject();
  }
}
```

- **FreeClientCreator**

```
package com.greatfree.remote;

import java.io.IOException;

import com.greatfree.reuse.Creatable;

/*
 * The class contains the method to create an instance of FreeClient by its IP address and port number.
It extends the interface of ResourceCreated and it is used as the resource creator in the
RetrievablePool. 09/17/20214
 */

// Created: 09/17/2014, Bing Li
public class FreeClientCreator implements Creatable<IPPort, FreeClient>
{
   // The method to create instance of FreeClient by its IP address and the port number. 09/17/2024,
   @Override
   public FreeClient createResourceInstance(IPPort source) throws IOException
   {
      // Invoke the constructor of FreeClient. 09/17/2014, Bing Li
      return new FreeClient(source.getIP(), source.getPort());
   }
}
```

- **FreeClientDisposer**

```
package com.greatfree.remote;

import com.greatfree.reuse.Disposable;

/*
 * The class implements the interface of Disposable and aims to invoke the dispose method of an instance of FreeClient to collect the resource. It is used a resource disposer in RetrievablePool. 09/17/2014, Bing Li
 */

// Created: 09/17/2014, Bing Li
public class FreeClientDisposer implements Disposable<FreeClient>
{
   // Dispose a FreeClient by invoking its dispose method. 09/17/2014, Bing Li
   @Override
   public void dispose(FreeClient t)
   {
      t.dispose();
   }
}
```

- **FreeClientPool**

```
package com.greatfree.remote;

import java.io.IOException;
import java.util.Set;

import com.greatfree.multicast.ServerMessage;
import com.greatfree.reuse.RetrievablePool;
import com.greatfree.util.UtilConfig;

/*
 * The pool, RetrievablePool, is mainly used for the resource of FreeClient. Some problems exist when instances of FreeClient are exposed outside since they might be disposed inside in the pool.
 * It is a better solution to wrap the instances of FreeClient and the management on them. The stuffs should be invisible to outside. For that, a new pool, FreeClientPool, is proposed. 11/19/2014, Bing Li
 */

public class FreeClientPool
{
  // Declare an instance of Retrievable to hide instances of FreeClient from outside. 11/19/2014, Bing Li
  private RetrievablePool<IPPort, FreeClient, FreeClientCreator, FreeClientDisposer> pool;

  /*
   * Initialize the pool. 11/19/2014, Bing Li
   */
  public FreeClientPool(int poolSize)
  {
    this.pool = new RetrievablePool<IPPort, FreeClient, FreeClientCreator, FreeClientDisposer>(poolSize, new FreeClientCreator(), new FreeClientDisposer());
  }

  /*
   * Dispose the resource, i.e., the pool in the case. 11/20/2014, Bing Li
   */
  public void dispose() throws IOException
  {
    this.pool.shutdown();
  }

  /*
   * Set the idle checking. 11/20/2014, Bing Li
   */
  public void setIdleChecker(long delay, long period, long maxIdleTime)
  {
    this.pool.setIdleChecker(delay, period, maxIdleTime);
  }

  /*
   * Send a message to an IP/port which is enclosed in the instance of IPPort. The method wraps three steps as below. Thus, those stuffs are invisible to users such that it avoids possible conflicts to share instances of FreeClient. 11/20/2014, Bing Li
   *
   *      Getting an instance of FreeClient
   *
   *      Sending the message
   *
   *      Collecting the instance of FreeClient.
   */
  public void send(IPPort ipPort, ServerMessage msg) throws IOException
  {
    // Get an instance of FreeClient by the instance of IPPort. 11/20/2014, Bing Li
    FreeClient client = this.pool.get(ipPort);
    // Check whether the instance of FreeClient is valid. 11/20/2014, Bing Li
    if (client != UtilConfig.NO_CLIENT)
    {
```

```
        // Send the message. 11/20/2014, Bing Li
        client.send(msg);
        // Collect the instance of FreeClient. 11/20/2014, Bing Li
        this.pool.collect(client);
      }
    }

    /*
     * Send a message to an IP/port. It also ensures initializing, sending and collecting are invisible to
users. 11/20/2014, Bing Li
     */
    public void send(String ip, int port, ServerMessage msg) throws IOException
    {
      // Get an instance of FreeClient by the IP/port. 11/20/2014, Bing Li
      FreeClient client = this.pool.get(new IPPort(ip, port));
      // Check whether the instance of FreeClient is valid. 11/20/2014, Bing Li
      if (client != UtilConfig.NO_CLIENT)
      {
        // Send the message. 11/20/2014, Bing Li
        client.send(msg);
        // Collect the instance of FreeClient. 11/20/2014, Bing Li
        this.pool.collect(client);
      }
    }

    /*
     * Send a message to a remote node by its key. It also ensures initializing, sending and collecting are
invisible to users. 11/20/2014, Bing Li
     */
    public void send(String clientKey, ServerMessage msg) throws IOException
    {
      // Get an instance of FreeClient by the client key. 11/20/2014, Bing Li
      FreeClient client = this.pool.get(clientKey);
      // Check whether the instance of FreeClient is valid. 11/20/2014, Bing Li
      if (client != UtilConfig.NO_CLIENT)
      {
        // Send the message. 11/20/2014, Bing Li
        client.send(msg);
        // Collect the instance of FreeClient. 11/20/2014, Bing Li
        this.pool.collect(client);
      }
    }

    /*
     * Send a request message to a remote node in the form of IPPort and then wait until its corresponding
response is received. It also ensures initializing, sending and collecting are invisible to users.
11/20/2014, Bing Li
     */
    public ServerMessage request(IPPort ipPort, ServerMessage req) throws IOException,
ClassNotFoundException
    {
      // Get an instance of FreeClient by the instance of IPPort. 11/20/2014, Bing Li
      FreeClient client = this.pool.get(ipPort);
      // Check whether the instance of FreeClient is valid. 11/20/2014, Bing Li
      if (client != UtilConfig.NO_CLIENT)
      {
        // Send the message and wait until the corresponding response is received. 11/20/2014, Bing Li
        ServerMessage res = client.request(req);
        // Collect the instance of FreeClient. 11/20/2014, Bing Li
        this.pool.collect(client);
        // Return the response. 11/20/2014, Bing Li
        return res;
      }
      // If the instance of FreeClient is not valid, return null. 11/20/2014, Bing Li
      return null;
    }

    /*
```

```
    * Send a request message to a remote node in the form of IP/port and then wait until its
corresponding response is received. It also ensures initializing, sending and collecting are invisible to
users. 11/20/2014, Bing Li
    */
   public ServerMessage request(String ip, int port, ServerMessage req) throws IOException,
ClassNotFoundException
   {
      // Get an instance of FreeClient by the instance of IPPort. 11/20/2014, Bing Li
      FreeClient client = this.pool.get(new IPPort(ip, port));
      // Check whether the instance of FreeClient is valid. 11/20/2014, Bing Li
      if (client != UtilConfig.NO_CLIENT)
      {
         // Send the message and wait until the corresponding response is received. 11/20/2014, Bing Li
         ServerMessage res = client.request(req);
         // Collect the instance of FreeClient. 11/20/2014, Bing Li
         this.pool.collect(client);
         // Return the response. 11/20/2014, Bing Li
         return res;
      }
      // If the instance of FreeClient is not valid, return null. 11/20/2014, Bing Li
      return null;
   }

   /*
    * Send a request message to a remote node by its client key and then wait until its corresponding
response is received. It also ensures initializing, sending and collecting are invisible to users.
11/20/2014, Bing Li
    */
   public ServerMessage request(String clientKey, ServerMessage req) throws IOException,
ClassNotFoundException
   {
      // Get an instance of FreeClient by the client key. 11/20/2014, Bing Li
      FreeClient client = this.pool.get(clientKey);
      // Check whether the instance of FreeClient is valid. 11/20/2014, Bing Li
      if (client != UtilConfig.NO_CLIENT)
      {
         // Send the message and wait until the corresponding response is received. 11/20/2014, Bing Li
         ServerMessage res = client.request(req);
         // Collect the instance of FreeClient. 11/20/2014, Bing Li
         this.pool.collect(client);
         // Return the response. 11/20/2014, Bing Li
         return res;
      }
      // If the instance of FreeClient is not valid, return null. 11/20/2014, Bing Li
      return null;
   }

   /*
    * Check whether the client is existed. 11/20/2014, Bing Li
    */
   public boolean isClientExisted(String ip, int port)
   {
      // Check whether the source to create the instance of FreeClient is existed. 11/20/2014, Bing Li
      return this.pool.isSourceExisted(new IPPort(ip, port));
   }

   /*
    * Get the IP of a client key. 11/20/2014, Bing Li
    */
   public String getIP(String clientKey)
   {
      return this.pool.getSource(clientKey).getIP();
   }

   /*
    * Get the client count in the pool. 11/20/2014, Bing Li
    */
   public int getClientSize()
```

```
    {
        return this.pool.getSourceSize();
    }

    /*
     * Get the instance of IPPort by the client key. 11/20/2014, Bing Li
     */
    public IPPort getIPPort(String clientKey)
    {
        return this.pool.getSource(clientKey);
    }

    /*
     * Get the client keys in the pool. 11/20/2014, Bing Li
     */
    public Set<String> getNodeKeys()
    {
        return this.pool.getAllObjectKeys();
    }

    /*
     * Get the count of the clients which are working. 11/20/2014, Bing Li
     */
    public int getBusyClientCount()
    {
        return this.pool.getBusyResourceSize();
    }

    /*
     * Get the count of the clients which are idle. 11/20/2014, Bing Li
     */
    public int getIdleClientCount()
    {
        return this.pool.getIdleResourceSize();
    }

    /*
     * Remote a client by its client key. 11/20/2014, Bing Li
     */
    public void removeClient(String clientKey) throws IOException
    {
        this.pool.removeResource(clientKey);
    }
}
```

- **RemoteReader**

```
package com.greatfree.remote;

import java.io.IOException;

import com.greatfree.exceptions.RemoteReadException;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.reuse.FreeReaderPool;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.MessageConfig;

/*
 * The class is responsible for sending a request to the remote end, waiting for the response and
returning it to the local end which sends the request. 09/21/2014, Bing Li
 */

// Created: 09/21/2014, Bing Li
public class RemoteReader
{
  // The pool to manage the instances of FreeClient. 11/06/2014, Bing Li
  private FreeReaderPool clientPool;

  /*
   * Initialize. 11/06/2014, Bing Li
   */
  private RemoteReader()
  {
  }

  // A singleton is defined. 11/06/2014, Bing Li
  private static RemoteReader instance = new RemoteReader();

  public static RemoteReader REMOTE()
  {
    if (instance == null)
    {
      instance = new RemoteReader();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Shutdown the reader. 11/07/2014, Bing Li
   */
  public void shutdown()
  {
    this.clientPool.shutdown();
  }

  /*
   * Initialize the reader. 11/07/2014, Bing Li
   */
  public void init(int poolSize)
  {
    this.clientPool = new FreeReaderPool(poolSize);
  }

  /*
   * Notify that the corresponding ObjectOutputStream is initialized. 11/07/2014, Bing Li
   */
  public void notifyOutStreamDone()
```

```
  {
    this.clientPool.notifyOutStreamDone();
  }

  /*
   * Send a request to the remote node at the IP address and the port number and wait until the
response is received. 09/21/2014, Bing Li
   *
   * The parameters:
   *
   *      String nodeKey: the local client key that is unique to differentiate it from others client on the
remote server;
   *
   *      String ip: the IP address of the remote node;
   *
   *      int port: the port number of the remote node;
   *
   *      ServerMessage request: the request sent to the remote node;
   *
   * The return value:
   *
   *      ServerMessage: the response to be received after the request is sent.
   *
   */
  public ServerMessage read(String nodeKey, String ip, int port, ServerMessage request) throws
RemoteReadException, IOException, ClassNotFoundException
  {
    // Get the instance of FreeClient by the IP address and the port number. 09/21/2014, Bing Li
    FreeClient client = this.clientPool.get(nodeKey, new IPPort(ip, port));
    if (client != ServerConfig.NO_CLIENT)
    {
      // The lock is hold by each particular instance of FreeClient. It guarantees that the sending,
receiving and collecting become an atomic operation, which can never be interrupted by other
concurrent operations of the client. 09/21/2014, Bing Li
      client.getLock().lock();
      try
      {
        // Send a request, wait for the response and return it. 09/21/2014, Bing Li
        return client.request(request);
      }
      finally
      {
        // Collect the client. 09/21/2014, Bing Li
        this.clientPool.collect(client);
        client.getLock().unlock();
      }
    }
    return MessageConfig.NO_MESSAGE;
  }

  /*
   * Send a request to the remote node at the client key and wait until the response is received.
09/21/2014, Bing Li
   *
   * The parameters:
   *
   *      String nodeKey: the local client key that is unique to differentiate it from others client on the
remote server;
   *
   *      String clientKey: the client key that represents the remote node, upon which the IP address and
the port can be retrieved from the client pool;
   *
   *      ServerMessage request: the request sent to the remote node;
   *
   * The return value:
   *
   *      ServerMessage: the response to be received after the request is sent.
   *
```

```
   */
   public ServerMessage read(String nodeKey, String clientKey, ServerMessage request) throws
RemoteReadException, IOException, ClassNotFoundException
   {
      // Get the instance of FreeClient by the client key. 09/21/2014, Bing Li
      FreeClient client = clientPool.get(nodeKey, clientKey);
      if (client != ServerConfig.NO_CLIENT)
      {
         // The lock is hold by each particular instance of FreeClient. It guarantees that the sending,
receiving and collecting become an atomic operation, which can never be interrupted by other
concurrent operations of the client. 09/21/2014, Bing Li
         client.getLock().lock();
         try
         {
            // Send a request, wait for the response and return it. 09/21/2014, Bing Li
            return client.request(request);
         }
         finally
         {
            // Collect the client. 09/21/2014, Bing Li
            clientPool.collect(client);
            client.getLock().unlock();
         }
      }
      return MessageConfig.NO_MESSAGE;
   }
}
```

- **SyncRemoteEventer**

```
package com.greatfree.remote;

import java.io.IOException;

import com.greatfree.multicast.ServerMessage;

/*
 * The eventer sends notifications to remote servers in a synchronous manner without waiting for responses. The sending method, notify(), are blocking. 11/05/2014, Bing Li
 */

// Created: 11/05/2014, Bing Li
public class SyncRemoteEventer<Notification extends ServerMessage>
{
	// The pool for FreeClient is needed to issue relevant clients to send notifications. 11/05/2014, Bing Li
	private FreeClientPool clientPool;

	/*
	 * Usually, the FreeClient pool is shared by multiple eventers. So, it is assigned to the eventer when initializing the eventer. 11/05/2014, Bing Li
	 */
	public SyncRemoteEventer(FreeClientPool clientPool)
	{
		this.clientPool = clientPool;
	}

	/*
	 * The resource consumed by the eventer is the FreeClient pool. Because it is shared, it should not be disposed here. Just leave the interface. 11/05/2014, Bing Li
	 */
	public void dispose()
	{
	}

	/*
	 * Send the notification to the remote server. Since no asynchronous mechanisms are available, the method is blocking. 11/05/2014, Bing Li
	 */
	public void notify(String ip, int port, Notification message) throws IOException, InterruptedException
	{
		// Send the notification. 11/23/2014, Bing Li
		this.clientPool.send(new IPPort(ip, port), message);
	}
}
```

- **IPNotification**

```
package com.greatfree.remote;

import com.greatfree.multicast.ServerMessage;

/*
 * This is an object to contain the instance of IPPort and the message to be sent to it. It is used by the
class of Eventer in most time. 11/20/2014, Bing Li
 */

// Created: 11/20/2014, Bing Li
public class IPNotification<Notification extends ServerMessage>
{
  // The message to be sent to the IP/port included in the object. 11/20/2014, Bing Li
  private Notification notification;
  // The IP/port, to which the notification to be sent. 11/20/2014, Bing Li
  private IPPort ipPort;

  /*
   * Initialize. 11/20/2014, Bing Li
   */
  public IPNotification(IPPort ipPort, Notification notification)
  {
    this.ipPort = ipPort;
    this.notification = notification;
  }

  /*
   * Expose the notification. 11/20/2014, Bing Li
   */
  public Notification getNotification()
  {
    return this.notification;
  }

  /*
   * Expose the instance of IPPort. 11/20/2014, Bing Li
   */
  public IPPort getIPPort()
  {
    return this.ipPort;
  }
}
```

- **Eventer**

```
package com.greatfree.remote;

import java.io.IOException;

import com.greatfree.concurrency.NotificationObjectQueue;
import com.greatfree.multicast.ServerMessage;

/*
 * This is a thread derived from NotificationObjectQueue. It keeps working until no objects are available in the queue. The thread keeps alive unless it is shutdown by a manager outside. 11/20/2014, Bing Li
 */

// Created: 11/20/2014, Bing Li
public class Eventer<Notification extends ServerMessage> extends NotificationObjectQueue<IPNotification<Notification>>
{
  // An instance of FreeClientPool through which the notification can be sent. 11/20/2014, Bing Li
  private FreeClientPool pool;
  // The time to be waited when no objects are available in the queue. 11/20/2014, Bing Li
  private long waitTime;

  /*
   * Initialize the eventer. It must notice that the primary component of the eventer, FreeClientPool, comes from outside. It represents that the eventer shares the pool with others. 11/20/2014, Bing Li
   */
  public Eventer(int queueSize, long waitTime, FreeClientPool pool)
  {
    super(queueSize);
    this.pool = pool;
    this.waitTime = waitTime;
  }

  /*
   * The task must be executed concurrently. 11/20/2014, Bing Li
   */
  public void run()
  {
    // The object to be dequeued from the queue. 11/20/2014, Bing Li
    IPNotification<Notification> notification;
    // Check whether the eventer is set to be shutdown. 11/20/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the queue is empty. 11/20/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the object from the queue. 11/20/2014, Bing Li
          notification = this.getNotification();
          try
          {
            // Since the object contains the IP/port and the notification, it is convenient to send the notification by the FreeClientPool. 11/20/2014, Bing Li
            this.pool.send(notification.getIPPort(), notification.getNotification());
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
          // Dispose the notification after it is sent out. 11/20/2014, Bing Li
          this.disposeObject(notification);
        }
        catch (InterruptedException e)
        {
```

```
                e.printStackTrace();
            }
        }
        try
        {
            // Wait for some time when no objects are available in the queue. 11/20/2014, Bing Li
            this.holdOn(this.waitTime);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
  }
}
```

- **AsyncRemoteEventer**

```
package com.greatfree.remote;

import java.util.HashMap;
import java.util.Map;
import java.util.Timer;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.concurrency.CheckIdleable;
import com.greatfree.concurrency.Collaborator;
import com.greatfree.concurrency.ThreadPool;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.util.CollectionSorter;
import com.greatfree.util.UtilConfig;

/*
 * This class aims to send notifications to a remote server asynchronously without waiting for responses. The sending methods are nonblocking. 11/20/2014, Bing Li
 */

// Created: 11/20/2014, Bing Li
public class AsyncRemoteEventer<Notification extends ServerMessage> extends Thread implements CheckIdleable
{
   // The eventers that are available to sent the notifications concurrently. They are indexed by the keys which are usually generated upon their IP/ports to be sent to. 11/20/2014, Bing Li
   private Map<String, Eventer<Notification>> eventers;
   // The queue which contains the notification to be sent. The notification contains the IP/port to be sent and the message to be sent. 11/20/2014, Bing Li
   private LinkedBlockingQueue<IPNotification<Notification>> notificationQueue;
   // The thread pool that starts and manages the eventer concurrently. It must be shared with others. 11/20/2014, Bing Li
   private ThreadPool threadPool;
   // The size of the event queue for each eventer. 11/20/2014, Bing Li
   private int eventQueueSize;
   // The count of eventers to be managed. 11/20/2014, Bing Li
   private int eventerSize;
   // The timer to manage the idle checker. 11/20/2014, Bing Li
   private Timer checkTimer;
   // The idle checker to monitor whether an eventer is idle long enough. 11/20/2014, Bing Li
   private EventerIdleChecker<AsyncRemoteEventer<Notification>> idleChecker;
   // The collaborator is used to pause the dispatcher when no notifications are available and notify to continue when new notifications are received. 11/20/2014, Bing Li
   private Collaborator collaborator;
   // The time to be waited when no notifications are available in the class. 11/20/2014, Bing Li
   private long eventingWaitTime;
   // The FreeClientPool that is used to initialize eventers. It must be shared with others. 11/20/2014, Bing Li
   private FreeClientPool clientPool;
   // The time to be waited when no notifications are available in each eventer. 11/20/2014, Bing Li
   private long eventerWaitTime;

   /*
    * Initialize. 11/20/2014, Bing Li
    */
   public AsyncRemoteEventer(FreeClientPool clientPool, ThreadPool threadPool, int eventQueueSize, int eventerSize, long eventingWaitTime, long eventerWaitTime)
   {
      this.eventers = new ConcurrentHashMap<String, Eventer<Notification>>();
      this.notificationQueue = new LinkedBlockingQueue<IPNotification<Notification>>();
      this.threadPool = threadPool;
      this.eventQueueSize = eventQueueSize;
      this.eventerSize = eventerSize;
      this.checkTimer = UtilConfig.NO_TIMER;
```

```
    this.collaborator = new Collaborator();
    this.eventingWaitTime = eventingWaitTime;
    this.clientPool = clientPool;
    this.eventerWaitTime = eventerWaitTime;
  }

  /*
   * Dispose the eventer dispatcher. 11/20/2014, Bing Li
   */
  public synchronized void dispose()
  {
    // Set the shutdown flag to be true. Thus, the loop in the dispatcher thread to schedule notification
loads is terminated. 11/20/2014, Bing Li
    this.collaborator.setShutdown();
    // Notify the dispatcher thread that is waiting for the notifications to terminate the waiting.
11/20/2014, Bing Li
    this.collaborator.signalAll();
    // Clear the notification queue. 11/20/2014, Bing Li
    if (this.notificationQueue != null)
    {
      this.notificationQueue.clear();
    }
    // Cancel the timer that controls the idle checking. 11/20/2014, Bing Li
    if (this.checkTimer != UtilConfig.NO_TIMER)
    {
      this.checkTimer.cancel();
    }
    // Terminate the periodically running thread for idle checking. 11/20/2014, Bing Li
    if (this.idleChecker != null)
    {
      this.idleChecker.cancel();
    }
    // Dispose all of eventers created during the dispatcher's running procedure. 11/20/2014, Bing Li
    for (Eventer<Notification> eventer : this.eventers.values())
    {
      eventer.dispose();
    }
    // Clear the eventer map. 11/20/2014, Bing Li
    this.eventers.clear();
  }

  /*
   * The method is called back by the idle checker periodically to monitor the idle states of eventers
within the dispatcher. 11/20/2014, Bing Li
   */
  @Override
  public void checkIdle()
  {
    // Check each eventer managed by the dispatcher. 11/20/2014, Bing Li
    for (Eventer<Notification> eventer : this.eventers.values())
    {
      // If the eventer is empty and idle, it is the one to be checked. 11/20/2014, Bing Li
      if (eventer.isEmpty() && eventer.isIdle())
      {
        // The algorithm to determine whether an eventer should be disposed or not is simple. When it is
checked to be idle, it is time to dispose it. 11/20/2014, Bing Li
        this.eventers.remove(eventer.getKey());
        // Dispose the eventer. 11/20/2014, Bing Li
        eventer.dispose();
        // Collect the resource of the eventer. 11/20/2014, Bing Li
        eventer = null;
      }
    }
  }

  /*
   * Set the idle checking parameters. 11/20/2014, Bing Li
   */
```

```
  @Override
  public void setIdleChecker(long idleCheckDelay, long idleCheckPeriod)
  {
    // Initialize the timer. 11/20/2014, Bing Li
    this.checkTimer = new Timer();
    // Initialize the idle checker. 11/20/2014, Bing Li
    this.idleChecker = new EventerIdleChecker<AsyncRemoteEventer<Notification>>(this);
    // Schedule the idle checking task. 11/20/2014, Bing Li
    this.checkTimer.schedule(this.idleChecker, idleCheckDelay, idleCheckPeriod);
  }

  /*
   * Send the notification asynchronously to the IP/port. 11/20/2014, Bing Li
   */
  public synchronized void notify(String ip, int ipPort, Notification notification)
  {
    // Put the notification and the IP/port into the queue by enclosing them into an instance of
IPNotification. 11/20/2014, Bing Li
    this.notificationQueue.add(new IPNotification<Notification>(new IPPort(ip, ipPort), notification));
    // Signal the potential waiting thread to schedule eventers to sent the notification just enqueued.
11/20/2014, Bing Li
    this.collaborator.signal();
  }

  /*
   * Send the notification asynchronously to a remote node by its key. 11/20/2014, Bing Li
   */
  public synchronized void notify(String clientKey, Notification notification)
  {
    // Put the notification and the IP/port into the queue by enclosing them into an instance of
IPNotification. The IP/port is retrieved from the FreeClientPool by the node key. 11/20/2014, Bing Li
    this.notificationQueue.add(new IPNotification<Notification>(this.clientPool.getIPPort(clientKey),
notification));
    // Signal the potential waiting thread to schedule eventers to sent the notification just enqueued.
11/20/2014, Bing Li
    this.collaborator.signal();
  }

  /*
   * The thread of the dispatcher is always running until no notifications to be sent. If too many
notifications are received, more eventers are created by the dispatcher. If notifications are limited, the
count of threads created by the dispatcher is also small. It is possible no any threads are alive when no
notifications are received for a long time. 11/20/2014, Bing Li
   */
  public void run()
  {
    // Declare a notification. 11/20/2014, Bing Li
    IPNotification<Notification> notification;
    // Initialize a task map to calculate the load of each eventer. 11/20/2014, Bing Li
    Map<String, Integer> taskMap = new HashMap<String, Integer>();
    // Declare a string to keep the selected eventer key. 11/20/2014, Bing Li
    String selectedThreadKey = UtilConfig.NO_KEY;
    // The dispatcher usually runs all of the time unless the local node is shutdown. To shutdown the
dispatcher, the shutdown flag of the collaborator is set to true. 11/20/2014, Bing Li
    while (!this.collaborator.isShutdown())
    {
      try
      {
        // Check whether notifications are available in the queue. 11/20/2014, Bing Li
        while (!this.notificationQueue.isEmpty())
        {
          // Dequeue the notification from the queue of the dispatcher. 11/20/2014, Bing Li
          notification = this.notificationQueue.take();

          // Since all of the eventers created by the dispatcher are saved in the map by their unique
keys, it is necessary to check whether any alive eventers are available. If so, it is possible to assign
tasks to them if they are not so busy. 11/20/2014, Bing Li
          while (this.eventers.size() > 0)
```

```
        {
          // Clear the map to start to calculate the loads of those eventers. 11/20/2014, Bing Li
          taskMap.clear();

          // Each eventer's workload is saved into the task map. 11/20/2014, Bing Li
          for (Eventer<Notification> thread : this.eventers.values())
          {
            taskMap.put(thread.getKey(), thread.getQueueSize());
          }
          // Select the eventer whose load is the least and keep the key of the eventer. 11/20/2014,
Bing Li
          selectedThreadKey = CollectionSorter.minValueKey(taskMap);
          // Since no concurrency is applied here, it is possible that the key is invalid. Thus, just check
here. 11/20/2014, Bing Li
          if (selectedThreadKey != null)
          {
            // Since no concurrency is applied here, it is possible that the key is out of the map. Thus,
just check here. 11/20/2014, Bing Li
            if (this.eventers.containsKey(selectedThreadKey))
            {
              try
              {
                // Check whether the eventer's load reaches the maximum value. 11/20/2014, Bing Li
                if (this.eventers.get(selectedThreadKey).isFull())
                {
                  // Check if the pool is full. If the least load eventer is full as checked by the above
condition, it denotes that all of the current alive eventers are full. So it is required to create an eventer to
respond the newly received notifications if the eventer count of the pool does not reach the maximum.
11/20/2014, Bing Li
                  if (this.eventers.size() < this.eventerSize)
                  {
                    // Create a new eventer. 11/20/2014, Bing Li
                    Eventer<Notification> thread = new Eventer<Notification>(this.eventQueueSize,
this.eventerWaitTime, this.clientPool);
                    // Save the newly created eventer into the map. 11/20/2014, Bing Li
                    this.eventers.put(thread.getKey(), thread);
                    // Enqueue the notification into the queue of the newly created eventer. Then, the
notification will be processed by the eventer. 11/20/2014, Bing Li
                    this.eventers.get(thread.getKey()).enqueue(notification);
                    // Start the eventer by the thread pool. 11/20/2014, Bing Li
                    this.threadPool.execute(this.eventers.get(thread.getKey()));
                  }
                  else
                  {
                    // Force to put the notification into the queue when the count of eventers reaches
the upper limit and each of the eventer's queue is full. 11/20/2014, Bing Li
                    this.eventers.get(selectedThreadKey).enqueue(notification);
                  }
                }
                else
                {
                  // If the least load eventer's queue is not full, just put the notification into the queue.
11/20/2014, Bing Li
                  this.eventers.get(selectedThreadKey).enqueue(notification);
                }

                // Jump out from the loop since the notification is put into a thread. 11/20/2014, Bing Li
                break;
              }
              catch (NullPointerException e)
              {
                // Since no concurrency is applied here, it is possible that a NullPointerException is
raised. If so, it means that the selected eventer is not available. Just continue to select anther one.
11/20/2014, Bing Li
                continue;
              }
            }
          }
```

```
            }
            // If no eventers are available, it needs to create a new one to take the notification. 11/20/2014,
Bing Li
            if (this.eventers.size() <= 0)
            {
              // Create a new eventer. 11/20/2014, Bing Li
              Eventer<Notification> thread = new Eventer<Notification>(this.eventQueueSize,
this.eventerWaitTime, this.clientPool);
              // Put it into the map for further reuse. 11/20/2014, Bing Li
              this.eventers.put(thread.getKey(), thread);
              // Take the notification. 11/20/2014, Bing Li
              this.threadPool.execute(this.eventers.get(thread.getKey()));
              // Start the thread. 11/20/2014, Bing Li
              this.eventers.get(thread.getKey()).enqueue(notification);
            }

            // If the dispatcher is shutdown, it is not necessary to keep processing the notifications. So,
jump out the loop and the eventer is dead. 11/20/2014, Bing Li
            if (this.collaborator.isShutdown())
            {
              break;
            }
          }

          // Check whether the dispatcher is shutdown or not. 11/20/2014, Bing Li
          if (!this.collaborator.isShutdown())
          {
            // If the dispatcher is still alive, it denotes that no notifications are available temporarily. Just
wait for a while. 11/20/2014, Bing Li
            this.collaborator.holdOn(this.eventingWaitTime);
          }
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
        catch (NullPointerException e)
        {
          e.printStackTrace();
        }
      }
    }
}
```

- **EventerIdleChecker**

```
package com.greatfree.remote;

import java.util.TimerTask;

import com.greatfree.concurrency.CheckIdleable;

/*
 * The class works with AsyncRemoteEventer to check whether an instance of Eventer is idle long
enough so that it should be disposed. 11/20/2014, Bing Li
 */

// Created: 11/20/2014, Bing Li
public class EventerIdleChecker<T extends CheckIdleable> extends TimerTask
{
   // The resource to be checked must implement the interface of CheckIdleable. 11/20/2014, Bing Li
   private T t;

   /*
    * Initialize the checker. 11/20/2014, Bing Li
    */
   public EventerIdleChecker(T t)
   {
      this.t = t;
   }

   /*
    * Check the resource by calling back its corresponding method concurrently and periodically.
11/20/2014, Bing Li
    */
   @Override
   public void run()
   {
      t.checkIdle();
   }
}
```

- **OutMessageStream**

```
package com.greatfree.remote;

import java.io.ObjectOutputStream;
import java.util.concurrent.locks.Lock;

import com.greatfree.multicast.ServerMessage;

/*
 * The class consists of the output stream that responds a client. The lock is used to keep responding
operations atomic. The request is any message that extends ServerMessage. 07/30/2014, Bing Li
 */

// Created: 07/30/2014, Bing Li
public class OutMessageStream<Message extends ServerMessage>
{
  // The output stream that is responsible for responding clients. 07/30/2014, Bing Li
  private ObjectOutputStream out;
  // The lock that keeps output operations atomic. 07/30/2014, Bing Li
  private Lock lock;
  // The message that extends ServerMessage. 07/30/2014, Bing Li
  private Message message;

  public OutMessageStream(ObjectOutputStream out, Lock lock, Message message)
  {
    // The output stream is shared by different threads that respond the same client concurrently.
07/30/2014, Bing Li
    this.out = out;
    // One client is assigned a unique lock such that responding to the client is not affected by
concurrency. 07/30/2014, Bing Li
    this.lock = lock;
    // The request from the client. 07/30/2014, Bing Li
    this.message = message;
  }

  /*
   * Expose the output stream. 07/30/2014, Bing Li
   */
  public ObjectOutputStream getOutStream()
  {
    return this.out;
  }

  /*
   * Expose the lock. 07/30/2014, Bing Li
   */
  public Lock getLock()
  {
    return this.lock;
  }

  /*
   * Expose the request. 07/30/2014, Bing Li
   */
  public Message getMessage()
  {
    return this.message;
  }

  /*
   * Dispose the request after the responding is done. 07/30/2014, Bing Li
   */
  public void disposeMessage()
  {
    this.message = null;
```

```
    }
}
```

- **ServerListener**

```
package com.greatfree.remote;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

import com.greatfree.concurrency.Collaborator;
import com.greatfree.concurrency.ThreadPool;

// The class acts as the listener to wait for a client's connection. To be more powerful, it involves a thread pool and the concurrency control mechanism. 07/30/2014, Bing Li

// Created: 07/17/2014, Bing Li
public class ServerListener extends ThreadPool
{
  // The TCP ServerSocket. 07/30/2014, Bing Li
  private ServerSocket serverSocket;
  // Since it is necessary to control the count of connected clients, the Collaborator is used. 07/30/2014, Bing Li
  private Collaborator collaborator;

  public ServerListener(ServerSocket serverSocket, int threadPoolSize, long keepAliveTime)
  {
    // Set parameters for the parent class, ThreadPool. 07/30/2014, Bing Li
    super(threadPoolSize, keepAliveTime);
    this.serverSocket = serverSocket;
    this.collaborator = new Collaborator();
  }

  /*
   * Shutdown the listener and the associated thread pool. 07/30/2014, Bing Li
   */
  public void shutdown()
  {
    this.collaborator.setShutdown();
    this.collaborator.signalAll();
    super.shutdown();
  }

  /*
   * Wait for connections. 07/30/2014, Bing Li
   */
  public Socket accept() throws IOException
  {
    return this.serverSocket.accept();
  }

  /*
   * Check whether the listener is shutdown. 07/30/2014, Bing Li
   */
  public boolean isShutdown()
  {
    return this.collaborator.isShutdown();
  }

  /*
   * Expose the collaborator. 07/30/2014, Bing Li
   */
  public Collaborator getCollaborator()
  {
    return this.collaborator;
  }

  /*
```

```
    * Wait for the available lock. 07/30/2014, Bing Li
    */
   public void holdOn() throws InterruptedException
   {
      this.collaborator.holdOn();
   }

   /*
    * Execute a thread. Usually, the thread is a ServerMessagePipe. It is possible some other tasks that
need to be executed concurrently, such as connecting to the remote server for an eventing local server.
07/30/2014, Bing Li
    */
   public void execute(Runnable thread)
   {
      super.execute(thread);
   }
}
```

- **ServerIO**

```
package com.greatfree.remote;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.net.SocketException;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

import com.greatfree.concurrency.Collaborator;
import com.greatfree.util.FreeObject;
import com.greatfree.util.Tools;

/*
 * The class encloses all the IO required stuffs to receive and respond a client's requests. 07/30/2014,
Bing Li
 */

// Created: 07/30/2014, Bing Li
abstract public class ServerIO extends FreeObject implements Runnable
{
  //
  private String clientKey;
  // The remote client/server socket. 07/30/2014, Bing Li
  private Socket clientSocket;
  private ObjectInputStream in;
  private ObjectOutputStream out;
  private boolean isShutdown;
  private String ip;
  private Lock lock;
  private Collaborator collaborator;

  public ServerIO(Socket clientSocket, Collaborator collaborator)
  {
    // Set the object key for the parent object, FreeObject. 07/30/2014, Bing Li
    super(Tools.getClientIPPortKey(clientSocket));
    this.clientSocket = clientSocket;

    // Get the IP address of the client. Usually, the IP is used to connect the remote server for an
eventing server. 07/30/2014, Bing Li
    this.ip = Tools.getClientIPAddress(clientSocket);
    // Create the key for the class upon the connecting client socket. The key is used to manage
ServerIO conveniently in a hash map. 07/30/2014, Bing Li
    this.clientKey = Tools.getKeyOfFreeClient(Tools.getClientIPAddress(clientSocket),
Tools.getClientIPPort(clientSocket));
    try
    {
      // Create the input stream to receive requests from the client. 07/30/2014, Bing Li
      this.in = new ObjectInputStream(this.clientSocket.getInputStream());
      // Create the output stream to respond the client. 07/30/2014, Bing Li
      this.out = new ObjectOutputStream(this.clientSocket.getOutputStream());
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }

    // Set the shutdown flag. 07/30/2014, Bing Li
    this.isShutdown = false;

    this.lock = new ReentrantLock();
    this.collaborator = collaborator;
  }
```

```
  /*
   * Dispose all of the resources contained in the class. 07/30/2014, Bing Li
   */
  public synchronized void shutdown() throws IOException
  {
    // The server listener for a client might be blocked for the limited size of ServerIO. Therefore, when
removing one, it is necessary to notify the listener to accept more clients. 08/04/2014, Bing Li
    this.collaborator.signal();
    if (this.isShutdown)
    {
      this.isShutdown = true;
    }
    this.in.close();
    this.out.close();
    this.clientSocket.close();
  }

  /*
   * Get the key of the class. 07/30/2014, Bing Li
   */
  public String getClientKey()
  {
    return this.clientKey;
  }

  /*
   * Expose the lock that is used to keep the operations to respond clients in an atomic manner.
07/30/2014, Bing Li
   */
  public Lock getLock()
  {
    return this.lock;
  }

  /*
   * Since the resources of ServerIP are limited, it must limit the count of connected clients. It is
performed by the collaborator's method of holdOn(). Therefore, when one Server is shutdown, the
method can notify the listener to allow more clients to enter. 07/30/2014, Bing Li
   */
  public void signal()
  {
    this.collaborator.signal();
  }

  /*
   * Get the IP address of the remote client/server. 07/30/2014, Bing Li
   */
  public String getIP()
  {
    return this.ip;
  }

  /*
   * Check whether the class is shutdown or not. 07/30/2014, Bing Li
   */
  public boolean isShutdown()
  {
    return this.isShutdown;
  }

  /*
   * Set the flag of shutdown to true. 07/30/2014, Bing Li
   */
  public synchronized void setShutdown()
  {
    this.isShutdown = true;
  }
```

```
    /*
     * Wait for requests from a remote client. 07/30/2014, Bing Li
     */
    public Object read() throws IOException, ClassNotFoundException, SocketException
    {
      return this.in.readObject();
    }

    /*
     * Get the output stream that is used to respond the remote client's requests. 07/30/2014, Bing Li
     */
    public ObjectOutputStream getOutStream()
    {
      return this.out;
    }

    /*
     * The concurrency method that must be implemented. 08/10/2014, Bing Li
     */
    @Override
    public void run()
    {
    }
}
```

- **ServerIORegistry**

```
package com.greatfree.remote;

import java.io.IOException;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;

import com.google.common.collect.Sets;
import com.greatfree.util.UtilConfig;

/*
 * The class is used to keep all of the ServerIOs, which are assigned to each client for the interactions between the server and the corresponding client. 08/04/2014, Bing Li
 */

// Created: 08/04/2014, Bing Li
public class ServerIORegistry<IO extends ServerIO>
{
  // A map to keep all connected ServerIOs. 08/04/2014, Bing Li
  private Map<String, IO> ioRegistry;

  /*
   * The map is initialized as the one which has the ability of concurrency control. 08/04/2014, Bing Li
   */
  public ServerIORegistry()
  {
    this.ioRegistry = new ConcurrentHashMap<String, IO>();
  }

  /*
   * Add a new connected ServerIO. 08/04/2014, Bing Li
   */
  public void addIO(IO io)
  {
    this.ioRegistry.put(io.getClientKey(), io);
  }

  /*
   * Get the count of the current available ServerIOs. 08/04/2014, Bing Li
   */
  public int getIOCount()
  {
    return this.ioRegistry.size();
  }

  /*
   * Get all of the available IPs. 08/04/2014, Bing Li
   */
  public Set<String> getIPs()
  {
    Set<String> ips = Sets.newHashSet();
    for (IO io : this.ioRegistry.values())
    {
      ips.add(io.getIP());
    }
    return ips;
  }

  /*
   * Get the IP of a particular client. 08/04/2014, Bing Li
   */
  public String getIP(String clientKey)
  {
    if (this.ioRegistry.containsKey(clientKey))
```

```
    {
      return this.ioRegistry.get(clientKey).getIP();
    }
    return UtilConfig.NO_IP;
  }

  /*
   * Remove a particular client. 08/04/2014, Bing Li
   */
  public void removeIO(IO io) throws IOException
  {
    if (this.ioRegistry.containsKey(io.getClientKey()))
    {
      this.ioRegistry.remove(io.getClientKey());
    }
    io.shutdown();
  }

  /*
   * Remove all of the ServerIOs. 08/04/2014, Bing Li
   */
  public void removeAllIOs() throws IOException
  {
    for (IO io : this.ioRegistry.values())
    {
      io.shutdown();
    }
    this.ioRegistry.clear();
  }
}
```

2.2 Reuse

- ## Creatable

```
package com.greatfree.reuse;

import java.io.IOException;

import com.greatfree.util.FreeObject;

/*
 * This is an interface to define a resource creator that initiates an instance of the resource in the
resource pool, such as RetrievablePool. The resource must derive from FreeObject. 08/26/2014, Bing Li
 */

// Created: 08/26/2014, Bing Li
public interface Creatable<Source, Resource extends FreeObject>
{
   /*
    * The interface to initiate an instance of the resource. The argument, source, encloses the initiation
values for the instance. 08/26/2014, Bing Li
    */
   public Resource createResourceInstance(Source source) throws IOException;
}
```

- **Disposable**

```
package com.greatfree.reuse;

import com.greatfree.util.FreeObject;

/*
 * The interface defines a method for the disposer that collects the resource in the resource pool, such
as RetrievablePool. The resource must derive from FreeObject. 08/26/2014, Bing Li
 */

// Created: 08/26/2014, Bing Li
public interface Disposable<Resource extends FreeObject>
{
   /*
    * The interface to dispose a resource. 08/26/2014, Bing Li
    */
   public void dispose(Resource rsc);
}
```

- **HashCreatable**

```
package com.greatfree.reuse;

import com.greatfree.util.HashFreeObject;

/*
 * The interface defines the method to create instances which extend HashFreeObject. 11/26/2014, Bing
Li
 */

// Created: 11/26/2014, Bing Li
public interface HashCreatable<Source, Resource extends HashFreeObject>
{
   public Resource createResourceInstance(Source source);
}
```

- **HashDisposable**

```
package com.greatfree.reuse;

import com.greatfree.util.HashFreeObject;

/*
 * The interface defines the method to dispose the objects that are derived from HashFreeObject.
11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public interface HashDisposable<Resource extends HashFreeObject>
{
   public void dispose(Resource t);
}
```

- **RunDisposable**

```
package com.greatfree.reuse;

/*
 * The class is an interface to define a thread's disposer. 07/30/2014, Bing Li
 */

// Created: 07/17/2014, Bing Li
public interface RunDisposable<Resource extends Runnable>
{
   public void dispose(Resource r);
   public void dispose(Resource r, long time);
}
```

- **ThreadDisposable**

```
package com.greatfree.reuse;

/*
 * This is an interface to dispose a thread. 09/20/2014, Bing Li
 */

// Created: 08/04/2014, Bing Li
public interface ThreadDisposable<Resource extends Thread>
{
   public void dispose(Resource r);
   public void dispose(Resource r, long time);
}
```

- **RetrievablePool**

```
package com.greatfree.reuse;

import java.io.IOException;
import java.util.Calendar;
import java.util.Date;
import java.util.Map;
import java.util.Set;
import java.util.Timer;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.locks.ReentrantLock;

import com.greatfree.concurrency.Collaborator;
import com.greatfree.util.CollectionSorter;
import com.greatfree.util.FreeObject;
import com.greatfree.util.Time;
import com.greatfree.util.UtilConfig;

/*
 * The class is a resource pool that aims to utilize the resources sufficiently with a lower cost. The pool is usually used for the resource of FreeClient. For each remote end, multiple FreeClients are initialized and managed by the pool. When it is necessary to interact with one remote end, it is convenient to obtain a FreeClient by the key or the initial values, i.e., the IP address and the port, that represent the remote end uniquely. That is why the pool is named the RetrievablePool. 09/02/2014, Bing Li
 *
 * The Source is the initial values to create a resource. It must extend the object, FreeObject. 09/02/2014, Bing Li
 *
 * The Resource is the resource that is managed by the pool. Also, it must derive from the object, FreeObject. 09/02/2014, Bing Li
 *
 * The Creator aims to initialize an instance of a resource when no idle ones are available and the maximum pool size is not reached. It should implement the interface, ResourceCreatable. 09/02/2014, Bing Li
 *
 * The Disposer is responsible for collecting or disposing the resources that are idle long enough. 09/02/2014, Bing Li
 *
 */

// Created: 08/26/2014, Bing Li
public class RetrievablePool<Source extends FreeObject, Resource extends FreeObject, Creator extends Creatable<Source, Resource>, Disposer extends Disposable<Resource>>
{
   // The map contains the resources that are being used. For each type of resources, a children map is initialized. The parent key represents the type of resources. The key of the children map is unique for each resource. For the case of FreeClient, the parent key is generated upon the IP address and the port of a remote end. Well, each client to the particular remote end has the children key, a hash value that is created randomly. 09/02/2014, Bing Li
   private Map<String, Map<String, Resource>> busyMap;
   // The map contains the resources that are idle temporarily. Similar to the busy map, for each type of resources, a children map is initialized. The parent key represents the type of resources. The key of the children map is unique for each resource. 09/02/2014, Bing Li
   private Map<String, Map<String, Resource>> idleMap;
   // The map contains the initial values that are used to initialize particular resources. The key is identical to that of the resource. Therefore, it is fine to retrieve the source to create the resource when no idle resources are available and the pool size does not reach the maximum value. For the case of FreeClient, the initial values are the IP address and the port. 09/02/2014, Bing Li
   private Map<String, Source> sourceMap;

   // The lock is responsible for managing the operations on busyMap, idleMap and sourceMap atomic. 10/12/2014, Bing Li
   private ReentrantLock rscLock;
   // The Collaborator is used in the pool to work in the way of notify/wait. A thread has to wait when the pool is full. When resources are available, it can be notified by the collaborator. The collaborator also
```

```
manages the termination of the pool. 09/02/2014, Bing Li
  private Collaborator collaborator;
  // The Timer controls the period to check the idle resources periodically. 11/06/2014, Bing Li
  private Timer checkTimer;
  // This is an instance of a class that is executed periodically to check idle resources. When a resource
is idle long enough, it is necessary to collect it. 09/02/2014, Bing Li
  private RetrievableIdleChecker<Source, Resource, Creator, Disposer> idleChecker;
  // The maximum time a resource can be idle. 09/02/2014, Bing Li
  private long maxIdleTime;
  // The size of the resources the pool can contain. 09/02/2014, Bing Li
  private int poolSize;
  // When no idle resources are available and the pool size is not reached, it is allowed to create a new
resource by the Creator. 09/02/2014, Bing Li
  private Creator creator;
  // When a resource is idle long enough, it is collected or disposed by the Disposer. 09/02/2014, Bing Li
  private Disposer disposer;

  /*
   * Initialize. 09/02/2014, Bing Li
   */
  public RetrievablePool(int poolSize, Creator creator, Disposer disposer)
  {
    this.busyMap = new ConcurrentHashMap<String, Map<String, Resource>>();
    this.idleMap = new ConcurrentHashMap<String, Map<String, Resource>>();
    this.sourceMap = new ConcurrentHashMap<String, Source>();

    this.rscLock = new ReentrantLock();
    this.collaborator = new Collaborator();
    this.poolSize = poolSize;
    this.creator = creator;
    // It is possible that the pool does not need to check the idle resources. It happens when the pool is
used in the case when the resource is not heavy and the consumed resources are low. If so, it is
unnecessary to initialize the timer. 09/02/2014, Bing Li
    this.checkTimer = UtilConfig.NO_TIMER;
    this.disposer = disposer;
  }

  /*
   * Shutdown the pool. 09/02/2014, Bing Li
   */
  public void shutdown()
  {
    // Set the shutdown flag to be true. Thus, the thread that runs the method of get() can be terminated.
09/02/2014, Bing Li
    this.collaborator.setShutdown();
    // Notify the thread that runs the method of get() to terminate and all of the threads that are waiting
for the resources that they are unblocked. 09/02/2014, Bing Li
    this.collaborator.signalAll();
    // Dispose all of the busy resources. It might lose data if no relevant management approaches are
adopted. 09/02/2014, Bing Li
    for (Map<String, Resource> resourceMap : this.busyMap.values())
    {
      for (Resource resource : resourceMap.values())
      {
        this.disposer.dispose(resource);
      }
    }
    this.busyMap.clear();

    // Dispose all of the idle resources. It might lose data if no relevant management approaches are
adopted. 09/02/2014, Bing Li
    for (Map<String, Resource> resourceMap : this.idleMap.values())
    {
      for (Resource resource : resourceMap.values())
      {
        this.disposer.dispose(resource);
      }
    }
```

```
      this.idleMap.clear();

      // Clear the sources. 09/02/2014, Bing Li
      this.sourceMap.clear();

      // Terminate the idle state checker that periodically runs. For it is possible that the checker is not initialized in some cases, it needs to check whether it is null or not. 09/02/2014, Bing Li
      if (this.idleChecker != null)
      {
        this.idleChecker.cancel();
      }
      // Terminate the timer that manages the period to run the idle state checker. For it is possible that the timer is not initialized in some cases, it needs to check whether it is null or not. 09/02/2014, Bing Li
      if (this.checkTimer != UtilConfig.NO_TIMER)
      {
        this.checkTimer.cancel();
      }
    }

    /*
     * Initialize the idle state checker and the timer to manage idle resources periodically when needed. 09/02/2014, Bing Li
     */
    public void setIdleChecker(long idleCheckDelay, long idleCheckPeriod, long maxIdleTime)
    {
      // Initialize the timer. 09/02/2014, Bing Li
      this.checkTimer = new Timer();
      // Initialize the checker. 09/02/2014, Bing Li
      this.idleChecker = new RetrievableIdleChecker<Source, Resource, Creator, Disposer>(this);
      // Schedule the task of checking idle states of resources. 09/02/2014, Bing Li
      this.checkTimer.schedule(this.idleChecker, idleCheckDelay, idleCheckPeriod);
      // Set the value of maximum idle time. 09/02/2014, Bing Li
      this.maxIdleTime = maxIdleTime;
    }

    /*
     * Check the idle resources. The method is called back by the IdleChecker. 09/02/2014, Bing Li
     */
    public void checkIdle()
    {
      // Get the current time which is used to calculate the idle time length. 09/02/2014, Bing Li
      Date currentTime = Calendar.getInstance().getTime();

      // The map is used to keep the sorted resources to select the longest idle one conveniently. 09/02/2014, Bing Li
      Map<String, Resource> sortedResourceMap;

      // Check whether the idle resources are idle long enough time. 09/02/2014, Bing Li
      this.rscLock.lock();
      for (Map<String, Resource> resourceMap : this.idleMap.values())
      {
        // Sort the idle resources by their idle time moment in the ascending order and the results are saved in the map of sortedResourceMap. Thus, the resources that exceed the maximum time length most probably are listed ahead. 09/02/2014, Bing Li
        sortedResourceMap = CollectionSorter.sortByValue(resourceMap);
        // Check the sorted resources. 09/02/2014, Bing Li
        for (Resource resource : sortedResourceMap.values())
        {
          // Calculate the idle time length and compare it with the maximum idle time. 09/02/2014, Bing Li
          if (Time.getTimespanInMilliSecond(currentTime, resource.getAccessedTime()) > this.maxIdleTime)
          {
            // If the idle time exceeds the maximum idle time, it denotes that the resource is not needed. So it must be collected or disposed. 09/06/2014, Bing Li
            resourceMap.remove(resource.getHashKey());
            // Dispose the resource by the disposer. 09/06/2014, Bing Li
            this.disposer.dispose(resource);
          }
```

```
            else
            {
              // If the idle time of the resource that has been idle for the longest time in the type does not
exceed the maximum value, it denotes that all of the resources of the type have not been idle long
enough. Thus, it needs to detect other types of resources. 09/06/2014, Bing Li
              break;
            }
          }
        }

        // The loop detects whether one type of idle resources is empty. If so, it needs to be removed from
the idle map. 09/06/2014, Bing Li
        for (String objectKey : this.idleMap.keySet())
        {
          // Check whether one type of idle resources is empty. 09/06/2014, Bing Li
          if (this.idleMap.get(objectKey).size() <= 0)
          {
            // Remove the type of idle resources that is empty from the map. 09/06/2014, Bing Li
            this.idleMap.remove(objectKey);
          }
        }
        this.rscLock.unlock();

        // Notify the blocked thread that the size of the assigned resources is lowered such that it is time for
it to get a resource it needs. A bunch of resources might be disposed during the procedure. It is
reasonable to signal all rather than signal a single waiting thread. 09/06/2014, Bing Li
        this.collaborator.signalAll();
      }

      /*
       * Dispose a resource explicitly when it is not needed. Usually, it is not invoked by the pool but by the
threads that consume the resource. It happens when it is confirmed that the resource is never needed in
a specific case. 09/06/2014, Bing Li
       */
      public void dispose(Resource res)
      {
        // Check whether the resource is null. 09/06/2014, Bing Li
        if (res != null)
        {
          // Check whether the pool is shutdown or not. If not, it must be managed by the rules of the pool.
09/06/2014, Bing Li
          if (!this.collaborator.isShutdown())
          {
            this.rscLock.lock();
            // Check whether the type of the resource is contained in the busy map. 09/06/2014, Bing Li
            if (this.busyMap.containsKey(res.getObjectKey()))
            {
              // Check whether the resource is contained in the busy map. 09/06/2014, Bing Li
              if (this.busyMap.get(res.getObjectKey()).containsKey(res.getHashKey()))
              {
                // Remove the resource from the busy map. 09/06/2014, Bing Li
                this.busyMap.get(res.getObjectKey()).remove(res.getHashKey());
                // Check if the type of the resource is empty in the busy map. 09/06/2014, Bing Li
                if (this.busyMap.get(res.getObjectKey()).size() <= 0)
                {
                  // Remove the type of the resource from the busy map. 09/06/2014, Bing Li
                  this.busyMap.remove(res.getObjectKey());
                }
              }
            }

            // Check whether the idle map contains the type of the resource. 09/06/2014, Bing Li
            if (this.idleMap.containsKey(res.getObjectKey()))
            {
              // Check whether the idle map contains the specific resource. 09/06/2014, Bing Li
              if (this.idleMap.get(res.getObjectKey()).containsKey(res.getHashKey()))
              {
                // Remove the resource from the idle map. 09/06/2014, Bing Li
```

```
            this.idleMap.get(res.getObjectKey()).remove(res.getHashKey());
            // Check if the type of the resource is empty in the idle map. 09/06/2014, Bing Li
            if (this.idleMap.get(res.getObjectKey()).size() <= 0)
            {
              // Remove the type of the resource from the idle map. 09/06/2014, Bing Li
              this.idleMap.remove(res.getObjectKey());
            }
          }
        }

        // Dispose the resource eventually after it is managed following the rules of the pool. 09/06/2014,
Bing Li
        this.disposer.dispose(res);

        this.rscLock.unlock();

        // Notify the thread that is blocked for the maximum size of the pool is reached. 09/06/2014, Bing
Li
        this.collaborator.signal();
      }
      else
      {
        // If the pool is shutdown, the resource is disposed directly. 09/06/2014, Bing Li
        this.disposer.dispose(res);
      }
    }
  }

  /*
   * Collect a resource. When the resource finishes its task, the method is invoked by the corresponding
thread such that the resource can be reused. 09/06/2014, Bing Li
   */
  public void collect(Resource res)
  {
    // Check whether the resource pool is shutdown. For the method is critical to the resource pool, it
does not make sense to go ahead if the pool is shutdown. 09/06/2014, Bing Li
    if (res != null && !this.collaborator.isShutdown())
    {
      this.rscLock.lock();
      // Check whether the busy map contains the type of the resource. 09/06/2014, Bing Li
      if (this.busyMap.containsKey(res.getObjectKey()))
      {
        // Check whether the resource is contained in the busy map. 09/06/2014, Bing Li
        if (this.busyMap.get(res.getObjectKey()).containsKey(res.getHashKey()))
        {
          // Remove the resource to be collected. 09/06/2014, Bing Li
          this.busyMap.get(res.getObjectKey()).remove(res.getHashKey());
          // Check whether the type of the resource is empty in the busy map. If so, it is proper to
remove it since it denotes that the type of the resource is not needed any longer. 09/06/2014, Bing Li
          if (this.busyMap.get(res.getObjectKey()).size() <= 0)
          {
            // Remove the type of the resource from the busy map. 09/06/2014, Bing Li
            this.busyMap.remove(res.getObjectKey());
          }
        }
      }

      // Set the accessed time stamp for the resource. The time stamp is the idle starting moment of the
resource. It is used to calculate whether the resource is idle for long enough. 09/06/2014, Bing Li
      res.setAccessedTime();
      // Check whether the idle map contains the type of the resource. 09/06/2014, Bing Li
      if (!this.idleMap.containsKey(res.getObjectKey()))
      {
        // If the type of the resource is not contained in the idle map, it needs to add the type first.
09/06/2014, Bing Li
        this.idleMap.put(res.getObjectKey(), new ConcurrentHashMap<String, Resource>());
        // Add the resource to the idle map. 09/06/2014, Bing Li
        this.idleMap.get(res.getObjectKey()).put(res.getHashKey(), res);
```

```
        }
        else
        {
          // If the type of the resource is already existed in the idle map, just add the resource to the idle
map. 09/06/2014, Bing Li
          this.idleMap.get(res.getObjectKey()).put(res.getHashKey(), res);
        }
        this.rscLock.unlock();

        // Notify the thread that is blocked for the maximum size of the pool is reached. 09/06/2014, Bing Li
        this.collaborator.signal();
      }
    }

    /*
     * Remove the type of resources explicitly by the key of the type. It is used in the case when a thread
confirms that one type of resources is not needed to be created in the pool. It is not used frequently. Just
keep an interface for possible cases. 09/06/2014, Bing Li
     */
    public void removeResource(String objectKey)
    {
      this.rscLock.lock();

      // Check whether the type of resources is existed in the busy map. 09/17/2014, Bing Li
      if (this.busyMap.containsKey(objectKey))
      {
        // Dispose all of the resources of the type if they are in the busy map. 09/17/2014, Bing Li
        for (Resource rsc : this.busyMap.get(objectKey).values())
        {
          this.disposer.dispose(rsc);
        }
        // Remove the type of resources from the busy map. 09/17/2014, Bing Li
        this.busyMap.remove(objectKey);
      }

      // Check whether the type of resources is existed in the idle map. 09/17/2014, Bing Li
      if (this.idleMap.containsKey(objectKey))
      {
        // Dispose all of the resources of the type if they are in the idle map. 09/17/2014, Bing Li
        for (Resource rsc : this.idleMap.get(objectKey).values())
        {
          this.disposer.dispose(rsc);
        }
        // Remove the type of resources from the idle map. 09/17/2014, Bing Li
        this.idleMap.remove(objectKey);
      }
      this.rscLock.unlock();

      // Notify all of the threads that are waiting for resources to keep on working if resources are available
for the removal from the idle map. 09/17/2014, Bing Li
      this.collaborator.signalAll();
    }

    /*
     * Check whether a specific resource is busy. 09/17/2014, Bing Li
     */
    public boolean isBusy(Resource resource)
    {
      this.rscLock.lock();
      try
      {
        // Check whether the type of the resource is existed in the busy map. 09/17/2014, Bing Li
        if (this.busyMap.containsKey(resource.getObjectKey()))
        {
          // Check whether the specific resource is existed in the busy map if the type of the resource is
existed in the map. 09/17/2014, Bing Li
          return this.busyMap.get(resource.getObjectKey()).containsKey(resource.getHashKey());
        }
```

```
        // If the type of the resource is not existed in the busy map, the resource is not busy. 09/17/2014,
Bing Li
        return false;
      }
      finally
      {
        this.rscLock.unlock();
      }
    }

    /*
     * Return all of the type keys of resources. 09/17/2014, Bing Li
     */
    public Set<String> getAllObjectKeys()
    {
      this.rscLock.lock();
      try
      {
        return this.sourceMap.keySet();
      }
      finally
      {
        this.rscLock.unlock();
      }
    }

    /*
     * Get the source of a particular type of resources. 09/17/2014, Bing Li
     */
    public Source getSource(String objectKey)
    {
      this.rscLock.lock();
      try
      {
        // Check whether the source is available in the source map. 09/17/2014, Bing Li
        if (this.sourceMap.containsKey(objectKey))
        {
          // Return the source if it is existed in the source map. 09/17/2014, Bing Li
          return this.sourceMap.get(objectKey);
        }
        // Return null if the source is not available in the source map. 09/17/2014, Bing Li
        return null;
      }
      finally
      {
        this.rscLock.unlock();
      }
    }

    /*
     * Get the size of sources. 09/17/2014, Bing Li
     */
    public int getSourceSize()
    {
      this.rscLock.lock();
      try
      {
        return this.sourceMap.size();
      }
      finally
      {
        this.rscLock.unlock();
      }
    }

    /*
     * Get the size of the pool. 11/03/2014, Bing Li
     */
```

```
  public synchronized int getPoolSize()
  {
    return this.poolSize;
  }

  /*
   * Check whether a particular source is available by the key of the source. 09/17/2014, Bing Li
   */
  public boolean isSourceExisted(String key)
  {
    this.rscLock.lock();
    try
    {
      return this.sourceMap.containsKey(key);
    }
    finally
    {
      this.rscLock.unlock();
    }
  }

  /*
   * Check whether a particular source is available by the source itself. 09/17/2014, Bing Li
   */
  public boolean isSourceExisted(Source src)
  {
    this.rscLock.lock();
    try
    {
      return this.sourceMap.containsKey(src.getObjectKey());
    }
    finally
    {
      this.rscLock.unlock();
    }
  }

  /*
   * Get the size of all of the types of busy resources at the moment when the method is invoked.
09/17/2014, Bing Li
   */
  public int getBusyResourceSize()
  {
    int busyResourceCount = 0;
    this.rscLock.lock();
    try
    {
      for (Map<String, Resource> resMap : this.busyMap.values())
      {
        busyResourceCount += resMap.size();
      }
      return busyResourceCount;
    }
    finally
    {
      this.rscLock.unlock();
    }
  }

  /*
   * Get the size of all of the types of idle resources at the moment when the method is invoked.
11/03/2014, Bing Li
   */
  public int getIdleResourceSize()
  {
    int idleResourceCount = 0;
    this.rscLock.lock();
    try
```

```
    {
      for (Map<String, Resource> resMap : this.idleMap.values())
      {
        idleResourceCount += resMap.size();
      }
      return idleResourceCount;
    }
    finally
    {
      this.rscLock.unlock();
    }
  }

  /*
   * Get the resource by its source. 09/17/2014, Bing Li
   */
  public synchronized Resource get(Source source) throws IOException
  {
    // The resource to be returned. 09/17/2014, Bing Li
    Resource rsc = null;
    // One particular type of resources that is the same type as the requested one. 09/17/2014, Bing Li
    Map<String, Resource> resourceMap;
    // The hash key of the resource that has the longest lifetime. 09/17/2014, Bing Li
    String oldestResourceKey;
    // The count of busy resources. 09/17/2014, Bing Li
    int busyResourceCount;
    // The count of idle resources 09/17/2014, Bing Li
    int idleResourceCount;

    // Since the procedure to get a particular resource might be blocked when the upper limit of the pool
is reached, it is required to keep a notify/wait mechanism to guarantee the procedure smooth.
09/17/2014, Bing Li
    while (!this.collaborator.isShutdown())
    {
      // Initialize the value of busyResourceCount. 09/17/2014, Bing Li
      busyResourceCount = 0;
      // Initialize the value of idleResourceCount. 09/17/2014, Bing Li
      idleResourceCount = 0;

      this.rscLock.lock();
      try
      {
        /*
         * Retrieve the resource from the idle map first. 09/17/2014, Bing Li
         */

        // Check whether the idle map contains the type of the resource. 09/17/2014, Bing Li
        if (this.idleMap.containsKey(source.getObjectKey()))
        {
          // The type of idle resources which are the candidates to be returned. 09/17/2014, Bing Li
          resourceMap = this.idleMap.get(source.getObjectKey());
          // Get the hash key of the resource that is idle longer than any others from the idle resources.
09/17/2014, Bing Li
          oldestResourceKey = CollectionSorter.minValueKey(resourceMap);
          // Check whether the hash key is valid. 09/17/2014, Bing Li
          if (oldestResourceKey != null)
          {
            // Get the resource that is idle than any others by its hash key. 09/17/2014, Bing Li
            rsc = resourceMap.get(oldestResourceKey);
            // Set the accessed time of the resource that is idle than any others to the current moment
since the resource will be used after it is returned. Now it becomes a busy one. 09/17/2014, Bing Li
            rsc.setAccessedTime();
            // Remove the resource from the idle map since it will become a busy one. 09/17/2014, Bing
Li
            this.idleMap.get(source.getObjectKey()).remove(oldestResourceKey);
            // Check whether the type of resources is empty in the idle map. 09/17/2014, Bing Li
            if (this.idleMap.get(source.getObjectKey()).size() <= 0)
            {
```

```
                // Remove the type of resources from the idle map if no any specific resources of the type
is in the idle map. 09/17/2014, Bing Li
                this.idleMap.remove(source.getObjectKey());
              }
              // Check whether the type of resources is available in the busy map. 09/17/2014, Bing Li
              if (!this.busyMap.containsKey(rsc.getObjectKey()))
              {
                // If the type of resources is not existed in the busy map, add the type and the particular
resource. 09/17/2014, Bing Li
                this.busyMap.put(rsc.getObjectKey(), new ConcurrentHashMap<String, Resource>());
                this.busyMap.get(rsc.getObjectKey()).put(oldestResourceKey, rsc);
              }
              else
              {
                // If the type of resources is existed in the busy map, add the particular resource.
09/17/2014, Bing Li
                this.busyMap.get(rsc.getObjectKey()).put(oldestResourceKey, rsc);
              }
            }
          }

          /*
           * If the idle map does not contain the requested resource, it is necessary to initialize a new one.
09/17/2014, Bing Li
           */

          // Check whether the resource is available after retrieving from the idle map. 10/12/2014, Bing Li
          if (rsc == null)
          {
            // Calculate the exact count of all of the busy resources. The value is used to check whether
the upper limit of the pool is reached. 09/17/2014, Bing Li
            for (Map<String, Resource> resMap : this.busyMap.values())
            {
              busyResourceCount += resMap.size();
            }

            // Calculate the exact count of all of the idle resources. The value is used to check whether the
upper limit of the pool is reached. 09/17/2014, Bing Li
            for (Map<String, Resource> resMap : this.idleMap.values())
            {
              idleResourceCount += resMap.size();
            }

            // Check whether the sum of the count of busy and idle resources reach the upper limit of the
pool. 09/17/2014, Bing Li
            if (busyResourceCount + idleResourceCount < this.poolSize)
            {
              // If the upper limit of the pool is not reached, it is time to create an instance by its source.
09/17/2014, Bing Li
              rsc = this.creator.createResourceInstance(source);
              // Check whether the newly created instance is valid. 09/17/2014, Bing Li
              if (rsc != null)
              {
                // Check whether the type of the resource is available in the busy map. 09/17/2014, Bing Li
                if (!this.busyMap.containsKey(rsc.getObjectKey()))
                {
                  // If the type of the resource is not available in the busy map, add the type and the
resource into it. 09/17/2014, Bing Li
                  this.busyMap.put(rsc.getObjectKey(), new ConcurrentHashMap<String, Resource>());
                  this.busyMap.get(rsc.getObjectKey()).put(rsc.getHashKey(), rsc);
                }
                else
                {
                  // If the type of the resource is available in the busy map, add the resource into it.
09/17/2014, Bing Li
                  this.busyMap.get(rsc.getObjectKey()).put(rsc.getHashKey(), rsc);
                }
                // Add the source to the source map. 09/17/2014, Bing Li
```

```
                this.sourceMap.put(rsc.getObjectKey(), source);
              }
              // Return the resource to the thread. 09/17/2014, Bing Li
              return rsc;
            }
          }
          else
          {
            // If the resource is obtained from the idle map and then return the resource to the thread.
09/17/2014, Bing Li
            return rsc;
          }
        }
        finally
        {
          this.rscLock.unlock();
        }

        /*
         * If no such resources are in the idle map and the upper limit of the pool is reached, the thread
that invokes the method has to wait for future possible updates. The possible updates include resource
disposals and idle resources being available. 09/17/2014, Bing Li
         */
        try
        {
          // Check whether the pool is shutdown. 09/17/2014, Bing Li
          if (!this.collaborator.isShutdown())
          {
            // If the pool is not shutdown, it is time to wait for some time. After that, the above procedure is
repeated if the pool is not shutdown. 09/17/2014, Bing Li
            this.collaborator.holdOn(UtilConfig.ONE_SECOND);
          }
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      return null;
    }

    /*
     * Get the resource by its type key. 09/17/2014, Bing Li
     */
    public Resource get(String objectKey) throws IOException
    {
      // Check whether the key is valid. 09/17/2014, Bing Li
      if (objectKey != UtilConfig.NO_KEY)
      {
        Source src;
        // The resource to be returned. 09/17/2014, Bing Li
        Resource rsc = null;
        // One particular type of resources that is the same type as the requested one. 09/17/2014, Bing Li
        Map<String, Resource> resources;
        // The hash key of the resource that has the longest lifetime. 09/17/2014, Bing Li
        String oldestResourceKey;
        // The count of busy resources. 09/17/2014, Bing Li
        int busyResourceCount;
        // The count of idle resources 09/17/2014, Bing Li
        int idleResourceCount;
        boolean isSourceAvailable = false;

        // Since the procedure to get a particular resource might be blocked when the upper limit of the
pool is reached, it is required to keep a notify/wait mechanism to guarantee the procedure smooth.
09/17/2014, Bing Li
        while (!this.collaborator.isShutdown())
        {
          this.rscLock.lock();
```

```
            try
            {
              // Check whether the type of the resource is existed in the source map. 09/17/2014, Bing Li
              if (this.sourceMap.containsKey(objectKey))
              {
                // Set the flag that the source is available. 11/03/2014, Bing Li
                isSourceAvailable = true;
                // Retrieve the source. 11/03/2014, Bing Li
                src = this.sourceMap.get(objectKey);
                // Initialize the value of busyResourceCount. 09/17/2014, Bing Li
                busyResourceCount = 0;
                // Initialize the value of busyResourceCount. 09/17/2014, Bing Li
                idleResourceCount = 0;

                /*
                 * Retrieve the resource from the idle map first. 09/17/2014, Bing Li
                 */

                // Check whether the idle map contains the type of the resource. 09/17/2014, Bing Li
                if (this.idleMap.containsKey(src.getObjectKey()))
                {
                  // The type of idle resources which are the candidates to be returned. 09/17/2014, Bing Li
                  resources = this.idleMap.get(src.getObjectKey());
                  // Get the hash key of the resource that is idle longer than any others from the idle
resources. 09/17/2014, Bing Li
                  oldestResourceKey = CollectionSorter.minValueKey(resources);
                  // Check whether the hash key is valid. 09/17/2014, Bing Li
                  if (oldestResourceKey != null)
                  {
                    // Get the resource that is idle than any others by its hash key. 09/17/2014, Bing Li
                    rsc = resources.get(oldestResourceKey);

                    // Set the accessed time of the resource that is idle than any others to the current
moment since the resource will be used after it is returned. Now it becomes a busy one. 09/17/2014,
Bing Li
                    rsc.setAccessedTime();

                    // Set the accessed time of the resource that is idle than any others to the current
moment since the resource will be used after it is returned. Now it becomes a busy one. 09/17/2014,
Bing Li
                    this.idleMap.get(src.getObjectKey()).remove(oldestResourceKey);
                    // Check whether the type of resources is empty in the idle map. 09/17/2014, Bing Li
                    if (this.idleMap.get(src.getObjectKey()).size() <= 0)
                    {
                      // Remove the type of resources from the idle map if no any specific resources of the
type is in the idle map. 09/17/2014, Bing Li
                      this.idleMap.remove(src.getObjectKey());
                    }

                    // Check whether the type of resources is available in the busy map. 09/17/2014, Bing Li
                    if (!this.busyMap.containsKey(rsc.getObjectKey()))
                    {
                      // If the type of resources is not existed in the busy map, add the type and the
particular resource. 09/17/2014, Bing Li
                      this.busyMap.put(rsc.getObjectKey(), new ConcurrentHashMap<String,
Resource>());
                      this.busyMap.get(rsc.getObjectKey()).put(oldestResourceKey, rsc);
                    }
                    else
                    {
                      // If the type of resources is not existed in the busy map, add the type and the
particular resource. 09/17/2014, Bing Li
                      this.busyMap.get(rsc.getObjectKey()).put(oldestResourceKey, rsc);
                    }
                  }
                }

                /*
```

```
             * If the idle map does not contain the requested resource, it is necessary to initialize a new
one. 09/17/2014, Bing Li
             */

            // Check whether the resource is available when retrieving from the idle map. 10/12/2014,
Bing Li
            if (rsc == null)
            {
              // Calculate the exact count of all of the busy resources. The value is used to check
whether the upper limit of the pool is reached. 09/17/2014, Bing Li
              for (Map<String, Resource> resMap : this.busyMap.values())
              {
                busyResourceCount += resMap.size();
              }

              // Calculate the exact count of all of the idle resources. The value is used to check whether
the upper limit of the pool is reached. 09/17/2014, Bing Li
              for (Map<String, Resource> resMap : this.idleMap.values())
              {
                idleResourceCount += resMap.size();
              }

              // Check whether the sum of the count of busy and idle resources reach the upper limit of
the pool. 09/17/2014, Bing Li
              if (busyResourceCount + idleResourceCount < this.poolSize)
              {
                // If the upper limit of the pool is not reached, it is time to create an instance by its
source. 09/17/2014, Bing Li
                rsc = this.creator.createResourceInstance(src);
                // Check whether the newly created instance is valid. 09/17/2014, Bing Li
                if (rsc != null)
                {
                  // Check whether the type of the resource is available in the busy map. 09/17/2014,
Bing Li
                  if (!this.busyMap.containsKey(rsc.getObjectKey()))
                  {
                    // If the type of the resource is not available in the busy map, add the type and the
resource into it. 09/17/2014, Bing Li
                    this.busyMap.put(rsc.getObjectKey(), new ConcurrentHashMap<String,
Resource>());
                    this.busyMap.get(rsc.getObjectKey()).put(rsc.getHashKey(), rsc);
                  }
                  else
                  {
                    // If the type of the resource is available in the busy map, add the resource into it.
09/17/2014, Bing Li
                    this.busyMap.get(rsc.getObjectKey()).put(rsc.getHashKey(), rsc);
                  }
                }
              }
              // Return the resource to the invoker. 09/17/2014, Bing Li
              return rsc;
            }
          }
          finally
          {
            this.rscLock.unlock();
          }

          // If the source is available, it is expeced to wait for idle resource. 11/03/2014, Bing Li
          if (isSourceAvailable)
          {
            /*
             * If no such resources in the idle map and the upper limit of the pool is reached, the thread
that invokes the method has to wait for future possible updates. The possible updates include resource
disposals and idle resources being available. 09/17/2014, Bing Li
             */
```

```
            try
            {
              // Check whether the pool is shutdown. 09/17/2014, Bing Li
              if (!this.collaborator.isShutdown())
              {
                // If the pool is not shutdown, it is time to wait for some time. After that, the above
procedure is repeated if the pool is not shutdown. 09/17/2014, Bing Li
                this.collaborator.holdOn(UtilConfig.ONE_SECOND);
              }
            }
            catch (InterruptedException e)
            {
              e.printStackTrace();
            }
          }
        }
      }
      // Return null if the key is invalid. 09/17/2014, Bing Li
      return null;
    }
}
```

- **RetrievableIdleChecker**

```
package com.greatfree.reuse;

import java.util.TimerTask;

import com.greatfree.util.FreeObject;

/*
 * The class runs periodically to check whether a resource being managed in a resource pool, such as RetrievablePool, is idle enough time. If so, it collects the resource. 08/26/2014, Bing Li
 */

// Created: 08/26/2014, Bing Li
public class RetrievableIdleChecker<Source extends FreeObject, Resource extends FreeObject, ResourceCreator extends Creatable<Source, Resource>, ResourceDisposer extends Disposable<Resource>> extends TimerTask
{
  // The resource pool contains all of the idle resources to be checked. 08/26/2014, Bing Li
  private RetrievablePool<Source, Resource, ResourceCreator, ResourceDisposer> pool;

  /*
   * Initialize the checker. 08/26/2014, Bing Li
   */
  public RetrievableIdleChecker(RetrievablePool<Source, Resource, ResourceCreator, ResourceDisposer> pool)
  {
    this.pool = pool;
  }

  /*
   * The timer must run asynchronously to check whether the resources are idle enough time. 08/26/2014, Bing Li
   */
  @Override
  public void run()
  {
    this.pool.checkIdle();
  }
}
```

- **QueuedPool**

```
package com.greatfree.reuse;

import java.io.IOException;
import java.util.Calendar;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.Queue;
import java.util.Set;
import java.util.Timer;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.locks.ReentrantLock;

import com.greatfree.concurrency.Collaborator;
import com.greatfree.util.FreeObject;
import com.greatfree.util.Time;
import com.greatfree.util.Tools;
import com.greatfree.util.UtilConfig;

/*
 * The pool aims to manage resources that are scheduled by their idle lengths. The one that is idle longer has the higher probability to be reused than the one that is idle shorter. 11/03/2014, Bing Li
 */

// Created: 11/03/2014, Bing Li
public class QueuedPool<Resource extends FreeObject, Creator extends Creatable<String, Resource>, Disposer extends Disposable<Resource>>
{
  // The map keeps all of the busy resources by their hash key. 11/03/2014, Bing Li
  private Map<String, Resource> busyMap;
  // The map keeps all of the idle resources in different queues by their type keys. Each type of idle resources has a dedicated queue. 11/03/2014, Bing Li
  private Map<String, Queue<Resource>> idleMap;
  // The lock is used to reach the goal of thread-safety. 11/03/2014, Bing Li
  private ReentrantLock rscLock;
  // The timer to be used to manage the period and schedule the task to check periodically whether idle resources are idle long enough. 11/03/2014, Bing Li
  private Timer checkTimer;
  // The instance of QueuedIdleChecker invokes the method of checkIdle() periodically and concurrently. 11/03/2014, Bing Li
  private QueuedIdleChecker<Resource, Creator, Disposer> idleChecker;
  // Declare the maximum idle time length. 11/03/2014, Bing Li
  private long maxIdleTime;
  // Declare the size of the pool. 11/03/2014, Bing Li
  private int poolSize;
  // The creator is used to create the instance of the resources managed by the pool. 11/03/2014, Bing Li
  private Creator creator;
  // The disposer is used to dispose the instance of the resources managed by the pool. 11/03/2014, Bing Li
  private Disposer disposer;
  // The collaborator is used to implement the mechanism of notify/wait to coordinate resource management for multiple threads. 11/03/2014, Bing Li
  private Collaborator collaborator;

  /*
   * Initialize. 11/03/2014, Bing Li
   */
  public QueuedPool(int poolSize, Creator creator, Disposer disposer)
  {
    this.busyMap = new HashMap<String, Resource>();
    this.idleMap = new HashMap<String, Queue<Resource>>();
    this.rscLock = new ReentrantLock();
    this.poolSize = poolSize;
```

```
    this.creator = creator;
    this.disposer = disposer;
    this.collaborator = new Collaborator();
  }

  /*
   * Shutdown the pool. 11/03/2014, Bing Li
   */
  public void shutdown() throws IOException
  {
    // Set the shutdown flag to be true. Thus, the loop in the method of get() can be terminated.
11/03/2014, Bing Li
    this.collaborator.setShutdown();
    // Notify the method of get() to terminate and all of the threads that are waiting for the resources are
unblocked. 11/03/2014, Bing Li
    this.collaborator.signalAll();

    // Dispose all of the busy resources. It might lose data if no relevant management approaches are
adopted. 11/03/2014, Bing Li
    for (Resource resource : this.busyMap.values())
    {
      this.disposer.dispose(resource);
    }
    this.busyMap.clear();

    // Dispose all of the idle resources. It might lose data if no relevant management approaches are
adopted. 11/03/2014, Bing Li
    Set<String> tKeys = this.idleMap.keySet();
    Resource resource;
    for (String key : tKeys)
    {
      while (this.idleMap.get(key).size() > 0)
      {
        resource = this.idleMap.get(key).poll();
        this.disposer.dispose(resource);
      }
    }
    this.idleMap.clear();

    // Terminate the idle state checker that periodically runs. For it is possible that the checker is not
initialized in some cases, it needs to check whether it is null or not. 11/03/2014, Bing Li
    if (this.idleChecker != null)
    {
      this.idleChecker.cancel();
    }
    // Terminate the timer that manages the period to run the idle state checker. For it is possible that the
timer is not initialized in some cases, it needs to check whether it is null or not. 11/03/2014, Bing Li
    if (this.checkTimer != UtilConfig.NO_TIMER)
    {
      this.checkTimer.cancel();
    }
  }

  /*
   * Dispose a resource explicitly when it is not needed. Usually, it is not invoked by the pool but by the
threads that consume the resource. It happens when it is confirmed that the resource is never needed in
a specific case. 11/03/2014, Bing Li
   */
  public void dispose(Resource rsc) throws IOException
  {
    this.rscLock.lock();
    // Check whether the resource is saved in the busy map. 11/03/2014, Bing Li
    if (this.busyMap.containsKey(rsc.getHashKey()))
    {
      // If the resource is in the busy map, remove it. 11/03/2014, Bing Li
      this.busyMap.remove(rsc.getHashKey());
      // Dispose the resource. 11/03/2014, Bing Li
      this.disposer.dispose(rsc);
```

```
    }
    else
    {
      // If the resource is not in the idle map, it is necessary to check whether it is saved in one of
queues of idle map. 11/03/2014, Bing Li
      if (this.idleMap.containsKey(rsc.getObjectKey()))
      {
        // If the resource type is in one of queues of the idle map, declare an instance of resource.
11/03/2014, Bing Li
        Resource idleRsc;
        // Get the first key of the first resource in the queue. 11/03/2014, Bing Li
        String firstKey = this.idleMap.get(rsc.getObjectKey()).peek().getHashKey();
        // Check whether the key of the first resource of the queue is equal to the hash key of the
resource to be disposed. 11/03/2014, Bing Li
        if (firstKey.equals(rsc.getHashKey()))
        {
          // If the above condition is fulfilled, it denotes that the resource is identical to the one to be
disposed. Thus, remove it from the queue. 11/03/2014, Bing Li
          this.idleMap.get(rsc.getObjectKey()).poll();
          // Dispose the resource. 11/03/2014, Bing Li
          this.disposer.dispose(rsc);
        }
        else
        {
          // If the above condition is not fulfilled, it is necessary to check whether the resource is queued
in the later position of the queue. 11/03/2014, Bing Li
          do
          {
            // Get the current first resource out of the queue. 11/03/2014, Bing Li
            idleRsc = this.idleMap.get(rsc.getObjectKey()).poll();
            // Check whether the resource removed from the queue is identical to the one to be
disposed. 11/03/2014, Bing Li
            if (idleRsc.getHashKey().equals(rsc.getHashKey()))
            {
              // If the above condition is fulfilled, it denotes that the resource is identical to the one to be
disposed. Thus, remove it from the queue. 11/03/2014, Bing Li
              this.disposer.dispose(rsc);
            }
            else
            {
              // If the above condition is not fulfilled, enqueue the resource just removed from the queue.
11/03/2014, Bing Li
              this.idleMap.get(rsc.getObjectKey()).add(idleRsc);
            }
          }
          // Check whether the resources in the queue are placed in the original position. If not, the loop
continues. If so, the loop needs to quit. 11/03/2014, Bing Li
          while (!firstKey.equals(this.idleMap.get(rsc.getObjectKey()).peek().getHashKey()));
        }
      }
      else
      {
        // If resource to be removed is neither in the busy map nor the idle map, dispose it directly.
11/03/2014, Bing Li
        this.disposer.dispose(rsc);
      }
    }
    this.rscLock.unlock();
    // Notify the thread that is blocked for the maximum size of the pool is reached. 11/03/2014, Bing Li
    this.collaborator.signal();
  }

  /*
   * The method creates an instance without any management. That is used only when the pool is
shutdown. It aims to avoid inconsistency when removing data by FileManager.RemoveFiles().
11/03/2014, Bing Li
   */
  public Resource create(String rscType) throws IOException, InterruptedException
```

```
  {
    return this.creator.createResourceInstance(rscType);
  }

  /*
   * Initialize the idle state checker and the timer to manage idle resources periodically when needed.
This method is not always indispensable. 11/03/2014, Bing Li
   */
  public void setIdleChecker(long idleCheckDelay, long idleCheckPeriod, long maxIdleTime)
  {
    // Initialize the timer. 11/03/2014, Bing Li
    this.checkTimer = new Timer();
    // Initialize the checker. 11/03/2014, Bing Li
    this.idleChecker = new QueuedIdleChecker<Resource, Creator, Disposer>(this);
    // Schedule the task of checking idle states of resources. 11/03/2014, Bing Li
    this.checkTimer.schedule(this.idleChecker, idleCheckDelay, idleCheckPeriod);
    // Set the value of maximum idle time. 11/03/2014, Bing Li
    this.maxIdleTime = maxIdleTime;
  }

  /*
   * Check the idle resources. The method is called back by the IdleChecker. 11/03/2014, Bing Li
   */
  public void checkIdle() throws IOException
  {
    this.rscLock.lock();
    Resource rsc;
    // Get the queue keys of the idle resources. 11/03/2014, Bing Li
    Set<String> tKeys = this.idleMap.keySet();
    // Get the current time which is used to calculate the idle time length. 11/03/2014, Bing Li
    Date currentTime = Calendar.getInstance().getTime();
    // Check idle resources of each queue. 11/03/2014, Bing Li
    for (String key : tKeys)
    {
      // Check whether a specific queue is empty. If not, the loop continues to detect whether a resource
is idle long enough. 11/03/2014, Bing Li
      while (this.idleMap.get(key).size() > 0)
      {
        // Check the first resource, but keep in the queue. 11/03/2014, Bing Li
        rsc = this.idleMap.get(key).peek();
        // Calculate the idle time length and compare it with the maximum idle time. 11/03/2014, Bing Li
        if (Time.getTimespanInMilliSecond(currentTime, rsc.getAccessedTime()) > this.maxIdleTime)
        {
          // Dequeue the resource that is idle long enough. 11/03/2014, Bing Li
          rsc = this.idleMap.get(key).poll();
          // Dispose the resource. 11/03/2014, Bing Li
          this.disposer.dispose(rsc);
        }
        else
        {
          // Since the first resource of the queue is the one that is idle longer than any others in the
queue, it denotes that all of the resources in the queue are not idle longer than the upper limit.
11/03/2014, Bing Li
          break;
        }
      }
    }
    this.rscLock.unlock();
    // Notify the blocked thread that the size of the assigned resources is lowered such that it is time for
it to get a resource it needs. A bunch of resources might be disposed during the procedure. It is
reasonable to signal all rather than signal a single waiting thread. 11/03/2014, Bing Li
    this.collaborator.signalAll();
  }

  /*
   * Collect a resource. When the resource finishes its task, the method is invoked by the corresponding
thread such that the resource can be reused. 11/03/2014, Bing Li
   */
```

```
  public void collect(Resource rsc)
  {
    this.rscLock.lock();
    // Check whether the resource is contained in the busy map. 11/03/2014, Bing Li
    if (this.busyMap.containsKey(rsc.getHashKey()))
    {
      // If it is, it is required to remove it from the busy map. 11/03/2014, Bing Li
      this.busyMap.remove(rsc.getHashKey());
    }
    // Set the idle starting moment, which is used to calculate whether the resource is idle long enough.
11/03/2014, Bing Li
    rsc.setAccessedTime();
    // Check whether a queue is existed for the particular resource in the idle map. 11/03/2014, Bing Li
    if (!this.idleMap.containsKey(rsc.getObjectKey()))
    {
      // If the idle map does not consist of the queue for the type of the resource, it is required to create a
new queue for it. 11/03/2014, Bing Li
      this.idleMap.put(rsc.getObjectKey(), new LinkedBlockingQueue<Resource>());
      this.idleMap.get(rsc.getObjectKey()).add(rsc);
    }
    else
    {
      // If the idle map consists of the queue for the type of the resource, just enqueue it into the queue.
11/03/2014, Bing Li
      this.idleMap.get(rsc.getObjectKey()).add(rsc);
    }
    this.rscLock.unlock();
    // Notify the thread that is blocked for the maximum size of the pool is reached. 11/03/2014, Bing Li
    this.collaborator.signal();
  }

  /*
   * Get the resource by its type name. 09/17/2014, Bing Li
   */
  public Resource get(String rscType) throws InstantiationException, IllegalAccessException,
IOException, InterruptedException
  {
    // Declare the resource to be returned to the thread that needs it. 11/03/2014, Bing Li
    Resource rsc = null;
    // Declare the idle size of all of the idle resources. 11/03/2014, Bing Li
    int idleSize;
    // Get the key of the resource type. 11/03/2014, Bing Li
    String queueKey = Tools.getHash(rscType);
    // The key of the longest idle queue. 11/03/2014, Bing Li
    String longestQueueKey = UtilConfig.NO_QUEUE_KEY;
    // The size of the longest idle queue. 11/03/2014, Bing Li
    int longestQueueSize;

    // Since the procedure to get a particular resource might be blocked when the upper limit of the pool
is reached, it is required to keep a notify/wait mechanism to guarantee the procedure smooth.
09/17/2014, Bing Li
    while (!this.collaborator.isShutdown())
    {
      this.rscLock.lock();
      try
      {
        // Check whether the idle map consists of the queue for the specific resource. 11/03/2014, Bing
Li
        if (this.idleMap.containsKey(queueKey))
        {
          // Check whether the queue for the resource is empty or not. 11/03/2014, Bing Li
          if (!this.idleMap.get(queueKey).isEmpty())
          {
            // If the queue is not empty, it denotes that idle resources of the specific type are available.
Then, get the first one out from the queue. 11/03/2014, Bing Li
            rsc = this.idleMap.get(queueKey).poll();
            // Check whether it is null. 11/03/2014, Bing Li
            if (rsc != null)
```

```
                {
                    // Set the busy starting time stamp. 11/03/2014, Bing Li
                    rsc.setAccessedTime();
                    // Put it into the busy map. 11/03/2014, Bing Li
                    this.busyMap.put(rsc.getHashKey(), rsc);
                    // Return the resource. 11/03/2014, Bing Li
                    return rsc;
                }
            }
        }

        /*
         * If the idle map does not contain the resource, it is necessary to create it if the pool size does
not reach the upper limit. 11/03/2014, Bing Li
         */

        // Initialize the idle resource size. 11/03/2014, Bing Li
        idleSize = 0;
        // Initialize the size of the longest idle queue. 11/03/2014, Bing Li
        longestQueueSize = 0;
        // Initialize the key of the longest idle queue. 11/03/2014, Bing Li
        longestQueueKey = UtilConfig.NO_QUEUE_KEY;
        // Calculate the count of all of the idle resources and the longest idle queue. 11/03/2014, Bing Li
        for (Map.Entry<String, Queue<Resource>> rscEntry : this.idleMap.entrySet())
        {
            // Calculate the count of all of the idle resources. 11/03/2014, Bing Li
            idleSize += rscEntry.getValue().size();
            // Calculate the longest idle queue size. 11/03/2014, Bing Li
            if (longestQueueSize < rscEntry.getValue().size())
            {
                // Get the size of the current longest queue. 11/03/2014, Bing Li
                longestQueueSize = rscEntry.getValue().size();
                // Get the key of the current longest queue. 11/03/2014, Bing Li
                longestQueueKey = rscEntry.getKey();
            }
        }

        // Check whether the sum of the count of busy and idle resources reach the upper limit of the
pool. 11/03/2014, Bing Li
        if (this.busyMap.size() + idleSize < this.poolSize)
        {
            // If the upper limit of the pool is not reached, create an instance of the resource. 11/03/2014,
Bing Li
            rsc = this.creator.createResourceInstance(rscType);
            // Put the newly created resource into the busy map. 11/03/2014, Bing Li
            this.busyMap.put(rsc.getHashKey(), rsc);
            // Return the newly created resource to the invoking thread. 11/03/2014, Bing Li
            return rsc;
        }
        else
        {
            // If the specified queue does not exist and the max size of pool is reached, it is proper to
dispose an idle one in the longest queue. 11/03/2014, Bing Li
            if (!longestQueueKey.equals(UtilConfig.NO_QUEUE_KEY))
            {
                // Get an idle resource out from the longest queue. 11/03/2014, Bing Li
                rsc = this.idleMap.get(longestQueueKey).poll();
                // Dispose the first resource of the longest idle queue. 11/03/2014, Bing Li
                this.disposer.dispose(rsc);
                // Create a new resource. 11/03/2014, Bing Li
                rsc = this.creator.createResourceInstance(rscType);
                // Put the newly created resource into the busy map. 11/03/2014, Bing Li
                this.busyMap.put(rsc.getHashKey(), rsc);
                // Return the newly created resource to the invoking thread. 11/03/2014, Bing Li
                return rsc;
            }
        }
    }
```

```
        finally
        {
          this.rscLock.unlock();
        }

        /*
         * If no such resources in the idle map and the upper limit of the pool is reached, the thread that
invokes the method has to wait for future possible updates. The possible updates include resource
disposals and idle resources being available. 11/03/2014, Bing Li
         */

        // Check whether the pool is shutdown. 11/03/2014, Bing Li
        if (!this.collaborator.isShutdown())
        {
          try
          {
            // If the pool is not shutdown, it is time to wait for some time. After that, the above procedure is
repeated if the pool is not shutdown. 11/03/2014, Bing Li
            this.collaborator.holdOn(UtilConfig.ONE_SECOND);
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
        }
      }
      return null;
    }
}
```

- **QueuedIdleChecker**

```
package com.greatfree.reuse;

import java.io.IOException;
import java.util.TimerTask;

import com.greatfree.util.FreeObject;

/*
 * The class aims to check periodically whether an idle resource is idle long enough to be disposed.
11/03/2014, Bing Li
 */

// Created: 11/03/2014, Bing Li
public class QueuedIdleChecker<Resource extends FreeObject, ResourceCreator extends
Creatable<String, Resource>, ResourceDisposer extends Disposable<Resource>> extends
TimerTask
{
  // Declare the pool. 11/03/2014, Bing Li
  private QueuedPool<Resource, ResourceCreator, ResourceDisposer> pool;

  /*
   * Initialize the pool, which is assigned from the outside. The method of idle checking is called
asynchronously by the class. 11/03/2014, Bing Li
   */
  public QueuedIdleChecker(QueuedPool<Resource, ResourceCreator, ResourceDisposer> pool)
  {
    this.pool = pool;
  }

  @Override
  public void run()
  {
    try
    {
      // Call the idle checking method. 11/03/2014, Bing Li
      this.pool.checkIdle();
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
  }
}
```

- **ResourcePool**

```
package com.greatfree.reuse;

import java.util.Calendar;
import java.util.Date;
import java.util.Map;
import java.util.Queue;
import java.util.Set;
import java.util.Timer;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.locks.ReentrantLock;

import com.greatfree.concurrency.Collaborator;
import com.greatfree.util.HashFreeObject;
import com.greatfree.util.Time;
import com.greatfree.util.UtilConfig;

/*
 * The pool aims to manage resources that are scheduled by their idle lengths. The one that is idle longer has the higher probability to be reused than the one that is idle shorter. Different from QueuedPool, this pool does not care about the type of resources. It assumes that all of resources in the pool are classified in the same type. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public class ResourcePool<Source, Resource extends HashFreeObject, Creator extends HashCreatable<Source, Resource>, Disposer extends HashDisposable<Resource>>
{
  // The map keeps all of the busy resources by their hash key. 11/26/2014, Bing Li
  private Map<String, Resource> busyMap;
  // The map keeps all of the idle resources in the same queue. Since usually they are the same type, it is unnecessary to differ them by their respective queues. 11/26/2014, Bing Li
  private Queue<Resource> idleQueue;
  // The lock is used to reach the goal of thread-safety. 11/26/2014, Bing Li
  private ReentrantLock rscLock;
  // The timer to be used to manage the period and schedule the task to check periodically whether idle resources are idle long enough. 11/26/2014, Bing Li
  private Timer checkTimer;
  // The instance of IdleChecker invokes the method of checkIdle() periodically and concurrently. 11/26/2014, Bing Li
  private IdleChecker<Source, Resource, Creator, Disposer> idleChecker;
  // Declare the maximum idle time length. 11/26/2014, Bing Li
  private long maxIdleTime;
  // Declare the size of the pool. 11/26/2014, Bing Li
  private int poolSize;
  // The creator is used to create the instance of the resources managed by the pool. 11/26/2014, Bing Li
  private Creator creator;
  // The disposer is used to dispose the instance of the resources managed by the pool. 11/26/2014, Bing Li
  private Disposer disposer;
  // The collaborator is used to implement the mechanism of notify/wait to coordinate resource management for multiple threads. 11/26/2014, Bing Li
  private Collaborator collaborator;
  // When resources are not available, it is required to wait for some time. This is the time length to wait. 11/26/2014, Bing Li
  private long waitTime;

  /*
   * Initialize. 11/26/2014, Bing Li
   */
  public ResourcePool(int poolSize, Creator creator, Disposer disposer, long waitTime)
  {
    this.busyMap = new ConcurrentHashMap<String, Resource>();
```

```
    this.idleQueue = new LinkedBlockingQueue<Resource>();
    this.rscLock = new ReentrantLock();

    this.poolSize = poolSize;
    this.creator = creator;
    this.checkTimer = UtilConfig.NO_TIMER;
    this.disposer = disposer;
    this.collaborator = new Collaborator();
    this.waitTime = waitTime;
  }

  /*
   * Shutdown the pool. 11/26/2014, Bing Li
   */
  public void shutdown()
  {
    // Set the shutdown flag to be true. Thus, the loop in the method of get() can be terminated.
11/26/2014, Bing Li
    this.collaborator.setShutdown();
    // Notify the method of get() to terminate and all of the threads that are waiting for the resources are
unblocked. 11/26/2014, Bing Li
    this.collaborator.signalAll();

    // Dispose all of the busy resources. It might lose data if no relevant management approaches are
adopted. 11/26/2014, Bing Li
    for (Resource t : this.busyMap.values())
    {
      this.disposer.dispose(t);
    }
    this.busyMap.clear();

    // Dispose all of the idle resources. It might lose data if no relevant management approaches are
adopted. 11/26/2014, Bing Li
    Resource t;
    while (this.idleQueue.size() > 0)
    {
      t = this.idleQueue.poll();
      this.disposer.dispose(t);
    }
    this.idleQueue.clear();

    // Terminate the idle state checker that periodically runs. For it is possible that the checker is not
initialized in some cases, it needs to check whether it is null or not. 11/26/2014, Bing Li
    if (this.idleChecker != null)
    {
      this.idleChecker.cancel();
    }
    // Terminate the timer that manages the period to run the idle state checker. For it is possible that the
timer is not initialized in some cases, it needs to check whether it is null or not. 11/26/2014, Bing Li
    if (this.checkTimer != UtilConfig.NO_TIMER)
    {
      this.checkTimer.cancel();
    }
  }

  /*
   * Initialize the idle state checker and the timer to manage idle resources periodically when needed.
This method is not always indispensable. 11/26/2014, Bing Li
   */
  public void setIdleChecker(long idleCheckDelay, long idleCheckPeriod, long maxIdleTime)
  {
    // Initialize the timer. 11/26/2014, Bing Li
    this.checkTimer = new Timer();
    // Initialize the checker. 11/26/2014, Bing Li
    this.idleChecker = new IdleChecker<Source, Resource, Creator, Disposer>(this);
    // Schedule the task of checking idle states of resources. 11/26/2014, Bing Li
    this.checkTimer.schedule(this.idleChecker, idleCheckDelay, idleCheckPeriod);
    // Set the value of maximum idle time. 11/26/2014, Bing Li
```

```
    this.maxIdleTime = maxIdleTime;
  }

  /*
   * Check the idle resources. The method is called back by the IdleChecker. 11/26/2014, Bing Li
   */
  public void checkIdle()
  {
    Resource t;
    // Get the current time which is used to calculate the idle time length. 11/26/2014, Bing Li
    Date currentTime = Calendar.getInstance().getTime();
    this.rscLock.lock();
    // Scan all of the idle resources. 11/26/2014, Bing Li
    while (this.idleQueue.size() > 0)
    {
      // Peek the first resource. If it is not idle long enough, the checking can be returned because the
first one must be idle for the most long time. 11/26/2014, Bing Li
      t = this.idleQueue.peek();
      // Estimate the idle time and check whether it is long enough. 11/26/2014, Bing Li
      if (Time.getTimespanInMilliSecond(currentTime, t.getAccessedTime()) > this.maxIdleTime)
      {
        // If it is idle long enough, it is dequeued from the queue. 11/26/2014, Bing Li
        t = this.idleQueue.poll();
        // Dispose the resource. 11/26/2014, Bing Li
        this.disposer.dispose(t);
      }
      else
      {
        // Since the first resource of the queue is the one that is idle longer than any others in the queue,
it denotes that all of the resources in the queue are not idle longer than the upper limit. 11/26/2014, Bing
Li
        break;
      }
    }
    this.rscLock.unlock();
    // Notify the blocked thread that the size of the assigned resources is lowered such that it is time for
it to get a resource it needs. A bunch of resources might be disposed during the procedure. It is
reasonable to signal all rather than signal a single waiting thread. 11/26/2014, Bing Li
    this.collaborator.signalAll();
  }

  /*
   * Dispose a resource explicitly when it is not needed. Usually, it is not invoked by the pool but by the
threads that consume the resource. It happens when it is confirmed that the resource is never needed in
a specific case. 11/26/2014, Bing Li
   */
  public void dispose(Resource t)
  {
    // Check whether the resource is valid. 11/26/2014, Bing Li
    if (t != null)
    {
      this.rscLock.lock();
      // Check whether the busy collection contains the resource to be disposed. 11/26/2014, Bing Li
      if (this.busyMap.containsKey(t.getHashKey()))
      {
        // If the resource exists in the collection, remove the resource from the collection. 11/26/2014,
Bing Li
        this.busyMap.remove(t.getHashKey());
      }
      // Dispose the resource. 11/26/2014, Bing Li
      this.disposer.dispose(t);
      this.rscLock.unlock();
      // Notify the thread that is blocked for the maximum size of the pool is reached. 11/26/2014, Bing Li
      this.collaborator.signal();
    }
  }

  /*
```

```
   * Check whether a specific resource is busy. 11/26/2014, Bing Li
   */
  public boolean isBusy(String resourceKey)
  {
    this.rscLock.lock();
    try
    {
      return this.busyMap.containsKey(resourceKey);
    }
    finally
    {
      this.rscLock.unlock();
    }
  }

  /*
   * Collect a resource. When the resource finishes its task, the method is invoked by the corresponding
thread such that the resource can be reused. 11/26/2014, Bing Li
   */
  public void collect(Resource t)
  {
    // Check whether the resource is valid. 11/26/2014, Bing Li
    if (t != null)
    {
      this.rscLock.lock();
      // Check whether the resource is contained in the busy collection. 11/26/2014, Bing Li
      if (this.busyMap.containsKey(t.getHashKey()))
      {
        // Remove it from the busy collection. 11/26/2014, Bing Li
        this.busyMap.remove(t.getHashKey());
      }
      // Set the accessed time stamp for the resource. The time stamp is the idle starting moment of the
resource. It is used to calculate whether the resource is idle for long enough. 11/26/2014, Bing Li
      t.setAccessedTime();
      // Enqueue the idle resource into the idle queue. 11/26/2014, Bing Li
      this.idleQueue.add(t);
      this.rscLock.unlock();
      // Notify the thread that is blocked for the maximum size of the pool is reached. 11/26/2014, Bing Li
      this.collaborator.signal();
    }
  }

  /*
   * Expose all the hash keys of the busy resources. 11/26/2014, Bing Li
   */
  public Set<String> getBusyKeys()
  {
    this.rscLock.lock();
    try
    {
      return this.busyMap.keySet();
    }
    finally
    {
      this.rscLock.unlock();
    }
  }

  /*
   * Get the busy resource from the busy collection. 11/26/2014, Bing Li
   */
  public Resource getBusyResource(String key)
  {
    this.rscLock.lock();
    try
    {
      if (this.busyMap.containsKey(key))
      {
```

```
            return this.busyMap.get(key);
          }
          return null;
        }
        finally
        {
          this.rscLock.unlock();
        }
      }

      /*
       * Get the resource by its source, which contains the arguments to create the resource. 11/26/2014,
Bing Li
       */
      public Resource get(Source source) throws InstantiationException, IllegalAccessException
      {
        Resource rsc = null;
        // Since the procedure to get a particular resource might be blocked when the upper limit of the pool
is reached, it is required to keep a notify/wait mechanism to guarantee the procedure smooth.
11/26/2014, Bing Li
        while (!this.collaborator.isShutdown())
        {
          this.rscLock.lock();
          try
          {
            // Check whether the idle queue is empty. 11/26/2014, Bing Li
            if (!this.idleQueue.isEmpty())
            {
              // Dequeue the idle resource. 11/26/2014, Bing Li
              rsc = this.idleQueue.poll();
              // Check whether the resource is valid. 11/26/2014, Bing Li
              if (rsc != null)
              {
                // Set the busy starting time stamp. 11/26/2014, Bing Li
                rsc.setAccessedTime();
                // Put it into the busy map. 11/26/2014, Bing Li
                this.busyMap.put(rsc.getHashKey(), rsc);
              }
            }

            // Check whether the resource is valid. 11/26/2014, Bing Li
            if (rsc == null)
            {
              /*
               * If the resource is invalid, it is necessary to create a new instance of the resource.
11/26/2014, Bing Li
               */

              // Check whether the count of the total existing resources exceeds the upper limit. 11/26/2014,
Bing Li
              if (this.busyMap.size() + this.idleQueue.size() < this.poolSize)
              {
                // Create a new instance of the resource. 11/26/2014, Bing Li
                rsc = this.creator.createResourceInstance(source);
                // Put the new instance of resource into the busy collection. 11/26/2014, Bing Li
                this.busyMap.put(rsc.getHashKey(), rsc);
                // Return the new instance of the resource. 11/26/2014, Bing Li
                return rsc;
              }
            }
            else
            {
              // Return the instance of the idle resource. 11/26/2014, Bing Li
              return rsc;
            }
          }
          finally
          {
```

```
            this.rscLock.unlock();
        }

        /*
         * If no such resources in the idle map and the upper limit of the pool is reached, the thread that
invokes the method has to wait for future possible updates. The possible updates include resource
disposals and idle resources being available. 11/26/2014, Bing Li
         */

        // Check whether the pool is shutdown. 11/26/2014, Bing Li
        if (!this.collaborator.isShutdown())
        {
          try
          {
            // If the pool is not shutdown, it is time to wait for some time. After that, the above procedure is
repeated if the pool is not shutdown. 11/26/2014, Bing Li
            this.collaborator.holdOn(this.waitTime);
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
        }
      }
      return null;
    }
}
```

- **IdleChecker**

```
package com.greatfree.reuse;

import java.util.TimerTask;

import com.greatfree.util.HashFreeObject;

/*
 * The idle checker works with the ResourcePool to check the idle states of resources. The checking
method is invoked by calling back of the ResourcePool. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public class IdleChecker<Source, Resource extends HashFreeObject, ResourceCreator extends
HashCreatable<Source, Resource>, ResourceDisposer extends HashDisposable<Resource>>
extends TimerTask
{
  // The resource pool which contains resources to be checked. 11/26/2014, Bing Li
  private ResourcePool<Source, Resource, ResourceCreator, ResourceDisposer> pool;

  /*
   * Initialize the idle checker. 11/26/2014, Bing Li
   */
  public IdleChecker(ResourcePool<Source, Resource, ResourceCreator, ResourceDisposer> pool)
  {
    this.pool = pool;
  }

  /*
   * Check the idle states of the resources concurrently.
   */
  @Override
  public void run()
  {
    this.pool.checkIdle();
  }
}
```

- **InteractiveDispatcher**

```
package com.greatfree.reuse;

import java.util.Calendar;
import java.util.Map;
import java.util.Set;
import java.util.Timer;
import java.util.concurrent.ConcurrentHashMap;

import com.greatfree.concurrency.InteractiveQueue;
import com.greatfree.concurrency.InteractiveThreadCreatable;
import com.greatfree.concurrency.Interactable;
import com.greatfree.concurrency.CheckIdleable;
import com.greatfree.concurrency.Collaborator;
import com.greatfree.concurrency.ThreadIdleChecker;
import com.greatfree.concurrency.Threader;
import com.greatfree.util.Time;
import com.greatfree.util.UtilConfig;

/*
 * This is a task dispatcher which schedules tasks to the special type of threads derived from InteractiveQueue. For the distinct designs, instances of managed threads can interact with the dispatcher for high quality management. 11/21/2014, Bing Li
 *
 * Different from classic dispatchers, the interactive pool splits the threads into two types, the fast one and the low one. However, the splitting is performed upon interactions between the dispatcher and those threads. 11/21/2014, Bing Li
 *
 */

// Created: 11/20/2014, Bing Li
public class InteractiveDispatcher<Task, Notifier extends Interactable<Task>, Function extends InteractiveQueue<Task, Notifier>, ThreadCreator extends InteractiveThreadCreatable<Task, Notifier, Function>, ThreadDisposer extends ThreadDisposable<Function>> implements CheckIdleable
{
	// The size of the fast thread pool. 11/21/2014, Bing Li
	private int fastPoolSize;
	// The size of the slow thread pool. 11/21/2014, Bing Li
	private int slowPoolSize;
	// When creating a new thread, it is required to specify the queue size for the thread. This is the argument. 11/21/2014, Bing Li
	private int threadQueueSize;
	// The maximum idle time for a thread. 11/21/2014, Bing Li
	private long idleTime;
	// The notifier which defines the methods, which are called back to notify the pool. 11/21/2014, Bing Li
	private Notifier notifier;
	// The map to contain fast threads. 11/21/2014, Bing Li
	private Map<String, Threader<Function, ThreadDisposer>> fastThreaderMap;
	// The map to contain slow threads. 11/21/2014, Bing Li
	private Map<String, Threader<Function, ThreadDisposer>> slowThreaderMap;
	// The creator to create instances of threads. 11/21/2014, Bing Li
	private ThreadCreator creator;
	// The disposer to dispose instances of threads. 11/21/2014, Bing Li
	private ThreadDisposer disposer;
	// The collaborator is in essence controlled by managed threads through callback. The pool's working steps are administered by those threads of the pool. 11/21/2014, Bing Li
	private Collaborator collaborator;
	// Declare a timer that controls the task of idle checking. 11/21/2014, Bing Li
	private Timer checkTimer;
	// Declare the checker to check whether created threads are idle long enough. 11/21/2014, Bing Li
	private ThreadIdleChecker<InteractiveDispatcher<Task, Notifier, Function, ThreadCreator, ThreadDisposer>> idleChecker;

	/*
	 * Initialize the pool. 11/21/2014, Bing Li
```

```
   */
  public InteractiveDispatcher(int fastPoolSize, int slowPoolSize, int threadQueueSize, long
idleTime, Notifier notifier, ThreadCreator creator, ThreadDisposer disposer)
  {
    this.fastPoolSize = fastPoolSize;
    this.slowPoolSize = slowPoolSize;
    this.threadQueueSize = threadQueueSize;
    this.idleTime = idleTime;
    this.notifier = notifier;
    this.fastThreaderMap = new ConcurrentHashMap<String, Threader<Function,
ThreadDisposer>>();
    this.slowThreaderMap = new ConcurrentHashMap<String, Threader<Function,
ThreadDisposer>>();
    this.creator = creator;
    this.disposer = disposer;
    this.collaborator = new Collaborator();
    this.checkTimer = UtilConfig.NO_TIMER;
  }

  /*
   * Dispose the pool. 11/21/2014, Bing Li
   */
  public void dispose() throws InterruptedException
  {
    // Set the flag of the collaborator. 11/21/2014, Bing Li
    this.collaborator.setShutdown();
    // Signal blocked threads to advance. 11/21/2014, Bing li
    this.collaborator.signalAll();
    // Stop the fast threads. 11/21/2014, Bing Li
    for (Threader<Function, ThreadDisposer> threader : this.fastThreaderMap.values())
    {
      threader.stop();
    }
    // Stop the slow threads. 11/21/2014, Bing Li
    for (Threader<Function, ThreadDisposer> threader : this.slowThreaderMap.values())
    {
      threader.stop();
    }
    // Cancel the timer that controls the idle checking. 11/21/2014, Bing Li
    if (this.checkTimer != UtilConfig.NO_TIMER)
    {
      this.checkTimer.cancel();
    }
    // Terminate the periodically running thread for idle checking. 11/21/2014, Bing Li
    if (this.idleChecker != null)
    {
      this.idleChecker.cancel();
    }
  }

  /*
   * Check whether the pool is shutdown or not. 11/21/2014, Bing Li
   */
  public boolean isShutdown()
  {
    return this.collaborator.isShutdown();
  }

  /*
   * Pause the pool. 11/21/2014, Bing Li
   */
  public void pause() throws InterruptedException
  {
    this.collaborator.holdOn();
  }

  /*
   * Check whether the pool is paused or not. 11/21/2014, Bing Li
```

```
   */
  public boolean isPaused()
  {
    return this.collaborator.isPaused();
  }

  /*
   * Demand the pool to wait. 11/21/2014, Bing Li
   */
  public void holdOn() throws InterruptedException
  {
    this.collaborator.holdOn();
  }

  /*
   * Demand the pool to keep on working. 11/21/2014, Bing Li
   */
  public void keepOn()
  {
    this.collaborator.keepOn();
  }

  /*
   * Get the fast thread by its key. 11/21/2014, Bing Li
   */
  public Function getFastFunction(String key)
  {
    // Check whether the fast thread is contained in the fast thread map. 11/21/2014, Bing Li
    if (this.fastThreaderMap.containsKey(key))
    {
      // Return the fast thread. 11/21/2014, Bing Li
      return this.fastThreaderMap.get(key).getFunction();
    }
    // Return null if the key is not contained in the fast thread map. 11/21/2014, Bing Li
    return null;
  }

  /*
   * Get the slow thread by its key. 11/21/2014, Bing Li
   */
  public Function getSlowFunction(String key)
  {
    // Check whether the slow thread is contained in the fast thread map. 11/21/2014, Bing Li
    if (this.slowThreaderMap.containsKey(key))
    {
      // Return the slow thread. 11/21/2014, Bing Li
      return this.slowThreaderMap.get(key).getFunction();
    }
    // Return null if the key is not contained in the slow thread map. 11/21/2014, Bing Li
    return null;
  }

  /*
   * Enqueue a new task into the pool for scheduling to a proper thread. 11/21/2014, Bing Li
   */
  public void enqueue(Task task)
  {
    // The scheduling procedure must be finished until the task is scheduled to a thread. 11/21/2014,
Bing Li
    while (true)
    {
      // Check whether the count of fast threads is less than the maximum size. 11/21/2014, Bing Li
      if (this.fastThreaderMap.size() < this.fastPoolSize)
      {
        // If the count of fast threads does not exceed the upper limit, check whether the count is greater
than zero. 11/21/2014, Bing Li
        if (this.fastThreaderMap.size() > 0)
        {
```

```
            // Scan each fast thread and schedule the task to the one whose tasks are not full. 11/21/2014, Bing Li
            for (Threader<Function, ThreadDisposer> threader : this.fastThreaderMap.values())
            {
              // Check whether the load of a fast thread is full. 11/21/2014, Bing Li
              if (!threader.getFunction().isFull())
              {
                // Enqueue the task to the fast thread. 11/21/2014, Bing Li
                threader.getFunction().enqueue(task);
                // End the scheduling. 11/21/2014, Bing Li
                return;
              }
            }
          }

          // If no fast threads are available or each existing fast threads are full, it is required to create a new thread. 11/21/2014, Bing Li
          Function thread = this.creator.createInteractiveThreadInstance(this.threadQueueSize, this.notifier);
          // Wrap the new created thread with threader. 11/21/2014, Bing Li
          Threader<Function, ThreadDisposer> threader = new Threader<Function, ThreadDisposer>(thread, this.disposer);
          // Enqueue the task into the thread. 11/21/2014, Bing Li
          threader.getFunction().enqueue(task);
          // Put the threader into the fast thread pool. 11/21/2014, Bing Li
          this.fastThreaderMap.put(thread.getKey(), threader);
          // Start the threader. 11/21/2014, Bing Li
          threader.start();
          return;
        }
        else
        {
          // If the count of fast threads exceeds the upper limit, it is required to check whether one of them is not full. If so, schedule the task to it. 11/21/2014, Bing Li
          for (Threader<Function, ThreadDisposer> threader : this.fastThreaderMap.values())
          {
            // Check whether one thread is full or not. 11/21/2014, Bing Li
            if (!threader.getFunction().isFull())
            {
              // Schedule the task to the thread. 11/21/2014, Bing Li
              threader.getFunction().enqueue(task);
              // End the scheduling. 11/21/2014, Bing Li
              return;
            }
          }

          // If the scheduling is not completed in the above steps, it denotes that threads are too busy. Thus, it is necessary to notify the relevant module to pause putting new tasks into the dispatcher temporarily. In addition, the pausing keeps until the existing tasks are done. The scheduling procedure must be waken up by relevant threads which finish the tasks. 11/21/2014, Bing Li
          this.notifier.pause();
        }
      }
    }

    /*
     * Get the fast thread keys. 11/21/2014, Bing Li
     */
    public Set<String> getFastThreadKeys()
    {
      return this.fastThreaderMap.keySet();
    }

    /*
     * Get the slow thread keys. 11/21/2014, Bing Li
     */
    public Set<String> getSlowThreadKeys()
    {
```

```
    return this.slowThreaderMap.keySet();
  }

  /*
   * Once if a fast thread becomes slow, it is required to put the thread into the slow pool. A fast
becomes slow when the task it is working on is heavy. For instance, crawling a URL might take much
time for network problems. 11/21/2014, Bing Li
   */
  public void alleviateSlow(String key) throws InterruptedException
  {
    // Check whether the fast pool contains the slow thread key. 11/21/2014, Bing Li
    if (this.fastThreaderMap.containsKey(key))
    {
      // Check whether the count of slow threads is less than the upper limit. 11/21/2014, Bing Li
      if (this.slowThreaderMap.size() < this.slowPoolSize)
      {
        // If the slow thread pool is still not full, it is time to get the new slow threader out from the fast
thread pool. 11/21/2014, Bing Li
        Threader<Function, ThreadDisposer> threader = this.fastThreaderMap.get(key);
        // Remove the new slow threader from the fast thread pool. 11/21/2014, Bing Li
        this.fastThreaderMap.remove(key);
        // Clear the waiting tasks in the new slow threader. It is necessary to indicate that the cleared
tasks are not taken care by the dispatcher. Additional mechanisms should be implemented outside. For
better solution, it must be revised. 11/21/2014, Bing Li
        threader.getFunction().clear();
        // Put the new slow threader into the slow thread pool. 11/21/2014, Bing Li
        this.slowThreaderMap.put(key, threader);
        // Since one fast thread is removed from the fast thread pool, it is necessary to notify the
scheduling procedure that might be blocked because the current size of the fast thread pool exceeds the
upper limit. 11/21/2014, Bing Li
        this.notifier.keepOn();
      }

      // If the size of the slow thread pool exceeds its upper limit, the new slow thread has to stay in the
fast thread pool in the version. It is suggested to revise here to accommodate the problem. 11/21/2014,
Bing Li
    }
  }

  /*
   * Once if a slow thread is identified as a fast one, it is time to put it into the fast thread pool. It usually
happens the slow thread has no tasks to do further. 11/21/2014, Bing Li
   */
  public void restoreFast(String key) throws InterruptedException
  {
    // Check whether the slow thread pool contains the thread key. 11/21/2014, Bing Li
    if (this.slowThreaderMap.containsKey(key))
    {
      // Get the threader from the slow thread pool. 11/21/2014, Bing Li
      Threader<Function, ThreadDisposer> threader = this.slowThreaderMap.get(key);
      // Remove the threader from the slow thread pool. 11/21/2014, Bing Li
      this.slowThreaderMap.remove(key);
      // Put the new fast thread into the fast thread pool. 11/21/2014, Bing Li
      this.fastThreaderMap.put(key, threader);
      // Since one fast thread is added to the fast thread pool, it is necessary to notify the scheduling
procedure that might be blocked because no empty or low load fast threads are available. 11/21/2014,
Bing Li
      this.notifier.keepOn();
    }
  }

  /*
   * The method is called back by the idle checker periodically to monitor the idle states of threads within
the dispatcher. 11/21/2014, Bing Li
   */
  @Override
  public void checkIdle()
  {
```

```
    // Check each thread in the fast thread pool. 11/21/2014, Bing Li
    for (Map.Entry<String, Threader<Function, ThreadDisposer>> threadEntry :
this.fastThreaderMap.entrySet())
    {
      // Check whether the fast thread is idle. 11/21/2014, Bing Li
      if (threadEntry.getValue().getFunction().getIdleTime() != Time.INIT_TIME)
      {
        // Check whether the thread is idle long enough. 11/21/2014, Bing Li
        if (Time.getTimespanInMilliSecond(Calendar.getInstance().getTime(),
threadEntry.getValue().getFunction().getEndTime()) > this.idleTime)
        {
          try
          {
            // Stop the thread that is idle long enough. 11/21/2014, Bing Li
            threadEntry.getValue().stop();
            // Remove the dead thread from the fast thread pool. 11/21/2014, Bing Li
            this.fastThreaderMap.remove(threadEntry.getKey());
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
        }
      }
    }

    // Since the threads in the slow thread pool are busy working, it is unreasonable to check them.
11/21/2014, Bing Li
  }

  /*
   * Set the idle checking parameters. 11/21/2014, Bing Li
   */
  @Override
  public void setIdleChecker(long idleCheckDelay, long idleCheckPeriod)
  {
    // Initialize the timer. 11/21/2014, Bing Li
    this.checkTimer = new Timer();
    // Initialize the idle checker. 11/21/2014, Bing Li
    this.idleChecker = new ThreadIdleChecker<InteractiveDispatcher<Task, Notifier, Function,
ThreadCreator, ThreadDisposer>>(this);
    // Schedule the idle checking task. 11/21/2014, Bing Li
    this.checkTimer.schedule(this.idleChecker, idleCheckDelay, idleCheckPeriod);
  }
}
```

- **ResourceCache**

```
package com.greatfree.reuse;

import java.util.Map;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.locks.ReentrantReadWriteLock;

import com.greatfree.util.CollectionSorter;
import com.greatfree.util.FreeObject;

/*
 * This class is a cache to save the resources that are used in a high probability. It is designed since the total amount of data is too large to be loaded into the memory. Therefore, only the ones that are used frequently are loaded into the cache. It is possible that some loaded ones are obsolete. It is necessary to load new ones that are used frequently into the cached and save the ones that are out of date into the database or the file system persistently. 09/22/2014, Bing Li
 */

// Created: 09/22/2014, Bing Li
public class ResourceCache<Resource extends FreeObject>
{
  // The size of the cache. 09/22/2014, Bing Li
  private long cacheSize;
  // The collection that keeps all of the resource in the cache. 09/22/2014, Bing Li
  private Map<String, Resource> resourceMap;
  // The lock is used to keep thread-safe. 11/28/2014, Bing Li
  private ReentrantReadWriteLock lock;

  /*
   * Initialize the cache. 09/22/2014, Bing Li
   */
  public ResourceCache(long cacheSize)
  {
    this.cacheSize = cacheSize;
    this.resourceMap = new ConcurrentHashMap<String, Resource>();
    this.lock = new ReentrantReadWriteLock();
  }

  /*
   * Dispose the cache. 09/22/2014, Bing Li
   */
  public void dispose()
  {
    if (this.resourceMap != null)
    {
      this.resourceMap.clear();
    }
  }

  /*
   * Clear the cache. 09/22/2014, Bing Li
   */
  public void clear()
  {
    this.resourceMap.clear();
  }

  /*
   * Reset the cache size. 09/22/2014, Bing Li
   */
  public synchronized void resetCacheSize(long cacheSize)
```

```
  {
    this.cacheSize = cacheSize;
  }

  /*
   * Get the cache size. 09/22/2014, Bing Li
   */
  public synchronized long getCacheSize()
  {
    return this.cacheSize;
  }

  /*
   * Assign resources to the cache. All of the existing resources are cleared and the ones to be set are
put into the cache. 09/22/2014, Bing Li
   */
  public void setResources(Map<String, Resource> resourceMap)
  {
    // Clear the existing resources. 09/22/2014, Bing Li
    this.clear();
    // Add the resources to the cache. 09/22/2014, Bing Li
    for (Resource resource : resourceMap.values())
    {
      // Add one piece of resource. 11/28/2014, Bing Li
      this.addResource(resource);
    }
  }

  /*
   * Add a new resource into the cache. 09/22/2014, Bing Li
   */
  public Resource addResource(Resource resource)
  {
    // It is possible that the cache is full such that an obsolete one must be removed. The instance is
declared here. 09/22/2014, Bing Li
    Resource removedRsc = null;
    // Keep thread safe. 09/22/2014, Bing Li
    this.lock.readLock().lock();
    // Check whether the size of the current resource exceeds the upper limit. 11/28/2014, Bing Li
    if (this.resourceMap.size() >= this.cacheSize)
    {
      // Selected the one that is accessed with the least probability. 11/28/2014, Bing Li
      String leastAccessedKey = CollectionSorter.minValueKey(this.resourceMap);
      // Check whether the key is valid. 11/28/2014, Bing Li
      if (leastAccessedKey != null)
      {
        // Get the resource that is accessed with the least probability. 11/28/2014, Bing Li
        removedRsc = this.resourceMap.get(leastAccessedKey);
        this.lock.readLock().unlock();
        this.lock.writeLock().lock();
        // Remove the resource from the cache. 11/28/2014, Bing Li
        this.resourceMap.remove(leastAccessedKey);
        this.lock.readLock().lock();
        this.lock.writeLock().unlock();
      }
    }
    this.lock.readLock().unlock();

    this.lock.writeLock().lock();
    // Put the new resource into the cache. 11/28/2014, Bing Li
    this.resourceMap.put(resource.getObjectKey(), resource);
    // Set the time stamp to access the new resource. 11/28/2014, Bing Li
    this.resourceMap.get(resource.getObjectKey()).setAccessedTime();
    this.lock.writeLock().unlock();
    // Return the removed resource for possible further processing. 11/28/2014, Bing Li
    return removedRsc;
  }
```

```
/*
 * Get one specific resource from the cache by its key. 11/28/2014, Bing Li
 */
public Resource getResource(String resourceKey)
{
  this.lock.readLock().lock();
  try
  {
    // Check whether the resource exists in the cache by its key. 11/28/2014, Bing Li
    if (this.resourceMap.containsKey(resourceKey))
    {
      // If the resource exists, set the accessed time stamp. 11/28/2014, Bing Li
      this.resourceMap.get(resourceKey).setAccessedTime();
      // Return the resource. 11/28/2014, Bing Li
      return this.resourceMap.get(resourceKey);
    }
  }
  finally
  {
    this.lock.readLock().unlock();
  }
  // Return null if the key does not exist. 11/28/2014, Bing Li
  return null;
}

/*
 * Check whether one specific resource exists in the cache by its key. 11/28/2014, Bing Li
 */
public boolean isResourceAvailable(String resourceKey)
{
  this.lock.readLock().lock();
  try
  {
    // Check whether the resource exists in the cache by its key. 11/28/2014, Bing Li
    if (this.resourceMap.containsKey(resourceKey))
    {
      // If the resource exists, set the accessed time stamp. 11/28/2014, Bing Li
      this.resourceMap.get(resourceKey).setAccessedTime();
      // Return true if the key exists. 11/28/2014, Bing Li
      return true;
    }
    else
    {
      // Return false if the key exists. 11/28/2014, Bing Li
      return false;
    }
  }
  finally
  {
    this.lock.readLock().unlock();
  }
}

/*
 * Remove resources from the cache by their keys. 11/28/2014, Bing Li
 */
public void removeResources(Set<String> resourceKeys)
{
  // Scan each key. 11/28/2014, Bing Li
  for (String resourceKey : resourceKeys)
  {
    // Remove the resource by its key. 11/28/2014, Bing Li
    this.removeResource(resourceKey);
  }
}

/*
 * Remove one particular resource from the cache by its key. 11/28/2014, Bing Li
```

```
     */
    public void removeResource(String resourceKey)
    {
        this.lock.readLock().lock();
        // Check whether the key of the resource exists. 11/28/2014, Bing Li
        if (this.resourceMap.containsKey(resourceKey))
        {
            this.lock.readLock().unlock();
            this.lock.writeLock().lock();
            // Remove the resource from the cache. 11/28/2014, Bing Li
            this.resourceMap.remove(resourceKey);
            this.lock.readLock().lock();
            this.lock.writeLock().unlock();
        }
        this.lock.readLock().unlock();
    }

    /*
     * Get all of the resources in the cache. 11/28/2014, Bing Li
     */
    public Map<String, Resource> getResources()
    {
        return this.resourceMap;
    }

    /*
     * Get the keys of all of the resource keys in the cache. 11/28/2014, Bing Li
     */
    public Set<String> getResourceKeys()
    {
        return this.resourceMap.keySet();
    }

    /*
     * Check whether the cache is empty or not. 11/28/2014, Bing Li
     */
    public boolean isEmpty()
    {
        return this.resourceMap.size() <= 0;
    }

    /*
     * Check whether the cache is full or not. 11/28/2014, Bing Li
     */
    public boolean isFull()
    {
        return this.resourceMap.size() >= this.cacheSize;
    }

    /*
     * Get the size of available resources in the cache. 11/28/2014, Bing Li
     */
    public int getResourceSize()
    {
        return this.resourceMap.size();
    }
}
```

- **FreeReaderPool**

```
package com.greatfree.reuse;

import java.io.IOException;
import java.util.Calendar;
import java.util.Date;
import java.util.Map;
import java.util.Set;
import java.util.Timer;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.locks.ReentrantLock;

import com.greatfree.concurrency.Collaborator;
import com.greatfree.remote.FreeClient;
import com.greatfree.remote.FreeClientCreator;
import com.greatfree.remote.FreeClientDisposer;
import com.greatfree.remote.IPPort;
import com.greatfree.util.CollectionSorter;
import com.greatfree.util.Time;
import com.greatfree.util.UtilConfig;

/*
 * This class is similar to RetrievablePool. In fact, it is a specific version of RetrievablePool. First, the resource managed by the pool is FreeClient. Second, it aims to initialize instances of FreeClient for not only output but also for input. 11/05/2014, Bing Li
 *
 * In Java, to initialize an instance of FreeClient for reading remote data, an ObjectInputSream is required. However, to initialize the input stream, the remote end must have a corresponding initialized ObjectOutputStream instance. If the remote end's ObjectOutputStream is not initialized in time, the instances of FreeClient cannot be created by the pool since it gets stuck. 11/05/2014, Bing Li
 *
 * To have the ability to read remotely, it must send a notification to the remote end to ensure the ObjectOutputStream has initialized. After that, an feedback message, InitReadFeedbackNotification, should be received if the output stream is available. Then, the instance of FreeClient can initialize ObjectInputStream. Therefore, it avoids the problem of getting stuck. 11/05/2014, Bing Li
 */

// Created: 11/05/2014, Bing Li
public class FreeReaderPool
{
   // The map contains the instances of FreeClient that are being used. For each remote server, a children map of FreeClient is initialized. The parent key represents the unique ID of a remote server. The key of the children map is unique for each instance of FreeClient. The parent key is generated upon the IP address and the port of a remote server. Well, each client to the particular remote end has the children key, a hash value that is created randomly. 11/06/2014, Bing Li
   private Map<String, Map<String, FreeClient>> busyMap;
   // The map contains the instances of FreeClient that are idle temporarily. Similar to the busy map, for each type of resources, a children map is initialized. The parent key represents the remote server. The key of the children map is unique for each instance of FreeClient. 11/06/2014, Bing Li
   private Map<String, Map<String, FreeClient>> idleMap;
   // The map contains the IPPorts that are used to initialize particular instances of FreeClient. The key is identical to that of the instance of FreeClient. Therefore, it is fine to retrieve the IPPort to create the instance of FreeClient when no idle clients are available and the pool size does not reach the maximum value. 11/06/2014, Bing Li
   private Map<String, IPPort> sourceMap;

   // The lock is responsible for managing the operation on busyMap, idleMap and sourceMap atomic. 10/01/2014, Bing Li
   private ReentrantLock rscLock;
   // The Collaborator is used in the pool to work in the way of notify/wait. A thread has to wait when the pool is full. When resources are available, it can be notified by the collaborator. The collaborator also manages the termination of the pool. 11/06/2014, Bing Li
   private Collaborator collaborator;
   // The Timer controls the period to check the idle instance of FreeClient periodically. 11/06/2014, Bing Li
```

```
  private Timer checkTimer;
  // This is an instance of a class that is executed periodically to check idle clients. When a client is idle long enough, it is necessary to collect it. 11/06/2014, Bing Li
  private FreeReaderIdleChecker idleChecker;
  // The maximum time a client can be idle. 11/06/2014, Bing Li
  private long maxIdleTime;
  // The size of the clients the pool can contain. 11/06/2014, Bing Li
  private int poolSize;
  // When no idle clients are available and the pool size is not reached, it is allowed to create a new instance of FreeClient by the Creator. 11/06/2014, Bing Li
  private FreeClientCreator creator;
  // When an instance of FreeClient is idle long enough, it is collected or disposed by the Disposer. 11/06/2014, Bing Li
  private FreeClientDisposer disposer;

  // This is the collaborator that is not included in RetrievablePool. It serves to wait for the corresponding remote server's notification such that the instance of FreeClient can be returned. 11/06/2014, Bing Li
  private Collaborator initReadCollaborator;

  /*
   * Initialize. 11/06/2014, Bing Li
   */
  public FreeReaderPool(int poolSize)
  {
    this.busyMap = new ConcurrentHashMap<String, Map<String, FreeClient>>();
    this.idleMap = new ConcurrentHashMap<String, Map<String, FreeClient>>();
    this.sourceMap = new ConcurrentHashMap<String, IPPort>();

    this.rscLock = new ReentrantLock();
    this.collaborator = new Collaborator();
    this.poolSize = poolSize;
    this.creator = new FreeClientCreator();

    // It is possible that the pool does not need to check the idle clients. If so, it is unnecessary to initialize the timer. 11/06/2014, Bing Li
    this.checkTimer = UtilConfig.NO_TIMER;
    this.disposer = new FreeClientDisposer();

    this.initReadCollaborator = new Collaborator();
  }

  /*
   * Shutdown the pool. 11/06/2014, Bing Li
   */
  public void shutdown()
  {
    // Set the shutdown flag to be true. Thus, the thread that runs the method of get() can be terminated. 11/06/2014, Bing Li
    this.collaborator.setShutdown();
    // Notify the thread that runs the method of get() to terminate and all of the threads that are waiting for the instances of FreeClient that they are unblocked. 11/06/2014, Bing Li
    this.collaborator.signalAll();
    // Dispose all of the busy clients. It might lose data if no relevant management approaches are adopted. 11/06/2014, Bing Li
    for (Map<String, FreeClient> resourceMap : this.busyMap.values())
    {
      for (FreeClient resource : resourceMap.values())
      {
        this.disposer.dispose(resource);
      }
    }
    this.busyMap.clear();

    // Dispose all of the idle FreeClients. It might lose data if no relevant management approaches are adopted. 11/06/2014, Bing Li
    for (Map<String, FreeClient> resourceMap : this.idleMap.values())
    {
      for (FreeClient resource : resourceMap.values())
```

```
        {
          this.disposer.dispose(resource);
        }
      }
      this.idleMap.clear();

      // Clear the sources. 11/06/2014, Bing Li
      this.sourceMap.clear();

      // Terminate the idle state checker that periodically runs. For it is possible that the checker is not initialized in some cases, it needs to check whether it is null or not. 11/06/2014, Bing Li
      if (this.idleChecker != null)
      {
        this.idleChecker.cancel();
      }
      // Terminate the timer that manages the period to run the idle state checker. For it is possible that the timer is not initialized in some cases, it needs to check whether it is null or not. 11/06/2014, Bing Li
      if (this.checkTimer != UtilConfig.NO_TIMER)
      {
        this.checkTimer.cancel();
      }
    }

    /*
     * Initialize the idle state checker and the timer to manage idle instances of FreeClient periodically when needed. 11/06/2014, Bing Li
     */
    public void setIdleChecker(long idleCheckDelay, long idleCheckPeriod, long maxIdleTime)
    {
      // Initialize the timer. 11/06/2014, Bing Li
      this.checkTimer = new Timer();
      // Initialize the checker. 11/06/2014, Bing Li
      this.idleChecker = new FreeReaderIdleChecker(this);
      // Schedule the task of checking idle states of FreeClient. 11/06/2014, Bing Li
      this.checkTimer.schedule(this.idleChecker, idleCheckDelay, idleCheckPeriod);
      // Set the value of maximum idle time. 11/06/2014, Bing Li
      this.maxIdleTime = maxIdleTime;
    }

    /*
     * Check the idle instances of FreeClient. The method is called back by the IdleChecker. 11/06/2014, Bing Li
     */
    public void checkIdle()
    {
      // Get the current time which is used to calculate the idle time length. 11/06/2014, Bing Li
      Date currentTime = Calendar.getInstance().getTime();

      // The map is used to keep the sorted instances of FreeClient to select the longest idle one conveniently. 11/06/2014, Bing Li
      Map<String, FreeClient> sortedResourceMap;

      // Check whether the idle instances of FreeClient are idle long enough time. 11/06/2014, Bing Li
      this.rscLock.lock();
      for (Map<String, FreeClient> resourceMap : this.idleMap.values())
      {
        // Sort the idle instances of FreeClient by their idle time moment in the ascending order and the results are saved in the map of sortedResourceMap. Thus, the resources that exceed the maximum time length most probably are listed ahead. 11/06/2014, Bing Li
        sortedResourceMap = CollectionSorter.sortByValue(resourceMap);
        // Check the sorted instances of FreeClient. 11/06/2014, Bing Li
        for (FreeClient resource : sortedResourceMap.values())
        {
          // Calculate the idle time length and compare it with the maximum idle time. 11/06/2014, Bing Li
          if (Time.getTimespanInMilliSecond(currentTime, resource.getAccessedTime()) > this.maxIdleTime)
          {
            // If the idle time exceeds the maximum idle time, it denotes that the instance of FreeClient is
```

```
not needed. So it must be collected or disposed. 11/06/2014, Bing Li
            resourceMap.remove(resource.getHashKey());
            // Dispose the client by the disposer. 11/06/2014, Bing Li
            this.disposer.dispose(resource);
          }
          else
          {
            // If the idle time of the instance of FreeClient that has been idle for the longest time in the type
does not exceed the maximum value, it denotes that all of the FreeClient instances of the type have not
been idle long enough. Thus, it needs to detect other types of instances. 11/06/2014, Bing Li
            break;
          }
        }
      }

      // The loop detects whether one type of idle FreeClient instances is empty. If so, it needs to be
removed from the idle map. 11/06/2014, Bing Li
      for (String objectKey : this.idleMap.keySet())
      {
        // Check whether one type of idle FreeClient instances is empty. 11/06/2014, Bing Li
        if (this.idleMap.get(objectKey).size() <= 0)
        {
          // Remove the type of idle FreeClient instances that is empty from the map. 11/06/2014, Bing Li
          this.idleMap.remove(objectKey);
        }
      }
      this.rscLock.unlock();

      // Notify the blocked thread that the size of the assigned FreeClient instances is lowered such that it
is time for it to get one it needs. A bunch of clients might be disposed during the procedure. It is
reasonable to signal all rather than signal a single waiting thread. 11/06/2014, Bing Li
      this.collaborator.signalAll();
    }

    /*
     * Dispose an instance of FreeClient explicitly when it is not needed. Usually, it is not invoked by the
pool but by the threads that consume the client. It happens when it is confirmed that the client is never
needed in a specific case. 11/06/2014, Bing Li
     */
    public void dispose(FreeClient client)
    {
      // Check whether the client is null. 11/06/2014, Bing Li
      if (client != null)
      {
        // Check whether the pool is shutdown or not. If not, it must be managed by the rules of the pool.
11/06/2014, Bing Li
        if (!this.collaborator.isShutdown())
        {
          this.rscLock.lock();
          // Check whether the type of FreeClient is contained in the busy map. 11/06/2014, Bing Li
          if (this.busyMap.containsKey(client.getObjectKey()))
          {
            // Check whether the client is contained in the busy map. 11/06/2014, Bing Li
            if (this.busyMap.get(client.getObjectKey()).containsKey(client.getHashKey()))
            {
              // Remove the client from the busy map. 11/06/2014, Bing Li
              this.busyMap.get(client.getObjectKey()).remove(client.getHashKey());
              // Check if the type of the client is empty in the busy map. 11/06/2014, Bing Li
              if (this.busyMap.get(client.getObjectKey()).size() <= 0)
              {
                // Remove the type of the FreeClient from the busy map.11/06/2014, Bing Li
                this.busyMap.remove(client.getObjectKey());
              }
            }
          }

          // Check whether the idle map contains the type of FreeClient. 11/06/2014, Bing Li
          if (this.idleMap.containsKey(client.getObjectKey()))
```

```
            {
               // Check whether the idle map contains the specific instance of FreeClient. 11/06/2014, Bing Li
               if (this.idleMap.get(client.getObjectKey()).containsKey(client.getHashKey()))
               {
                  // Remove the instance of FreeClient from the idle map. 11/06/2014, Bing Li
                  this.idleMap.get(client.getObjectKey()).remove(client.getHashKey());
                  // Check if the type of FreeClient is empty in the idle map. 11/06/2014, Bing Li
                  if (this.idleMap.get(client.getObjectKey()).size() <= 0)
                  {
                     // Remove the type of FreeClient from the idle map. 11/06/2014, Bing Li
                     this.idleMap.remove(client.getObjectKey());
                  }
               }
            }
            // Dispose the resource eventually after it is managed following the rules of the pool. 11/06/2014,
Bing Li
            this.disposer.dispose(client);
            this.rscLock.unlock();
            // Notify the thread that is blocked for the maximum size of the pool is reached. 11/06/2014, Bing
Li
            this.collaborator.signal();
         }
         else
         {
            // If the pool is shutdown, the resource is disposed directly. 11/06/2014, Bing Li
            this.disposer.dispose(client);
         }
      }
   }

   /*
    * Collect an instance of FreeClient. When the client finishes its task, the method is invoked by the
corresponding thread such that the client can be reused. 11/06/2014, Bing Li
    */
   public void collect(FreeClient client) throws NullPointerException
   {
      // Check whether the resource pool is shutdown. For the method is critical to the resource pool, it
does not make sense to go ahead if the pool is shutdown. 11/06/2014, Bing Li
      if (client != null && !this.collaborator.isShutdown())
      {
         this.rscLock.lock();
         // Check whether the busy map contains the type of FreeClient. 11/06/2014, Bing Li
         if (this.busyMap.containsKey(client.getObjectKey()))
         {
            // Check whether the instance of FreeClient is contained in the busy map. 11/06/2014, Bing Li
            if (this.busyMap.get(client.getObjectKey()).containsKey(client.getHashKey()))
            {
               // Remove the instance of FreeClient to be collected. 11/06/2014, Bing Li
               this.busyMap.get(client.getObjectKey()).remove(client.getHashKey());
               // Check whether the type of FreeClient is empty in the busy map. If so, it is proper to remove it
since it denotes that the type of FreeClient is not needed any longer. 11/06/2014, Bing Li
               if (this.busyMap.get(client.getObjectKey()).size() <= 0)
               {
                  // Remove the type of FreeClient from the busy map. 11/06/2014, Bing Li
                  this.busyMap.remove(client.getObjectKey());
               }
            }
         }

         // Set the accessed time stamp for the client. The time stamp is the idle starting moment of the
client. It is used to calculate whether the client is idle for long enough. 11/06/2014, Bing Li
         client.setAccessedTime();
         // Check whether the idle map contains the type of FreeClient. 11/06/2014, Bing Li
         if (!this.idleMap.containsKey(client.getObjectKey()))
         {
            // If the type of FreeClient is not contained in the idle map, it needs to add the type first.
11/06/2014, Bing Li
            this.idleMap.put(client.getObjectKey(), new ConcurrentHashMap<String, FreeClient>());
```

```
            // Add the instance of FreeClient to the idle map. 11/06/2014, Bing Li
            this.idleMap.get(client.getObjectKey()).put(client.getHashKey(), client);
          }
          else
          {
            // If the type of FreeClient is already existed in the idle map, just add the instance of FreeClient
to the idle map. 11/06/2014, Bing Li
            this.idleMap.get(client.getObjectKey()).put(client.getHashKey(), client);
          }
          this.rscLock.unlock();
          // Notify the thread that is blocked for the maximum size of the pool is reached. 11/06/2014, Bing Li
          this.collaborator.signal();
        }
      }

      /*
       * Remove the type of FreeClient explicitly by the key of the type. It is used in the case when a thread
confirms that one type of FreeClient is not needed to be created in the pool. It is not used frequently.
Just keep an interface for possible cases. 11/06/2014, Bing Li
       */
      public void removeResource(String objectKey)
      {
        this.rscLock.lock();
        // Check whether the type of FreeClient is existed in the busy map. 11/06/2014, Bing Li
        if (this.busyMap.containsKey(objectKey))
        {
          // Dispose all of the instances of FreeClient if they are in the busy map. 11/06/2014, Bing Li
          for (FreeClient rsc : this.busyMap.get(objectKey).values())
          {
            this.disposer.dispose(rsc);
          }
          // Remove the type of FreeClient from the busy map. 11/06/2014, Bing Li
          this.busyMap.remove(objectKey);
        }

        // Check whether the type of FreeClient is existed in the idle map. 11/06/2014, Bing Li
        if (this.idleMap.containsKey(objectKey))
        {
          // Dispose all of the instances of FreeClient if they are in the idle map. 11/06/2014, Bing Li
          for (FreeClient rsc : this.idleMap.get(objectKey).values())
          {
            this.disposer.dispose(rsc);
          }
          // Remove the type of FreeClient from the idle map. 11/06/2014, Bing Li
          this.idleMap.remove(objectKey);
        }
        this.rscLock.unlock();
        // Notify all of the threads that are waiting for FreeClient to keep on working if instances of FreeClient
are available for the removal from the idle map. 11/06/2014, Bing Li
        this.collaborator.signalAll();
      }

      /*
       * Check whether a specific instance of FreeClient is busy. 11/06/2014, Bing Li
       */
      public boolean isBusy(FreeClient client)
      {
        this.rscLock.lock();
        try
        {
          // Check whether the type of FreeClient is existed in the busy map. 11/06/2014, Bing Li
          if (this.busyMap.containsKey(client.getObjectKey()))
          {
            // Check whether the specific instance of FreeClient is existed in the busy map if the type of
FreeClient is existed in the map. 11/06/2014, Bing Li
            return this.busyMap.get(client.getObjectKey()).containsKey(client.getHashKey());
          }
          // If the type of FreeClient is not existed in the busy map, the client is not busy. 11/06/2014, Bing Li
```

```
      return false;
    }
    finally
    {
      this.rscLock.unlock();
    }
  }

  /*
   * Return all of the type keys of FreeClient. 11/06/2014, Bing Li
   */
  public Set<String> getAllObjectKeys()
  {
    this.rscLock.lock();
    try
    {
      return this.sourceMap.keySet();
    }
    finally
    {
      this.rscLock.unlock();
    }
  }

  /*
   * Get the IPPort of a particular type of FreeClient. 11/06/2014, Bing Li
   */
  public IPPort getSource(String objectKey)
  {
    this.rscLock.lock();
    try
    {
      // Check whether the source is available in the source map. 11/06/2014, Bing Li
      if (this.sourceMap.containsKey(objectKey))
      {
        // Return the source if it is existed in the source map. 11/06/2014, Bing Li
        return this.sourceMap.get(objectKey);
      }
      // Return null if the source is not available in the source map. 11/06/2014, Bing Li
      return null;
    }
    finally
    {
      this.rscLock.unlock();
    }
  }

  /*
   * Get the size of IPPorts. 11/06/2014, Bing Li
   */
  public int getSourceSize()
  {
    this.rscLock.lock();
    try
    {
      return this.sourceMap.size();
    }
    finally
    {
      this.rscLock.unlock();
    }
  }

  /*
   * Get the size of the pool. 11/06/2014, Bing Li
   */
  public synchronized int getPoolSize()
  {
```

```
    return this.poolSize;
  }

  /*
   * Check whether a particular IPPort is available by the key of the IPPort. 11/06/2014, Bing Li
   */
  public boolean isSourceExisted(String key)
  {
    this.rscLock.lock();
    try
    {
      return this.sourceMap.containsKey(key);
    }
    finally
    {
      this.rscLock.unlock();
    }
  }

  /*
   * Check whether a particular IPPort is available by the IPPort itself. 11/06/2014, Bing Li
   */
  public boolean IsSourceExisted(IPPort source)
  {
    this.rscLock.lock();
    try
    {
      return this.sourceMap.containsKey(source.getObjectKey());
    }
    finally
    {
      this.rscLock.unlock();
    }
  }

  /*
   * Get the size of all of the types of busy instances of FreeClient at the moment when the method is
invoked. 11/06/2014, Bing Li
   */
  public int getBusyResourceSize()
  {
    int busyResourceCount = 0;
    this.rscLock.lock();
    try
    {
      for (Map<String, FreeClient> resMap : this.busyMap.values())
      {
        busyResourceCount += resMap.size();
      }
    }
    finally
    {
      this.rscLock.unlock();
    }
    return busyResourceCount;
  }

  /*
   * Get the size of all of the types of idle instances of FreeClient at the moment when the method is
invoked. 11/06/2014, Bing Li
   */
  public int getIdleResourceSize()
  {
    int idleResourceCount = 0;
    this.rscLock.lock();
    try
    {
      for (Map<String, FreeClient> resMap : this.idleMap.values())
```

```
        {
          idleResourceCount += resMap.size();
        }
      }
      finally
      {
        this.rscLock.unlock();
      }
      return idleResourceCount;
    }

    /*
     * After receiving feedback from the remote server, the method is called to notify the relevant to
complete the initialization of ObjectInputStream of the FreeClient instance. 11/06/2014, Bing Li
     */
    public void notifyOutStreamDone()
    {
      this.initReadCollaborator.signal();
    }

    /*
     * Get the instance of FreeClient by the IP/port of the remote server. The argument, nodeKey, is used
to notify the remote server to retrieve the client to the local end such that the feedback can be sent.
11/06/2014, Bing Li
     */
    public FreeClient get(String nodeKey, IPPort src) throws IOException
    {
      // The instance of FreeClient to be returned. 11/06/2014, Bing Li
      FreeClient client = null;
      // One particular type of FreeClient that is the same type as the requested one. 11/06/2014, Bing Li
      Map<String, FreeClient> resourceMap;
      // The hash key of the FreeClient instance that has the longest lifetime. 11/06/2014, Bing Li
      String oldestResourceKey;
      // The count of busy instances of FreeClient. 11/06/2014, Bing Li
      int busyResourceCount;
      // The count of idle instances of FreeClient. 11/06/2014, Bing Li
      int idleResourceCount;

      // Since the procedure to get an instance of FreeClient might be blocked when the upper limit of the
pool is reached, it is required to keep a notify/wait mechanism to guarantee the procedure smooth.
11/06/2014, Bing Li
      while (!this.collaborator.isShutdown())
      {
        // Initialize the value of busyResourceCount. 11/06/2014, Bing Li
        busyResourceCount = 0;
        // Initialize the value of idleResourceCount. 11/06/2014, Bing Li
        idleResourceCount = 0;

        this.rscLock.lock();
        try
        {
          /*
           * Retrieve the FreeClient instance from the idle map first. 11/06/2014, Bing Li
           */

          // Check whether the idle map contains the type of FreeClient. 11/06/2014, Bing Li
          if (this.idleMap.containsKey(src.getObjectKey()))
          {
            // The type of idle FreeClient instances which are the candidates to be returned. 11/06/2014,
Bing Li
            resourceMap = this.idleMap.get(src.getObjectKey());
            // Get the hash key of the client that is idle longer than any others from the idle FreeClient
instances. 11/06/2014, Bing Li
            oldestResourceKey = CollectionSorter.minValueKey(resourceMap);
            // Check whether the hash key is valid. 11/06/2014, Bing Li
            if (oldestResourceKey != null)
            {
              // Get the instance of FreeClient that is idle than any others by its hash key. 11/06/2014,
```

```
Bing Li
                client = resourceMap.get(oldestResourceKey);
                // Set the accessed time of the client that is idle than any others to the current moment since
the client will be used after it is returned. Now it becomes a busy one. 11/06/2014, Bing Li
                client.setAccessedTime();
                // Remove the instance of FreeClient from the idle map since it will become a busy one.
11/06/2014, Bing Li
                this.idleMap.get(src.getObjectKey()).remove(oldestResourceKey);
                // Check whether the type of FreeClient is empty in the idle map. 11/06/2014, Bing Li
                if (this.idleMap.get(src.getObjectKey()).size() <= 0)
                {
                  // Remove the type of FreeClient from the idle map if no any specific instances of the type
is in the idle map. 11/06/2014, Bing Li
                  this.idleMap.remove(src.getObjectKey());
                }
                // Check whether the type of FreeClient is available in the busy map. 11/06/2014, Bing Li
                if (!this.busyMap.containsKey(client.getObjectKey()))
                {
                  // If the type of FreeClient is not existed in the busy map, add the type and the particular
instance. 11/06/2014, Bing Li
                  this.busyMap.put(client.getObjectKey(), new ConcurrentHashMap<String, FreeClient>());
                  this.busyMap.get(client.getObjectKey()).put(oldestResourceKey, client);
                }
                else
                {
                  // If the type of FreeClient is existed in the busy map, add the particular instance.
11/06/2014, Bing Li
                  this.busyMap.get(client.getObjectKey()).put(oldestResourceKey, client);
                }
              }
            }

            /*
             * If the idle map does not contain the requested instance of FreeClient, it is necessary to
initialize a new one. 11/06/2014, Bing Li
             */

            // Check whether the instance of FreeClient is available after retrieving from the idle map.
11/06/2014, Bing Li
            if (client == null)
            {
              // Calculate the exact count of all of the busy instances of FreeClient. The value is used to
check whether the upper limit of the pool is reached. 11/06/2014, Bing Li
              for (Map<String, FreeClient> resMap : this.busyMap.values())
              {
                busyResourceCount += resMap.size();
              }

              // Calculate the exact count of all of the idle instances of FreeClient. The value is used to
check whether the upper limit of the pool is reached. 11/06/2014, Bing Li
              for (Map<String, FreeClient> resMap : this.idleMap.values())
              {
                idleResourceCount += resMap.size();
              }

              // Check whether the sum of the count of busy and idle instances of FreeClient reach the
upper limit of the pool. 11/06/2014, Bing Li
              if (busyResourceCount + idleResourceCount < this.poolSize)
              {
                // If the upper limit of the pool is not reached, it is time to create an instance of FreeClient by
its IPPort. 11/06/2014, Bing Li
                client = this.creator.createResourceInstance(src);
                // Check whether the newly created instance of FreeClient is valid. 11/06/2014, Bing Li
                if (client != null)
                {
                  // Check whether the type of FreeClient is available in the busy map. 11/06/2014, Bing Li
                  if (!this.busyMap.containsKey(client.getObjectKey()))
                  {
```

```
                    // If the type of FreeClient is not available in the busy map, add the type and the instance
into it. 11/06/2014, Bing Li
                    this.busyMap.put(client.getObjectKey(), new ConcurrentHashMap<String,
FreeClient>());
                    this.busyMap.get(client.getObjectKey()).put(client.getHashKey(), client);
                  }
                  else
                  {
                    // If the type of FreeClient is available in the busy map, add the instance into it.
11/06/2014, Bing Li
                    this.busyMap.get(client.getObjectKey()).put(client.getHashKey(), client);
                  }
                  // Add the IPPort to the source map. 11/06/2014, Bing Li
                  this.sourceMap.put(client.getObjectKey(), src);
                }

                try
                {
                  // The line sends a message to the remote server. If it receives the message and send a
feedback, it indicates that the ObjectOutputStream is initialized on the server. Then, the local
ObjectInputStream can be initialized. It avoids the possible getting blocked. 11/06/2014, Bing Li
                  client.initRead(nodeKey);
                  // Wait for feedback from the remote server. 11/06/2014, Bing Li
                  this.initReadCollaborator.holdOn(UtilConfig.INIT_READ_WAIT_TIME);
                  // After the feedback is received, the ObjectInputStream can be initialized. 11/06/2014,
Bing Li
                  client.setInputStream();
                }
                catch (InterruptedException e)
                {
                  e.printStackTrace();
                }
                // Return the instance of FreeClient to the thread. 11/06/2014, Bing Li
                return client;
              }
            }
            else
            {
              // If the instance of FreeClient is obtained from the idle map and then return it to the thread.
11/06/2014, Bing Li
              return client;
            }
          }
          finally
          {
            this.rscLock.unlock();
          }

          /*
           * If no such instances of FreeClient are in the idle map and the upper limit of the pool is reached,
the thread that invokes the method has to wait for future possible updates. The possible updates include
FreeClient instances disposals and idle ones being available. 11/06/2014, Bing Li
           */
          try
          {
            // Check whether the pool is shutdown. 11/06/2014, Bing Li
            if (!this.collaborator.isShutdown())
            {
              // If the pool is not shutdown, it is time to wait for some time. After that, the above procedure is
repeated if the pool is not shutdown. 11/06/2014, Bing Li
              this.collaborator.holdOn(UtilConfig.ONE_SECOND);
            }
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
        }
```

```
    return null;
  }

  /*
   * Get the instance of FreeClient by the key of IPPort of the remote server. The argument, nodeKey, is
used to notify the remote server to retrieve the client to the local end such that the feedback can be sent.
11/06/2014, Bing Li
   */
  public FreeClient get(String nodeKey, String objectKey) throws IOException
  {
    // Check whether the key is valid. 11/06/2014, Bing Li
    if (objectKey != UtilConfig.NO_KEY)
    {
      // Declare an instance of IPPort. 11/06/2014, Bing Li
      IPPort src;
      // The instance of FreeClient to be returned. 11/06/2014, Bing Li
      FreeClient client = null;
      // One particular type of FreeClient that is the same type as the requested one. 11/06/2014, Bing Li
      Map<String, FreeClient> resourceMap;
      // The hash key of the FreeClient instance that has the longest lifetime. 11/06/2014, Bing Li
      String oldestResourceKey;
      // The count of busy instances of FreeClient. 11/06/2014, Bing Li
      int busyResourceCount;
      // The count of idle instances of FreeClient. 11/06/2014, Bing Li
      int idleResourceCount;

      // A flag to indicate whether the relevant IPPort is available. 11/06/2014, Bing Li
      boolean isSourceAvailable = false;
      // Since the procedure to get an instance of FreeClient might be blocked when the upper limit of
the pool is reached, it is required to keep a notify/wait mechanism to guarantee the procedure smooth.
11/06/2014, Bing Li
      while (!this.collaborator.isShutdown())
      {
        this.rscLock.lock();
        try
        {
          // Check if the IPPort is available. 11/06/2014, Bing Li
          if (this.sourceMap.containsKey(objectKey))
          {
            // Set the flag that the IPPort is available. 11/06/2014, Bing Li
            isSourceAvailable = true;
            // Retrieve the IPPort. 11/06/2014, Bing Li
            src = this.sourceMap.get(objectKey);
            // Initialize the value of busyResourceCount. 11/06/2014, Bing Li
            busyResourceCount = 0;
            // Initialize the value of idleResourceCount. 11/06/2014, Bing Li
            idleResourceCount = 0;

            /*
             * Retrieve the FreeClient instance from the idle map first. 11/06/2014, Bing Li
             */

            // Check whether the idle map contains the type of FreeClient. 11/06/2014, Bing Li
            if (this.idleMap.containsKey(src.getObjectKey()))
            {
              // The type of idle FreeClient instances which are the candidates to be returned.
11/06/2014, Bing Li
              resourceMap = this.idleMap.get(src.getObjectKey());
              // Get the hash key of the client that is idle longer than any others from the idle FreeClient
instances. 11/06/2014, Bing Li
              oldestResourceKey = CollectionSorter.minValueKey(resourceMap);
              // Check whether the hash key is valid. 11/06/2014, Bing Li
              if (oldestResourceKey != null)
              {
                // Get the instance of FreeClient that is idle than any others by its hash key. 11/06/2014,
Bing Li
                client = resourceMap.get(oldestResourceKey);
                // Set the accessed time of the client that is idle than any others to the current moment
```

```
since the client will be used after it is returned. Now it becomes a busy one. 11/06/2014, Bing Li
                    client.setAccessedTime();
                    // Remove the instance of FreeClient from the idle map since it will become a busy one.
11/06/2014, Bing Li
                    this.idleMap.get(src.getObjectKey()).remove(oldestResourceKey);
                    // Check whether the type of FreeClient is empty in the idle map. 11/06/2014, Bing Li
                    if (this.idleMap.get(src.getObjectKey()).size() <= 0)
                    {
                      // Remove the type of FreeClient from the idle map if no any specific instances of the
type is in the idle map. 11/06/2014, Bing Li
                      this.idleMap.remove(src.getObjectKey());
                    }
                    // Check whether the type of FreeClient is available in the busy map. 11/06/2014, Bing Li
                    if (!this.busyMap.containsKey(client.getObjectKey()))
                    {
                      // If the type of FreeClient is not existed in the busy map, add the type and the
particular instance. 11/06/2014, Bing Li
                      this.busyMap.put(client.getObjectKey(), new ConcurrentHashMap<String,
FreeClient>());
                      this.busyMap.get(client.getObjectKey()).put(oldestResourceKey, client);
                    }
                    else
                    {
                      // If the type of FreeClient is existed in the busy map, add the particular instance.
11/06/2014, Bing Li
                      this.busyMap.get(client.getObjectKey()).put(oldestResourceKey, client);
                    }
                  }
                }

                /*
                 * If the idle map does not contain the requested instance of FreeClient, it is necessary to
initialize a new one. 11/06/2014, Bing Li
                 */

                // Check whether the instance of FreeClient is available after retrieving from the idle map.
11/06/2014, Bing Li
                if (client == null)
                {
                  // Calculate the exact count of all of the busy instances of FreeClient. The value is used to
check whether the upper limit of the pool is reached. 11/06/2014, Bing Li
                  for (Map<String, FreeClient> resMap : this.busyMap.values())
                  {
                    busyResourceCount += resMap.size();
                  }

                  // Calculate the exact count of all of the idle instances of FreeClient. The value is used to
check whether the upper limit of the pool is reached. 11/06/2014, Bing Li
                  for (Map<String, FreeClient> resMap : this.idleMap.values())
                  {
                    idleResourceCount += resMap.size();
                  }

                  // Check whether the sum of the count of busy and idle instances of FreeClient reach the
upper limit of the pool. 11/06/2014, Bing Li
                  if (busyResourceCount + idleResourceCount < this.poolSize)
                  {
                    // If the upper limit of the pool is not reached, it is time to create an instance of FreeClient
by its IPPort. 11/06/2014, Bing Li
                    client = this.creator.createResourceInstance(src);
                    // Check whether the newly created instance of FreeClient is valid. 11/06/2014, Bing Li
                    if (client != null)
                    {
                      // Check whether the type of FreeClient is available in the busy map. 11/06/2014, Bing
Li
                      if (!this.busyMap.containsKey(client.getObjectKey()))
                      {
                        // If the type of FreeClient is not available in the busy map, add the type and the
```

```
instance into it. 11/06/2014, Bing Li
                        this.busyMap.put(client.getObjectKey(), new ConcurrentHashMap<String,
FreeClient>());
                        this.busyMap.get(client.getObjectKey()).put(client.getHashKey(), client);
                      }
                      else
                      {
                        // If the type of FreeClient is available in the busy map, add the instance into it.
11/06/2014, Bing Li
                        this.busyMap.get(client.getObjectKey()).put(client.getHashKey(), client);
                      }
                    }

                    try
                    {
                      // The line sends a message to the remote server. If it receives the message, it
indicates that the ObjectOutputStream is initialized on the server. Then, the local ObjectInputStream can
be initialized. It avoids the possible getting blocked. 11/06/2014, Bing Li
                      client.initRead(nodeKey);
                      // Wait for feedback from the remote server. 11/06/2014, Bing Li
                      this.initReadCollaborator.holdOn(UtilConfig.INIT_READ_WAIT_TIME);
                      // After the feedback is received, the ObjectInputStream can be initialized. 11/06/2014,
Bing Li
                      client.setInputStream();
                    }
                    catch (InterruptedException e)
                    {
                      e.printStackTrace();
                    }
                  }
                }
                // Return the instance of FreeClient to the thread. 11/06/2014, Bing Li
                return client;
              }
            }
            finally
            {
              this.rscLock.unlock();
            }

            /*
             * If no such instances of FreeClient are in the idle map and the upper limit of the pool is
reached, the thread that invokes the method has to wait for future possible updates. The possible
updates include FreeClient instances disposals and idle ones being available. 11/06/2014, Bing Li
             */

            // Check if the IPPort is available. 11/06/2014, Bing Li
            if (isSourceAvailable)
            {
              try
              {
                // Check whether the pool is shutdown. 11/06/2014, Bing Li
                if (!this.collaborator.isShutdown())
                {
                  // If the pool is not shutdown, it is time to wait for some time. After that, the above
procedure is repeated if the pool is not shutdown. 11/06/2014, Bing Li
                  this.collaborator.holdOn(UtilConfig.ONE_SECOND);
                }
              }
              catch (InterruptedException e)
              {
                e.printStackTrace();
              }
            }
            else
            {
              return UtilConfig.NO_CLIENT;
            }
```

```
            }
        }
        return UtilConfig.NO_CLIENT;
    }
}
```

- **FreeReaderIdleChecker**

```
package com.greatfree.reuse;

import java.util.TimerTask;

/*
 * The class is used to call back the method of checkIdle of the instance of FreeReaderPool. 11/05/2014,
Bing Li
 */

// Created: 11/05/2014, Bing Li
public class FreeReaderIdleChecker extends TimerTask
{
   // Declare an instance of FreeReaderPool. 11/05/2014, Bing Li
   private FreeReaderPool pool;

   /*
    * The instance of FreeReaderPool is initialized outside and assigned to the object. 11/05/2014, Bing
Li
    */
   public FreeReaderIdleChecker(FreeReaderPool pool)
   {
      this.pool = pool;
   }

   /*
    * Invoke the method, checkIdle(), periodically. 11/05/2014, Bing Li
    */
   @Override
   public void run()
   {
      this.pool.checkIdle();
   }
}
```

- **MulticastMessageDisposer**

```
package com.greatfree.reuse;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.CopyOnWriteArrayList;

import com.greatfree.concurrency.MessageBindable;
import com.greatfree.multicast.ServerMulticastMessage;

/*
 * This is an implementation of the MessageBindable interface. It is usually used by threads which share the same multicast messages. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public class MulticastMessageDisposer<Message extends ServerMulticastMessage> implements MessageBindable<Message>
{
  // The collection keeps each message's state, i.e., the thread keys that share the message and whether the message has been already processed by a particular thread. 11/26/2014, Bing Li
  private Map<String, Map<String, Boolean>> messageStates;
  // All of the thread keys to be synchronized are saved in the list. 11/26/2014, Bing Li
  private CopyOnWriteArrayList<String> threadKeys;

  /*
   * Initialize the message disposer. 11/26/2014, Bing Li
   */
  public MulticastMessageDisposer()
  {
    this.messageStates = new ConcurrentHashMap<String, Map<String, Boolean>>();
    this.threadKeys = new CopyOnWriteArrayList<String>();
  }

  /*
   * Add the thread key that shares the notification to the message disposer. 11/26/2014, Bing Li
   */
  @Override
  public void addThread(String key)
  {
    this.threadKeys.add(key);
  }

  /*
   * Put one message into the message disposer for synchronization and disposal. 11/26/2014, Bing Li
   */
  @Override
  public synchronized void set(Message message)
  {
    // Check whether the message exists in the disposer. 11/26/2014, Bing Li
    if (!this.messageStates.containsKey(message.getKey()))
    {
      // If the message does not exist, save it. 11/26/2014, Bing Li
      this.messageStates.put(message.getKey(), new ConcurrentHashMap<String, Boolean>());
      // For the new message, it must associate it with all of threads. 11/26/2014, Bing Li
      for (String key : this.threadKeys)
      {
        // The flag, false, indicates that the thread has not finished processing the message. Therefore, the disposal cannot be performed. 11/26/2014, Bing Li
        this.messageStates.get(message.getKey()).put(key, false);
      }
    }
  }

  /*
```

```
    * When a thread finishes processing a message, it must invoke the method, bind(), to notify the
message disposer for synchronization. 11/26/2014, Bing Li
    */
   @Override
   public synchronized void bind(String threadKey, Message message)
   {
      // Update the flag to true. It indicates that the message has already been processed by a particular
thread. 11/26/2014, Bing Li
      this.messageStates.get(message.getKey()).put(threadKey, true);
      // Initialize a count for synchronization management. 11/26/2014, Bing Li
      int count = 0;
      // Scan each thread that shares the message for the issue of whether they have already processed
the message or not. 11/26/2014, Bing Li
      for (String key : this.threadKeys)
      {
         // Check if one message is finished processing by a thread. 11/26/2014, Bing Li
         if (this.messageStates.get(message.getKey()).get(key))
         {
            // Increment the count if one thread has finished processing the message. 11/26/2014, Bing Li
            count++;
         }
      }
      // Check whether the count exceeds the count of all of the threads. 11/26/2014, Bing Li
      if (count >= this.threadKeys.size())
      {
         // If the above condition is fulfilled, it denotes that all of the threads have already consumed the
message. 11/26/2014, Bing Li
         this.messageStates.remove(message.getKey());
         // Then, further processing on the message can be performed here. In this case, the message is
disposed. 11/26/2014, Bing Li
         message = null;
      }
   }

   /*
    * Dispose the message disposer. 11/26/2014, Bing Li
    */
   @Override
   public void dispose()
   {
      this.messageStates.clear();
      this.threadKeys.clear();
      this.messageStates = null;
      this.threadKeys = null;
   }
}
```

2.3 Concurrency

- **Threader**

```
package com.greatfree.concurrency;

import com.greatfree.reuse.ThreadDisposable;

/*
 * This is a class that is used to simplify the procedure to invoke a single thread that is derived from the class of Thread. 07/17/2014, Bing Li
 */

// Created: 08/04/2014, Bing Li
public class Threader<Function extends Thread, FunctionDisposer extends ThreadDisposable<Function>>
{
  // The task that needs to be taken concurrently. 08/04/2014, Bing Li
  private Function function;

  // The class aims to dispose the task being executed concurrently. 08/04/2014, Bing Li
  private FunctionDisposer disposer;

  public Threader(Function func, FunctionDisposer disposer)
  {
    // The constructor encapsulates the thread initialization task. It looks cleaner than the situation in which the procedure is exposed to developers. 08/04/2014, Bing Li
    this.function = func;
    this.function.setDaemon(true);
    this.disposer = disposer;
  }

  public Threader(Function func, FunctionDisposer disposer, boolean isDaemon)
  {
    // The constructor encapsulates the thread initialization task. It looks cleaner than the situation in which the procedure is exposed to developers. 08/04/2014, Bing Li
    this.function = func;
    this.function.setDaemon(isDaemon);
    this.disposer = disposer;
  }

  /*
   * Start the initialized thread. 08/04/2014, Bing Li
   */
  public void start()
  {
    this.function.start();
  }

  /*
   * Stop the thread and wait for its death for some time. 08/04/2014, Bing Li
   */
  public void stop(long waitTime) throws InterruptedException
  {
    this.disposer.dispose(this.function, waitTime);
  }

  /*
   * Stop the thread by waiting for its death. 08/04/2014, Bing Li
   */
  public void stop() throws InterruptedException
  {
    this.disposer.dispose(this.function);
  }

  /*
   * Expose the task. 08/04/2014, Bing Li
   */
```

```
  public Function getFunction()
  {
    return this.function;
  }

  /*
   * Detect whether the thread is alive. 08/04/2014, Bing Li
   */
  public boolean isAlive()
  {
    return this.function.isAlive();
  }
}
```

- **Runner**

```
package com.greatfree.concurrency;

import com.greatfree.reuse.RunDisposable;

/*
 * This is a class that is used to simplify the procedure to invoke a single thread that implements the
interface of Runnable. 07/17/2014, Bing Li
 */

// Created: 07/17/2014, Bing Li
public class Runner<Function extends Runnable, FunctionDisposer extends
RunDisposable<Function>>
{
   // The thread that supports the concurrency. 08/04/2014, Bing Li
   private Thread thread;

   // The task that needs to be taken concurrently. 08/04/2014, Bing Li
   private Function function;

   // The class aims to dispose the task being executed concurrently. 08/04/2014, Bing Li
   private FunctionDisposer disposer;

   public Runner(Function func, FunctionDisposer disposer)
   {
      // The constructor encapsulates the thread initialization task. It looks cleaner than the situation in
which the procedure is exposed to developers. 08/04/2014, Bing Li
      this.function = func;
      this.thread = new Thread(this.function);
      this.thread.setDaemon(true);
      this.disposer = disposer;
   }

   public Runner(Function func, FunctionDisposer disposer, boolean isDaemon)
   {
      // The constructor encapsulates the thread initialization task. It looks cleaner than the situation in
which the procedure is exposed to developers. 08/04/2014, Bing Li
      this.function = func;
      this.thread = new Thread(this.function);
      this.thread.setDaemon(isDaemon);
      this.disposer = disposer;
   }

   /*
    * Start the initialized thread. 08/04/2014, Bing Li
    */
   public void start()
   {
      this.thread.start();
   }

   /*
    * Stop the thread and wait for its death for some time. 08/04/2014, Bing Li
    */
   public void stop(long waitTime) throws InterruptedException
   {
      this.disposer.dispose(this.function, waitTime);
   }

   /*
    * Stop the thread by waiting for its death. 08/04/2014, Bing Li
    */
   public void stop() throws InterruptedException
   {
      this.disposer.dispose(this.function);
```

```
    }

    /*
     * Expose the task. 08/04/2014, Bing Li
     */
    public Function getFunction()
    {
      return this.function;
    }

    /*
     * Detect whether the thread is alive. 08/04/2014, Bing Li
     */
    public boolean isAlive()
    {
      return this.thread.isAlive();
    }

    /*
     * Interrupt the enclosed thread. 08/04/2014, Bing Li
     */
    public void interrupt()
    {
      this.thread.interrupt();
    }
  }
```

- **ThreadPool**

```
package com.greatfree.concurrency;

import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.RejectedExecutionException;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import com.greatfree.util.UtilConfig;

/*
 * The class is a wrapper of the built-in class, ThreadPoolExecutor to ease the procedure of managing
threads. 07/17/2014, Bing Li
 */

// Created: 07/17/2014, Bing Li
public class ThreadPool
{
  // The built-in thread pool from JDK. 08/10/2014, Bing Li
  private ThreadPoolExecutor threadPool;
  // The core pool size of the thread pool. 08/10/2014, Bing Li
  private int corePoolSize;
  // The max pool size of the thread pool. 08/10/2014, Bing Li
  private int maxPoolSize;
  // The time to keeo alive for an idle thread. 08/10/2014, Bing Li
  private long keepAliveTime;
  // The queue that all the threads are scheduled and kept. 08/10/2014, Bing Li
  private LinkedBlockingQueue<Runnable> taskQueue;

  /*
   * Initialize the properties. 08/10/2014, Bing Li
   */
  public ThreadPool(int corePoolSize, long keepAliveTime)
  {
    this.taskQueue = new LinkedBlockingQueue<Runnable>();
    this.corePoolSize = corePoolSize;
    this.maxPoolSize = corePoolSize + UtilConfig.ADDTIONAL_THREAD_POOL_SIZE;
    this.keepAliveTime = keepAliveTime;
    this.threadPool = new ThreadPoolExecutor(this.corePoolSize, this.maxPoolSize,
this.keepAliveTime, TimeUnit.MILLISECONDS, this.taskQueue);
  }

  /*
   * Shutdown the thread pool. 08/10/2014, Bing Li
   */
  public synchronized void shutdown()
  {
    this.taskQueue.clear();
    this.threadPool.shutdownNow();
  }

  /*
   * Get the core size of the thread pool. 08/10/2014, Bing Li
   */
  public int getCorePoolSize()
  {
    return this.threadPool.getCorePoolSize();
  }

  /*
   * Create a new thread pool if the previous one is shutdown. 08/10/2014, Bing Li
   */
  public void reset()
  {
    if (this.threadPool.isShutdown())
```

```
    {
      if (this.keepAliveTime == Long.MAX_VALUE)
      {
        this.threadPool = new ThreadPoolExecutor(this.corePoolSize, this.maxPoolSize,
this.keepAliveTime, TimeUnit.MILLISECONDS, this.taskQueue);
      }
      else
      {
        this.threadPool = new ThreadPoolExecutor(this.corePoolSize, this.maxPoolSize,
this.keepAliveTime, TimeUnit.MILLISECONDS, this.taskQueue);
      }
    }
  }

  /*
   * Execute the thread that derives from the class of Thread. 08/10/2014, Bing Li
   */
  public void execute(Thread thread)
  {
    try
    {
      this.threadPool.execute(thread);
    }
    catch (RejectedExecutionException e)
    {
      e.printStackTrace();
    }
  }

  /*
   * Execute the thread that implements the interface of Runnable. 08/10/2014, Bing Li
   */
  public void execute(Runnable thread)
  {
    try
    {
      this.threadPool.execute(thread);
    }
    catch (RejectedExecutionException e)
    {
      e.printStackTrace();
    }
  }
}
```

- **ServerMessageDispatcher**

```
package com.greatfree.concurrency;

import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;

/*
 * This is the base of a server message dispatcher. All of the messages sent to the server are dispatched
by the class concurrently. 07/30/2014, Bing Li
 */

// Created: 07/30/2014, Bing Li
public class ServerMessageDispatcher<Message extends ServerMessage> implements
Dispatchable<OutMessageStream<ServerMessage>>
{
  // Declare an instance of ThreadPool that is used to execute threads concurrently. 11/07/2014, Bing Li
  private ThreadPool pool;

  /*
   * Initialize the dispatcher. 11/07/2014, Bing Li
   */
  public ServerMessageDispatcher(int corePoolSize, long keepAliveTime)
  {
    this.pool = new ThreadPool(corePoolSize, keepAliveTime);
  }

  /*
   * The empty implementation to process messages. 11/07/2014, Bing Li
   */
  @Override
  public void consume(OutMessageStream<ServerMessage> msg)
  {
  }

  /*
   * The empty implementation to shutdown the dispatcher. 11/07/2014, Bing Li
   */
  @Override
  public void shutdown()
  {
    this.pool.shutdown();
  }

  /*
   * The empty implementation to start the thread. 11/07/2014, Bing Li
   */
  @Override
  public void execute(Thread thread)
  {
  }
}
```

- **ThreadIdleChecker**

```
package com.greatfree.concurrency;

import java.util.TimerTask;

/*
 * This is a callback thread that runs periodically to call the idle checking method of the thread being
monitored. 11/04/2014, Bing Li
 */

// Created: 11/04/2014, Bing Li
public class ThreadIdleChecker<ThreadInstance extends CheckIdleable> extends TimerTask
{
   // Declare an instance of thread, which must implement the interface of CheckIdleable. 11/04/2014,
Bing Li
   private ThreadInstance thread;

   /*
    * A thread being monitored is input by the initialization. 11/04/2014, Bing Li
    */
   public ThreadIdleChecker(ThreadInstance thread)
   {
      this.thread = thread;
   }

   // Check the thread idle state periodically. 11/04/2014, Bing Li
   @Override
   public void run()
   {
      thread.checkIdle();
   }
}
```

- **MessageProducer**

```
package com.greatfree.concurrency;

import java.util.Queue;
import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;

/*
 * This is a producer/consumer pattern class to input received messages into a concurrency mechanism,
the server dispatcher, smoothly. 07/30/2014, Bing Li
 */

// Created: 07/30/2014, Bing Li
public class MessageProducer<Consumer extends ServerMessageDispatcher<ServerMessage>>
extends Thread
{
  // A dispatcher that processes messages concurrently. 08/04/2014, Bing Li
  private Consumer consumer;
  // A queue to schedule messages in the way of first-in-first-out. 08/04/2014, Bing Li
  private Queue<OutMessageStream<ServerMessage>> queue;
  // The collaborator supports the concurrency control. 08/04/2014, Bing Li
  private Collaborator collaborator;

  // Initializing ... 08/04/2014, Bing Li
  public MessageProducer(Consumer consumer)
  {
    // The consumer is defined and initialized outside the message producer. 08/22/2014, Bing Li
    this.consumer = consumer;
    // Initialize a concurrency controlled queue to keep the messages in a thread-safe way. 08/22/2014,
Bing Li
    this.queue = new LinkedBlockingQueue<OutMessageStream<ServerMessage>>();
    // Initialize a collaborator for the notify-wait mechanism. 08/22/2014, Bing Li
    this.collaborator = new Collaborator();
  }

  /*
   * Disposing ... 08/04/2014, Bing Li
   */
  public void dispose()
  {
    // Set the shutdown flag to true. 08/22/2014, Bing Li
    this.collaborator.setShutdown();
    // Notify all the threads that hold the lock to check the shutdown flag. 08/22/2014, Bing Li
    this.collaborator.signalAll();
    // Since the message producer is shutdown, the queue can be cleared. 08/22/2014, Bing Li
    this.queue.clear();
    // Shutdown the consumer. 08/22/2014, Bing Li
    this.consumer.shutdown();
  }

  /*
   * Push new messages into the queue and notify the associated thread to process. 08/04/2014, Bing Li
   */
  public synchronized void produce(OutMessageStream<ServerMessage> message)
  {
    // Push the messages into the queue. 08/22/2014, Bing Li
    this.queue.add(message);
    // Notify the running thread to process the message. 08/22/2014, Bing Li
    this.collaborator.signal();
  }

  /*
   *    Wait for the available messages to process. 08/04/2014, Bing Li
```

```
   */
  public void run()
  {
    try
    {
      OutMessageStream<ServerMessage> message;
      // An always running loop to keep the thread alive all the time until it is disposed explicitly.
08/22/2014, Bing Li
      while (!this.collaborator.isShutdown())
      {
        // Check whether the queue is empty or not. If messages exist in the queue, all of them must be
processed until the queue is empty. 08/22/2014, Bing Li
        while (!this.queue.isEmpty())
        {
          // Dequeue a message in the queue. 08/22/2014, Bing Li
          message = this.queue.poll();
          // Process the message by the consumer, which is defined outside the class. 08/22/2014, Bing
Li
          this.consumer.consume(message);
        }
        // If the message queue is empty, the thread needs to wait until messages are received.
08/22/2014, Bing Li
        this.collaborator.holdOn();
      }
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
  }
}
```

- **Collaborator**

```
package com.greatfree.concurrency;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

import com.greatfree.util.Tools;

/*
 * The class an integrated one that helps developers to control concurrency. 07/30/2014, Bing Li
 */

// Created: 07/17/2014, Bing Li
public class Collaborator
{
   // A unique key of the class. 07/30/2014, Bing Li
   private String key;
   // The waitLock is the design of JDK Lock. It is used to achieve the goal of concurrency control. 07/30/2014, Bing Li
   private Lock waitLock;
   // The waitCondition is the design of JDK Condition. It is used to achieve the goal of concurrency control. 07/30/2014, Bing Li
   private Condition waitCondition;
   // The flag of shutdown. 07/30/2014, Bing Li
   private boolean isShutdown;
   // The flag of waiting. 07/30/2014, Bing Li
   private boolean isPaused;

   public Collaborator()
   {
      // The key is assigned to a unique String. 07/30/2014, Bing Li
      this.key = Tools.generateUniqueKey();
      this.waitLock = new ReentrantLock();
      this.waitCondition = this.waitLock.newCondition();
      this.isShutdown = false;
      this.isPaused = false;
   }

   /*
    * The class is used to detect whether a managed thread is shutdown. 07/30/2014, Bing Li
    */
   public synchronized boolean isShutdown()
   {
      return this.isShutdown;
   }

   /*
    * The class sets the status of a managed thread to shut down. The thread is shut down when the current task is finished. 07/30/2014, Bing Li
    */
   public synchronized void setShutdown()
   {
      this.isShutdown = true;
   }

   /*
    * To reuse a thread, it is required to reset the shutdown flag by the method. 07/30/2014, Bing Li
    */
   public synchronized void reset()
   {
      this.isShutdown = false;
   }
```

```
  /*
   * The method notifies a thread to acquire the lock to keep going. 07/30/2014, Bing Li
   */
  public void signal()
  {
    this.waitLock.lock();
    this.waitCondition.signal();
    this.waitLock.unlock();
  }

  /*
   * The method notifies all of the managed threads to acquire the lock to keep going. 07/30/2014, Bing
Li
   */
  public void signalAll()
  {
    this.waitLock.lock();
    this.waitCondition.signalAll();
    this.waitLock.unlock();
  }

  /*
   * The method forces the managed thread to wait until it is notified that a lock is available. 07/30/2014,
Bing Li
   */
  public void holdOn() throws InterruptedException
  {
    // Check whether the flag of shutdown is set. The waiting is performed only when the flag of
shutdown is not set. 11/20/2014, Bing Li
    if (!this.isShutdown())
    {
      this.waitLock.lock();
      this.waitCondition.await();
      this.waitLock.unlock();
    }
  }

  /*
   * The method forces the managed thread to wait for some time in the unit of millisecond until it is
notified that a lock is available. 07/30/2014, Bing Li
   */
  public void holdOn(long waitTime) throws InterruptedException
  {
    // Check whether the flag of shutdown is set. The waiting is performed only when the flag of
shutdown is not set. 11/20/2014, Bing Li
    if (!this.isShutdown())
    {
      this.waitLock.lock();
      this.waitCondition.await(waitTime, TimeUnit.MILLISECONDS);
      this.waitLock.unlock();
    }
  }

  /*
   * The key is unique to the class. It is used to keep multiple Collaborators conveniently in a collection,
such as HashMap. 07/30/2014, Bing Li
   */
  public synchronized String getKey()
  {
    return this.key;
  }

  /*
   * To generate a new key for the class. It is used scarcely. Sometimes when multicasting requests, the
key needs to be reset to wait for corresponding responses, e.g., RootRequestBroadcastor. 07/30/2014,
Bing Li
   */
  public synchronized String resetKey()
```

```
    {
        this.key = Tools.generateUniqueKey();
        return this.key;
    }

    /*
     * Set the state of isPause to true. 07/30/2014, Bing Li
     */
    public synchronized void setPause()
    {
        this.isPaused = true;
    }

    /*
     * Set the state of isPaused to false and notify the thread to acquire a lock to continue to work.
07/30/2014, Bing Li
     */
    public void keepOn()
    {
        this.waitLock.lock();
        this.isPaused = false;
        this.waitCondition.signal();
        this.waitLock.unlock();
    }

    /*
     * Check the state of isPaused. 07/30/2014, Bing Li
     */
    public synchronized boolean isPaused()
    {
        return this.isPaused;
    }
}
```

- **ConsumerThread**

```
package com.greatfree.concurrency;

import java.util.Queue;
import java.util.concurrent.LinkedBlockingQueue;

/*
 * This is an implementation of the pattern of producer/consumer. 11/11/2014, Bing Li
 */

// Created: 11/11/2014, Bing Li
public class ConsumerThread<Food, Consumer extends Consumable<Food>> extends Thread
{
  // Declare a queue to take the task. 11/11/2014, Bing Li
  private Queue<Food> queue;
  // Declare an instance of Collaborator that is used to coordinate threads to produce and consume products. 11/11/2014, Bing Li
  private Collaborator collaborator;
  // The consumer that consumes products. It defined how to process the injected food/products. 11/11/2014, Bing Li
  private Consumable<Food> consumer;
  // This is a flag to indicate the all of the food (products) is put into the queue. 11/11/2014, Bing Li
  private boolean isAllFoodQueued;
  // When the queue is empty but some food (products) is still not put into the queue, it is required to wait for some time. The argument defines the length of waiting time. 11/11/2014, Bing Li
  private long waitTime;

  /*
   * Initialize the consumer. 11/20/2014, Bing Li
   */
  public ConsumerThread(Consumable<Food> consumer, long waitTime)
  {
    this.queue = new LinkedBlockingQueue<Food>();
    this.collaborator = new Collaborator();
    this.consumer = consumer;
    this.isAllFoodQueued = false;
    this.waitTime = waitTime;
  }

  /*
   * Dispose the consumer thread. 11/20/2014, Bing Li
   */
  public void dispose()
  {
    // Set the flag of shutdown to terminate the loop in the consumer. 11/20/2014, Bing Li
    this.collaborator.setShutdown();
    // Signal all of the potentially waiting threads that the consumer is dead. They will not keep waiting for the signal. 11/20/2014, Bing Li
    this.collaborator.signalAll();
    // Clear the queue. 11/20/2014, Bing Li
    this.queue.clear();
    try
    {
      // Waiting for the consumer thread to be dead. 11/20/2014, Bing Li
      this.join();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
  }

  /*
   * Return the length of products/food to be consumed. 11/20/2014, Bing Li
   */
```

```
  public int queueLength()
  {
    return this.queue.size();
  }

  /*
   * Produce food such that the consumer can keep eating. 11/20/2014, Bing Li
   */
  public void produce(Food food)
  {
    // Enqueue the food. 11/20/2014, Bing Li
    this.queue.add(food);
    // Notify the waiting thread that is hungry. 11/20/2014, Bing Li
    this.signal();
  }

  /*
   * When no food is available, set the relevant flag. Thus, the consumer can be terminated its running.
11/20/2014, Bing Li
   */
  public synchronized void setIsFoodQueued(boolean isAllFoodQueued)
  {
    // Update the current state of whether food is still available. 11/20/2014, Bing Li
    this.isAllFoodQueued = isAllFoodQueued;
    // Check whether all of the food is enqueued. 11/20/2014, Bing Li
    if (this.isAllFoodQueued)
    {
      // Notify the waiting consumer that it is time to be dismissed. 11/20/2014, Bing Li
      this.signal();
    }
  }

  /*
   * Signal the consumer thread. 11/20/2014, Bing Li
   */
  private void signal()
  {
    this.collaborator.signal();
  }

  /*
   * Consume food/products concurrently. 11/20/2014, Bing Li
   */
  public void run()
  {
    // Declare an instance of Food, i.e., a product. 11/20/2014, Bing Li
    Food food;
    // Check whether the consumer is already shutdown. 11/20/2014, Bing Li
    while (!this.collaborator.isShutdown())
    {
      // Check whether the queue is empty. 11/20/2014, Bing Li
      while (!this.queue.isEmpty())
      {
        // Dequeue the food. 11/2014, Bing Li
        food = this.queue.poll();
        // Consume the food. 11/20/2014, Bing Li
        this.consumer.consume(food);
      }
      // Check whether all of the food is already enqueued. 11/20/2014, Bing Li
      if (!this.isAllFoodQueued)
      {
        // If not all of the food is enqueued, it needs to wait for future newly enqueued food. 11/20/2014,
Bing Li
        try
        {
          this.collaborator.holdOn(this.waitTime);
        }
        catch (InterruptedException e)
```

```
            {
                e.printStackTrace();
            }
        }
        else
        {
            // If all of the food is consumed, it is time to terminate the consumer thread. 11/20/2014, Bing Li
            return;
        }
    }
  }
}
```

- **Consumable**

```
package com.greatfree.concurrency;

/*
 * The interface defines the method for a consumer in the producer/consumer pattern. 11/30/2014, Bing
Li
 */

// Created: 11/11/2014, Bing Li
public interface Consumable<Food>
{
   public void consume(Food food);
}
```

- **CheckIdleable**

```
package com.greatfree.concurrency;

/*
 * The interface defines the signatures of two methods for a thread's idle checking. 11/04/2014, Bing Li
 */

// Created: 11/04/2014, Bing Li
public interface CheckIdleable
{
   // The method signature to check the idle state. 11/04/2014, Bing Li
   public void checkIdle();
   // The method signature to set the configurations of idle checking. 11/04/2014, Bing Li
   public void setIdleChecker(long idleCheckDelay, long idleCheckPeriod);
}
```

- **Dispatchable**

```
package com.greatfree.concurrency;

import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;

/*
 * It defines some interfaces that are needed in ServerMessageDispatcher. 11/07/2014, Bing Li
 */

// Created: 11/07/2014, Bing Li
public interface Dispatchable<Message extends OutMessageStream<ServerMessage>>
{
   // The interface to shutdown the dispatcher. 11/07/2014, Bing Li
   public void shutdown();
   // The interface to process the received messages concurrently. 11/07/2014, Bing Li
   public void consume(Message msg);
   // Start a thread. 11/07/2014, Bing Li
   public void execute(Thread thread);
}
```

- **NotificationQueue**

```
package com.greatfree.concurrency;

import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.multicast.ServerMessage;
import com.greatfree.util.Tools;
import com.greatfree.util.UtilConfig;

/*
 * A fundamental thread that receives and processes notifications in the form of messages concurrently. Notifications are put into a queue and prepare for further processing. It must be derived by sub classes to process specific notifications. 11/04/2014, Bing Li
 */

// Created: 11/04/2014, Bing Li
public class NotificationQueue<Notification extends ServerMessage> extends Thread
{
	// Declare the key for the notification thread. 11/04/2014, Bing Li
	private String key;
	// Declare an instance of LinkedBlockingQueue to take received notifications. 11/04/2014, Bing Li
	private LinkedBlockingQueue<Notification> queue;
	// Declare the size of the queue. 11/04/2014, Bing Li
	private int taskSize;
	// The notify/wait mechanism to implement the producer/consumer pattern. 11/04/2014, Bing Li
	private Collaborator collaborator;
	// The flag that indicates the busy/idle state of the thread. 11/04/2014, Bing Li
	private boolean isIdle;

	/*
	 * Initialize the notification thread. This constructor has no limit on the size of the queue. 11/04/2014, Bing Li
	 */
	public NotificationQueue()
	{
		// Generate the key. 11/04/2014, Bing Li
		this.key = Tools.generateUniqueKey();
		// Initialize the queue without any size constraint. 11/04/2014, Bing Li
		this.queue = new LinkedBlockingQueue<Notification>();
		// Ignore the value of taskSize. 11/04/2014, Bing Li
		this.taskSize = UtilConfig.NO_QUEUE_SIZE;
		// Initialize the collaborator. 11/04/2014, Bing Li
		this.collaborator = new Collaborator();
		// Set the idle state to false. 11/04/2014, Bing Li
		this.isIdle = false;
	}

	/*
	 * Initialize the notification thread. This constructor has a limit on the size of the queue. 11/04/2014, Bing Li
	 */
	public NotificationQueue(int taskSize)
	{
		// Generate the key. 11/04/2014, Bing Li
		this.key = Tools.generateUniqueKey();
		// Initialize the queue with the particular size constraint. 11/04/2014, Bing Li
		this.queue = new LinkedBlockingQueue<Notification>(taskSize);
		// Set the value of taskSize. 11/04/2014, Bing Li
		this.taskSize = taskSize;
		// Initialize the collaborator. 11/04/2014, Bing Li
		this.collaborator = new Collaborator();
		// Set the idle state to false. 11/04/2014, Bing Li
		this.isIdle = false;
	}
```

```
  /*
   * Dispose the notification thread. 11/04/2014, Bing Li
   */
  public synchronized void dispose()
  {
    // Set the flag to be the state of being shutdown. 11/04/2014, Bing Li
    this.collaborator.setShutdown();
    // Notify the thread being waiting to go forward. Since the shutdown flag is set, the thread must die
for the notification. 11/04/2014, Bing Li
    this.collaborator.signalAll();
    try
    {
      // Wait for the thread to die. 11/04/2014, Bing Li
      this.join();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
    // Clear the queue to release resources. 11/04/2014, Bing Li
    if (this.queue != null)
    {
      this.queue.clear();
    }
  }

  /*
   * Expose the key for the convenient management. 11/04/2014, Bing Li
   */
  public String getKey()
  {
    return this.key;
  }

  /*
   * Enqueue a notification into the thread. 11/04/2014, Bing Li
   */
  public void enqueue(Notification notification) throws IllegalStateException
  {
    // Set the state of the thread to be busy. 11/04/2014, Bing Li
    this.setBusy();
    // Enqueue the notification. 11/04/2014, Bing Li
    this.queue.add(notification);
    // Notify the waiting thread to keep on working since new notifications are received. 09/22/2014, Bing
Li
    this.collaborator.signal();
  }

  /*
   * Set the state to be busy. 11/04/2014, Bing Li
   */
  private synchronized void setBusy()
  {
    this.isIdle = false;
  }

  /*
   * Set the state to be idle. 11/04/2014, Bing Li
   */
  private synchronized void setIdle()
  {
    this.isIdle = true;
  }

  /*
   * Get the current size of the queue. 11/04/2014, Bing Li
   */
  public int getQueueSize()
```

```
  {
    return this.queue.size();
  }

  /*
   * The method intends to stop the thread temporarily when no notifications are available. A thread is
identified as being idle immediately after the temporary waiting is finished. 11/04/2014, Bing Li
   */
  public void holdOn(long waitTime) throws InterruptedException
  {
    // Wait for some time, which is determined by the value of waitTime. 11/04/2014, Bing Li
    this.collaborator.holdOn(waitTime);
    // Set the state of the thread to be idle after waiting for some time. 11/04/2014, Bing Li
    this.setIdle();
  }

  /*
   * Check whether the shutdown flag of the thread is set or not. It might take some time for the thread to
be shutdown practically even though the flag is set. 11/04/2014, Bing Li
   */
  public boolean isShutdown()
  {
    return this.collaborator.isShutdown();
  }

  /*
   * Check whether the current size of the queue reaches the upper limit. 11/04/2014, Bing Li
   */
  public boolean isFull()
  {
    return this.queue.size() >= this.taskSize;
  }

  /*
   * Check whether the shutdown flag of the thread is set or not. It might take some time for the thread to
be shutdown practically even though the flag is set. 11/04/2014, Bing Li
   */
  public boolean isEmpty()
  {
    return this.queue.size() <= 0;
  }

  /*
   * Check whether the thread is idle or not. 11/04/2014, Bing Li
   */
  public synchronized boolean isIdle()
  {
    return this.isIdle;
  }

  /*
   * Dequeue the notification from the queue. 11/04/2014, Bing Li
   */
  public Notification getNotification() throws InterruptedException
  {
    return this.queue.take();
  }

  /*
   * Get the notification but leave the notification in the queue. 11/04/2014, Bing Li
   */
  public Notification peekNotification()
  {
    return this.queue.peek();
  }

  /*
   * Dispose the notification. 09/22/2014, Bing Li
```

```
     */
    public synchronized void disposeMessage(Notification notification)
    {
        notification = null;
    }
}
```

- **NotifictionDispatcher**

```
package com.greatfree.concurrency;

import java.util.HashMap;
import java.util.Map;
import java.util.Timer;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.multicast.ServerMessage;
import com.greatfree.util.CollectionSorter;
import com.greatfree.util.UtilConfig;

/*
 * This is an important class that enqueues notifications and creates threads to process them concurrently. It works in the way like a dispatcher. That is why it is named. 11/04/2014, Bing Li
 */

// Created: 11/04/2014, Bing Li
public class NotificationDispatcher<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>, ThreadCreator extends NotificationThreadCreatable<Notification, NotificationThread>> extends Thread implements CheckIdleable
{
	// Declare a map to contain all of the threads. 11/04/2014, Bing Li
	private Map<String, NotificationThread> threadMap;
	// Declare a queue to contain notifications. Here the queue is specified when declaring. It aims to use the method take() instead of poll() since take() waits when no data in the queue while poll() returns null. The effect of take() is usually preferred. 11/04/2014, Bing Li
	private LinkedBlockingQueue<Notification> notificationQueue;
	// Declare a thread pool that is used to run a thread. 11/04/2014, Bing Li
	private ThreadPool threadPool;
	// Declare a thread creator that is used to initialize a thread instance. 11/04/2014, Bing Li
	private ThreadCreator threadCreator;
	// Declare the maximum task length for each thread to be created. 11/04/2014, Bing Li
	private int maxTaskSize;
	// Declare the maximum thread count that can be created in the dispatcher. 11/04/2014, Bing Li
	private int maxThreadSize;
	// Declare a timer that controls the task of idle checking. 11/04/2014, Bing Li
	private Timer checkTimer;
	// Declare the checker to check whether created threads are idle long enough. 11/04/2014, Bing Li
	private ThreadIdleChecker<NotificationDispatcher<Notification, NotificationThread, ThreadCreator>> idleChecker;
	// The collaborator is used to pause the dispatcher when no notifications are available and notify to continue when new notifications are received. 11/04/2014, Bing Li
	private Collaborator workCollaborator;
	// The time to wait when no notifications are available. 11/04/2014, Bing Li
	private long dispatcherWaitTime;
	// A flag that indicates whether a local thread pool is used or a global one is used. Since it might get problems if the global pool is shutdown inside the class. Thus, it is required to set the flag. 11/19/2014, Bing Li
	private boolean isSelfThreadPool;

	/*
	 * When a server dispatcher has a local thread pool for each message dispatcher, the constructor is used. 11/04/2014, Bing Li
	 */
	public NotificationDispatcher(int poolSize, long keepAliveTime, ThreadCreator threadCreator, int maxTaskSize, int maxThreadSize, long dispatcherWaitTime)
	{
		this.threadMap = new ConcurrentHashMap<String, NotificationThread>();
		this.notificationQueue = new LinkedBlockingQueue<Notification>();
		this.threadPool = new ThreadPool(poolSize, keepAliveTime);
		this.threadCreator = threadCreator;
		this.maxTaskSize = maxTaskSize;
```

```
    this.maxThreadSize = maxThreadSize;
    this.checkTimer = UtilConfig.NO_TIMER;
    this.workCollaborator = new Collaborator();
    this.dispatcherWaitTime = dispatcherWaitTime;
    this.isSelfThreadPool = true;
  }

  /*
   * When a server dispatcher has a global thread pool, the constructor is used. 11/04/2014, Bing Li
   */
  public NotificationDispatcher(ThreadPool threadPool, ThreadCreator threadCreator, int
maxTaskSize, int maxThreadSize, long dispatcherWaitTime)
  {
    this.threadMap = new ConcurrentHashMap<String, NotificationThread>();
    this.notificationQueue = new LinkedBlockingQueue<Notification>();
    this.threadPool = threadPool;
    this.threadCreator = threadCreator;
    this.maxTaskSize = maxTaskSize;
    this.maxThreadSize = maxThreadSize;
    this.checkTimer = UtilConfig.NO_TIMER;
    this.workCollaborator = new Collaborator();
    this.dispatcherWaitTime = dispatcherWaitTime;
    this.isSelfThreadPool = true;
  }

  /*
   * Dispose the notification dispatcher. 11/04/2014, Bing Li
   */
  public synchronized void dispose()
  {
    // Set the shutdown flag to be true. Thus, the loop in the dispatcher thread to schedule notification
loads is terminated. 11/04/2014, Bing Li
    this.workCollaborator.setShutdown();
    // Notify the dispatcher thread that is waiting for the notifications to terminate the waiting.
11/04/2014, Bing Li
    this.workCollaborator.signalAll();
    // Clear the notification queue. 11/04/2014, Bing Li
    if (this.notificationQueue != null)
    {
      this.notificationQueue.clear();
    }
    // Cancel the timer that controls the idle checking. 11/04/2014, Bing Li
    if (this.checkTimer != UtilConfig.NO_TIMER)
    {
      this.checkTimer.cancel();
    }
    // Terminate the periodically running thread for idle checking. 11/04/2014, Bing Li
    if (this.idleChecker != null)
    {
      this.idleChecker.cancel();
    }
    // Dispose all of threads created during the dispatcher's running procedure. 11/04/2014, Bing Li
    for (NotificationThread thread : this.threadMap.values())
    {
      thread.dispose();
    }
    // Clear the threads. 11/04/2014, Bing Li
    this.threadMap.clear();
    // Check whether the thread pool is local or not. 11/19/2014, Bing Li
    if (this.isSelfThreadPool)
    {
      // Shutdown the thread pool if it belongs to the instance of the class only. 11/04/2014, Bing Li
      this.threadPool.shutdown();
    }
    // Dispose the thread creator. 11/04/2014, Bing Li
    this.threadCreator = null;
  }
```

```
  /*
   * Set the idle checking parameters. 11/04/2014, Bing Li
   */
  @Override
  public void setIdleChecker(long idleCheckDelay, long idleCheckPeriod)
  {
    // Initialize the timer. 11/04/2014, Bing Li
    this.checkTimer = new Timer();
    // Initialize the idle checker. 11/04/2014, Bing Li
    this.idleChecker = new ThreadIdleChecker<NotificationDispatcher<Notification, NotificationThread,
ThreadCreator>>(this);
    // Schedule the idle checking task. 11/04/2014, Bing Li
    this.checkTimer.schedule(this.idleChecker, idleCheckDelay, idleCheckPeriod);
  }

  /*
   * The method is called back by the idle checker periodically to monitor the idle states of threads within
the dispatcher. 11/04/2014, Bing Li
   */
  @Override
  public void checkIdle()
  {
    // Check each thread managed by the dispatcher. 11/04/2014, Bing Li
    for (NotificationThread thread : this.threadMap.values())
    {
      // If the thread is empty and idle, it is the one to be checked. 11/04/2014, Bing Li
      if (thread.isEmpty() && thread.isIdle())
      {
        // The algorithm to determine whether a thread should be disposed or not is simple. When it is
checked to be idle, it is time to dispose it. 11/04/2014, Bing Li
        this.threadMap.remove(thread.getKey());
        // Dispose the thread. 11/04/2014, Bing Li
        thread.dispose();
        // Collect the resource of the thread. 11/04/2014, Bing Li
        thread = null;
      }
    }
  }

  /*
   * Enqueue the newly received notification into the dispatcher. 11/04/2014, Bing Li
   */
  public synchronized void enqueue(Notification notification)
  {
    // Enqueue the notification into the queue. 11/04/2014, Bing Li
    this.notificationQueue.add(notification);
    // Notify the dispatcher thread, which is possibly blocked when no requests are available, to keep
working. 11/04/2014, Bing Li
    this.workCollaborator.signal();
  }

  /*
   * The thread of the dispatcher is always running until no notifications from remote nodes will be
received. If too many notifications are received, more threads are created by the dispatcher. If
notifications are limited, the count of threads created by the dispatcher is also small. It is possible no any
threads are alive when no notifications are received for a long time. 11/04/2014, Bing Li
   */
  public void run()
  {
    // Declare a notification. 11/04/2014, Bing Li
    Notification notification;
    // Initialize a task map to calculate the load of each thread. 11/05/2014, Bing Li
    Map<String, Integer> threadTaskMap = new HashMap<String, Integer>();
    // Declare a string to keep the selected thread key. 11/04/2014, Bing Li
    String selectedThreadKey = UtilConfig.NO_KEY;
    // The dispatcher usually runs all of the time unless the server is shutdown. To shutdown the
dispatcher, the shutdown flag of the collaborator is set to true. 11/05/2014, Bing Li
    while (!this.workCollaborator.isShutdown())
```

```
    {
      try
      {
        // Check whether notifications are received and saved in the queue. 11/05/2014, Bing Li
        while (!this.notificationQueue.isEmpty())
        {
          // Dequeue the notification from the queue of the dispatcher. 11/05/2014, Bing Li
          notification = this.notificationQueue.take();

          // Since all of the threads created by the dispatcher are saved in the map by their unique keys, it is necessary to check whether any alive threads are available. If so, it is possible to assign tasks to them if they are not so busy. 11/05/2014, Bing Li
          while (this.threadMap.size() > 0)
          {
            // Clear the map to start to calculate the loads of those threads. 11/05/2014, Bing Li
            threadTaskMap.clear();

            // Each thread's workload is saved into the threadTaskMap. 11/05/2014, Bing Li
            for (NotificationThread thread : this.threadMap.values())
            {
              threadTaskMap.put(thread.getKey(), thread.getQueueSize());
            }
            // Select the thread whose load is the least and keep the key of the thread. 11/05/2014, Bing Li
            selectedThreadKey = CollectionSorter.minValueKey(threadTaskMap);
            // Since no concurrency is applied here, it is possible that the key is invalid. Thus, just check here. 11/19/2014, Bing Li
            if (selectedThreadKey != null)
            {
              // Since no concurrency is applied here, it is possible that the key is out of the map. Thus, just check here. 11/19/2014, Bing Li
              if (this.threadMap.containsKey(selectedThreadKey))
              {
                try
                {
                  // Check whether the thread's load reaches the maximum value. 11/05/2014, Bing Li
                  if (this.threadMap.get(selectedThreadKey).isFull())
                  {
                    // Check if the pool is full. If the least load thread is full as checked by the above condition, it denotes that all of the current alive threads are full. So it is required to create a thread to respond the newly received notifications if the thread count of the pool does not reach the maximum. 11/05/2014, Bing Li
                    if (this.threadMap.size() < this.maxThreadSize)
                    {
                      // Create a new thread. 11/05/2014, Bing Li
                      NotificationThread thread = this.threadCreator.createNotificationThreadInstance(this.maxTaskSize);
                      // Save the newly created thread into the map. 11/05/2014, Bing Li
                      this.threadMap.put(thread.getKey(), thread);
                      // Enqueue the notification into the queue of the newly created thread. Then, the notification will be processed by the thread. 11/05/2014, Bing Li
                      this.threadMap.get(thread.getKey()).enqueue(notification);
                      // Start the thread by the thread pool. 11/05/2014, Bing Li
                      this.threadPool.execute(this.threadMap.get(thread.getKey()));
                    }
                    else
                    {
                      // Force to put the notification into the queue when the count of threads reaches the upper limit and each of the thread's queue is full. 11/05/2014, Bing Li
                      this.threadMap.get(selectedThreadKey).enqueue(notification);
                    }
                  }
                  else
                  {
                    // If the least load thread's queue is not full, just put the notification into the queue. 11/05/2014, Bing Li
                    this.threadMap.get(selectedThreadKey).enqueue(notification);
                  }
```

```
                    // Jump out from the loop since the notification is put into a thread. 11/05/2014, Bing Li
                    break;
                  }
                  catch (NullPointerException e)
                  {
                    // Since no concurrency is applied here, it is possible that a NullPointerException is
raised. If so, it means that the selected thread is not available. Just continue to select anther one.
11/19/2014, Bing Li
                    continue;
                  }
                }
              }
            }
            // If no threads are available, it needs to create a new one to take the notification. 11/05/2014,
Bing Li
            if (this.threadMap.size() <= 0)
            {
              // Create a new thread. 11/05/2014, Bing Li
              NotificationThread thread =
this.threadCreator.createNotificationThreadInstance(this.maxTaskSize);
              // Put it into the map for further reuse. 11/05/2014, Bing Li
              this.threadMap.put(thread.getKey(), thread);
              // Take the notification. 11/05/2014, Bing Li
              this.threadMap.get(thread.getKey()).enqueue(notification);
              // Start the thread. 11/05/2014, Bing Li
              this.threadPool.execute(this.threadMap.get(thread.getKey()));
            }

            // If the dispatcher is shutdown, it is not necessary to keep processing the notifications. So,
jump out the loop and the thread is dead. 11/05/2014, Bing Li
            if (this.workCollaborator.isShutdown())
            {
              break;
            }
          }

          // Check whether the dispatcher is shutdown or not. 11/05/2014, Bing Li
          if (!this.workCollaborator.isShutdown())
          {
            // If the dispatcher is still alive, it denotes that no notifications are available temporarily. Just
wait for a while. 11/05/2014, Bing Li
            this.workCollaborator.holdOn(this.dispatcherWaitTime);
          }
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
    }
  }
}
```

- **NotificationThreadCreatable**

```
package com.greatfree.concurrency;

import com.greatfree.multicast.ServerMessage;

/*
 * The interface defines the method signature to create the instances of NotificationQueue. 11/04/2014,
Bing Li
 */

// Created: 11/04/2014, Bing Li
public interface NotificationThreadCreatable<Notification extends ServerMessage,
NotificationThread extends NotificationQueue<Notification>>
{
   public NotificationThread createNotificationThreadInstance(int taskSize);
}
```

- **RequestQueue**

```
package com.greatfree.concurrency;

import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.Queue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.locks.Lock;

import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.util.Tools;

/*
 * This is a thread that receive requests from a client, put those messages into a queue and prepare for further processing. It must be derived by sub classes to provide the real responses for the requests. 09/22/2014, Bing Li
 */

// Created: 09/22/2014, Bing Li
public class RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage> extends Thread
{
	// The unique key of the thread. It is convenient for managing it by a table-like mechanism. 09/22/2014, Bing Li
	private String key;
	// The queue that saves the request stream, which extends the OutMessageStream, including the associated output stream,  the lock and the request. 09/22/2014, Bing Li
	private Queue<Stream> queue;
	// The maximum size of the queue. 09/22/2014, Bing Li
	private int maxTaskSize;
	// It is necessary to keep the thread waiting when no requests are available. The collaborator is used to notify the thread to keep working when requests are received. When the thread is idle enough, it can be collected. The collaborator is also used to control the life cycle of the thread. 09/22/2014, Bing Li
	private Collaborator collaborator;
	// The flag that represents whether the thread is busy or idle. 09/22/2014, Bing Li
	private boolean isIdle;

	/*
	 * Initialize an instance. 09/22/2014, Bing Li
	 */
	public RequestQueue(int maxTaskSize)
	{
		// Generate a unique key for the instance of the class. 09/22/2014, Bing Li
		this.key = Tools.generateUniqueKey();
		// Initialize the queue to keep received the request streams, which consist of requests and their relevant streams and locks. It is critical to set up the queue without the limit of the length since the message is forced to be put into the queue when the count of threads reach the maximum and each one's queue is full. 09/22/2014, Bing Li
		this.queue = new LinkedBlockingQueue<Stream>();
		this.maxTaskSize = maxTaskSize;
		this.collaborator = new Collaborator();
		// Setting the idle is false means that the thread is busy when being initialized. 09/22/2014, Bing Li
		this.isIdle = false;
	}

	/*
	 * Dispose the instance of the class. 09/22/2014, Bing Li
	 */
	public synchronized void dispose()
	{
		// Set the flag to be the state of being shutdown. 09/22/2014, Bing Li
		this.collaborator.setShutdown();
		// Notify the thread being waiting to go forward. Since the shutdown flag is set, the thread must die for the notification. 09/22/2014, Bing Li
```

```
    this.collaborator.signal();
    try
    {
      // Wait for the thread to die. 09/22/2014, Bing Li
      this.join();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
    // Clear the queue to release resources. 09/22/2014, Bing Li
    if (this.queue != null)
    {
      this.queue.clear();
    }
  }

  /*
   * Expose the key for the convenient management. 09/22/2014, Bing Li
   */
  public String getKey()
  {
    return this.key;
  }

  /*
   * Enqueue the wrapper, the request stream, for the request, the output stream and the lock.
09/22/2014, Bing Li
   */
  public void enqueue(Stream request)
  {
    // Set the state of the thread to be busy. 09/22/2014, Bing Li
    this.setBusy();
    // Enqueue the request and its relevant output stream and lock. 09/22/2014, Bing Li
    this.queue.add(request);
    // Notify the waiting thread to keep on working since new requests are received. 09/22/2014, Bing Li
    this.collaborator.signal();
  }

  /*
   * Set the state to be busy. 09/22/2014, Bing Li
   */
  private synchronized void setBusy()
  {
    this.isIdle = false;
  }

  /*
   * Set the state to be idle. 09/22/2014, Bing Li
   */
  private synchronized void setIdle()
  {
    this.isIdle = true;
  }

  /*
   * The method intends to stop the thread temporarily when no requests are available. A thread is
identified as being idle immediately after the temporary waiting is finished. 09/22/2014, Bing Li
   */
  public void holdOn(long waitTime) throws InterruptedException
  {
    // Wait for some time, which is determined by the value of waitTime. 09/22/2014, Bing Li
    this.collaborator.holdOn(waitTime);
    // Set the state of the thread to be idle after waiting for some time. 09/22/2014, Bing Li
    this.setIdle();
  }

  /*
```

```
   * Check whether the shutdown flag of the thread is set or not. It might take some time for the thread to
be shutdown practically even though the flag is set. 09/22/2014, Bing Li
   */
  public boolean isShutdown()
  {
    return this.collaborator.isShutdown();
  }

  /*
   * Check whether the queue is empty. 09/22/2014, Bing Li
   */
  public boolean isEmpty()
  {
    return this.queue.size() <= 0;
  }

  /*
   * Check whether the current size of the queue reaches the upper limit. 09/22/2014, Bing Li
   */
  public boolean isFull()
  {
    return this.queue.size() >= this.maxTaskSize;
  }

  /*
   * Check whether the thread is idle or not. 09/22/2014, Bing Li
   */
  public synchronized boolean isIdle()
  {
    return this.isIdle;
  }

  /*
   * Get the current size of the queue. 09/22/2014, Bing Li
   */
  public int getQueueSize()
  {
    return this.queue.size();
  }

  /*
   * Dequeue the request stream from the queue. The stream includes the request, the output stream
and the lock. They are dequeued for processing and responding. 09/22/2014, Bing Li
   */
  public Stream getRequest()
  {
    return this.queue.poll();
  }

  /*
   * After the response is created, the method is responsible for sending it back to the client. 09/22/2014,
Bing Li
   */
  public synchronized void respond(ObjectOutputStream out, Lock outLock, Response response)
throws IOException
  {
    // The lock is shared by all of threads that use the output stream. It makes sure that the responding
operations are performed in an atomic way.
    outLock.lock();
    try
    {
      // Send the response to the remote end. 09/17/2014, Bing Li
      out.writeObject(response);
      // It is required to invoke the below methods to avoid the memory leak. 09/22/2014, Bing Li
      out.flush();
      out.reset();
    }
    finally
```

```
      {
        outLock.unlock();
      }
    }

    /*
     * Dispose the request stream and the response. 09/22/2014, Bing Li
     */
    public synchronized void disposeMessage(Stream request, Response response)
    {
      request.disposeMessage();
      response = null;
    }

    /*
     * Dispose the request stream. Sometimes the response needs to be forwarded. So it should be
disposed. 09/22/2014, Bing Li
     */
    public synchronized void disposeMessage(Stream request)
    {
      request.disposeMessage();
    }
}
```

- **RequestDispatcher**

```
package com.greatfree.concurrency;

import java.util.HashMap;
import java.util.Map;
import java.util.Timer;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.util.CollectionSorter;
import com.greatfree.util.UtilConfig;

/*
 * This is an important class that enqueues requests and creates threads to respond them concurrently.
It works in the way like a dispatcher. That is why it is named. 11/04/2014, Bing Li
 */

// Created: 11/04/2014, Bing Li
public class RequestDispatcher<Request extends ServerMessage, Stream extends
OutMessageStream<Request>, Response extends ServerMessage, RequestThread extends
RequestQueue<Request, Stream, Response>, ThreadCreator extends
RequestThreadCreatable<Request, Stream, Response, RequestThread>> extends Thread
implements CheckIdleable
{
  // Declare a map to contain all of the threads. 11/04/2014, Bing Li
  private Map<String, RequestThread> threadMap;
  // Declare a queue to contain message streams. 11/04/2014, Bing Li
  private LinkedBlockingQueue<Stream> requestQueue;
  // Declare a thread pool that is used to run a thread. 11/04/2014, Bing Li
  private ThreadPool threadPool;
  // Declare a thread creator that is used to initialize a thread instance. 11/04/2014, Bing Li
  private ThreadCreator threadCreator;
  // Declare the maximum task length for each thread to be created. 11/04/2014, Bing Li
  private int maxTaskSize;
  // Declare the maximum thread count that can be created in the dispatcher. 11/04/2014, Bing Li
  private int maxThreadSize;
  // Declare a timer that controls the task of idle checking. 11/04/2014, Bing Li
  private Timer checkTimer;
  // Declare the checker to check whether created threads are idle long enough. 11/04/2014, Bing Li
  private ThreadIdleChecker<RequestDispatcher<Request, Stream, Response, RequestThread,
ThreadCreator>> idleChecker;
  // The collaborator is used to pause the dispatcher when no requests are available and notify to
continue when new requests are received. 11/04/2014, Bing Li
  private Collaborator workCollaborator;
  // The time to wait when no requests are available. 11/04/2014, Bing Li
  private long dispatcherWaitTime;

  /*
   * Initialize the request dispatcher. 11/04/2014, Bing Li
   */
  public RequestDispatcher(int poolSize, long keepAliveTime, ThreadCreator threadCreator, int
maxTaskSize, int maxThreadSize, long dispatcherWaitTime)
  {
    this.threadMap = new ConcurrentHashMap<String, RequestThread>();
    this.requestQueue = new LinkedBlockingQueue<Stream>();
    this.threadPool = new ThreadPool(poolSize, keepAliveTime);
    this.threadCreator = threadCreator;
    this.maxTaskSize = maxTaskSize;
    this.maxThreadSize = maxThreadSize;
    this.checkTimer = UtilConfig.NO_TIMER;
    this.workCollaborator = new Collaborator();
    this.dispatcherWaitTime = dispatcherWaitTime;
  }
```

```
  /*
   * Dispose the request dispatcher. 11/04/2014, Bing Li
   */
  public synchronized void dispose()
  {
    // Set the shutdown flag to be true. Thus, the loop in the dispatcher thread to schedule requests load
is terminated. 11/04/2014, Bing Li
    this.workCollaborator.setShutdown();
    // Notify the dispatcher thread that is waiting for the requests to terminate the waiting. 11/04/2014,
Bing Li
    this.workCollaborator.signalAll();
    // Clear the request queue. 11/04/2014, Bing Li
    if (this.requestQueue != null)
    {
      this.requestQueue.clear();
    }
    // Cancel the timer that controls the idle checking. 11/04/2014, Bing Li
    if (this.checkTimer != UtilConfig.NO_TIMER)
    {
      this.checkTimer.cancel();
    }
    // Terminate the periodically running thread for idle checking. 11/04/2014, Bing Li
    if (this.idleChecker != null)
    {
      this.idleChecker.cancel();
    }
    // Dispose all of threads created during the dispatcher's running procedure. 11/04/2014, Bing Li
    for (RequestThread thread : this.threadMap.values())
    {
      thread.dispose();
    }
    // Clear the threads. 11/04/2014, Bing Li
    this.threadMap.clear();
    // Shutdown the thread pool. 11/04/2014, Bing Li
    this.threadPool.shutdown();
    // Dispose the thread creator. 11/04/2014, Bing Li
    this.threadCreator = null;
  }

  /*
   * The method is called back by the idle checker periodically to monitor the idle states of threads within
the dispatcher. 11/04/2014, Bing Li
   */
  @Override
  public void checkIdle()
  {
    // Check each thread managed by the dispatcher. 11/04/2014, Bing Li
    for (RequestThread thread : this.threadMap.values())
    {
      // If the thread is empty and idle, it is the one to be checked. 11/04/2014, Bing Li
      if (thread.isEmpty() && thread.isIdle())
      {
        // The algorithm to determine whether a thread should be disposed or not is simple. When it is
checked to be idle, it is time to dispose it. 11/04/2014, Bing Li
        this.threadMap.remove(thread.getKey());
        // Dispose the thread. 11/04/2014, Bing Li
        thread.dispose();
        // Collect the resource of the thread. 11/04/2014, Bing Li
        thread = null;
      }
    }
  }

  /*
   * Set the idle checking parameters. 11/04/2014, Bing Li
   */
  @Override
```

```
  public void setIdleChecker(long idleCheckDelay, long idleCheckPeriod)
  {
    // Initialize the timer. 11/04/2014, Bing Li
    this.checkTimer = new Timer();
    // Initialize the idle checker. 11/04/2014, Bing Li
    this.idleChecker = new ThreadIdleChecker<RequestDispatcher<Request, Stream, Response,
RequestThread, ThreadCreator>>(this);
    // Schedule the idle checking task. 11/04/2014, Bing Li
    this.checkTimer.schedule(this.idleChecker, idleCheckDelay, idleCheckPeriod);
  }

  /*
   * Enqueue the newly received request stream into the dispatcher. 11/04/2014, Bing Li
   */
  public synchronized void enqueue(Stream request)
  {
    // Enqueue the request stream into the queue. 11/04/2014, Bing Li
    this.requestQueue.add(request);
    // Notify the dispatcher thread, which is possibly blocked when no requests are available, to keep
working. 11/04/2014, Bing Li
    this.workCollaborator.signal();
  }

  /*
   * The thread of the dispatcher is always running until no requests from clients will be received. If too
many requests are received, more threads are created by the dispatcher to respond users in time. If
requests are limited, the count of threads created by the dispatcher is also small. It is possible no any
threads are alive when no requests are received for a long time. 11/04/2014, Bing Li
   */
  public void run()
  {
    // Declare a request stream. 11/04/2014, Bing Li
    Stream request;
    // Initialize a thread map to calculate the load of each thread. 11/04/2014, Bing Li
    Map<String, Integer> threadTaskMap = new HashMap<String, Integer>();
    // Declare a string to keep the selected thread key. 11/04/2014, Bing Li
    String selectedThreadKey = UtilConfig.NO_KEY;
    // The dispatcher usually runs all of the time unless the server is shutdown. To shutdown the
dispatcher, the shutdown flag of the collaborator is set to true. 11/04/2014, Bing Li
    while (!this.workCollaborator.isShutdown())
    {
      try
      {
        // Check whether requests are received and saved in the queue. 11/04/2014, Bing Li
        while (!this.requestQueue.isEmpty())
        {
          // Dequeue the request from the queue of the dispatcher. 11/04/2014, Bing Li
          request = this.requestQueue.take();

          // Since all of the threads created by the dispatcher are saved in the map by their unique keys,
it is necessary to check whether any alive threads are available. If so, it is possible to assign tasks to
them if they are not so busy. 11/04/2014, Bing Li
          while (this.threadMap.size() > 0)
          {
            // Clear the map to start to calculate the load those threads. 11/04/2014, Bing Li
            threadTaskMap.clear();

            // Each thread's workload is saved into the threadTaskMap. 11/04/2014, Bing Li
            for (RequestThread thread : this.threadMap.values())
            {
              threadTaskMap.put(thread.getKey(), thread.getQueueSize());
            }
            // Select the thread whose load is the least and keep the key of the thread. 11/04/2014, Bing
Li
            selectedThreadKey = CollectionSorter.minValueKey(threadTaskMap);
            // Since no concurrency is applied here, it is possible that the key is invalid. Thus, just check
here. 11/19/2014, Bing Li
            if (selectedThreadKey != null)
```

```
            {
                // Since no concurrency is applied here, it is possible that the key is out of the map. Thus,
just check here. 11/19/2014, Bing Li
                if (this.threadMap.containsKey(selectedThreadKey))
                {
                  try
                  {
                    // Check whether the thread's load reaches the maximum value. 11/04/2014, Bing Li
                    if (this.threadMap.get(selectedThreadKey).isFull())
                    {
                      // Check if the pool is full. If the least load thread is full as checked by the above
condition, it denotes that all of the current alive threads are full. So it is required to create a thread to
respond the newly received requests if the thread count of the pool does not reach the maximum.
11/04/2014, Bing Li
                      if (this.threadMap.size() < this.maxThreadSize)
                      {
                        // Create a new thread. 11/04/2014, Bing Li
                        RequestThread thread =
this.threadCreator.createRequestThreadInstance(this.maxTaskSize);
                        // Save the newly created thread into the map. 11/04/2014, Bing Li
                        this.threadMap.put(thread.getKey(), thread);
                        // Enqueue the request into the queue of the newly created thread. Then, the
request will be processed and responded by the thread. 11/04/2014, Bing Li
                        this.threadMap.get(thread.getKey()).enqueue(request);
                        // Start the thread by the thread pool. 11/04/2014, Bing Li
                        this.threadPool.execute(this.threadMap.get(thread.getKey()));
                      }
                      else
                      {
                        // Force to put the request into the queue when the count of threads reaches the
upper limit and each of the thread's queue is full. 11/04/2014, Bing Li
                        this.threadMap.get(selectedThreadKey).enqueue(request);
                      }
                    }
                    else
                    {
                      // If the least load thread's queue is not full, just put the request into the queue.
11/04/2014, Bing Li
                      this.threadMap.get(selectedThreadKey).enqueue(request);
                    }

                    // Jump out from the loop since the request is put into a thread. 11/04/2014, Bing Li
                    break;
                  }
                  catch (NullPointerException e)
                  {
                    // Since no concurrency is applied here, it is possible that a NullPointerException is
raised. If so, it means that the selected thread is not available. Just continue to select anther one.
11/19/2014, Bing Li
                    continue;
                  }
                }
              }
            }
            // If no threads are available, it needs to create a new one to take the request. 11/04/2014,
Bing Li
            if (this.threadMap.size() <= 0)
            {
              // Create a new thread. 11/04/2014, Bing Li
              RequestThread thread =
this.threadCreator.createRequestThreadInstance(this.maxTaskSize);
              // Put it into the map for further reuse. 11/04/2014, Bing Li
              this.threadMap.put(thread.getKey(), thread);
              // Take the request. 11/04/2014, Bing Li
              this.threadMap.get(thread.getKey()).enqueue(request);
              // Start the thread. 11/04/2014, Bing Li
              this.threadPool.execute(this.threadMap.get(thread.getKey()));
            }
```

```
            // If the dispatcher is shutdown, it is not necessary to keep processing the requests. So, jump out the loop and the thread is dead. 11/04/2014, Bing Li
            if (this.workCollaborator.isShutdown())
            {
              break;
            }
          }

          // Check whether the dispatcher is shutdown or not. 11/04/2014, Bing Li
          if (!this.workCollaborator.isShutdown())
          {
            // If the dispatcher is still alive, it denotes that no requests are available temporarily. Just wait for a while. 11/04/2014, Bing Li
            this.workCollaborator.holdOn(this.dispatcherWaitTime);
          }
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
    }
  }
}
```

- **RequestThreadCreatable**

```
package com.greatfree.concurrency;

import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;

/*
 * The interface defines a method signature that creates a thread to respond users' requests.
11/04/2014, Bing Li
 */

// Created: 11/04/2014, Bing Li
public interface RequestThreadCreatable<Request extends ServerMessage, Stream extends
OutMessageStream<Request>, Response extends ServerMessage, RequestThread extends
RequestQueue<Request, Stream, Response>>
{
   // The method signature to create an instance of RequestThread to respond users' requests.
11/04/2014, Bing Li
   public RequestThread createRequestThreadInstance(int taskSize);
}
```

- **NotificationObjectQueue**

```
package com.greatfree.concurrency;

import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.util.Tools;
import com.greatfree.util.UtilConfig;

/*
 * A fundamental thread that receives and processes notifications as an object concurrently. Notifications are put into a queue and prepare for further processing. It must be derived by sub classes to process specific notifications. 11/20/2014, Bing Li
 */

// Created: 11/20/2014, Bing Li
public class NotificationObjectQueue<Notification> extends Thread
{
	// Declare the key for the notification thread. 11/20/2014, Bing Li
	private String key;
	// Declare an instance of LinkedBlockingQueue to take received notifications. 11/20/2014, Bing Li
	private LinkedBlockingQueue<Notification> queue;
	// Declare the size of the queue. 11/20/2014, Bing Li
	private int taskSize;
	// The notify/wait mechanism to implement the producer/consumer pattern. 11/20/2014, Bing Li
	private Collaborator collaborator;
	// The flag that represents the busy/idle state of the thread. 11/20/2014, Bing Li
	private boolean isIdle;

	/*
	 * Initialize the notification thread. This constructor has no limit on the size of the queue. 11/20/2014, Bing Li
	 */
	public NotificationObjectQueue()
	{
		// Generate the key. 11/20/2014, Bing Li
		this.key = Tools.generateUniqueKey();
		// Initialize the queue without any size constraint. 11/20/2014, Bing Li
		this.queue = new LinkedBlockingQueue<Notification>();
		// Ignore the value of taskSize. 11/20/2014, Bing Li
		this.taskSize = UtilConfig.NO_QUEUE_SIZE;
		// Initialize the collaborator. 11/20/2014, Bing Li
		this.collaborator = new Collaborator();
		// Set the idle state to false. 11/20/2014, Bing Li
		this.isIdle = false;
	}

	/*
	 * Initialize the notification thread. This constructor has a limit on the size of the queue. 11/20/2014, Bing Li
	 */
	public NotificationObjectQueue(int taskSize)
	{
		// Generate the key. 11/20/2014, Bing Li
		this.key = Tools.generateUniqueKey();
		// Initialize the queue with the particular size constraint. 11/20/2014, Bing Li
		this.queue = new LinkedBlockingQueue<Notification>();
		// Set the value of taskSize. 11/20/2014, Bing Li
		this.taskSize = taskSize;
		// Initialize the collaborator. 11/20/2014, Bing Li
		this.collaborator = new Collaborator();
		// Set the idle state to false. 11/20/2014, Bing Li
		this.isIdle = false;
	}

	/*
```

```
    * Dispose the notification thread. 11/20/2014, Bing Li
    */
   public synchronized void dispose()
   {
      // Set the flag to be the state of being shutdown. 11/20/2014, Bing Li
      this.collaborator.setShutdown();
      // Notify the thread being waiting to go forward. Since the shutdown flag is set, the thread must die
for the notification. 11/20/2014, Bing Li
      this.collaborator.signalAll();
      try
      {
         // Wait for the thread to die. 11/20/2014, Bing Li
         this.join();
      }
      catch (InterruptedException e)
      {
         e.printStackTrace();
      }
      // Clear the queue to release resources. 11/20/2014, Bing Li
      if (this.queue != null)
      {
         this.queue.clear();
      }
   }

   /*
    * Expose the key for the convenient management. 11/20/2014, Bing Li
    */
   public String getKey()
   {
      return this.key;
   }

   /*
    * Enqueue a notification into the thread. 11/20/2014, Bing Li
    */
   public void enqueue(Notification notification)
   {
      // Set the state of the thread to be busy. 11/20/2014, Bing Li
      this.setBusy();
      // Enqueue the notification. 11/20/2014, Bing Li
      this.queue.add(notification);
      // Notify the waiting thread to keep on working since new notifications are received. 11/20/2014, Bing
Li
      this.collaborator.signal();
   }

   /*
    * Set the state to be busy. 11/20/2014, Bing Li
    */
   private synchronized void setBusy()
   {
      this.isIdle = false;
   }

   /*
    * Set the state to be idle. 11/20/2014, Bing Li
    */
   private synchronized void setIdle()
   {
      this.isIdle = true;
   }

   /*
    * The method intends to stop the thread temporarily when no notifications are available. A thread is
identified as being idle immediately after the temporary waiting is finished. 11/20/2014, Bing Li
    */
   public void holdOn(long waitTime) throws InterruptedException
```

```
    {
        // Wait for some time, which is determined by the value of waitTime. 11/20/2014, Bing Li
        this.collaborator.holdOn(waitTime);
        // Set the state of the thread to be idle after waiting for some time. 11/20/2014, Bing Li
        this.setIdle();
    }

    /*
     * Check whether the shutdown flag of the thread is set or not. It might take some time for the thread to
be shutdown practically even though the flag is set. 11/20/2014, Bing Li
     */
    public boolean isShutdown()
    {
        return this.collaborator.isShutdown();
    }

    /*
     * Check whether the current size of the queue reaches the upper limit. 11/20/2014, Bing Li
     */
    public boolean isFull()
    {
        return this.queue.size() >= this.taskSize;
    }

    /*
     * Check whether the shutdown flag of the thread is set or not. It might take some time for the thread to
be shutdown practically even though the flag is set. 11/20/2014, Bing Li
     */
    public boolean isEmpty()
    {
        return this.queue.size() <= 0;
    }

    /*
     * Check whether the thread is idle or not. 11/20/2014, Bing Li
     */
    public synchronized boolean isIdle()
    {
        return this.isIdle;
    }

    /*
     * Get the current size of the queue. 11/20/2014, Bing Li
     */
    public int getQueueSize()
    {
        return this.queue.size();
    }

    /*
     * Get the notification but leave the notification in the queue. 11/20/2014, Bing Li
     */
    public Notification peekNotification()
    {
        return this.queue.peek();
    }

    /*
     * Dequeue the notification stream from the queue. 11/20/2014, Bing Li
     */
    public Notification getNotification() throws InterruptedException
    {
        return this.queue.take();
    }

    /*
     * Dispose the notification. 11/20/2014, Bing Li
     */
```

```
  public synchronized void disposeObject(Notification notification)
  {
    notification = null;
  }
}
```

- **InteractiveQueue**

```
package com.greatfree.concurrency;

import java.util.Calendar;
import java.util.Date;
import java.util.LinkedList;
import java.util.Queue;
import java.util.concurrent.locks.ReentrantReadWriteLock;

import com.greatfree.util.Time;
import com.greatfree.util.Tools;

/*
 * This is the base class that must be derived to implement a thread that holds the methods which could be called to notify the interactive dispatcher. 11/20/2014, Bing Li
 */

// Created: 11/20/2014, Bing Li
public class InteractiveQueue<Task, Notifier extends Interactable<Task>> extends Thread
{
  // The key to represent the thread. 11/20/2014, Bing Li
  private String key;
  // The queue to keep tasks that should be processed by the thread. 11/20/2014, Bing Li
  private Queue<Task> queue;
  // The maximum task size. 11/20/2014, Bing Li
  private int taskSize;
  // The notify/wait mechanism to coordinate the thread. 11/20/2014, Bing Li
  private Collaborator collaborator;
  // The notifier that interacts with the callee. 11/20/2014, Bing Li
  private Notifier notifier;
  // The starting time of the thread. 11/20/2014, Bing Li
  private Date startTime;
  // The ending time of the thread. 11/20/2014, Bing Li
  private Date endTime;
  // The idle time of the thread. 11/20/2014, Bing Li
  private Date idleTime;
  // The current task the thread is working on. 11/20/2014, Bing Li
  private Task currentTask;
  // The read/write lock to manage the concurrency on the thread. 11/20/2014, Bing Li
  private ReentrantReadWriteLock lock;

  /*
   * Initialize. The notifier is defined outside. 11/20/2014, Bing Li
   */
  public InteractiveQueue(int taskSize, Notifier notifier)
  {
    this.key = Tools.generateUniqueKey();
    this.queue = new LinkedList<Task>();
    this.taskSize = taskSize;
    this.collaborator = new Collaborator();
    this.notifier = notifier;
    this.startTime = Time.INIT_TIME;
    this.endTime = Time.INIT_TIME;
    this.idleTime = Time.INIT_TIME;
    this.lock = new ReentrantReadWriteLock();
  }

  /*
   * Dispose the thread. 11/20/2014, Bing Li
   */
  public synchronized void dispose()
  {
    // Set the shutdown flag to be true. Thus, the loop in the dispatcher thread to schedule tasks is terminated. 11/21/2014, Bing Li
    this.collaborator.setShutdown();
```

```
    // Notify the dispatcher thread that is waiting for tasks to terminate the waiting. 11/21/2014, Bing Li
    this.collaborator.signalAll();
    try
    {
      // Wait for the thread to end. 11/21/2014, Bing Li
      this.join();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
    // Clear the queue. 11/21/2014, Bing Li
    if (this.queue != null)
    {
      this.queue.clear();
    }
  }

  /*
   * Expose the key of the thread. 11/21/2014, Bing Li
   */
  public String getKey()
  {
    return this.key;
  }

  /*
   * Expose the starting time of the thread. 11/21/2014, Bing Li
   */
  public Date getStartTime()
  {
    this.lock.readLock().lock();
    try
    {
      return this.startTime;
    }
    finally
    {
      this.lock.readLock().unlock();
    }
  }

  /*
   * Expose the ending time of the thread. 11/21/2014, Bing Li
   */
  public Date getEndTime()
  {
    this.lock.readLock().lock();
    try
    {
      return this.endTime;
    }
    finally
    {
      this.lock.readLock().unlock();
    }
  }

  /*
   * Set the starting time. 11/21/2014, Bing Li
   */
  public void setStartTime()
  {
    this.lock.writeLock().lock();
    // Set the starting time. 11/21/2014, Bing Li
    this.startTime = Calendar.getInstance().getTime();
    // An initial time is set to the idle time. It denotes that the thread is busy. 11/21/2014, Bing Li
    this.idleTime = Time.INIT_TIME;
```

```
    // An initial time is set to the ending time. It denotes that the thread is still in the process of working, not ending yet. 11/21/2014, Bing Li
    this.endTime = Time.INIT_TIME;
    this.lock.writeLock().unlock();
  }

  /*
   * Set the ending time. 11/21/2014, Bing Li
   */
  public void setEndTime()
  {
    this.lock.writeLock().lock();
    // An initial time is set to the starting time. It denotes that the thread is ended. 11/21/2014, Bing Li
    this.startTime = Time.INIT_TIME;
    // Set the ending time. 11/21/2014, Bing Li
    this.endTime = Calendar.getInstance().getTime();
    this.lock.writeLock().unlock();
  }

  /*
   * Enqueue a task to the thread. 11/21/2014, Bing Li
   */
  public void enqueue(Task task)
  {
    this.lock.writeLock().lock();
    // When a task is enqueued, it indicates that it must not be idle. Thus, the idle time is set to an initial time. 11/21/2014, Bing Li
    this.idleTime = Time.INIT_TIME;
    // Enqueue the new task. 11/21/2014, Bing Li
    this.queue.add(task);
    this.lock.writeLock().unlock();
    // Notify the thread since now it might be waiting for new tasks. 11/21/2014, Bing Li
    this.collaborator.signal();
  }

  /*
   * Wait for some time. It happens when no tasks are available. 11/21/2014, Bing Li
   */
  public void holdOn(long waitTime) throws InterruptedException
  {
    // Wait for some time. 11/21/2014, Bing Li
    this.collaborator.holdOn(waitTime);
    this.lock.writeLock().lock();
    // Set the idle time after waiting for some time. It represents that the thread is idle. 11/21/2014, Bing Li
    this.idleTime = Calendar.getInstance().getTime();
    this.lock.writeLock().unlock();
  }

  /*
   * Check whether the thread is shutdown. 11/21/2014, Bing Li
   */
  public boolean isShutdown()
  {
    return this.collaborator.isShutdown();
  }

  /*
   * Check whether the queue of the thread is full. 11/21/2014, Bing Li
   */
  public boolean isFull()
  {
    this.lock.readLock().lock();
    try
    {
      // If the queue size is greater than or equal to the maximum task size, it denotes that the thread is full. 11/21/2014, Bing Li
      return this.queue.size() >= this.taskSize;
```

```
    }
    finally
    {
      this.lock.readLock().unlock();
    }
  }

  /*
   * Check whether the queue of the thread is empty. 11/21/2014, Bing Li
   */
  public boolean isEmpty()
  {
    this.lock.readLock().lock();
    try
    {
      // Check if the queue is valid. 11/21/2014, Bing Li
      if (this.queue != null)
      {
        // Check whether the size of the queue is less than zero. 11/21/2014, Bing Li
        if (this.queue.size() <= 0)
        {
          // If no tasks are available in the queue, it needs to notify the manager, usually a thread pool,
of the thread to speed up the thread if it is identified as a slow one at the moment. 11/21/2014, Bing Li
          this.notifier.restoreFast(this.key);
          // Tell the invoker that the thread is empty. 11/21/2014, Bing Li
          return true;
        }
        else
        {
          // Tell the invoker that the thread is not empty. 11/21/2014, Bing Li
          return false;
        }
      }
      // Tell the invoker that the thread is empty. 11/21/2014, Bing Li
      return true;
    }
    finally
    {
      this.lock.readLock().unlock();
    }
  }

  /*
   * Expose the current queue size. 11/21/2014, Bing Li
   */
  public int getQueueSize()
  {
    this.lock.readLock().lock();
    try
    {

      return this.queue.size();
    }
    finally
    {
      this.lock.readLock().unlock();
    }
  }

  /*
   * Clear the queue. 11/21/2014, Bing Li
   */
  public void clear()
  {
    this.lock.writeLock().lock();
    this.queue.clear();
    this.lock.writeLock().unlock();
  }
```

```
  /*
   * Expose the idle time. 11/21/2014, Bing Li
   */
  public Date getIdleTime()
  {
    this.lock.readLock().lock();
    try
    {
      return this.idleTime;
    }
    finally
    {
      this.lock.readLock().unlock();
    }
  }

  /*
   * Expose the current task. 11/21/2014, Bing Li
   */
  public Task getCurrentTask()
  {
    this.lock.readLock().lock();
    try
    {
      return this.currentTask;
    }
    finally
    {
      this.lock.readLock().unlock();
    }
  }

  /*
   * Notify the manager, usually a thread pool, of the thread that a specific task is done in the thread.
11/21/2014, Bing Li
   */
  public void done(Task task)
  {
    this.notifier.done(task);
  }

  /*
   * Dequeue a task from the thread. 11/21/2014, Bing Li
   */
  public Task getTask() throws InterruptedException
  {
    this.lock.writeLock().lock();
    try
    {
      // Check if the queue is valid. 11/21/2014, Bing Li
      if (this.queue != null)
      {
        // Check whether the queue contains tasks. 11/21/2014, Bing Li
        if (!this.queue.isEmpty())
        {
          // Dequeue a task if the queue is not empty. 11/21/2014, Bing Li
          this.currentTask = this.queue.poll();
          // Check whether the queue size is less than the maximum task size. 11/21/2014, Bing Li
          if (this.queue.size() < this.taskSize)
          {
            // Notify the thread manager, usually a thread pool, to let the thread keep on working. It
might get stuck for overload. 11/21/2014, Bing Li
            this.notifier.keepOn();
          }
          // Return the task. 11/21/2014, Bing Li
          return this.currentTask;
        }
```

```
        }
        // No task is available if the queue is invalid. 11/21/2014, Bing Li
        return null;
      }
      finally
      {
        this.lock.writeLock().unlock();
      }
    }

    /*
     * Dispose one task. It happens when the task is finished. 11/21/2014, Bing Li
     */
    public void disposeObject(Task notification)
    {
      notification = null;
    }
}
```

- **InteractiveThreadCreatable**

```
package com.greatfree.concurrency;

/*
 * In general, a pool needs to have the ability to create instances of managed resources. The Creatable interface is responsible for that. 11/21/2014, Bing Li
 *
 * The interface defines the method to create instances of InteractiveThread, which is derived from InteractiveQueue. 11/21/2014, Bing Li
 *
 */

// Created: 11/21/2014, Bing Li
public interface InteractiveThreadCreatable<Notification, Notifier extends Interactable<Notification>, InteractiveThread extends InteractiveQueue<Notification, Notifier>>
{
   public InteractiveThread createInteractiveThreadInstance(int taskSize, Notifier notifier);
}
```

- **Interactable**

```
package com.greatfree.concurrency;

/*
 * Here, the calling-back is a mechanism to build a mechanism between the caller and the callee to interact. The caller notifies the callee by calling the methods provided by the callee such that the callee can respond to the caller. The caller does that when its running circumstance is changed in a certain situation. 11/20/2014, Bing Li
 *
 *  In the tutorial, the callee is a thread pool, i.e., InteractiveDispatcher. With the support of the interfaces below, the caller can help the pool to determine how to manage the threads within it. 11/20/2014, Bing Li
 */

// Created: 11/20/2014, Bing Li
public interface Interactable<Task>
{
   // Pause the execution of a thread. 11/20/2014, Bing Li
   public void pause();
   // Continue to work on a task. 11/20/2014, Bing Li
   public void keepOn();
   // Restore to a fast state for a thread by specifying its key. 11/20/2014, Bing Li
   public void restoreFast(String key);
   // Notify the task is finished. 11/20/2014, Bing Li
   public void done(Task task);
}
```

- **MessageBindable**

```
package com.greatfree.concurrency;

/*
 * Some behaviors, such as dispose(), on the messages must be synchronized among threads. For example, if no synchronization, it is possible that a message is disposed while it is consumed in another one. 11/26/2014, Bing Li
 *
 * The methods for those behaviors and the ones to synchronize them are defined in the interface. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public interface MessageBindable<ServerMessage>
{
   // Add the thread key. 11/26/2014, Bing Li
   public void addThread(String key);
   // Set the message that the synchronize needs to take care. 11/26/2014, Bing Li
   public void set(ServerMessage message);
   // Bind the message with the thread that needs to be synchronized. 11/26/2014, Bing Li
   public void bind(String threadKey, ServerMessage message);
   // Dispose the resources of the binder. 11/26/2014, Bing Li
   public void dispose();
}
```

- **BoundNotificationQueue**

```
package com.greatfree.concurrency;

import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.multicast.ServerMessage;
import com.greatfree.util.Tools;
import com.greatfree.util.UtilConfig;

/*
 * The thread is different from NotificationQueue in the sense that it deals with the case when a notification is shared by multiple threads rather than just one. Therefore, it is necessary to implement a synchronization mechanism among those threads. 11/26/2014, Bing Li
 *
 * For example, if no synchronization, it is possible that a message is disposed while it is consumed in another one. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public class BoundNotificationQueue<Notification extends ServerMessage, Binder extends MessageBindable<Notification>> extends Thread
{
  // Declare the key for the notification thread. 11/26/2014, Bing Li
  private String key;
  // The thread is managed by a dispatcher. The key represents the dispatcher. 11/26/2014, Bing Li
  private String dispatcherKey;
  // Declare an instance of LinkedBlockingQueue to take received notifications. 11/26/2014, Bing Li
  private LinkedBlockingQueue<Notification> queue;
  // Declare the size of the queue. 11/26/2014, Bing Li
  private int taskSize;
  // The notify/wait mechanism to implement the producer/consumer pattern. 11/26/2014, Bing Li
  private Collaborator collaborator;
  // The flag that indicates the busy/idle state of the thread. 11/26/2014, Bing Li
  private boolean isIdle;
  // The binder that synchronizes the threads that share the notification. 11/26/2014, Bing Li
  private Binder binder;

  /*
   * Initialize the bound notification thread. This constructor has no limit on the size of the queue. It is required to input the binder and the dispatcher key. 11/26/2014, Bing Li
   */
  public BoundNotificationQueue(Binder binder, String dispatcherKey)
  {
    // Generate the key. 11/26/2014, Bing Li
    this.key = Tools.generateUniqueKey();
    this.dispatcherKey = dispatcherKey;
    this.queue = new LinkedBlockingQueue<Notification>();
    this.taskSize = UtilConfig.NO_QUEUE_SIZE;
    this.collaborator = new Collaborator();
    this.isIdle = false;
    this.binder = binder;
  }

  /*
   * Initialize the bound notification thread. This constructor has a limit on the size of the queue. It is required to input the binder and the dispatcher key. 11/26/2014, Bing Li
   */
  public BoundNotificationQueue(int taskSize, String dispatcherKey, Binder binder)
  {
    this.key = Tools.generateUniqueKey();
    this.dispatcherKey = dispatcherKey;
    this.queue = new LinkedBlockingQueue<Notification>();
    this.taskSize = taskSize;
    this.collaborator = new Collaborator();
    this.isIdle = false;
```

```
    this.binder = binder;
  }

  /*
   * Dispose the bound notification thread. 11/26/2014, Bing Li
   */
  public synchronized void dispose()
  {
    // Set the flag to be the state of being shutdown. 11/26/2014, Bing Li
    this.collaborator.setShutdown();
    // Notify the thread being waiting to go forward. Since the shutdown flag is set, the thread must die
for the notification. 11/26/2014, Bing Li
    this.collaborator.signalAll();
    try
    {
      // Wait for the thread to die. 11/26/2014, Bing Li
      this.join();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
    // Clear the queue to release resources. 11/26/2014, Bing Li
    if (this.queue != null)
    {
      this.queue.clear();
    }
  }

  /*
   * Expose the key for the convenient management. 11/26/2014, Bing Li
   */
  public String getKey()
  {
    return this.key;
  }

  /*
   * Expose the dispatcher key. 11/26/2014, Bing Li
   */
  public String getDispatcherKey()
  {
    return this.dispatcherKey;
  }

  /*
   * Enqueue a notification into the thread. 11/26/2014, Bing Li
   */
  public void enqueue(Notification notification) throws IllegalStateException
  {
    // Set the state of the thread to be busy. 11/26/2014, Bing Li
    this.setBusy();
    // Enqueue the notification. 11/26/2014, Bing Li
    this.queue.add(notification);
    // Notify the waiting thread to keep on working since new notifications are received. 11/26/2014, Bing
Li
    this.collaborator.signal();
  }

  /*
   * Set the state to be busy. 11/26/2014, Bing Li
   */
  private synchronized void setBusy()
  {
    this.isIdle = false;
  }

  /*
```

```
   * Set the state to be idle. 11/26/2014, Bing Li
   */
  private synchronized void setIdle()
  {
    this.isIdle = true;
  }

  /*
   * Get the current size of the queue. 11/26/2014, Bing Li
   */
  public int getQueueSize()
  {
    return this.queue.size();
  }

  /*
   * The method intends to stop the thread temporarily when no notifications are available. A thread is
identified as being idle immediately after the temporary waiting is finished. 11/26/2014, Bing Li
   */
  public void holdOn(long waitTime) throws InterruptedException
  {
    // Wait for some time, which is determined by the value of waitTime. 11/26/2014, Bing Li
    this.collaborator.holdOn(waitTime);
    // Set the state of the thread to be idle after waiting for some time. 11/26/2014, Bing Li
    this.setIdle();
  }

  /*
   * Check whether the shutdown flag of the thread is set or not. It might take some time for the thread to
be shutdown practically even though the flag is set. 11/26/2014, Bing Li
   */
  public boolean isShutdown()
  {
    return this.collaborator.isShutdown();
  }

  /*
   * Check whether the current size of the queue reaches the upper limit. 11/26/2014, Bing Li
   */
  public boolean isFull()
  {
    return this.queue.size() >= this.taskSize;
  }

  /*
   * Check whether the shutdown flag of the thread is set or not. It might take some time for the thread to
be shutdown practically even though the flag is set. 11/26/2014, Bing Li
   */
  public boolean isEmpty()
  {
    return this.queue.size() <= 0;
  }

  /*
   * Check whether the thread is idle or not. 11/26/2014, Bing Li
   */
  public synchronized boolean isIdle()
  {
    return this.isIdle;
  }

  /*
   * Dequeue the notification from the queue. 11/26/2014, Bing Li
   */
  public Notification getNotification() throws InterruptedException
  {
    return this.queue.take();
  }
```

```
    /*
     * Get the notification but leave the notification in the queue. 11/26/2014, Bing Li
     */
    public Notification peekNotification()
    {
      return this.queue.peek();
    }

    /*
     * Notify the bound notification is consumed by the thread, which is represented by the key. The binder
then determines whether the behaviors that affect them can be executed. 11/26/2014, Bing Li
     */
    public synchronized void bind(String threadKey, Notification notification)
    {
      this.binder.bind(threadKey, notification);
    }
}
```

- **BoundNotificationDispatcher**

```
package com.greatfree.concurrency;

import java.util.HashMap;
import java.util.Map;
import java.util.Timer;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.multicast.ServerMessage;
import com.greatfree.util.CollectionSorter;
import com.greatfree.util.Tools;
import com.greatfree.util.UtilConfig;

/*
 * This is a dispatcher to manage threads which need to share notifications. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public class BoundNotificationDispatcher<Notification extends ServerMessage, Binder extends
MessageBindable<Notification>, NotificationThread extends BoundNotificationQueue<Notification,
Binder>, ThreadCreator extends BoundNotificationThreadCreatable<Notification, Binder,
NotificationThread>> extends Thread implements CheckIdleable
{
   // Define a key for the dispatcher. It is required to synchronize the procedure of sharing notifications.
11/26/2014, Bing Li
   private String key;
   // Declare a map to contain all of the threads. 11/26/2014, Bing Li
   private Map<String, NotificationThread> threadMap;
   // Declare a queue to contain notifications. Here the queue is specified when declaring. It aims to use
the method take() instead of poll() since take() waits when no data in the queue while poll() returns null.
The effect of take() is usually preferred. 11/26/2014, Bing Li
   private LinkedBlockingQueue<Notification> notificationQueue;
   // Declare a thread pool that is used to run a thread. 11/26/2014, Bing Li
   private ThreadPool threadPool;
   // The binder that synchronizes threads and do something after all of them have already processed the
share notifications. 11/26/2014, Bing Li
   private Binder binder;
   // Declare a thread creator that is used to initialize a thread instance. 11/26/2014, Bing Li
   private ThreadCreator threadCreator;
   // The maximum task length for each thread to be created. 11/26/2014, Bing Li
   private int maxTaskSize;
   // The maximum thread count that can be created in the dispatcher. 11/26/2014, Bing Li
   private int maxThreadSize;
   // a timer that controls the task of idle checking. 11/26/2014, Bing Li
   private Timer checkTimer;
   // The checker to scan whether created threads are idle long enough. 11/26/2014, Bing Li
   private ThreadIdleChecker<BoundNotificationDispatcher<Notification, Binder, NotificationThread,
ThreadCreator>> idleChecker;
   // The collaborator is used to pause the dispatcher when no notifications are available and notify to
continue when new notifications are received. 11/26/2014, Bing Li
   private Collaborator workCollaborator;
   // The time to wait when no notifications are available. 11/26/2014, Bing Li
   private long dispatcherWaitTime;
   // A flag that indicates whether a local thread pool is used or a global one is used. Since it might get
problems if the global pool is shutdown inside the class. Thus, it is required to set the flag. 11/26/2014,
Bing Li
   private boolean isSelfThreadPool;

   /*
    * When a server dispatcher has a local thread pool for each message dispatcher, the constructor is
used. 11/26/2013, Bing Li
    */
   public BoundNotificationDispatcher(int poolSize, long keepAliveTime, Binder binder, ThreadCreator
threadCreator, int maxTaskSize, int maxThreadSize, long dispatcherKeepAliveTime)
```

```
  {
    this.key = Tools.generateUniqueKey();
    this.threadMap = new ConcurrentHashMap<String, NotificationThread>();
    this.notificationQueue = new LinkedBlockingQueue<Notification>();
    this.threadPool = new ThreadPool(poolSize, keepAliveTime);
    this.binder = binder;
    this.binder.addThread(this.key);
    this.threadCreator = threadCreator;
    this.maxTaskSize = maxTaskSize;
    this.maxThreadSize = maxThreadSize;
    this.checkTimer = UtilConfig.NO_TIMER;
    this.workCollaborator = new Collaborator();
    this.dispatcherWaitTime = dispatcherKeepAliveTime;
    this.isSelfThreadPool = true;
  }

  /*
   * When a server dispatcher has a global thread pool, the constructor is used. 11/26/2014, Bing Li
   */
  public BoundNotificationDispatcher(ThreadPool threadPool, Binder binder, ThreadCreator
threadCreator, int maxTaskSize, int maxThreadSize, long dispatcherKeepAliveTime)
  {
    this.key = Tools.generateUniqueKey();
    this.threadMap = new ConcurrentHashMap<String, NotificationThread>();
    this.notificationQueue = new LinkedBlockingQueue<Notification>();
    this.threadPool = threadPool;
    this.binder = binder;
    this.binder.addThread(this.key);
    this.threadCreator = threadCreator;
    this.maxTaskSize = maxTaskSize;
    this.maxThreadSize = maxThreadSize;
    this.checkTimer = UtilConfig.NO_TIMER;
    this.workCollaborator = new Collaborator();
    this.dispatcherWaitTime = dispatcherKeepAliveTime;
    this.isSelfThreadPool = false;
  }

  /*
   * Dispose the bound notification dispatcher. 11/04/2014, Bing Li
   */
  public synchronized void dispose()
  {
    // Set the shutdown flag to be true. Thus, the loop in the dispatcher thread to schedule notification
loads is terminated. 11/04/2014, Bing Li
    this.workCollaborator.setShutdown();
    // Notify the dispatcher thread that is waiting for the notifications to terminate the waiting.
11/04/2014, Bing Li
    this.workCollaborator.signalAll();
    // Clear the notification queue. 11/04/2014, Bing Li
    if (this.notificationQueue != null)
    {
      this.notificationQueue.clear();
    }
    // Cancel the timer that controls the idle checking. 11/04/2014, Bing Li
    if (this.checkTimer != UtilConfig.NO_TIMER)
    {
      this.checkTimer.cancel();
    }
    // Terminate the periodically running thread for idle checking. 11/04/2014, Bing Li
    if (this.idleChecker != null)
    {
      this.idleChecker.cancel();
    }
    // Dispose all of threads created during the dispatcher's running procedure. 11/04/2014, Bing Li
    for (NotificationThread thread : this.threadMap.values())
    {
      thread.dispose();
    }
```

```
    // Clear the threads. 11/04/2014, Bing Li
    this.threadMap.clear();
    // Check whether the thread pool is local or not. 11/19/2014, Bing Li
    if (this.isSelfThreadPool)
    {
      // Shutdown the thread pool if it belongs to the instance of the class only. 11/04/2014, Bing Li
      this.threadPool.shutdown();
    }
    // Dispose the thread creator. 11/04/2014, Bing Li
    this.threadCreator = null;
  }

  /*
   * Expose the dispatcher key. 11/27/2014, Bing Li
   */
  public String getKey()
  {
    return this.key;
  }

  /*
   * The method is called back by the idle checker periodically to monitor the idle states of threads within
the dispatcher. 11/27/2014, Bing Li
   */
  @Override
  public void checkIdle()
  {
    // Check each thread managed by the dispatcher. 11/27/2014, Bing Li
    for (NotificationThread thread : this.threadMap.values())
    {
      // If the thread is empty and idle, it is the one to be checked. 11/27/2014, Bing Li
      if (thread.isEmpty() && thread.isIdle())
      {
        // The algorithm to determine whether a thread should be disposed or not is simple. When it is
checked to be idle, it is time to dispose it. 11/27/2014, Bing Li
        this.threadMap.remove(thread.getKey());
        // Dispose the thread. 11/27/2014, Bing Li
        thread.dispose();
        // Collect the resource of the thread. 11/27/2014, Bing Li
        thread = null;
      }
    }
  }

  /*
   * Set the idle checking parameters. 11/27/2014, Bing Li
   */
  @Override
  public void setIdleChecker(long idleCheckDelay, long idleCheckPeriod)
  {
    // Initialize the timer. 11/27/2014, Bing Li
    this.checkTimer = new Timer();
    // Initialize the idle checker. 11/27/2014, Bing Li
    this.idleChecker = new ThreadIdleChecker<BoundNotificationDispatcher<Notification, Binder,
NotificationThread, ThreadCreator>>(this);
    // Schedule the idle checking task. 11/27/2014, Bing Li
    this.checkTimer.schedule(this.idleChecker, idleCheckDelay, idleCheckPeriod);
  }

  /*
   * Enqueue the newly received notification into the dispatcher. 11/27/2014, Bing Li
   */
  public synchronized void enqueue(Notification notification)
  {
    // Set the notification to the binder. Then, it indicates that the notification is shared by other threads.
It must be synchronized by the binder. 11/27/2014, Bing Li
    this.binder.set(notification);
    // Enqueue the notification into the queue. 11/27/2014, Bing Li
```

```
    this.notificationQueue.add(notification);
    // Notify the dispatcher thread, which is possibly blocked when no requests are available, to keep working. 11/27/2014, Bing Li
    this.workCollaborator.signal();
  }

  /*
   * The thread of the dispatcher is always running until no notifications from remote nodes will be received. If too many notifications are received, more threads are created by the dispatcher. If notifications are limited, the count of threads created by the dispatcher is also small. It is possible no any threads are alive when no notifications are received for a long time. 11/27/2014, Bing Li
   */
  public void run()
  {
    // Declare a notification. 11/27/2014, Bing Li
    Notification notification;
    // Initialize a task map to calculate the load of each thread. 11/27/2014, Bing Li
    Map<String, Integer> threadTaskMap = new HashMap<String, Integer>();
    // Declare a string to keep the selected thread key. 11/27/2014, Bing Li
    String selectedThreadKey = UtilConfig.NO_KEY;
    // The dispatcher usually runs all of the time unless the server is shutdown. To shutdown the dispatcher, the shutdown flag of the collaborator is set to true. 11/27/2014, Bing Li
    while (!this.workCollaborator.isShutdown())
    {
      try
      {
        // Check whether notifications are received and saved in the queue. 11/27/2014, Bing Li
        while (!this.notificationQueue.isEmpty())
        {
          // Dequeue the notification from the queue of the dispatcher. 11/27/2014, Bing Li
          notification = this.notificationQueue.take();

          // Since all of the threads created by the dispatcher are saved in the map by their unique keys, it is necessary to check whether any alive threads are available. If so, it is possible to assign tasks to them if they are not so busy. 11/27/2014, Bing Li
          while (this.threadMap.size() > 0)
          {
            // Clear the map to start to calculate the loads of those threads. 11/27/2014, Bing Li
            threadTaskMap.clear();

            // Each thread's workload is saved into the threadTaskMap. 11/27/2014, Bing Li
            for (NotificationThread thread : this.threadMap.values())
            {
              threadTaskMap.put(thread.getKey(), thread.getQueueSize());
            }
            // Select the thread whose load is the least and keep the key of the thread. 11/27/2014, Bing Li
            selectedThreadKey = CollectionSorter.minValueKey(threadTaskMap);
            // Since no concurrency is applied here, it is possible that the key is invalid. Thus, just check here. 11/27/2014, Bing Li
            if (selectedThreadKey != null)
            {
              // Since no concurrency is applied here, it is possible that the key is out of the map. Thus, just check here. 11/27/2014, Bing Li
              if (this.threadMap.containsKey(selectedThreadKey))
              {
                try
                {
                  // Check whether the thread's load reaches the maximum value. 11/27/2014, Bing Li
                  if (this.threadMap.get(selectedThreadKey).isFull())
                  {
                    // Check if the pool is full. If the least load thread is full as checked by the above condition, it denotes that all of the current alive threads are full. So it is required to create a thread to respond the newly received notifications if the thread count of the pool does not reach the maximum. 11/27/2014, Bing Li
                    if (this.threadMap.size() < this.maxThreadSize)
                    {
                      // Create a new thread. 11/27/2014, Bing Li
```

```
                        NotificationThread thread =
this.threadCreator.createNotificationThreadInstance(this.maxTaskSize, this.key, this.binder);
                        // Save the newly created thread into the map. 11/27/2014, Bing Li
                        this.threadMap.put(thread.getKey(), thread);
                        // Enqueue the notification into the queue of the newly created thread. Then, the
notification will be processed by the thread. 11/27/2014, Bing Li
                        this.threadMap.get(thread.getKey()).enqueue(notification);
                        // Start the thread by the thread pool. 11/27/2014, Bing Li
                        this.threadPool.execute(this.threadMap.get(thread.getKey()));
                      }
                      else
                      {
                        // Force to put the notification into the queue when the count of threads reaches
the upper limit and each of the thread's queue is full. 11/27/2014, Bing Li
                        this.threadMap.get(selectedThreadKey).enqueue(notification);
                      }
                    }
                    else
                    {
                      // If the least load thread's queue is not full, just put the notification into the queue.
11/27/2014, Bing Li
                      this.threadMap.get(selectedThreadKey).enqueue(notification);
                    }
                    // Jump out from the loop since the notification is put into a thread. 11/27/2014, Bing Li
                    break;
                  }
                  catch (NullPointerException e)
                  {
                    // Since no concurrency is applied here, it is possible that a NullPointerException is
raised. If so, it means that the selected thread is not available. Just continue to select anther one.
11/27/2014, Bing Li
                    continue;
                  }
                }
              }
            }
            // If no threads are available, it needs to create a new one to take the notification. 11/27/2014,
Bing Li
            if (this.threadMap.size() <= 0)
            {
              // Create a new thread. 11/27/2014, Bing Li
              NotificationThread thread =
this.threadCreator.createNotificationThreadInstance(this.maxTaskSize, this.key, this.binder);
              // Put it into the map for further reuse. 11/27/2014, Bing Li
              this.threadMap.put(thread.getKey(), thread);
              // Take the notification. 11/27/2014, Bing Li
              this.threadMap.get(thread.getKey()).enqueue(notification);
              // Start the thread. 11/27/2014, Bing Li
              this.threadPool.execute(this.threadMap.get(thread.getKey()));
            }
            // If the dispatcher is shutdown, it is not necessary to keep processing the notifications. So,
jump out the loop and the thread is dead. 11/27/2014, Bing Li
            if (this.workCollaborator.isShutdown())
            {
              break;
            }
          }

          // Check whether the dispatcher is shutdown or not. 11/27/2014, Bing Li
          if (!this.workCollaborator.isShutdown())
          {
            // If the dispatcher is still alive, it denotes that no notifications are available temporarily. Just
wait for a while. 11/27/2014, Bing Li
            this.workCollaborator.holdOn(this.dispatcherWaitTime);
          }
        }
        catch (InterruptedException e)
        {
```

```
            e.printStackTrace();
        }
    }
  }
}
```

- **BoundNotificationThreadCreatable**

```
package com.greatfree.concurrency;

import com.greatfree.multicast.ServerMessage;

/*
 * The interface defines the method to create the instance of BoundNotificationQueue. It works with
BoundNotificationDispatcher. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public interface BoundNotificationThreadCreatable<Notification extends ServerMessage, Binder
extends MessageBindable<Notification>, NotificationThread extends
BoundNotificationQueue<Notification, Binder>>
{
   public NotificationThread createNotificationThreadInstance(int taskSize, String dispatcherKey,
Binder binder);
}
```

- **BroadcastRequestQueue**

```
package com.greatfree.concurrency;

import java.io.IOException;
import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.multicast.BroadcastRequest;
import com.greatfree.multicast.BroadcastResponse;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.remote.IPPort;
import com.greatfree.util.Tools;
import com.greatfree.util.UtilConfig;

/*
 * The thread is the base one to support implementing broadcast requests in a concurrent way. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class BroadcastRequestQueue<Request extends BroadcastRequest, Response extends BroadcastResponse> extends Thread
{
   // The unique key of the thread. It is convenient for managing it by a table-like mechanism. 11/29/2014, Bing Li
   private String key;
   // The queue to take the received broadcast requests. 11/29/2014, Bing Li
   private LinkedBlockingQueue<Request> queue;
   // The IP/port of the broadcast original initiator. The response must be sent back to it. 11/29/2014, Bing Li
   private IPPort ipPort;
   // The maximum size of the queue. 11/29/2014, Bing Li
   private int taskSize;
   // It is necessary to keep the thread waiting when no requests are available. The collaborator is used to notify the thread to keep working when requests are received. When the thread is idle enough, it can be collected. The collaborator is also used to control the life cycle of the thread. 11/29/2014, Bing Li
   private Collaborator collaborator;
   // The flag that represents whether the thread is busy or idle. 11/29/2014, Bing Li
   private boolean isIdle;
   // The TCP client pool. With it, it is able to get the instance of the client to connect the initiator of the broadcast. 11/29/2014, Bing Li
   private FreeClientPool pool;

   /*
    * Initialize an instance. 11/29/2014, Bing Li
    */
   public BroadcastRequestQueue(IPPort ipPort, FreeClientPool pool)
   {
      this.key = Tools.generateUniqueKey();
      this.queue = new LinkedBlockingQueue<Request>();
      this.ipPort = ipPort;
      this.taskSize = UtilConfig.NO_QUEUE_SIZE;
      this.collaborator = new Collaborator();
      this.isIdle = false;
      this.pool = pool;
   }

   /*
    * Initialize an instance. 11/29/2014, Bing Li
    */
   public BroadcastRequestQueue(IPPort ipPort, FreeClientPool pool, int taskSize)
   {
      this.key = Tools.generateUniqueKey();
      this.queue = new LinkedBlockingQueue<Request>();
      this.ipPort = ipPort;
      this.taskSize = taskSize;
```

```
    this.collaborator = new Collaborator();
    this.isIdle = false;
    this.pool = pool;
  }

  /*
   * Dispose the instance of the class. 11/29/2014, Bing Li
   */
  public synchronized void dispose()
  {
    // Set the flag to be the state of being shutdown. 11/29/2014, Bing Li
    this.collaborator.setShutdown();
    // Notify the thread being waiting to go forward. Since the shutdown flag is set, the thread must die
for the notification. 11/29/2014, Bing Li
    this.collaborator.signalAll();
    try
    {
      // Wait for the thread to die. 11/29/2014, Bing Li
      this.join();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
    // Clear the queue to release resources. 11/29/2014, Bing Li
    if (this.queue != null)
    {
      this.queue.clear();
    }
  }

  /*
   * Expose the key for the convenient management. 11/29/2014, Bing Li
   */
  public String getKey()
  {
    return this.key;
  }

  /*
   * Enqueue the broadcast request. 11/29/2014, Bing Li
   */
  public void enqueue(Request request)
  {
    // Set the state of the thread to be busy. 11/29/2014, Bing Li
    this.setBusy();
    // Enqueue the request and its relevant output stream and lock. 11/29/2014, Bing Li
    this.queue.add(request);
    // Notify the waiting thread to keep on working since new requests are received. 11/29/2014, Bing Li
    this.collaborator.signal();
  }

  /*
   * Set the state to be busy. 11/29/2014, Bing Li
   */
  private synchronized void setBusy()
  {
    this.isIdle = false;
  }

  /*
   * Set the state to be idle. 11/29/2014, Bing Li
   */
  private synchronized void setIdle()
  {
    this.isIdle = true;
  }
```

```
  /*
   * The method intends to stop the thread temporarily when no requests are available. A thread is
identified as being idle immediately after the temporary waiting is finished. 11/29/2014, Bing Li
   */
  public void holdOn(long waitTime) throws InterruptedException
  {
    // Wait for some time, which is determined by the value of waitTime. 11/29/2014, Bing Li
    this.collaborator.holdOn(waitTime);
    // Set the state of the thread to be idle after waiting for some time. 11/29/2014, Bing Li
    this.setIdle();
  }

  /*
   * Check whether the shutdown flag of the thread is set or not. It might take some time for the thread to
be shutdown practically even though the flag is set. 11/29/2014, Bing Li
   */
  public boolean isShutdown()
  {
    return this.collaborator.isShutdown();
  }

  /*
   * Check whether the queue is empty. 11/29/2014, Bing Li
   */
  public boolean isEmpty()
  {
    return this.queue.size() <= 0;
  }

  /*
   * Check whether the current size of the queue reaches the upper limit. 11/29/2014, Bing Li
   */
  public boolean isFull()
  {
    return this.queue.size() >= this.taskSize;
  }

  /*
   * Check whether the thread is idle or not. 11/29/2014, Bing Li
   */
  public synchronized boolean isIdle()
  {
    return this.isIdle;
  }

  /*
   * Get the current size of the queue. 11/29/2014, Bing Li
   */
  public int getQueueSize()
  {
    return this.queue.size();
  }

  /*
   * Dequeue the request from the queue. 11/29/2014, Bing Li
   */
  public Request getRequest() throws InterruptedException
  {
    return this.queue.take();
  }

  /*
   * After the response is created, the method is responsible for sending it back to the anycast initiator.
11/29/2014, Bing Li
   */
  public synchronized void respond(Response response) throws IOException
  {
    this.pool.send(this.ipPort, response);
```

```
    }

    /*
     * Dispose the request and the response. 11/29/2014, Bing Li
     */
    public synchronized void DisposeMessage(Request request, Response response)
    {
      request = null;
      response = null;
    }

    /*
     * Dispose the request. 11/29/2014, Bing Li
     */
    public synchronized void DisposeMessage(Request request)
    {
      request = null;
    }

    /*
     * Dispose the response. 11/29/2014, Bing Li
     */
    public synchronized void DisposeMessage(Response response)
    {
      response = null;
    }
}
```

- **BroadcastRequestDispatcher**

```
package com.greatfree.concurrency;

import java.util.HashMap;
import java.util.Map;
import java.util.Timer;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.multicast.BroadcastRequest;
import com.greatfree.multicast.BroadcastResponse;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.remote.IPPort;
import com.greatfree.util.CollectionSorter;
import com.greatfree.util.UtilConfig;

/*
 * This is an important class that enqueues requests and creates broadcast queue threads to respond them concurrently. If the local data is not available, it is necessary to forward the request to the local node's children. 11/29/2014, Bing Li
 *
 * It works in the way like a scheduler or a dispatcher. That is why it is named. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class BroadcastRequestDispatcher<Request extends BroadcastRequest, Response extends BroadcastResponse, RequestThread extends BroadcastRequestQueue<Request, Response>, RequestThreadCreator extends BroadcastRequestThreadCreatable<Request, Response, RequestThread>> extends Thread implements CheckIdleable
{
   // Declare a map to contain all of the threads. 11/29/2014, Bing Li
   private Map<String, RequestThread> threadMap;
   // Declare a queue to contain requests. 11/29/2014, Bing Li
   private LinkedBlockingQueue<Request> requestQueue;
   // Declare a thread pool that is used to run a thread. 11/29/2014, Bing Li
   private FreeClientPool pool;
   // The IP/port is the initiator of the broadcast requesting. Keep the information to send the response to it once if the result is obtained. 11/29/2014, Bing Li
   private IPPort serverAddress;
   // Declare a thread pool that is used to run a thread. 11/29/2014, Bing Li
   private ThreadPool threadPool;
   // Declare a thread creator that is used to initialize a thread instance. 11/29/2014, Bing Li
   private RequestThreadCreator threadCreator;
   // Declare the maximum task length for each thread to be created. 11/29/2014, Bing Li
   private int maxTaskSize;
   // Declare the maximum thread count that can be created in the dispatcher. 11/29/2014, Bing Li
   private int maxThreadSize;
   // Declare a timer that controls the task of idle checking. 11/29/2014, Bing Li
   private Timer checkTimer;
   // Declare the checker to check whether created threads are idle long enough. 11/29/2014, Bing Li
   private ThreadIdleChecker<BroadcastRequestDispatcher<Request, Response, RequestThread, RequestThreadCreator>> idleChecker;
   // The collaborator is used to pause the dispatcher when no requests are available and notify to continue when new requests are received. 11/29/2014, Bing Li
   private Collaborator collaborator;
   // The time to wait when no requests are available. 11/29/2014, Bing Li
   private long dispatcherWaitTime;
   // The flag indicates whether the dispatcher has its own thread pool or shares with others. 11/29/2014, Bing Li
   private boolean isSelfThreadPool;

   /*
    * Initialize the broadcast request dispatcher. The constructor is called when the thread pool is owned by the dispatcher only. 11/29/2014, Bing Li
    */
```

```
  public BroadcastRequestDispatcher(FreeClientPool pool, String serverAddress, int serverPort, int
poolSize, long keepAliveTime, RequestThreadCreator actionThreadCreator, int maxTaskSize, int
maxThreadSize, long dispatcherKeepAliveTime)
  {
    this.pool = pool;
    this.serverAddress = new IPPort(serverAddress, serverPort);
    this.threadMap = new ConcurrentHashMap<String, RequestThread>();
    this.requestQueue = new LinkedBlockingQueue<Request>();
    this.threadPool = new ThreadPool(poolSize, keepAliveTime);
    this.threadCreator = actionThreadCreator;
    this.maxTaskSize = maxTaskSize;
    this.maxThreadSize = maxThreadSize;
    this.checkTimer = UtilConfig.NO_TIMER;
    this.collaborator = new Collaborator();
    this.dispatcherWaitTime = dispatcherKeepAliveTime;
    this.isSelfThreadPool = true;
  }

  /*
   * Dispose the broadcast request dispatcher. The constructor is called when the thread pool is shared.
11/29/2014, Bing Li
   */
  public BroadcastRequestDispatcher(FreeClientPool pool, String serverAddress, int serverPort,
ThreadPool threadPool, RequestThreadCreator actionThreadCreator, int maxTaskSize, int
maxThreadSize, long dispatcherKeepAliveTime)
  {
    this.pool = pool;
    this.serverAddress = new IPPort(serverAddress, serverPort);
    this.threadMap = new ConcurrentHashMap<String, RequestThread>();
    this.requestQueue = new LinkedBlockingQueue<Request>();
    this.threadPool = threadPool;
    this.threadCreator = actionThreadCreator;
    this.maxTaskSize = maxTaskSize;
    this.maxThreadSize = maxThreadSize;
    this.checkTimer = UtilConfig.NO_TIMER;
    this.collaborator = new Collaborator();
    this.dispatcherWaitTime = dispatcherKeepAliveTime;
    this.isSelfThreadPool = false;
  }

  /*
   * Dispose the request dispatcher. 11/29/2014, Bing Li
   */
  public synchronized void dispose()
  {
    // Set the shutdown flag to be true. Thus, the loop in the dispatcher thread to schedule requests load
is terminated. 11/29/2014, Bing Li
    this.collaborator.setShutdown();
    // Notify the dispatcher thread that is waiting for the requests to terminate the waiting. 11/29/2014,
Bing Li
    this.collaborator.signalAll();
    // Clear the request queue. 11/29/2014, Bing Li
    if (this.requestQueue != null)
    {
      this.requestQueue.clear();
    }
    // Cancel the timer that controls the idle checking. 11/29/2014, Bing Li
    if (this.checkTimer != UtilConfig.NO_TIMER)
    {
      this.checkTimer.cancel();
    }
    // Terminate the periodically running thread for idle checking. 11/29/2014, Bing Li
    if (this.idleChecker != null)
    {
      this.idleChecker.cancel();
    }
    // Dispose all of threads created during the dispatcher's running procedure. 11/29/2014, Bing Li
    for (RequestThread thread : this.threadMap.values())
```

```
    {
      thread.dispose();
    }
    // Clear the threads. 11/29/2014, Bing Li
    this.threadMap.clear();
    // Check whether the thread is owned by the dispatcher only. 11/29/2014, Bing Li
    if (this.isSelfThreadPool)
    {
      // Shutdown the thread pool. 11/29/2014, Bing Li
      this.threadPool.shutdown();
    }
    // Dispose the thread creator. 11/29/2014, Bing Li
    this.threadCreator = null;
  }

  /*
   * Set the idle checking parameters. 11/29/2014, Bing Li
   */
  @Override
  public void setIdleChecker(long idleCheckDelay, long idleCheckPeriod)
  {
    // Initialize the timer. 11/29/2014, Bing Li
    this.checkTimer = new Timer();
    // Initialize the idle checker. 11/29/2014, Bing Li
    this.idleChecker = new ThreadIdleChecker<BroadcastRequestDispatcher<Request, Response,
RequestThread, RequestThreadCreator>>(this);
    // Schedule the idle checking task. 11/29/2014, Bing Li
    this.checkTimer.schedule(this.idleChecker, idleCheckDelay, idleCheckPeriod);
  }

  /*
   * The method is called back by the idle checker periodically to monitor the idle states of threads within
the dispatcher. 11/29/2014, Bing Li
   */
  @Override
  public void checkIdle()
  {
    // Check each thread managed by the dispatcher. 11/29/2014, Bing Li
    for (RequestThread thread : this.threadMap.values())
    {
      // If the thread is empty and idle, it is the one to be checked. 11/29/2014, Bing Li
      if (thread.isEmpty() && thread.isIdle())
      {
        // The algorithm to determine whether a thread should be disposed or not is simple. When it is
checked to be idle, it is time to dispose it. 11/29/2014, Bing Li
        this.threadMap.remove(thread.getKey());
        // Dispose the thread. 11/29/2014, Bing Li
        thread.dispose();
        // Collect the resource of the thread. 11/29/2014, Bing Li
        thread = null;
      }
    }
  }

  /*
   * Enqueue the newly received request into the dispatcher. 11/29/2014, Bing Li
   */
  public synchronized void enqueue(Request request)
  {
    // Enqueue the request into the queue. 11/29/2014, Bing Li
    this.requestQueue.add(request);
    // Notify the dispatcher thread, which is possibly blocked when no requests are available, to keep
working. 11/29/2014, Bing Li
    this.collaborator.signal();
  }

  /*
   * The thread of the dispatcher is always running until no requests from clients will be received. If too
```

```
many requests are received, more threads are created by the dispatcher to respond users in time. If
requests are limited, the count of threads created by the dispatcher is also small. It is possible no any
threads are alive when no requests are received for a long time. 11/29/2014, Bing Li
   */
  public void run()
  {
    // Declare a request. 11/29/2014, Bing Li
    Request request;
    // Initialize a thread map to calculate the load of each thread. 11/29/2014, Bing Li
    Map<String, Integer> threadTaskMap = new HashMap<String, Integer>();
    // Declare a string to keep the selected thread key. 11/29/2014, Bing Li
    String selectedThreadKey = UtilConfig.NO_KEY;
    // The dispatcher usually runs all of the time unless the server is shutdown. To shutdown the
dispatcher, the shutdown flag of the collaborator is set to true. 11/29/2014, Bing Li
    while (!this.collaborator.isShutdown())
    {
      try
      {
        // Check whether requests are received and saved in the queue. 11/29/2014, Bing Li
        while (!this.requestQueue.isEmpty())
        {
          // Dequeue the request from the queue of the dispatcher. 11/29/2014, Bing Li
          request = this.requestQueue.take();
          // Since all of the threads created by the dispatcher are saved in the map by their unique keys,
it is necessary to check whether any alive threads are available. If so, it is possible to assign tasks to
them if they are not so busy. 11/29/2014, Bing Li
          while (this.threadMap.size() > 0)
          {
            // Clear the map to start to calculate the load those threads. 11/29/2014, Bing Li
            threadTaskMap.clear();

            // Each thread's workload is saved into the threadTaskMap. 11/29/2014, Bing Li
            for (RequestThread thread : this.threadMap.values())
            {
              threadTaskMap.put(thread.getKey(), thread.getQueueSize());
            }
            // Select the thread whose load is the least and keep the key of the thread. 11/29/2014, Bing
Li
            selectedThreadKey = CollectionSorter.minValueKey(threadTaskMap);
            // Since no concurrency is applied here, it is possible that the key is invalid. Thus, just check
here. 11/19/2014, Bing Li
            if (selectedThreadKey != null)
            {
              // Since no concurrency is applied here, it is possible that the key is out of the map. Thus,
just check here. 11/19/2014, Bing Li
              if (this.threadMap.containsKey(selectedThreadKey))
              {
                try
                {
                  // Check whether the thread's load reaches the maximum value. 11/29/2014, Bing Li
                  if (this.threadMap.get(selectedThreadKey).isFull())
                  {
                    // Check if the pool is full. If the least load thread is full as checked by the above
condition, it denotes that all of the current alive threads are full. So it is required to create a thread to
respond the newly received requests if the thread count of the pool does not reach the maximum.
11/29/2014, Bing Li
                    if (this.threadMap.size() < this.maxThreadSize)
                    {
                      // Create a new thread. 11/29/2014, Bing Li
                      RequestThread thread =
this.threadCreator.createRequestThreadInstance(this.serverAddress, this.pool, this.maxTaskSize);
                      // Save the newly created thread into the map. 11/29/2014, Bing Li
                      this.threadMap.put(thread.getKey(), thread);
                      // Enqueue the request into the queue of the newly created thread. Then, the
request will be processed and responded by the thread. 11/29/2014, Bing Li
                      this.threadMap.get(thread.getKey()).enqueue(request);
                      // Start the thread by the thread pool. 11/29/2014, Bing Li
                      this.threadPool.execute(this.threadMap.get(thread.getKey()));
```

```
                    }
                    else
                    {
                      // Force to put the request into the queue when the count of threads reaches the
upper limit and each of the thread's queue is full. 11/29/2014, Bing Li
                      this.threadMap.get(selectedThreadKey).enqueue(request);
                    }
                  }
                  else
                  {
                    // If the least load thread's queue is not full, just put the request into the queue.
11/29/2014, Bing Li
                    this.threadMap.get(selectedThreadKey).enqueue(request);
                  }
                  // Jump out from the loop since the request is put into a thread. 11/29/2014, Bing Li
                  break;
                }
                catch (NullPointerException e)
                {
                  // Since no concurrency is applied here, it is possible that a NullPointerException is
raised. If so, it means that the selected thread is not available. Just continue to select anther one.
11/29/2014, Bing Li
                  continue;
                }
              }
            }
          }
          // If no threads are available, it needs to create a new one to take the request. 11/29/2014,
Bing Li
          if (this.threadMap.size() <= 0)
          {
            // Create a new thread. 11/29/2014, Bing Li
            RequestThread thread =
this.threadCreator.createRequestThreadInstance(this.serverAddress, this.pool, this.maxTaskSize);
            // Put it into the map for further reuse. 11/29/2014, Bing Li
            this.threadMap.put(thread.getKey(), thread);
            // Take the request. 11/29/2014, Bing Li
            this.threadMap.get(thread.getKey()).enqueue(request);
            // Start the thread. 11/29/2014, Bing Li
            this.threadPool.execute(this.threadMap.get(thread.getKey()));
          }
          // If the dispatcher is shutdown, it is not necessary to keep processing the requests. So, jump
out the loop and the thread is dead. 11/29/2014, Bing Li
          if (this.collaborator.isShutdown())
          {
            break;
          }
        }
        // Check whether the dispatcher is shutdown or not. 11/29/2014, Bing Li
        if (!this.collaborator.isShutdown())
        {
          // If the dispatcher is still alive, it denotes that no requests are available temporarily. Just wait
for a while. 11/29/2014, Bing Li
          this.collaborator.holdOn(this.dispatcherWaitTime);
        }
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **BroadcastRequestThreadCreatable**

```
package com.greatfree.concurrency;

import com.greatfree.multicast.BroadcastRequest;
import com.greatfree.multicast.BroadcastResponse;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.remote.IPPort;

/*
 * The interface to define the method to create a thread for processing broadcast requests concurrently.
11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public interface BroadcastRequestThreadCreatable<Request extends BroadcastRequest,
Response extends BroadcastResponse, RequestThread extends BroadcastRequestQueue<Request,
Response>>
{
   public RequestThread createRequestThreadInstance(IPPort ipPort, FreeClientPool pool, int
taskSize);
}
```

- **AnycastRequestQueue**

```
package com.greatfree.concurrency;

import java.io.IOException;
import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.multicast.AnycastRequest;
import com.greatfree.multicast.AnycastResponse;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.remote.IPPort;
import com.greatfree.util.Tools;
import com.greatfree.util.UtilConfig;

/*
 * The thread is the base one to support implementing anycast requests in a concurrent way.
11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class AnycastRequestQueue<Request extends AnycastRequest, Response extends
AnycastResponse> extends Thread
{
   // The unique key of the thread. It is convenient for managing it by a table-like mechanism. 11/29/2014,
Bing Li
   private String key;
   // The queue to take the received anycast requests. 11/29/2014, Bing Li
   private LinkedBlockingQueue<Request> queue;
   // The IP/port of the anycast original initiator. The response must be sent back to it. 11/29/2014, Bing
Li
   private IPPort ipPort;
   // The maximum size of the queue. 11/29/2014, Bing Li
   private int taskSize;
   // It is necessary to keep the thread waiting when no requests are available. The collaborator is used
to notify the thread to keep working when requests are received. When the thread is idle enough, it can
be collected. The collaborator is also used to control the life cycle of the thread. 11/29/2014, Bing Li
   private Collaborator collaborator;
   // The flag that represents whether the thread is busy or idle. 11/29/2014, Bing Li
   private boolean isIdle;
   // The TCP client pool. With it, it is able to get the instance of the client to connect the initiator of the
anycast. 11/29/2014, Bing Li
   private FreeClientPool pool;

   /*
    * Initialize an instance. 11/29/2014, Bing Li
    */
   public AnycastRequestQueue(IPPort ipPort, FreeClientPool pool)
   {
      this.key = Tools.generateUniqueKey();
      this.queue = new LinkedBlockingQueue<Request>();
      this.ipPort = ipPort;
      this.taskSize = UtilConfig.NO_QUEUE_SIZE;
      this.collaborator = new Collaborator();
      this.isIdle = false;
      this.pool = pool;
   }

   /*
    * Initialize an instance. 11/29/2014, Bing Li
    */
   public AnycastRequestQueue(IPPort ipPort, FreeClientPool pool, int taskSize)
   {
      this.key = Tools.generateUniqueKey();
      this.queue = new LinkedBlockingQueue<Request>();
      this.ipPort = ipPort;
      this.taskSize = taskSize;
```

```
    this.collaborator = new Collaborator();
    this.isIdle = false;
    this.pool = pool;
  }

  /*
   * Dispose the instance of the class. 11/29/2014, Bing Li
   */
  public synchronized void dispose()
  {
    // Set the flag to be the state of being shutdown. 11/29/2014, Bing Li
    this.collaborator.setShutdown();
    // Notify the thread being waiting to go forward. Since the shutdown flag is set, the thread must die
for the notification. 11/29/2014, Bing Li
    this.collaborator.signalAll();
    try
    {
      // Wait for the thread to die. 11/29/2014, Bing Li
      this.join();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
    // Clear the queue to release resources. 11/29/2014, Bing Li
    if (this.queue != null)
    {
      this.queue.clear();
    }
  }

  /*
   * Expose the key for the convenient management. 11/29/2014, Bing Li
   */
  public String getKey()
  {
    return this.key;
  }

  /*
   * Enqueue the anycast request. 11/29/2014, Bing Li
   */
  public void enqueue(Request request)
  {
    // Set the state of the thread to be busy. 11/29/2014, Bing Li
    this.setBusy();
    // Enqueue the request and its relevant output stream and lock. 11/29/2014, Bing Li
    this.queue.add(request);
    // Notify the waiting thread to keep on working since new requests are received. 11/29/2014, Bing Li
    this.collaborator.signal();
  }

  /*
   * Set the state to be busy. 11/29/2014, Bing Li
   */
  private synchronized void setBusy()
  {
    this.isIdle = false;
  }

  /*
   * Set the state to be idle. 11/29/2014, Bing Li
   */
  private synchronized void setIdle()
  {
    this.isIdle = true;
  }
```

```
  /*
   * The method intends to stop the thread temporarily when no requests are available. A thread is
identified as being idle immediately after the temporary waiting is finished. 11/29/2014, Bing Li
   */
  public void holdOn(long waitTime) throws InterruptedException
  {
    // Wait for some time, which is determined by the value of waitTime. 11/29/2014, Bing Li
    this.collaborator.holdOn(waitTime);
    // Set the state of the thread to be idle after waiting for some time. 11/29/2014, Bing Li
    this.setIdle();
  }

  /*
   * Check whether the shutdown flag of the thread is set or not. It might take some time for the thread to
be shutdown practically even though the flag is set. 11/29/2014, Bing Li
   */
  public boolean isShutdown()
  {
    return this.collaborator.isShutdown();
  }

  /*
   * Check whether the queue is empty. 11/29/2014, Bing Li
   */
  public boolean isEmpty()
  {
    return this.queue.size() <= 0;
  }

  /*
   * Check whether the current size of the queue reaches the upper limit. 11/29/2014, Bing Li
   */
  public boolean isFull()
  {
    return this.queue.size() >= this.taskSize;
  }

  /*
   * Check whether the thread is idle or not. 11/29/2014, Bing Li
   */
  public synchronized boolean isIdle()
  {
    return this.isIdle;
  }

  /*
   * Get the current size of the queue. 11/29/2014, Bing Li
   */
  public int getQueueSize()
  {
    return this.queue.size();
  }

  /*
   * Dequeue the request from the queue. 11/29/2014, Bing Li
   */
  public Request getRequest() throws InterruptedException
  {
    return this.queue.take();
  }

  /*
   * After the response is created, the method is responsible for sending it back to the anycast initiator.
11/29/2014, Bing Li
   */
  public synchronized void respond(Response response) throws IOException
  {
    this.pool.send(this.ipPort, response);
```

```
    }

    /*
     * Dispose the request and the response. 11/29/2014, Bing Li
     */
    public synchronized void disposeMessage(Request request, Response response)
    {
      request = null;
      response = null;
    }

    /*
     * Dispose the request. 11/29/2014, Bing Li
     */
    public synchronized void disposeMessage(Request request)
    {
      request = null;
    }

    /*
     * Dispose the response. 11/29/2014, Bing Li
     */
    public synchronized void disposeMessage(Response response)
    {
      response = null;
    }
}
```

- **AnycastRequestDispatcher**

```
package com.greatfree.concurrency;

import java.util.HashMap;
import java.util.Map;
import java.util.Timer;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.multicast.AnycastRequest;
import com.greatfree.multicast.AnycastResponse;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.remote.IPPort;
import com.greatfree.util.CollectionSorter;
import com.greatfree.util.UtilConfig;

/*
 * This is an important class that enqueues requests and creates anycast queue threads to respond them concurrently. If the local data is not available, it is necessary to forward the request to the local node's children. 11/29/2014, Bing Li
 *
 * It works in the way like a scheduler or a dispatcher. That is why it is named. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class AnycastRequestDispatcher<Request extends AnycastRequest, Response extends AnycastResponse, RequestThread extends AnycastRequestQueue<Request, Response>, RequestThreadCreator extends AnycastRequestThreadCreatable<Request, Response, RequestThread>> extends Thread implements CheckIdleable
{
	// Declare a map to contain all of the threads. 11/29/2014, Bing Li
	private Map<String, RequestThread> threadMap;
	// Declare a queue to contain requests. 11/29/2014, Bing Li
	private LinkedBlockingQueue<Request> requestQueue;
	// Declare a thread pool that is used to run a thread. 11/29/2014, Bing Li
	private FreeClientPool pool;
	// The IP/port is the initiator of the anycast requesting. Keep the information to send the response to it once if the result is obtained. 11/29/2014, Bing Li
	private IPPort serverAddress;
	// Declare a thread pool that is used to run a thread. 11/29/2014, Bing Li
	private ThreadPool threadPool;
	// Declare a thread creator that is used to initialize a thread instance. 11/29/2014, Bing Li
	private RequestThreadCreator threadCreator;
	// Declare the maximum task length for each thread to be created. 11/29/2014, Bing Li
	private int maxTaskSize;
	// Declare the maximum thread count that can be created in the dispatcher. 11/29/2014, Bing Li
	private int maxThreadSize;
	// Declare a timer that controls the task of idle checking. 11/29/2014, Bing Li
	private Timer checkTimer;
	// Declare the checker to check whether created threads are idle long enough. 11/29/2014, Bing Li
	private ThreadIdleChecker<AnycastRequestDispatcher<Request, Response, RequestThread, RequestThreadCreator>> idleChecker;
	// The collaborator is used to pause the dispatcher when no requests are available and notify to continue when new requests are received. 11/29/2014, Bing Li
	private Collaborator workCollaborator;
	// The time to wait when no requests are available. 11/29/2014, Bing Li
	private long dispatcherWaitTime;
	// The flag indicates whether the dispatcher has its own thread pool or shares with others. 11/29/2014, Bing Li
	private boolean isSelfThreadPool;

	/*
	 * Initialize the anycast request dispatcher. The constructor is called when the thread pool is owned by the dispatcher only. 11/29/2014, Bing Li
	 */
```

```
  public AnycastRequestDispatcher(FreeClientPool pool, String serverAddress, int serverPort, int
poolSize, long keepAliveTime, RequestThreadCreator threadCreator, int maxTaskSize, int
maxThreadSize, long dispatcherKeepAliveTime)
  {
    this.pool = pool;
    this.serverAddress = new IPPort(serverAddress, serverPort);
    this.threadMap = new ConcurrentHashMap<String, RequestThread>();
    this.requestQueue = new LinkedBlockingQueue<Request>();
    this.threadPool = new ThreadPool(poolSize, keepAliveTime);
    this.threadCreator = threadCreator;
    this.maxTaskSize = maxTaskSize;
    this.maxThreadSize = maxThreadSize;
    this.checkTimer = UtilConfig.NO_TIMER;
    this.workCollaborator = new Collaborator();
    this.dispatcherWaitTime = dispatcherKeepAliveTime;
    this.isSelfThreadPool = true;
  }

  /*
   * Dispose the anycast request dispatcher. The constructor is called when the thread pool is shared.
11/29/2014, Bing Li
   */
  public AnycastRequestDispatcher(FreeClientPool pool, String serverAddress, int serverPort,
ThreadPool threadPool, RequestThreadCreator threadCreator, int maxTaskSize, int maxThreadSize,
long dispatcherKeepAliveTime)
  {
    this.pool = pool;
    this.serverAddress = new IPPort(serverAddress, serverPort);
    this.threadMap = new ConcurrentHashMap<String, RequestThread>();
    this.requestQueue = new LinkedBlockingQueue<Request>();
    this.threadPool = threadPool;
    this.threadCreator = threadCreator;
    this.maxTaskSize = maxTaskSize;
    this.maxThreadSize = maxThreadSize;
    this.checkTimer = UtilConfig.NO_TIMER;
    this.workCollaborator = new Collaborator();
    this.dispatcherWaitTime = dispatcherKeepAliveTime;
    this.isSelfThreadPool = false;
  }

  /*
   * Dispose the request dispatcher. 11/29/2014, Bing Li
   */
  public synchronized void dispose()
  {
    // Set the shutdown flag to be true. Thus, the loop in the dispatcher thread to schedule requests load
is terminated. 11/29/2014, Bing Li
    this.workCollaborator.setShutdown();
    // Notify the dispatcher thread that is waiting for the requests to terminate the waiting. 11/29/2014,
Bing Li
    this.workCollaborator.signalAll();
    // Clear the request queue. 11/29/2014, Bing Li
    if (this.requestQueue != null)
    {
      this.requestQueue.clear();
    }
    // Cancel the timer that controls the idle checking. 11/29/2014, Bing Li
    if (this.checkTimer != UtilConfig.NO_TIMER)
    {
      this.checkTimer.cancel();
    }
    // Terminate the periodically running thread for idle checking. 11/29/2014, Bing Li
    if (this.idleChecker != null)
    {
      this.idleChecker.cancel();
    }
    // Dispose all of threads created during the dispatcher's running procedure. 11/29/2014, Bing Li
    for (RequestThread thread : this.threadMap.values())
```

```
    {
      thread.dispose();
    }
    // Clear the threads. 11/29/2014, Bing Li
    this.threadMap.clear();
    // Check whether the thread is owned by the dispatcher only. 11/29/2014, Bing Li
    if (this.isSelfThreadPool)
    {
      // Shutdown the thread pool. 11/29/2014, Bing Li
      this.threadPool.shutdown();
    }
    // Dispose the thread creator. 11/29/2014, Bing Li
    this.threadCreator = null;
  }

  /*
   * The method is called back by the idle checker periodically to monitor the idle states of threads within
the dispatcher. 11/29/2014, Bing Li
   */
  @Override
  public void checkIdle()
  {
    // Check each thread managed by the dispatcher. 11/29/2014, Bing Li
    for (RequestThread thread : this.threadMap.values())
    {
      // If the thread is empty and idle, it is the one to be checked. 11/29/2014, Bing Li
      if (thread.isEmpty() && thread.isIdle())
      {
        // The algorithm to determine whether a thread should be disposed or not is simple. When it is
checked to be idle, it is time to dispose it. 11/29/2014, Bing Li
        this.threadMap.remove(thread.getKey());
        // Dispose the thread. 11/29/2014, Bing Li
        thread.dispose();
        // Collect the resource of the thread. 11/29/2014, Bing Li
        thread = null;
      }
    }
  }

  /*
   * Set the idle checking parameters. 11/29/2014, Bing Li
   */
  @Override
  public void setIdleChecker(long idleCheckDelay, long idleCheckPeriod)
  {
    // Initialize the timer. 11/29/2014, Bing Li
    this.checkTimer = new Timer();
    // Initialize the idle checker. 11/29/2014, Bing Li
    this.idleChecker = new ThreadIdleChecker<AnycastRequestDispatcher<Request, Response,
RequestThread, RequestThreadCreator>>(this);
    // Schedule the idle checking task. 11/29/2014, Bing Li
    this.checkTimer.schedule(this.idleChecker, idleCheckDelay, idleCheckPeriod);
  }

  /*
   * Enqueue the newly received request into the dispatcher. 11/29/2014, Bing Li
   */
  public synchronized void enqueue(Request request)
  {
    // Enqueue the request into the queue. 11/29/2014, Bing Li
    this.requestQueue.add(request);
    // Notify the dispatcher thread, which is possibly blocked when no requests are available, to keep
working. 11/29/2014, Bing Li
    this.workCollaborator.signal();
  }

  /*
   * The thread of the dispatcher is always running until no requests from clients will be received. If too
```

```
many requests are received, more threads are created by the dispatcher to respond users in time. If
requests are limited, the count of threads created by the dispatcher is also small. It is possible no any
threads are alive when no requests are received for a long time. 11/29/2014, Bing Li
   */
  public void run()
  {
    // Declare a request. 11/29/2014, Bing Li
    Request request;
    // Initialize a thread map to calculate the load of each thread. 11/29/2014, Bing Li
    Map<String, Integer> threadTaskMap = new HashMap<String, Integer>();
    // Declare a string to keep the selected thread key. 11/29/2014, Bing Li
    String selectedThreadKey = UtilConfig.NO_KEY;
    // The dispatcher usually runs all of the time unless the server is shutdown. To shutdown the
dispatcher, the shutdown flag of the collaborator is set to true. 11/29/2014, Bing Li
    while (!this.workCollaborator.isShutdown())
    {
      try
      {
        // Check whether requests are received and saved in the queue. 11/29/2014, Bing Li
        while (!this.requestQueue.isEmpty())
        {
          // Dequeue the request from the queue of the dispatcher. 11/29/2014, Bing Li
          request = this.requestQueue.take();
          // Since all of the threads created by the dispatcher are saved in the map by their unique keys,
it is necessary to check whether any alive threads are available. If so, it is possible to assign tasks to
them if they are not so busy. 11/29/2014, Bing Li
          while (this.threadMap.size() > 0)
          {
            // Clear the map to start to calculate the load those threads. 11/29/2014, Bing Li
            threadTaskMap.clear();

            // Each thread's workload is saved into the threadTaskMap. 11/29/2014, Bing Li
            for (RequestThread thread : this.threadMap.values())
            {
              threadTaskMap.put(thread.getKey(), thread.getQueueSize());
            }
            // Select the thread whose load is the least and keep the key of the thread. 11/29/2014, Bing
Li
            selectedThreadKey = CollectionSorter.minValueKey(threadTaskMap);
            // Since no concurrency is applied here, it is possible that the key is invalid. Thus, just check
here. 11/19/2014, Bing Li
            if (selectedThreadKey != null)
            {
              // Since no concurrency is applied here, it is possible that the key is out of the map. Thus,
just check here. 11/19/2014, Bing Li
              if (this.threadMap.containsKey(selectedThreadKey))
              {
                try
                {
                  // Check whether the thread's load reaches the maximum value. 11/29/2014, Bing Li
                  if (this.threadMap.get(selectedThreadKey).isFull())
                  {
                    // Check if the pool is full. If the least load thread is full as checked by the above
condition, it denotes that all of the current alive threads are full. So it is required to create a thread to
respond the newly received requests if the thread count of the pool does not reach the maximum.
11/29/2014, Bing Li
                    if (this.threadMap.size() < this.maxThreadSize)
                    {
                      // Create a new thread. 11/29/2014, Bing Li
                      RequestThread thread =
this.threadCreator.createRequestThreadInstance(this.serverAddress, this.pool, this.maxTaskSize);
                      // Save the newly created thread into the map. 11/29/2014, Bing Li
                      this.threadMap.put(thread.getKey(), thread);
                      // Enqueue the request into the queue of the newly created thread. Then, the
request will be processed and responded by the thread. 11/29/2014, Bing Li
                      this.threadMap.get(thread.getKey()).enqueue(request);
                      // Start the thread by the thread pool. 11/29/2014, Bing Li
                      this.threadPool.execute(this.threadMap.get(thread.getKey()));
```

```
                    }
                    else
                    {
                      // Force to put the request into the queue when the count of threads reaches the
upper limit and each of the thread's queue is full. 11/29/2014, Bing Li
                      this.threadMap.get(selectedThreadKey).enqueue(request);
                    }
                  }
                  else
                  {
                    // If the least load thread's queue is not full, just put the request into the queue.
11/29/2014, Bing Li
                    this.threadMap.get(selectedThreadKey).enqueue(request);
                  }
                  // Jump out from the loop since the request is put into a thread. 11/29/2014, Bing Li
                  break;
                }
                catch (NullPointerException e)
                {
                  // Since no concurrency is applied here, it is possible that a NullPointerException is
raised. If so, it means that the selected thread is not available. Just continue to select anther one.
11/29/2014, Bing Li
                  continue;
                }
              }
            }
          }
          // If no threads are available, it needs to create a new one to take the request. 11/29/2014,
Bing Li
          if (this.threadMap.size() <= 0)
          {
            // Create a new thread. 11/29/2014, Bing Li
            RequestThread thread =
this.threadCreator.createRequestThreadInstance(this.serverAddress, this.pool, this.maxTaskSize);
            // Put it into the map for further reuse. 11/29/2014, Bing Li
            this.threadMap.put(thread.getKey(), thread);
            // Take the request. 11/29/2014, Bing Li
            this.threadMap.get(thread.getKey()).enqueue(request);
            // Start the thread. 11/29/2014, Bing Li
            this.threadPool.execute(this.threadMap.get(thread.getKey()));
          }
          // If the dispatcher is shutdown, it is not necessary to keep processing the requests. So, jump
out the loop and the thread is dead. 11/29/2014, Bing Li
          if (this.workCollaborator.isShutdown())
          {
            break;
          }
        }
        // Check whether the dispatcher is shutdown or not. 11/29/2014, Bing Li
        if (!this.workCollaborator.isShutdown())
        {
          // If the dispatcher is still alive, it denotes that no requests are available temporarily. Just wait
for a while. 11/29/2014, Bing Li
          this.workCollaborator.holdOn(this.dispatcherWaitTime);
        }
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **AnycastRequestThreadCreatable**

```
package com.greatfree.concurrency;

import com.greatfree.multicast.AnycastRequest;
import com.greatfree.multicast.AnycastResponse;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.remote.IPPort;

/*
 * The interface to define the method to create a thread for processing anycast requests concurrently.
11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public interface AnycastRequestThreadCreatable<Request extends AnycastRequest, Response
extends AnycastResponse, RequestThread extends AnycastRequestQueue<Request, Response>>
{
   public RequestThread createRequestThreadInstance(IPPort ipPort, FreeClientPool pool, int
taskSize);
}
```

- **BoundBroadcastRequestQueue**

```
package com.greatfree.concurrency;

import java.io.IOException;
import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.multicast.BroadcastRequest;
import com.greatfree.multicast.BroadcastResponse;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.remote.IPPort;
import com.greatfree.util.Tools;
import com.greatfree.util.UtilConfig;

/*
 * When processing broadcast requests, no matter whether the local memory node contains the matched data, it is required to forward the request to its children. That is the difference between the anycast and the broadcast. However, it is suggested that the retrieval and data forwarding can be done concurrently and they do not affect with one another. The thread is designed for the goal since they are synchronized once after both of them finish their critical tasks. Therefore, they do not affect each other. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class BoundBroadcastRequestQueue<Request extends BroadcastRequest, Response extends BroadcastResponse, RequestBinder extends MessageBindable<Request>> extends Thread
{
  // The unique key of the thread. It is convenient for managing it by a table-like mechanism. 11/29/2014, Bing Li
  private String key;
  // The dispatcher key that controls the thread. It represents one of threads that share the broadcast request. 11/29/2014, Bing Li
  private String dispatcherKey;
  // The queue to take the received broadcast requests. 11/29/2014, Bing Li
  private LinkedBlockingQueue<Request> queue;
  // The IP/port of the broadcast original initiator. The response must be sent back to it. 11/29/2014, Bing Li
  private IPPort ipPort;
  // The maximum size of the queue. 11/29/2014, Bing Li
  private int taskSize;
  // It is necessary to keep the thread waiting when no requests are available. The collaborator is used to notify the thread to keep working when requests are received. When the thread is idle enough, it can be collected. The collaborator is also used to control the life cycle of the thread. 11/29/2014, Bing Li
  private Collaborator collaborator;
  // The flag that represents whether the thread is busy or idle. 11/29/2014, Bing Li
  private boolean isIdle;
  // The TCP client pool. With it, it is able to get the instance of the client to connect the initiator of the broadcast. 11/29/2014, Bing Li
  private FreeClientPool pool;
  // The binder that receives notifications from all of threads that share the broadcast. After all of the threads have completed their concurrent tasks, the binder can do something that must be done in a synchronous way. 11/29/2014, Bing Li
  private RequestBinder reqBinder;

  /*
   * Initialize an instance. 11/29/2014, Bing Li
   */
  public BoundBroadcastRequestQueue(IPPort ipPort, FreeClientPool pool, String dispatcherKey, RequestBinder reqBinder)
  {
    this.key = Tools.generateUniqueKey();
    this.queue = new LinkedBlockingQueue<Request>();
    this.ipPort = ipPort;
    this.taskSize = UtilConfig.NO_QUEUE_SIZE;
    this.collaborator = new Collaborator();
```

```
    this.isIdle = false;
    this.pool = pool;
    this.dispatcherKey = dispatcherKey;
    this.reqBinder = reqBinder;
  }

  /*
   * Initialize an instance. 11/29/2014, Bing Li
   */
  public BoundBroadcastRequestQueue(IPPort ipPort, FreeClientPool pool, int taskSize, String
dispatcherKey, RequestBinder reqBinder)
  {
    this.key = Tools.generateUniqueKey();
    this.queue = new LinkedBlockingQueue<Request>();
    this.ipPort = ipPort;
    this.taskSize = taskSize;
    this.collaborator = new Collaborator();
    this.isIdle = false;
    this.pool = pool;
    this.dispatcherKey = dispatcherKey;
    this.reqBinder = reqBinder;
  }

  /*
   * Dispose the instance of the class. 11/29/2014, Bing Li
   */
  public synchronized void dispose()
  {
    // Set the flag to be the state of being shutdown. 11/29/2014, Bing Li
    this.collaborator.setShutdown();
    // Notify the thread being waiting to go forward. Since the shutdown flag is set, the thread must die
for the notification. 11/29/2014, Bing Li
    this.collaborator.signalAll();
    try
    {
      // Wait for the thread to die. 11/29/2014, Bing Li
      this.join();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
    // Clear the queue to release resources. 11/29/2014, Bing Li
    if (this.queue != null)
    {
      this.queue.clear();
    }
  }

  /*
   * Expose the key for the convenient management. 11/29/2014, Bing Li
   */
  public String getKey()
  {
    return this.key;
  }

  /*
   * Expose the dispatcher key. 11/29/2014, Bing Li
   */
  public String getDispatcherKey()
  {
    return this.dispatcherKey;
  }

  /*
   * Enqueue the broadcast request. 11/29/2014, Bing Li
   */
```

```
  public void enqueue(Request request) throws IllegalStateException
  {
    // Set the state of the thread to be busy. 11/29/2014, Bing Li
    this.setBusy();
    // Enqueue the request and its relevant output stream and lock. 11/29/2014, Bing Li
    this.queue.add(request);
    // Notify the waiting thread to keep on working since new requests are received. 11/29/2014, Bing Li
    this.collaborator.signal();
  }

  /*
   * Set the state to be busy. 11/29/2014, Bing Li
   */
  private synchronized void setBusy()
  {
    this.isIdle = false;
  }

  /*
   * Set the state to be idle. 11/29/2014, Bing Li
   */
  private synchronized void setIdle()
  {
    this.isIdle = true;
  }

  /*
   * Get the current size of the queue. 11/29/2014, Bing Li
   */
  public int getQueueSize()
  {
    return this.queue.size();
  }

  /*
   * The method intends to stop the thread temporarily when no requests are available. A thread is
identified as being idle immediately after the temporary waiting is finished. 11/29/2014, Bing Li
   */
  public void holdOn(long waitTime) throws InterruptedException
  {
    this.collaborator.holdOn(waitTime);
    this.setIdle();
  }

  /*
   * Check whether the shutdown flag of the thread is set or not. It might take some time for the thread to
be shutdown practically even though the flag is set. 11/29/2014, Bing Li
   */
  public boolean isShutdown()
  {
    return this.collaborator.isShutdown();
  }

  /*
   * Check whether the current size of the queue reaches the upper limit. 11/29/2014, Bing Li
   */
  public boolean isFull()
  {
    return this.queue.size() >= this.taskSize;
  }

  /*
   * Check whether the queue is empty. 11/29/2014, Bing Li
   */
  public boolean isEmpty()
  {
    return this.queue.size() <= 0;
  }
```

```
  /*
   * Check whether the thread is idle or not. 11/29/2014, Bing Li
   */
  public synchronized boolean isIdle()
  {
    return this.isIdle;
  }

  /*
   * Dequeue the request from the queue. 11/29/2014, Bing Li
   */
  public Request getRequest() throws InterruptedException
  {
    return this.queue.take();
  }

  /*
   * After the response is created, the method is responsible for sending it back to the anycast initiator.
11/29/2014, Bing Li
   */
  public synchronized void respond(Response response) throws IOException
  {
    this.pool.send(this.ipPort, response);
  }

  /*
   * Notify the binder that one thread completes its task. 11/29/2014, Bing Li
   */
  public synchronized void bind(String threadKey, Request request)
  {
    this.reqBinder.bind(threadKey, request);
  }

  /*
   * Dispose the request and the response. 11/29/2014, Bing Li
   */
  public synchronized void disposeResponse(Response response)
  {
    response = null;
  }
}
```

- **BoundBroadcastRequestDispatcher**

```
package com.greatfree.concurrency;

import java.util.HashMap;
import java.util.Map;
import java.util.Timer;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.multicast.BroadcastRequest;
import com.greatfree.multicast.BroadcastResponse;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.remote.IPPort;
import com.greatfree.util.CollectionSorter;
import com.greatfree.util.Tools;
import com.greatfree.util.UtilConfig;

/*
 * This is a dispatcher to manage broadcast request threads which need to share requests. The threads must be synchronized by the binder. 11/26/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class BoundBroadcastRequestDispatcher<Request extends BroadcastRequest, Response extends BroadcastResponse, RequestBinder extends MessageBindable<Request>, RequestThread extends BoundBroadcastRequestQueue<Request, Response, RequestBinder>, RequestThreadCreator extends BoundBroadcastRequestThreadCreatable<Request, Response, RequestBinder, RequestThread>> extends Thread implements CheckIdleable
{
  // The unique key of the dispatcher. It represents the thread in the binder. 11/29/2014, Bing Li
  private String key;
  // Declare a map to contain all of the threads. 11/29/2014, Bing Li
  private Map<String, RequestThread> threadMap;
  // Declare a queue to contain requests. 11/29/2014, Bing Li
  private LinkedBlockingQueue<Request> requestQueue;
  // Declare a thread pool that is used to run a thread. 11/29/2014, Bing Li
  private FreeClientPool pool;
  // The IP/port is the initiator of the broadcast requesting. Keep the information to send the response to it once if the result is obtained. 11/29/2014, Bing Li
  private IPPort serverAddress;
  // Declare a thread pool that is used to run a thread. 11/29/2014, Bing Li
  private ThreadPool threadPool;
  // Declare a thread creator that is used to initialize a thread instance. 11/29/2014, Bing Li
  private RequestThreadCreator threadCreator;
  // Declare the maximum task length for each thread to be created. 11/29/2014, Bing Li
  private int maxTaskSize;
  // Declare the maximum thread count that can be created in the dispatcher. 11/29/2014, Bing Li
  private int maxThreadSize;
  // Declare a timer that controls the task of idle checking. 11/29/2014, Bing Li
  private Timer checkTimer;
  // Declare the checker to check whether created threads are idle long enough. 11/29/2014, Bing Li
  private ThreadIdleChecker<BoundBroadcastRequestDispatcher<Request, Response, RequestBinder, RequestThread, RequestThreadCreator>> idleChecker;
  // The collaborator is used to pause the dispatcher when no requests are available and notify to continue when new requests are received. 11/29/2014, Bing Li
  private Collaborator collaborator;
  // The time to wait when no requests are available. 11/29/2014, Bing Li
  private long dispatcherWaitTime;
  // The binder that controls the synchronization for the threads that share the request. Only after all of them accomplish their own tasks concurrently, some synchronization tasks can be handled. It is managed by the binder. 11/29/2014, Bing Li
  private RequestBinder reqBinder;
  // The flag indicates whether the dispatcher has its own thread pool or shares with others. 11/29/2014, Bing Li
  private boolean isSelfThreadPool;
```

```
  /*
   * Initialize the broadcast request dispatcher. The constructor is called when the thread pool is owned
by the dispatcher only. 11/29/2014, Bing Li
   */
  public BoundBroadcastRequestDispatcher(FreeClientPool pool, String serverAddress, int
serverPort, int poolSize, long keepAliveTime, RequestBinder reqBinder, RequestThreadCreator
threadCreator, int maxTaskSize, int maxThreadSize, long dispatcherKeepAliveTime)
  {
    this.key = Tools.generateUniqueKey();
    this.pool = pool;
    this.serverAddress = new IPPort(serverAddress, serverPort);
    this.threadMap = new ConcurrentHashMap<String, RequestThread>();
    this.requestQueue = new LinkedBlockingQueue<Request>();
    this.threadPool = new ThreadPool(poolSize, keepAliveTime);
    this.threadCreator = threadCreator;
    this.maxTaskSize = maxTaskSize;
    this.maxThreadSize = maxThreadSize;
    this.checkTimer = UtilConfig.NO_TIMER;
    this.collaborator = new Collaborator();
    this.dispatcherWaitTime = dispatcherKeepAliveTime;
    this.reqBinder = reqBinder;
    this.reqBinder.addThread(this.key);
    this.isSelfThreadPool = true;
  }

  /*
   * Dispose the broadcast request dispatcher. The constructor is called when the thread pool is shared.
11/29/2014, Bing Li
   */
  public BoundBroadcastRequestDispatcher(FreeClientPool pool, String serverAddress, int
serverPort, ThreadPool threadPool, RequestBinder reqBinder, RequestThreadCreator threadCreator,
int maxTaskSize, int maxThreadSize, long dispatcherKeepAliveTime)
  {
    this.key = Tools.generateUniqueKey();
    this.pool = pool;
    this.serverAddress = new IPPort(serverAddress, serverPort);
    this.threadMap = new ConcurrentHashMap<String, RequestThread>();
    this.requestQueue = new LinkedBlockingQueue<Request>();
    this.threadPool = threadPool;
    this.threadCreator = threadCreator;
    this.maxTaskSize = maxTaskSize;
    this.maxThreadSize = maxThreadSize;
    this.checkTimer = UtilConfig.NO_TIMER;
    this.collaborator = new Collaborator();
    this.dispatcherWaitTime = dispatcherKeepAliveTime;
    this.reqBinder = reqBinder;
    this.reqBinder.addThread(this.key);
    this.isSelfThreadPool = false;
  }

  /*
   * Dispose the request dispatcher. 11/29/2014, Bing Li
   */
  public synchronized void dispose()
  {
    // Set the shutdown flag to be true. Thus, the loop in the dispatcher thread to schedule requests load
is terminated. 11/29/2014, Bing Li
    this.collaborator.setShutdown();
    // Notify the dispatcher thread that is waiting for the requests to terminate the waiting. 11/29/2014,
Bing Li
    this.collaborator.signalAll();
    // Clear the request queue. 11/29/2014, Bing Li
    if (this.requestQueue != null)
    {
      this.requestQueue.clear();
    }
    // Cancel the timer that controls the idle checking. 11/29/2014, Bing Li
```

```
    if (this.checkTimer != UtilConfig.NO_TIMER)
    {
      this.checkTimer.cancel();
    }
    // Terminate the periodically running thread for idle checking. 11/29/2014, Bing Li
    if (this.idleChecker != null)
    {
      this.idleChecker.cancel();
    }
    // Dispose all of threads created during the dispatcher's running procedure. 11/29/2014, Bing Li
    for (RequestThread thread : this.threadMap.values())
    {
      thread.dispose();
    }
    // Clear the threads. 11/29/2014, Bing Li
    this.threadMap.clear();
    // Check whether the thread is owned by the dispatcher only. 11/29/2014, Bing Li
    if (this.isSelfThreadPool)
    {
      // Shutdown the thread pool. 11/29/2014, Bing Li
      this.threadPool.shutdown();
    }
    // Dispose the thread creator. 11/29/2014, Bing Li
    this.threadCreator = null;
  }

  /*
   * The method is called back by the idle checker periodically to monitor the idle states of threads within
the dispatcher. 11/29/2014, Bing Li
   */
  @Override
  public void checkIdle()
  {
    // If the thread is empty and idle, it is the one to be checked. 11/29/2014, Bing Li
    for (RequestThread thread : this.threadMap.values())
    {
      // If the thread is empty and idle, it is the one to be checked. 11/29/2014, Bing Li
      if (thread.isEmpty() && thread.isIdle())
      {
        // The algorithm to determine whether a thread should be disposed or not is simple. When it is
checked to be idle, it is time to dispose it. 11/29/2014, Bing Li
        this.threadMap.remove(thread.getKey());
        // Dispose the thread. 11/29/2014, Bing Li
        thread.dispose();
        // Collect the resource of the thread. 11/29/2014, Bing Li
        thread = null;
      }
    }
  }

  /*
   * Set the idle checking parameters. 11/29/2014, Bing Li
   */
  @Override
  public void setIdleChecker(long idleCheckDelay, long idleCheckPeriod)
  {
    // Initialize the timer. 11/29/2014, Bing Li
    this.checkTimer = new Timer();
    // Initialize the idle checker. 11/29/2014, Bing Li
    this.idleChecker = new ThreadIdleChecker<BoundBroadcastRequestDispatcher<Request,
Response, RequestBinder, RequestThread, RequestThreadCreator>>(this);
    // Schedule the idle checking task. 11/29/2014, Bing Li
    this.checkTimer.schedule(this.idleChecker, idleCheckDelay, idleCheckPeriod);
  }

  /*
   * Enqueue the newly received request into the dispatcher. 11/29/2014, Bing Li
   */
```

```
  public synchronized void enqueue(Request request)
  {
    // Put the shared request into the binder for synchronization management. 11/29/2014, Bing Li
    this.reqBinder.set(request);
    // Enqueue the request into the queue. 11/29/2014, Bing Li
    this.requestQueue.add(request);
    // Notify the dispatcher thread, which is possibly blocked when no requests are available, to keep
working. 11/29/2014, Bing Li
    this.collaborator.signal();
  }

  /*
   * The thread of the dispatcher is always running until no requests from clients will be received. If too
many requests are received, more threads are created by the dispatcher to respond users in time. If
requests are limited, the count of threads created by the dispatcher is also small. It is possible no any
threads are alive when no requests are received for a long time. 11/29/2014, Bing Li
   */
  public void run()
  {
    // Declare a request. 11/29/2014, Bing Li
    Request request;
    // Initialize a thread map to calculate the load of each thread. 11/29/2014, Bing Li
    Map<String, Integer> threadTaskMap = new HashMap<String, Integer>();
    // Declare a string to keep the selected thread key. 11/29/2014, Bing Li
    String selectedThreadKey = UtilConfig.NO_KEY;
    // The dispatcher usually runs all of the time unless the server is shutdown. To shutdown the
dispatcher, the shutdown flag of the collaborator is set to true. 11/29/2014, Bing Li
    while (!this.collaborator.isShutdown())
    {
      try
      {
        // Check whether requests are received and saved in the queue. 11/29/2014, Bing Li
        while (!this.requestQueue.isEmpty())
        {
          // Dequeue the request from the queue of the dispatcher. 11/29/2014, Bing Li
          request = this.requestQueue.take();
          // Since all of the threads created by the dispatcher are saved in the map by their unique keys,
it is necessary to check whether any alive threads are available. If so, it is possible to assign tasks to
them if they are not so busy. 11/29/2014, Bing Li
          while (this.threadMap.size() > 0)
          {
            // Clear the map to start to calculate the load those threads. 11/29/2014, Bing Li
            threadTaskMap.clear();

            // Each thread's workload is saved into the threadTaskMap. 11/29/2014, Bing Li
            for (RequestThread thread : this.threadMap.values())
            {
              threadTaskMap.put(thread.getKey(), thread.getQueueSize());
            }
            // Select the thread whose load is the least and keep the key of the thread. 11/29/2014, Bing
Li
            selectedThreadKey = CollectionSorter.minValueKey(threadTaskMap);
            // Since no concurrency is applied here, it is possible that the key is invalid. Thus, just check
here. 11/19/2014, Bing Li
            if (selectedThreadKey != null)
            {
              // Since no concurrency is applied here, it is possible that the key is out of the map. Thus,
just check here. 11/19/2014, Bing Li
              if (this.threadMap.containsKey(selectedThreadKey))
              {
                try
                {
                  // Check whether the thread's load reaches the maximum value. 11/29/2014, Bing Li
                  if (this.threadMap.get(selectedThreadKey).isFull())
                  {
                    // Check if the pool is full. If the least load thread is full as checked by the above
condition, it denotes that all of the current alive threads are full. So it is required to create a thread to
respond the newly received requests if the thread count of the pool does not reach the maximum.
```

```
11/29/2014, Bing Li
                        if (this.threadMap.size() < this.maxThreadSize)
                        {
                          // Create a new thread. 11/29/2014, Bing Li
                          RequestThread thread =
this.threadCreator.createRequestThreadInstance(this.serverAddress, this.pool, this.maxTaskSize,
this.key, this.reqBinder);
                          // Save the newly created thread into the map. 11/29/2014, Bing Li
                          this.threadMap.put(thread.getKey(), thread);
                          // Enqueue the request into the queue of the newly created thread. Then, the
request will be processed and responded by the thread. 11/29/2014, Bing Li
                          this.threadMap.get(thread.getKey()).enqueue(request);
                          // Start the thread by the thread pool. 11/29/2014, Bing Li
                          this.threadPool.execute(this.threadMap.get(thread.getKey()));
                        }
                        else
                        {
                          // Force to put the request into the queue when the count of threads reaches the
upper limit and each of the thread's queue is full. 11/29/2014, Bing Li
                          this.threadMap.get(selectedThreadKey).enqueue(request);
                        }
                      }
                      else
                      {
                        // If the least load thread's queue is not full, just put the request into the queue.
11/29/2014, Bing Li
                        this.threadMap.get(selectedThreadKey).enqueue(request);
                      }
                      // Jump out from the loop since the request is put into a thread. 11/29/2014, Bing Li
                    break;
                    }
                    catch (NullPointerException e)
                    {
                      // Since no concurrency is applied here, it is possible that a NullPointerException is
raised. If so, it means that the selected thread is not available. Just continue to select anther one.
11/29/2014, Bing Li
                      continue;
                    }
                  }
                }
              }
              // If no threads are available, it needs to create a new one to take the request. 11/29/2014,
Bing Li
              if (this.threadMap.size() <= 0)
              {
                // Create a new thread. 11/29/2014, Bing Li
                RequestThread thread =
this.threadCreator.createRequestThreadInstance(this.serverAddress, this.pool, this.maxTaskSize,
this.key, this.reqBinder);
                // Put it into the map for further reuse. 11/29/2014, Bing Li
                this.threadMap.put(thread.getKey(), thread);
                // Take the request. 11/29/2014, Bing Li
                this.threadMap.get(thread.getKey()).enqueue(request);
                // Start the thread. 11/29/2014, Bing Li
                this.threadPool.execute(this.threadMap.get(thread.getKey()));
              }
              // If the dispatcher is shutdown, it is not necessary to keep processing the requests. So, jump
out the loop and the thread is dead. 11/29/2014, Bing Li
              if (this.collaborator.isShutdown())
              {
                break;
              }
            }
            // Check whether the dispatcher is shutdown or not. 11/29/2014, Bing Li
            if (!this.collaborator.isShutdown())
            {
              // If the dispatcher is still alive, it denotes that no requests are available temporarily. Just wait
for a while. 11/29/2014, Bing Li
```

```
            this.collaborator.holdOn(this.dispatcherWaitTime);
          }
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
    }
  }
```

- **BoundBroadcastRequestThreadCreatable**

```
package com.greatfree.concurrency;

import com.greatfree.multicast.BroadcastRequest;
import com.greatfree.multicast.BroadcastResponse;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.remote.IPPort;

/*
 * The interface defines the method to create the bound broadcast request thread. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public interface BoundBroadcastRequestThreadCreatable<Request extends BroadcastRequest,
Response extends BroadcastResponse, RequestBinder extends MessageBindable<Request>,
RequestThread extends BoundBroadcastRequestQueue<Request, Response, RequestBinder>>
{
   public RequestThread createRequestThreadInstance(IPPort ipPort, FreeClientPool pool, int
taskSize, String dispatcherKey, RequestBinder reqBinder);
}
```

2.4 Multicast

- **Tree**

```
package com.greatfree.multicast;

import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;

import com.greatfree.util.CollectionSorter;
import com.greatfree.util.UtilConfig;

/*
 * It aims to construct a tree to raise the quality of multicasting. The tree in the case is simple, such as each node having an equal number of children. The tree is acceptable when all of the nodes have a closed computing capacity and most of them run within a stable computing environment. For a heterogeneous environment, a more complicated tree or other topologies must be applied. 11/10/2014, Bing Li
 */

// Created: 11/10/2014, Bing Li
public class Tree
{
	/*
	 * Construct a tree. 11/10/2014, Bing Li
	 *
	 * Parameters:
	 *
	 *	String: rootKey, the root of the tree;
	 *
	 *	int: rootBranchCount, the count of the root;
	 *
	 *	Map<String, Integer>: nodes, all of nodes, excluding the root, in the tree. In addition, besides all of the keys of the nodes, the capacity of each node is also kept in the collection. The value is equal to the number of children the node can support when transmitting data.
	 */
	public static Map<String, List<String>> constructTree(String rootKey, int rootBranchCount, Map<String, Integer> nodes)
	{
		// Define a collection to take the tree. 11/10/2014, Bing Li
		Map<String, List<String>> tree = new HashMap<String, List<String>>();
		// Besides the root, the tree consists of has a number of children. All of them are numbered from the largest to the smallest according to their capacities. Moreover, for the root, it must has the count, rootBranchCount, of children. They are also numbered from the left to the right. The integer, firstChildIndex, represents the index of the leftmost one and the one, endChildIndex represents the rightmost one. 11/10/2014, Bing Li
		int firstChildIndex;
		int endChildIndex;
		// When constructing the tree, it is necessary to select the parent node. The parentNodeKey keeps the parent node temporarily. 11/10/2014, Bing Li
		String parentNodeKey;
		// The childNodeKey keeps the child node temporarily. 11/10/2014, Bing Li
		String childNodeKey;
		// Sort the nodes according to their capacities in the order from the larger to the smaller. 11/10/2014, Bing Li
		Map<String, Integer> nodeCapacityMap = CollectionSorter.sortDescendantByValue(nodes);
		// Assign the sorted nodes into a list, which is convenient to be accessed by the integer index. The smaller index represents a larger capacity, and vice versa. In general, the node which has a higher capacity must be placed in the higher level within the tree. The sorting supports the following algorithm reaches the goal. 11/10/2014, Bing Li
		List<String> nodeKeys = new LinkedList<String>(nodeCapacityMap.keySet());
		// The rootKey should also be counted. So plus one here. 11/10/2014, Bing Li
		int nodeSize = nodeKeys.size() + 1;
		// For each node, a certain number of children are assigned. Therefore, the tree is constructed. 11/10/2014, Bing Li
		for (int i = 0; i < nodeSize; i ++)
```

```
    {
        // To get the leftmost child for the node i, it is required to input the root children count and all of the
nodes' capacities. 11/10/2014, Bing Li
        firstChildIndex = getFirstChildIndex(i, rootBranchCount, nodeCapacityMap);
        // Check whether the first child index is less than the total node count to ensure it is valid.
Otherwise, it means all of the nodes have got their children such that the procedure can be terminated.
11/10/2014, Bing Li
        if (firstChildIndex < nodeSize)
        {
          // If i is equal to 0, it denotes that the current parent node key is the root key. 11/10/2014, Bing Li
          if (i == 0)
          {
            // Assign the root key to the parent node key. 11/10/2014, Bing Li
            parentNodeKey = rootKey;
          }
          else
          {
            // If i is not equal to 0, it is convenient to get the key by its index. 11/10/2014, Bing Li
            parentNodeKey = nodeKeys.get(i - 1);
          }

          // Check if i is equal to 0. 11/10/2014, Bing Li
          if (i == 0)
          {
            // Since i is equal to 0, it denotes that it is the first child index of the root key is found.
According to the capacity of the root, it is convenient to get the rightmost child of the root. 11/10/2014,
Bing Li
            endChildIndex = firstChildIndex + rootBranchCount - 1;
          }
          else
          {
            // Since i is not equal to 0, it denotes that it is the first child index of another node is found.
According to the capacity of the node, it is convenient to get its rightmost child. 11/10/2014, Bing Li
            endChildIndex = firstChildIndex + nodeCapacityMap.get(parentNodeKey) - 1;
          }

          // After the leftmost, firstChildIndex, and the rightmost, endChildIndex, are found, it is easy to get
all of the other node keys between them. 11/10/2014, Bing Li
          for (int j = firstChildIndex; j <= endChildIndex; j++)
          {
            // Check whether j is larger than the node size. If it is, it denotes that all of the nodes have got
their children. Then, the algorithm is terminated. 11/10/2014, Bing Li
            if (j < nodeSize)
            {
              // If j is less than the node size, get the key according to the index, j. 11/10/2014, Bing Li
              childNodeKey = nodeKeys.get(j - 1);
              // Check if the current parent key is put into the collection, tree, to be returned. 11/10/2014,
Bing Li
              if (!tree.containsKey(parentNodeKey))
              {
                // If the collection, tree, does not include the parent key, add it and its child key.
11/10/2014, Bing Li
                tree.put(parentNodeKey, new LinkedList<String>());
                tree.get(parentNodeKey).add(childNodeKey);
              }
              else
              {
                // If the collection, tree, includes the parent key, add its child key. 11/10/2014, Bing Li
                tree.get(parentNodeKey).add(childNodeKey);
              }
            }
            else
            {
              // Return the tree. 11/10/2014, Bing Li
              return tree;
            }
          }
        }
```

```
        else
        {
          // Return the tree. 11/10/2014, Bing Li
          return tree;
        }
      }
      // Return the tree. 11/10/2014, Bing Li
      return tree;
    }

    /*
     * The method constructs a tree in which each node, except the root node, has the identical capacity.
11/10/2014, Bing Li
     *
     * Parameters:
     *
     *    String: rootKey, the root of the tree;
     *
     *    List<String>: nodeKeys, all of the node keys to be added into the tree;
     *
     *    int: rootBranchCount, the root capacity or the branch count;
     *
     *    int: branchCount, the capacity or the branch count of each node, except the root.
     *
     */
    public static Map<String, List<String>> constructTree(String rootKey, List<String> nodeKeys, int
rootBranchCount, int branchCount)
    {
      // Define a collection to take the tree. 11/10/2014, Bing Li
      Map<String, List<String>> tree = new HashMap<String, List<String>>();
      // Besides the root, the tree consists of has a number of children. All of them are numbered from the
largest to the smallest according to their capacities. Moreover, for the root, it must has the count,
rootBranchCount, of children. They are also numbered from the left to the right. The integer,
firstChildIndex, represents the index of the leftmost one and the one, endChildIndex represents the
rightmost one. 11/10/2014, Bing Li
      int firstChildIndex;
      int endChildIndex;
      // When constructing the tree, it is necessary to select the parent node. The parentNodeKey keeps
the parent node temporarily. 11/10/2014, Bing Li
      String parentNodeKey;
      // The childNodeKey keeps the child node temporarily. 11/10/2014, Bing Li
      String childNodeKey;
      // The rootKey should also be counted. So plus one here. 11/10/2014, Bing Li
      int nodeSize = nodeKeys.size() + 1;
      // For each node, a certain number of children are assigned. Therefore, the tree is constructed.
11/10/2014, Bing Li
      for (int i = 0; i < nodeSize; i++)
      {
        // To get the leftmost child for the node i, it is required to input the root children count and all of the
nodes' capacities, which are the same in this case. 11/10/2014, Bing Li
        firstChildIndex = getFirstChildIndex(i, rootBranchCount, branchCount);
        // Check whether the first child index is less than the total node count to ensure it is valid.
Otherwise, it means all of the nodes have got their children such that the procedure can be terminated.
11/10/2014, Bing Li
        if (firstChildIndex < nodeSize)
        {
          // If i is equal to 0, it denotes that the current parent node key is the root key. 11/10/2014, Bing Li
          if (i == 0)
          {
            // Assign the root key to the parent node key. 11/10/2014, Bing Li
            parentNodeKey = rootKey;
          }
          else
          {
            // If i is not equal to 0, it is convenient to get the key by its index. 11/10/2014, Bing Li
            parentNodeKey = nodeKeys.get(i - 1);
          }
```

```
            // Check if i is equal to 0. 11/10/2014, Bing Li
            if (i == 0)
            {
              // Since i is equal to 0, it denotes that it is the first child index of the root key is found. According to the capacity of the root, it is convenient to get the rightmost child of the root. 11/10/2014, Bing Li
              endChildIndex = firstChildIndex + rootBranchCount - 1;
            }
            else
            {
              // Since i is not equal to 0, it denotes that it is the first child index of another node is found. According to the capacity of the node, it is convenient to get its rightmost child. 11/10/2014, Bing Li
              endChildIndex = firstChildIndex + branchCount - 1;
            }

            // After the leftmost, firstChildIndex, and the rightmost, endChildIndex, are found, it is easy to get all of the other node keys between them. 11/10/2014, Bing Li
            for (int j = firstChildIndex; j <= endChildIndex; j ++)
            {
              // Check whether j is larger than the node size. If it is, it denotes that all of the nodes have got their children. Then, the algorithm is terminated. 11/10/2014, Bing Li
              if (j < nodeSize)
              {
                // If j is less than the node size, get the key according to the index, j. 11/10/2014, Bing Li
                childNodeKey = nodeKeys.get(j - 1);
                // Check if the current parent key is put into the collection, tree, to be returned. 11/10/2014, Bing Li
                if (!tree.containsKey(parentNodeKey))
                {
                  // If the collection, tree, does not include the parent key, add it and its child key. 11/10/2014, Bing Li
                  tree.put(parentNodeKey, new LinkedList<String>());
                  tree.get(parentNodeKey).add(childNodeKey);
                }
                else
                {
                  // If the collection, tree, includes the parent key, add its child key. 11/10/2014, Bing Li
                  tree.get(parentNodeKey).add(childNodeKey);
                }
              }
              else
              {
                // Return the tree. 11/10/2014, Bing Li
                return tree;
              }
            }
          }
          else
          {
            // Return the tree. 11/10/2014, Bing Li
            return tree;
          }
        }
        // Return the tree. 11/10/2014, Bing Li
        return tree;
      }

      /*
       * This method aims to construct a tree, in which each node, including the root, has the identical capacity or the branch count. The first node in nodeKeys is regarded as the root. 11/10/2014, Bing Li
       */
      public static Map<String, List<String>> constructTree(List<String> nodeKeys, int branchCount)
      {
        // Define a collection to take the tree. 11/10/2014, Bing Li
        Map<String, List<String>> tree = new HashMap<String, List<String>>();
        // Besides the root, the tree consists of has a number of children. All of them are numbered from the largest to the smallest according to their capacities. Moreover, for the root, it must has the count, rootBranchCount, of children. They are also numbered from the left to the right. The integer,
```

```
firstChildIndex, represents the index of the leftmost one and the one, endChildIndex represents the
rightmost one. 11/10/2014, Bing Li
    int firstChildIndex;
    int endChildIndex;
    // When constructing the tree, it is necessary to select the parent node. The parentNodeKey keeps
the parent node temporarily. 11/10/2014, Bing Li
    String parentNodeKey;
    // The childNodeKey keeps the child node temporarily. 11/10/2014, Bing Li
    String childNodeKey;
    // For each node, a certain number of children are assigned. Therefore, the tree is constructed.
11/10/2014, Bing Li
    for (int i = 0; i < nodeKeys.size(); i++)
    {
      // To get the leftmost child for the node i, it is required to input the root children count and all of the
nodes' capacities, which are the same in this case. 11/10/2014, Bing Li
      firstChildIndex = getFirstChildIndex(i, branchCount);
      // Check whether the first child index is less than the total node count to ensure it is valid.
Otherwise, it means all of the nodes have got their children such that the procedure can be terminated.
11/10/2014, Bing Li
      if (firstChildIndex < nodeKeys.size())
      {
        // Get the parent node key by retrieving the node index. 11/10/2014, Bing Li
        parentNodeKey = nodeKeys.get(i);
        // Since each node has the identical capacity or the branch count, it is easy to estimate the
rightmost child index. 11/10/2014, Bing Li
        endChildIndex = firstChildIndex + branchCount - 1;
        // After the leftmost, firstChildIndex, and the rightmost, endChildIndex, are found, it is easy to get
all of the other node keys between them. 11/10/2014, Bing Li
        for (int j = firstChildIndex; j <= endChildIndex; j++)
        {
          // Check whether j is larger than the node size. If it is, it denotes that all of the nodes have got
their children. Then, the algorithm is terminated. 11/10/2014, Bing Li
          if (j < nodeKeys.size())
          {
            // If j is less than the node size, get the key according to the index, j. 11/10/2014, Bing Li
            childNodeKey = nodeKeys.get(j);
            // Check if the current parent key is put into the collection, tree, to be returned. 11/10/2014,
Bing Li
            if (!tree.containsKey(parentNodeKey))
            {
              // If the collection, tree, does not include the parent key, add it and its child key.
11/10/2014, Bing Li
              tree.put(parentNodeKey, new LinkedList<String>());
              tree.get(parentNodeKey).add(childNodeKey);
            }
            else
            {
              // If the collection, tree, includes the parent key, add its child key. 11/10/2014, Bing Li
              tree.get(parentNodeKey).add(childNodeKey);
            }
          }
          else
          {
            // Return the tree. 11/10/2014, Bing Li
            return tree;
          }
        }
      }
      else
      {
        // Return the tree. 11/10/2014, Bing Li
        return tree;
      }
    }
    // Return the tree. 11/10/2014, Bing Li
    return tree;
  }
```

```
  /*
   * This method aims to construct a tree, in which each node, including the root, has the identical capacity or the branch count. The root key is specified, excluding from the node keys. 11/10/2014, Bing Li
   */
  public static Map<String, List<String>> constructTree(String rootKey, List<String> nodeKeys, int branchCount)
  {
    // Define a collection to take the tree. 11/10/2014, Bing Li
    Map<String, List<String>> tree = new HashMap<String, List<String>>();
    // Besides the root, the tree consists of has a number of children. All of them are numbered from the largest to the smallest according to their capacities. Moreover, for the root, it must has the count, rootBranchCount, of children. They are also numbered from the left to the right. The integer, firstChildIndex, represents the index of the leftmost one and the one, endChildIndex represents the rightmost one. 11/10/2014, Bing Li
    int firstChildIndex;
    int endChildIndex;
    // When constructing the tree, it is necessary to select the parent node. The parentNodeKey keeps the parent node temporarily. 11/10/2014, Bing Li
    String parentNodeKey;
    // The childNodeKey keeps the child node temporarily. 11/10/2014, Bing Li
    String childNodeKey;
    // The rootKey should also be counted. So plus one here. 11/10/2014, Bing Li
    int nodeSize = nodeKeys.size() + 1;
    // For each node, a certain number of children are assigned. Therefore, the tree is constructed. 11/10/2014, Bing Li
    for (int i = 0; i < nodeSize; i ++)
    {
      // To get the leftmost child for the node i, it is required to input the root children count and all of the nodes' capacities, which are the same in this case. 11/10/2014, Bing Li
      firstChildIndex = getFirstChildIndex(i, branchCount);
      // Check whether the first child index is less than the total node count to ensure it is valid. Otherwise, it means all of the nodes have got their children such that the procedure can be terminated. 11/10/2014, Bing Li
      if (firstChildIndex < nodeSize)
      {
        // If i is equal to 0, it denotes that the current parent node key is the root key. 11/10/2014, Bing Li
        if (i == 0)
        {
          // Assign the root key to the parent node key. 11/10/2014, Bing Li
          parentNodeKey = rootKey;
        }
        else
        {
          // If i is not equal to 0, it is convenient to get the key by its index. 11/10/2014, Bing Li
          parentNodeKey = nodeKeys.get(i - 1);
        }
        // Since each node has the identical capacity or the branch count, it is easy to estimate the rightmost child index. 11/10/2014, Bing Li
        endChildIndex = firstChildIndex + branchCount - 1;
        // After the leftmost, firstChildIndex, and the rightmost, endChildIndex, are found, it is easy to get all of the other node keys between them. 11/10/2014, Bing Li
        for (int j = firstChildIndex; j <= endChildIndex; j ++)
        {
          // Check whether j is larger than the node size. If it is, it denotes that all of the nodes have got their children. Then, the algorithm is terminated. 11/10/2014, Bing Li
          if (j < nodeSize)
          {
            // If j is less than the node size, get the key according to the index, j. 11/10/2014, Bing Li
            childNodeKey = nodeKeys.get(j - 1);
            // Check if the current parent key is put into the collection, tree, to be returned. 11/10/2014, Bing Li
            if (!tree.containsKey(parentNodeKey))
            {
              // If the collection, tree, does not include the parent key, add it and its child key. 11/10/2014, Bing Li
              tree.put(parentNodeKey, new LinkedList<String>());
              tree.get(parentNodeKey).add(childNodeKey);
```

```
            }
            else
            {
              // If the collection, tree, includes the parent key, add its child key. 11/10/2014, Bing Li
              tree.get(parentNodeKey).add(childNodeKey);
            }
          }
          else
          {
            // Return the tree. 11/10/2014, Bing Li
            return tree;
          }
        }
      }
      else
      {
        // Return the tree. 11/10/2014, Bing Li
        return tree;
      }
    }
    // Return the tree. 11/10/2014, Bing Li
    return tree;
  }

  /*
   * With the constructed tree, get all of the children keys, from the immediate ones to the leaves, of a
parent node. 11/10/2014, Bing Li
   */
  public static List<String> getAllChildrenKeys(Map<String, List<String>> tree, String
parentNodeKey)
  {
    // Check whether the tree contains the parent node. 11/10/2014, Bing Li
    if (tree.containsKey(parentNodeKey))
    {
      // Initialize a list to keep the children keys. 11/10/2014, Bing Li
      List<String> allChildrenKeys = new LinkedList<String>();
      // Get the immediate children keys. 11/10/2014, Bing Li
      List<String> childrenKeys = tree.get(parentNodeKey);
      // Keep the immediate children keys. 11/10/2014, Bing Li
      allChildrenKeys.addAll(childrenKeys);
      // For each of the immediate child, get its children keys, from the immediate ones to the leaves.
11/10/2014, Bing Li
      for (String childrenKey : childrenKeys)
      {
        // Invoke the method recursively. 11/10/2014, Bing Li
        childrenKeys = getAllChildrenKeys(tree, childrenKey);
        // Check if the children keys are valid. 11/10/2014, Bing Li
        if (childrenKeys != UtilConfig.NO_CHILDREN_KEYS)
        {
          // Keep the immediate children keys. 11/10/2014, Bing Li
          allChildrenKeys.addAll(childrenKeys);
        }
      }
      // Return the children list. 11/10/2014, Bing Li
      return allChildrenKeys;
    }
    // Return null if the parent node key is not included in the tree. 11/10/2014, Bing Li
    return UtilConfig.NO_CHILDREN_KEYS;
  }

  /*
   * Get the tree level of interior nodes. 11/10/2014, Bing Li
   */
  private static int getTreeInteriorLevelCount(int treeNodeCount, int treeBranchCount)
  {
    // Check if the root branch is equal to 0. 11/10/2014, Bing Li
    if (treeBranchCount == 0)
    {
```

```
        // If the root branch is 0, it does not make sense in practice. Return 0. 11/10/2014, Bing Li
        return 0;
      }
      else
      {
        // Initialize the interior level as 0. 11/10/2014, Bing Li
        int interiorLevel = 0;
        // Initialize the currently node count in the tree as 0. 11/10/2014, Bing Li
        int currentNodeCount = 0;
        // Check whether the current node count in the tree is greater than the tree node count to be
added. 11/10/2014, Bing Li
        while (currentNodeCount < treeNodeCount)
        {
          // If the current node count in the tree is still less than the tree node count to be added, the
interior level should be incremented. 11/10/2014, Bing Li
          interiorLevel++;
          // The count of the interior nodes is equal to that of the root capacity multiplying the interior level
to the power of each node's capacity. 11/10/2014, Bing Li
          currentNodeCount += Math.pow(treeBranchCount, interiorLevel);
        }
        // Since the current node count in the tree is estimated from 0 and the interior level is incremented
at the value, the returned value should be subtracted 1. 11/10/2014, Bing Li
        return interiorLevel - 1;
      }
    }

    /*
     * Get the tree level of interior nodes. 11/10/2014, Bing Li
     */
    private static int getTreeInteriorLevelCount(int treeNodeCount, int rootBranchCount, int
treeBranchCount)
    {
      // Check if the root branch is equal to 0. 11/10/2014, Bing Li
      if (rootBranchCount == 0)
      {
        // If the root branch is 0, it does not make sense in practice. Return 0. 11/10/2014, Bing Li
        return 0;
      }
      else
      {
        // Initialize the interior level as 0. 11/10/2014, Bing Li
        int interiorLevel = 0;
        // Initialize the currently node count in the tree as 0. 11/10/2014, Bing Li
        int currentNodeCount = 0;
        // Check whether the current node count in the tree is greater than the tree node count to be
added. 11/10/2014, Bing Li
        while (currentNodeCount < treeNodeCount)
        {
          // If the current node count in the tree is still less than the tree node count to be added, the
interior level should be incremented. 11/10/2014, Bing Li
          interiorLevel++;
          // Since each node, except the root, has the identical capacity, the node count on the certain
level of the tree is easy to be estimated. 11/10/2014, Bing Li
          if (interiorLevel == 1)
          {
            // When the level is 1, the count of the interior nodes is equal to the root capacity. 11/10/2014,
Bing Li
            currentNodeCount += rootBranchCount;
          }
          else
          {
            // Otherwise, the count of the interior nodes is equal to that of the root capacity multiplying the
interior level to the power of each node's capacity. 11/10/2014, Bing Li
            currentNodeCount += rootBranchCount * Math.pow(treeBranchCount, interiorLevel);
          }
        }
        // Since the current node count in the tree is estimated from 0 and the interior level is incremented
at the value, the returned value should be subtracted 1. 11/10/2014, Bing Li
```

```
        return interiorLevel - 1;
      }
    }

    /*
     * Get the tree level of interior nodes. 11/10/2014, Bing Li
     */
    private static int getTreeInteriorLevelCount(int treeNodeCount, int rootBranchCount, Map<String,
Integer> nodeCapacityMap)
    {
      // Check if the root branch is equal to 0. 11/10/2014, Bing Li
      if (rootBranchCount == 0)
      {
        // If the root branch is 0, it does not make sense in practice. Return 0. 11/10/2014, Bing Li
        return 0;
      }
      else
      {
        // Initialize the interior level as 0. 11/10/2014, Bing Li
        int interiorLevel = 0;
        // Initialize the currently node count in the tree as 0. 11/10/2014, Bing Li
        int currentNodeCount = 0;
        // Check whether the current node count in the tree is greater than the tree node count to be
added. 11/10/2014, Bing Li
        while (currentNodeCount < treeNodeCount)
        {
          // If the current node count in the tree is still less than the tree node count to be added, the
interior level should be incremented. 11/10/2014, Bing Li
          interiorLevel++;
          // With the predefined interior level, each node's capacity and the root branch, invoke the method
to get the current node count in the tree. 11/10/2014, Bing Li
          currentNodeCount = getCompleteTreeNodeCount(nodeCapacityMap, rootBranchCount,
interiorLevel);
        }
        // Since the current node count in the tree is estimated from 0 and the interior level is incremented
at the value, the returned value should be subtracted 1. 11/10/2014, Bing Li
        return interiorLevel - 1;
      }
    }

    /*
     * Get all of the tree node count in the form of the complete tree upon the tree level, each node's
capacity and the root branch. 11/10/2014, Bing Li
     */
    private static int getCompleteTreeNodeCount(Map<String, Integer> nodeCapacityMap, int
rootBranchCount, int levelCount)
    {
      // If the level count is 0, the tree node count is 1. 11/10/2014, Bing Li
      if (levelCount == 0)
      {
        return 1;
      }
      // If the level count is 1, the tree node count is equal to the sum of the root node plus the root branch
count. 11/10/2014, Bing Li
      else if (levelCount == 1)
      {
        return rootBranchCount + 1;
      }
      else
      {
        // Get the tree node count in the form of the complete tree of the father level. This is a recursive
call. 11/10/2014, Bing Li
        int upperFatherLevelsTreeNodeCount = getCompleteTreeNodeCount(nodeCapacityMap,
rootBranchCount, levelCount - 1);
        // Get the tree node count in the form of the complete tree of the grandfather level. This is a
recursive call. 11/10/2014, Bing Li
        int upperGrandfatherLevelsTreeNodeCount = getCompleteTreeNodeCount(nodeCapacityMap,
rootBranchCount, levelCount - 2);
```

```
        // Initialize the complete tree node count as the value of the tree node count in the form of the
complete tree of the father level. 11/10/2014, Bing Li
        int completeTreeNodeCount = upperFatherLevelsTreeNodeCount;
        // Save the node keys into a list so as to retrieve the node key by the node index. 11/10/2014, Bing
Li
        List<String> nodeKeys = new LinkedList<String>(nodeCapacityMap.keySet());
        // Add the capacities of nodes on the father level to the tree node count on the father level. The
total count is the node count of the current level. 11/10/2014, Bing Li
        for (int i = upperGrandfatherLevelsTreeNodeCount; i < upperFatherLevelsTreeNodeCount; i++)
        {
          // The index, i, is valid only if it is less than the total node size minus 1. 11/10/2014, Bing Li
          if (i - 1< nodeKeys.size())
          {
            // Add the capacity of one node's capacity on the father level. 11/10/2014, Bing Li
            completeTreeNodeCount += nodeCapacityMap.get(nodeKeys.get(i - 1));
          }
          else
          {
            // If the index, i, is not valid, jump out the loop. 11/10/2014, Bing Li
            break;
          }
        }
        // Return the tree node in the form of the complete tree. 11/10/2014, Bing Li
        return completeTreeNodeCount;
      }
    }

    /*
     * Get the count of the tree level once if the tree has the certain number of nodes and their capacities
are identical. 11/10/2014, Bing Li
     */
    private static int getTreeLevelCount(int treeNodeCount, int treeBranchCount)
    {
      // Invoke the method to get the tree level of interior nodes. To get the total level of the tree, add one.
11/10/2014, Bing Li
      return getTreeInteriorLevelCount(treeNodeCount, treeBranchCount) + 1;
    }

    /*
     * Get the count of the tree level once if the tree has the certain number of nodes and their capacities
are identical except that of the root. 11/10/2014, Bing Li
     */
    private static int getTreeLevelCount(int treeNodeCount, int rootBranchCount, int
treeBranchCount)
    {
      // Invoke the method to get the tree level of interior nodes. To get the total level of the tree, add one.
11/10/2014, Bing Li
      return getTreeInteriorLevelCount(treeNodeCount, rootBranchCount, treeBranchCount) + 1;
    }

    /*
     * Get the count of the tree level once if the tree has the certain number of nodes and their capacities
are known. 11/10/2014, Bing Li
     */
    private static int getTreeLevelCount(int treeNodeCount, int rootBranchCount, Map<String,
Integer> nodeCapacityMap)
    {
      // Invoke the method to get the tree level of interior nodes. To get the total level of the tree, add one.
11/10/2014, Bing Li
      return getTreeInteriorLevelCount(treeNodeCount, rootBranchCount, nodeCapacityMap) + 1;
    }

    /*
     * Once if the tree level is known and each node has the identical capacity or the identical branch
count, the count of the tree nodes in the form of the complete tree is the sum of each node's capacity. It
can be estimated with the support of power. 11/10/2014, Bing Li
     */
    private static int getCompleteTreeNodeCount(int treeBranchCount, double treeLevelCount)
```

```
  {
    int interiorNodeCount = 0;
    for (int i = 0; i <= treeLevelCount; i++)
    {
      interiorNodeCount += Math.pow(treeBranchCount, i);
    }
    return interiorNodeCount;
  }

  /*
   * Once if the tree level is known and each node has the identical capacity or the identical branch
count except that of the root, the count of the tree nodes in the form of the complete tree is the sum of
each node's capacity. It can be estimated with the support of power. 11/10/2014, Bing Li
   */
  private static int getCompleteTreeNodeCount(int rootBranchCount, int treeBranchCount, double
treeLevelCount)
  {
    int interiorNodeCount = 0;
    for (int i = 0; i <= treeLevelCount; i++)
    {
      if (i == 0)
      {
        interiorNodeCount += 1;
      }
      else
      {
        interiorNodeCount += rootBranchCount * Math.pow(treeBranchCount, i - 1);
      }
    }
    return interiorNodeCount;
  }

  /*
   * The method gets the tree level a node resides. 11/10/2014, Bing Li
   */
  private static int getTreeLevelResided(int nodeIndex, int treeBranchCount)
  {
    // Get the count of the tree level once if the tree has the certain number of nodes and their capacities
are identical. Since the node index starts from zero, it is required to add one to the index to invoke the
method. 11/10/2014, Bing Li
    return getTreeLevelCount(nodeIndex + 1, treeBranchCount);
  }

  /*
   * The method gets the tree level a node resides. 11/10/2014, Bing Li
   */
  private static int getTreeLevelResided(int nodeIndex, int rootBranchCount, int treeBranchCount)
  {
    // Get the count of the tree level once if the tree has the certain number of nodes and their capacities
are identical except that of the root. Since the node index starts from zero, it is required to add one to the
index to invoke the method. 11/10/2014, Bing Li
    return getTreeLevelCount(nodeIndex + 1, rootBranchCount, treeBranchCount);
  }

  /*
   * The method gets the tree level a node resides. The node is represented by its index after sorting by
node capacities in the descendant order. 11/10/2014, Bing Li
   */
  private static int getTreeLevelResided(int nodeIndex, int rootBranchCount, Map<String, Integer>
nodeCapacityMap)
  {
    // Get the count of the tree level once if the tree has the certain number of nodes and their capacities
are known. Since the node index starts from zero, it is required to add one to the index to invoke the
method. 11/10/2014, Bing Li
    return getTreeLevelCount(nodeIndex + 1, rootBranchCount, nodeCapacityMap);
  }

  /*
```

```
   * The method gets the leftmost index, firstChildIndex, of a node's children. Each node has the
identical capacity or the branch count. 11/10/2014, Bing Li
   */
  private static int getFirstChildIndex(int currentNodeIndex, int treeBranchCount)
  {
    // Check whether the index of the current node is greater than 0. 11/10/2014, Bing Li
    if (currentNodeIndex > 0)
    {
      // Get the level of the tree the current node resides. 11/10/2014, Bing Li
      int treeLevelResided = getTreeLevelResided(currentNodeIndex, treeBranchCount);
      // Check if the resided tree level is greater than 0. 11/10/2014, Bing Li
      if (treeLevelResided > 0)
      {
        // If the resided tree level is obtained, it is reasonable to get the node count of all of upper levels.
11/10/2014, Bing Li
        int upperLevelNodeCount = getCompleteTreeNodeCount(treeBranchCount, treeLevelResided -
1);
        // After getting the node count of all of upper levels, it is reasonable to get the index of the
current node is sorted on its own level. 11/10/2014, Bing Li
        int currentLevelNodeIndex = currentNodeIndex - upperLevelNodeCount;
        /*
         *  Since except the root node, each node has the same capacity, to get the leftmost child index,
the value is the sum of
         *
         *      The total count of the nodes before the current node on the level multiplying the branch
count;
         *
         *      The count of all of the nodes in the tree of the level.
         *
         *  11/10/2014, Bing Li
         *
         */
        return currentLevelNodeIndex * treeBranchCount +
getCompleteTreeNodeCount(treeBranchCount, treeLevelResided);
      }
      else
      {
        // According to the algorithm, the resided tree level is equal to 0 only if the tree contains only one
node, i.e., the root node. It does not make sense in practice. Therefore, if the level is not greater than 0,
the leftmost node index of 0. 11/10/2014, Bing Li
        return 0;
      }
    }
    else
    {
      // If the index of the current node is not greater than 0, it denotes it is the root. Then, its leftmost
child index is 1. Return 1. 11/10/2014, Bing Li
      return 1;
    }
  }

  /*
   * The method gets the leftmost index, firstChildIndex, of a node's children. 11/10/2014, Bing Li
   */
  private static int getFirstChildIndex(int currentNodeIndex, int rootBranchCount, int
treeBranchCount)
  {
    // Check whether the index of the current node is greater than 0. 11/10/2014, Bing Li
    if (currentNodeIndex > 0)
    {
      // Get the level of the tree the current node resides. 11/10/2014, Bing Li
      int treeLevelResided = getTreeLevelResided(currentNodeIndex, rootBranchCount,
treeBranchCount);
      // Check if the resided tree level is greater than 0. 11/10/2014, Bing Li
      if (treeLevelResided > 0)
      {
        // If the resided tree level is obtained, it is reasonable to get the node count of all of upper levels.
11/10/2014, Bing Li
```

```
            int upperLevelNodeCount = getCompleteTreeNodeCount(rootBranchCount, treeBranchCount,
treeLevelResided - 1);
            // After getting the node count of all of upper levels, it is reasonable to get the index of the
current node is sorted on its own level. 11/10/2014, Bing Li
            int currentLevelNodeIndex = currentNodeIndex - upperLevelNodeCount;
            /*
             * Since except the root node, each node has the same capacity, to get the leftmost child index,
the value is the sum of
             *
             *      The total count of the nodes before the current node on the level multiplying the branch
count;
             *
             *      The count of all of the nodes in the tree of the level.
             *
             * 11/10/2014, Bing Li
             *
             */
            return currentLevelNodeIndex * treeBranchCount +
getCompleteTreeNodeCount(rootBranchCount, treeBranchCount, treeLevelResided);
          }
          else
          {
            // According to the algorithm, the resided tree level is equal to 0 only if the tree contains only one
node, i.e., the root node. It does not make sense in practice. Therefore, if the level is not greater than 0,
the leftmost node index of 0. 11/10/2014, Bing Li
            return 0;
          }
        }
        else
        {
          // If the index of the current node is not greater than 0, it denotes it is the root. Then, its leftmost
child index is 1. Return 1. 11/10/2014, Bing Li
          return 1;
        }
      }

      /*
       * The method gets the leftmost index, firstChildIndex, of a node's children. 11/10/2014, Bing Li
       */
      private static int getFirstChildIndex(int currentNodeIndex, int rootBranchCount, Map<String,
Integer> nodeCapacityMap)
      {
        // Check whether the index of the current node is greater than 0. 11/10/2014, Bing Li
        if (currentNodeIndex > 0)
        {
          // Get the level of the tree the current node resides. 11/10/2014, Bing Li
          int treeLevelResided = getTreeLevelResided(currentNodeIndex, rootBranchCount,
nodeCapacityMap);
          // Check if the resided tree level is greater than 0. 11/10/2014, Bing Li
          if (treeLevelResided > 0)
          {
            // If the resided tree level is obtained, it is reasonable to get the node count of all of upper levels.
11/10/2014, Bing Li
            int upperLevelsNodeCount = getCompleteTreeNodeCount(nodeCapacityMap,
rootBranchCount, treeLevelResided - 1);
            // After getting the node count of all of upper levels, it is reasonable to get the index of the
current node is sorted on its own level. 11/10/2014, Bing Li
            int currentLevelNodeIndex = currentNodeIndex - upperLevelsNodeCount;
            // Then, get the count of nodes after the new level the current node resides has added.
11/10/2014, Bing Li
            int currentTreeNodeCount = getCompleteTreeNodeCount(nodeCapacityMap, rootBranchCount,
treeLevelResided);
            // Declare the node index to retrieve the corresponding node key. 11/10/2014, Bing Li
            int nodeIndex;
            // If the current node happens to be the first one on its level, then the index of its leftmost child is
equal to the count of nodes in all of the upper levels, including the one the current node resides.
11/10/2014, Bing Li
            int firstChildIndex = currentTreeNodeCount;
```

```
            // If the current node is not the first one on its level, which is the common case, then it is
necessary to add the total capacities of the nodes before the current node on the same level. Initializing
the following list is convenient to get the node key on the node index. 11/10/2014, Bing Li
            List<String> nodeKeys = new LinkedList<String>(nodeCapacityMap.keySet());
            // Add the total capacities of the nodes before the current node on the same level is not the first
one on the level. 11/10/2014, Bing Li
            for (int i = 0; i < currentLevelNodeIndex; i ++)
            {
               // Get the node index of the node before the current node on the same level. 11/10/2014, Bing
Li
               nodeIndex = i + upperLevelsNodeCount - 1;
               // Check whether the node index is less than the node size. 11/10/2014, Bing Li
               if (nodeIndex < nodeCapacityMap.size())
               {
                  // Add the capacity of one node that resides before the current node on the same level.
11/10/2014, Bing Li
                  firstChildIndex += nodeCapacityMap.get(nodeKeys.get(nodeIndex));
               }
               else
               {
                  // If the the node index is not less than the node size, it does not make sense. Just break
here. 11/10/2014, Bing Li
                  break;
               }
            }
            // Return the index of leftmost child of the current node. 11/10/2014, Bing Li
            return firstChildIndex;
         }
         else
         {
            // According to the algorithm, the resided tree level is equal to 0 only if the tree contains only one
node, i.e., the root node. It does not make sense in practice. Therefore, if the level is not greater than 0,
the leftmost node index of 0. 11/10/2014, Bing Li
            return 0;
         }
      }
      else
      {
         // If the index of the current node is not greater than 0, it denotes it is the root. Then, its leftmost
child index is 1. Return 1. 11/10/2014, Bing Li
         return 1;
      }
   }
}
```

- **ServerMessage**

```
package com.greatfree.multicast;

import java.io.Serializable;

/*
 * The class is the base for all messages transmitted between remote clients/servers. 07/30/2014, Bing
Li
 */

// Created: 07/30/2014, Bing Li
public class ServerMessage implements Serializable
{
   private static final long serialVersionUID = 955012179332438088L;

   // The key of the message. When a message needs to be shared by multiple threads and the relevant
synchronization control is required, the key must be initialized. It must be unique in the process.
09/19/2014, Bing Li
   private String key;
   // The type of the message. 09/19/2014, Bing Li
   private int type;

   /*
    * Initialize the message without initializing the key. 09/19/2014, Bing Li
    */
   public ServerMessage(int type)
   {
      this.type = type;
   }

   /*
    * Initialize the message. The constructor initializes the type and the key. 09/19/2014, Bing Li
    */
   public ServerMessage(int type, String key)
   {
      this.type = type;
      this.key = key;
   }

   /*
    * Expose the type. 09/19/2014, Bing Li
    */
   public int getType()
   {
      return this.type;
   }

   /*
    * Expose the key. 09/19/2014, Bing Li
    */
   public String getKey()
   {
      return this.key;
   }
}
```

- **ServerMulticastMessage**

```
package com.greatfree.multicast;

import java.util.Map;

import com.greatfree.util.UtilConfig;


/*
 * This is the base class to implement the message that can be multicast among a bunch of nodes. 11/09/2014, Bing Li
 */

// Created: 11/09/2014, Bing Li
public class ServerMulticastMessage extends ServerMessage
{
	private static final long serialVersionUID = -6067174732413112924L;

	// The nodes to receive the message to be multicast. 11/09/2014, Bing Li
	private Map<String, String> childrenNodes;

	/*
	 * Initialize the message to be multicast. 11/10/2014, Bing Li
	 *
	 * The argument, messageType, represents the type of the message;
	 *
	 * The argument, key, represents the unique key of the message;
	 *
	 * The argument, childrenNodes, keeps all of the nodes that receive the message. The key of the collection is the unique key of the client and the value is the IP address. Here, the port number is omitted since usually the port of all of the nodes in a cluster is the same.
	 *
	 */
	public ServerMulticastMessage(int messageType, String key, Map<String, String> childrenNodes)
	{
		super(messageType, key);
		this.childrenNodes = childrenNodes;
	}

	/*
	 * Another constructor of the message. This is invoked by the node which has no grandsons. 11/10/2014, Bing Li
	 */
	public ServerMulticastMessage(int messageType, String key)
	{
		super(messageType, key);
		this.childrenNodes = UtilConfig.NO_NODES;
	}

	/*
	 * Set the children nodes. The method is usually invoked when it is failed to send a message to a particular node. Therefore, it is necessary to choose another node and reset the children nodes to it after removing the failed one. 11/10/2014, Bing Li
	 */
	public void setChildrenNodes(Map<String, String> childrenNodes)
	{
		this.childrenNodes = childrenNodes;
	}

	/*
	 * Get the IP of a specific child by its key. With the IP, it is convenient to connect the child by the FreeClient pool. 11/10/2014, Bing Li
	 */
	public String getChildrenNodeIP(String childrenKey)
	{
```

```
    // Check whether the child key exists. 11/10/2014, Bing Li
    if (this.childrenNodes.containsKey(childrenKey))
    {
      // Return the IP of the child. 11/10/2014, Bing Li
      return this.childrenNodes.get(childrenKey);
    }
    // Return null if the child key does not exist. 11/10/2014, Bing Li
    return UtilConfig.NO_IP;
  }

  /*
   * Get the collection of all of the children. The method is invoked when it is received by a particular
node and it is required for the node to keep transfer the message to those children. After getting the keys
of the children, the node is able to construct a tree for those children to raise the multicast efficiency.
11/10/2014, Bing Li
   */
  public Map<String, String> getChildrenNodes()
  {
    return this.childrenNodes;
  }
}
```

- **BroadcastRequest**

```
package com.greatfree.multicast;

import java.util.HashMap;

/*
 * The message is a broadcast request to be sent through all of the distributed nodes to retrieve required data. For multicasting is required, it extends ServerMulticastMessage. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class BroadcastRequest extends ServerMulticastMessage
{
  private static final long serialVersionUID = -5477115431261715561L;

  // To implement the broadcast, the request contains the collaborator key that is associated with the instance of Collaborator that is waiting at the source node that sends the broadcast request. Once if the required data is obtained at one node, it must be returned to the source from the destination. The collaborator must be signaled by the responses. As a broadcast, all of the nodes must respond before the requesting is finished. 11/28/2014, Bing Li
  private String collaboratorKey;

  /*
   * Initialize. The parameters are explained as follows. 11/28/2014, Bing Li
   *
   *      int dataType: the message type;
   *
   *      String key: the message key;
   *
   *      String collaboratorKey: the collaborator key pointing to the instance of Collaborator to collect all of the results by synchronization
   *
   *      HashMap<String, String> childrenServerMap: when an intermediate node receives the request, it must forward the request to its children with the assistance of information.
   */
  public BroadcastRequest(int dataType, String key, String collaboratorKey, HashMap<String, String> childrenServerMap)
  {
    super(dataType, key, childrenServerMap);
    this.collaboratorKey = collaboratorKey;
  }

  /*
   * Initialize. This constructor is used for the case when the requestor has no children nodes. 11/28/2014, Bing Li
   */
  public BroadcastRequest(int dataType, String key, String collaboratorKey)
  {
    super(dataType, key);
    this.collaboratorKey = collaboratorKey;
  }

  /*
   * Expose the collaborator. 11/28/2014, Bing Li
   */
  public String getCollaboratorKey()
  {
    return this.collaboratorKey;
  }
}
```

- **BroadcastResponse**

```
package com.greatfree.multicast;

/*
 * The message is a broadcast response to be responded to the initial requester after retrieving the
required data. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class BroadcastResponse extends ServerMessage
{
  private static final long serialVersionUID = 200920679139533863L;

  // The message key. 11/28/2014, Bing Li
  private String key;
  // The collaborator to assist the synchronization for collecting broadcast retrieved data. 11/28/2014,
Bing Li
  private String collaboratorKey;

  /*
   * Initialize. The parameters are explained as follows. 11/28/2014, Bing Li
   *
   *      int dataType: the message type;
   *
   *      String key: the message key;
   *
   *      String collaboratorKey: the collaborator key pointing to the instance of Collaborator to collect all
of the results by synchronization
   */
  public BroadcastResponse(int dataType, String key, String collaboratorKey)
  {
    super(dataType);
    this.key = key;
    this.collaboratorKey = collaboratorKey;
  }

  /*
   * Expose the key. 11/28/2014, Bing Li
   */
  public String getKey()
  {
    return this.key;
  }

  /*
   * Expose the collaborator key. 11/28/2014, Bing Li
   */
  public String getCollaboratorKey()
  {
    return this.collaboratorKey;
  }
}
```

- **AnycastRequest**

```
package com.greatfree.multicast;

import java.util.HashMap;

/*
 * The request is a multicast one that is sent to all of the nodes in a cluster. However, once if one node at least responds the request positively, the multicast requesting must be terminated. That is the difference from the broadcast requesting. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class AnycastRequest extends ServerMulticastMessage
{
  private static final long serialVersionUID = 1555137356272335216L;

  // To implement the anycast, the request contains the collaborator key that is associated with the instance of Collaborator that is waiting at the source node that sends the anycast request. Once if the required data is obtained at one node, it must be returned to the source from the destination. The collaborator must be signaled by the responses. As an anycast, only one response from all of the nodes can terminate the requesting. 11/28/2014, Bing Li
  private String collaboratorKey;

  /*
   * Initialize. The parameters are explained as follows. 11/29/2014, Bing Li
   *
   *      int dataType: the message type;
   *
   *      String key: the message key;
   *
   *      String collaboratorKey: the collaborator key pointing to the instance of Collaborator to collect all of the results by synchronization
   *
   *      HashMap<String, String> childrenServerMap: when an intermediate node receives the request, it must forward the request to its children with the assistance of information.
   */
  public AnycastRequest(int messageType, String key, String collaboratorKey, HashMap<String, String> childrenServerMap)
  {
    super(messageType, key);
    this.collaboratorKey = collaboratorKey;
  }

  /*
   * Initialize. This constructor is used for the case when the requestor has no children nodes. 11/28/2014, Bing Li
   */
  public AnycastRequest(int messageType, String key, String collaboratorKey)
  {
    super(messageType, key);
    this.collaboratorKey = collaboratorKey;
  }

  /*
   * Expose the collaborator. 11/28/2014, Bing Li
   */
  public String getCollaboratorKey()
  {
    return this.collaboratorKey;
  }
}
```

- **AnycastResponse**

```
package com.greatfree.multicast;

/*
 * The message is an anycast response to be responded to the initial requester after retrieving the
required data. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class AnycastResponse extends ServerMessage
{
  private static final long serialVersionUID = 1170590433477187242L;

  // The collaborator to assist the synchronization for collecting broadcast retrieved data. 11/29/2014,
Bing Li
  private String collaboratorKey;

  /*
   * Initialize. The parameters are explained as follows. 11/29/2014, Bing Li
   *
   *      int dataType: the message type;
   *
   *      String collaboratorKey: the collaborator key pointing to the instance of Collaborator to collect all
of the results by synchronization
   */
  public AnycastResponse(int type, String collaboratorKey)
  {
    super(type);
    this.collaboratorKey = collaboratorKey;
  }

  /*
   * Expose the collaborator key. 11/29/2014, Bing Li
   */
  public String getCollaboratorKey()
  {
    return this.collaboratorKey;
  }
}
```

- **RootObjectMulticastor**

```
package com.greatfree.multicast;

import java.io.IOException;
import java.io.Serializable;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;

import com.greatfree.remote.FreeClientPool;
import com.greatfree.util.HashFreeObject;
import com.greatfree.util.Rand;
import com.greatfree.util.UtilConfig;

/*
 * The code is a core component to achieve the multicast goal among a bunch of nodes. The nodes are usually organized into a particular topology to raise the multicast efficiency. In the case, a tree is constructed for the nodes. For different situations, a more appropriate topology can be selected. 11/10/2014, Bing Li
 *
 * In addition, the tree constructed is simple. The root node has a branch count and other nodes have the identical branch count. That indicates that the root node might have a different capacity to send messages concurrently from others. And, all of the nodes except the root has the same computing power and bandwidth. They work in a stable computing environment. The case is available in most enterprise level computing centers. 11/10/2014, Bing Li
 *
 * Moreover, the multicastor provides an interface to send objects to a bunch of nodes. That is one of the reasons why it is named RootObjectMulticastor. 11/10/2014, Bing Li
 *
 * Finally, this multicastor runs on the root node of the multicasting tree. Each child node has another design. 11/10/2014, Bing Li
 *
 */

// Created: 11/09/2014, Bing Li
public class RootObjectMulticastor<ObjectedData extends Serializable, Message extends ServerMulticastMessage, MessageCreator extends ObjectMulticastCreatable<Message, ObjectedData>> extends HashFreeObject
{
   // Declare an instance of FreeClient pool, which is used to manage instances of FreeClient to send messages to relevant nodes. 11/10/2014, Bing Li
   private FreeClientPool clientPool;
   // Declare the root node's branch count, which can be understood as the capacity of the root node to send messages concurrently. 11/10/2014, Bing Li
   private int rootBranchCount;
   // Declare the other node's branch count, which can be understood as the capacity of the nodes to send messages concurrently. In this case,  11/10/2014, Bing Li
   private int treeBranchCount;
   // The creator to generate multicast messages. 11/10/2014, Bing Li
   private MessageCreator messageCreator;

   /*
    * Initialize the multicastor. It is noted that the pool to manage instances of FreeClient is from the outside of the multicastor. It denotes that the multicastor shares the pool with others. 11/10/2014, Bing Li
    */
   public RootObjectMulticastor(FreeClientPool clientPool, int rootBranchCount, int treeBranchCount, MessageCreator messageCreator)
   {
      this.clientPool = clientPool;
      this.rootBranchCount = rootBranchCount;
      this.treeBranchCount = treeBranchCount;
      this.messageCreator = messageCreator;
   }
```

```
  /*
   * Since all of the resources are shared with others, it is unreasonable to dispose them inside the
multicastor. Just leave the method for future revisions. 11/10/2014, Bing li
   */
  public void dispose()
  {
  }

  /*
   * Disseminate the instance of ObjectedData. 11/10/2014, Bing Li
   */
  public void disseminate(ObjectedData obj) throws InstantiationException, IllegalAccessException,
IOException, InterruptedException
  {
    // Declare a message to contain the instance of ObjectedData, object. 11/10/2014, Bing Li
    Message msg;
    // Declare a tree to support the high efficient multicasting. 11/10/2014, Bing Li
    Map<String, List<String>> tree;
    // Declare a list to take children keys. 11/10/2014, Bing Li
    List<String> allChildrenKeys;
    // Declare a map to take remote nodes' IPs. 11/10/2014, Bing Li
    HashMap<String, String> remoteNodeIPs;
    // An integer to keep the new parent node index. 11/10/2014, Bing Li
    int newParentNodeIndex;
    // Check whether the FreeClient pool has the count of nodes than the root capacity. 11/10/2014,
Bing Li
    if (this.clientPool.getClientSize() > this.rootBranchCount)
    {
      // Construct a tree if the count of nodes is larger than the capacity of the root. Without the tree, the
root node has to send messages concurrently out of its capacity. To lower its load, the tree is required.
All the nodes to received the multicast data are from the FreeClient pool. 11/10/2014, Bing Li
      tree = Tree.constructTree(UtilConfig.ROOT_KEY, new
LinkedList<String>(this.clientPool.getNodeKeys()), this.rootBranchCount, this.treeBranchCount);
      // After the tree is constructed, the root only needs to send messages to its immediate children
only. The loop does that by getting the root's children from the tree and sening the message one by one.
11/10/2014, Bing Li
      for (String childrenKey : tree.get(UtilConfig.ROOT_KEY))
      {
        // Get all of the children keys of the immediate child of the root. 11/10/2014, Bing Li
        allChildrenKeys = Tree.getAllChildrenKeys(tree, childrenKey);
        // Check if the children keys are valid. 11/10/2014, Bing Li
        if (allChildrenKeys != UtilConfig.NO_CHILDREN_KEYS)
        {
          // Initialize a map to keep the IPs of those children nodes of the immediate child of the root.
11/10/2014, Bing Li
          remoteNodeIPs = new HashMap<String, String>();
          // Retrieve the IP of each of the child node of the immediate child of the root and save the IPs
into the map. 11/10/2014, Bing Li
          for (String childrenKeyInTree : allChildrenKeys)
          {
            // Retrieve the IP of a child node of the immediate child of the root and save the IP into the
map. 11/10/2014, Bing Li
            remoteNodeIPs.put(childrenKeyInTree, this.clientPool.getIP(childrenKeyInTree));
          }
          // Create the message to the immediate child of the root. The message is created by enclosing
the object to be sent and the IPs of all of the children nodes of the immediate child of the root.
11/10/2014, Bing Li
          msg = this.messageCreator.createInstanceWithChildren(obj, remoteNodeIPs);
          // Check if the instance of FreeClient of the immediate child of the root is valid and all of the
children keys of the immediate child of the root are not empty. If both of the conditions are true, the loop
continues. 11/10/2014, Bing Li
          while (allChildrenKeys.size() > 0)
          {
            try
            {
              // Send the message to the immediate child of the root. 11/10/2014, Bing Li
              this.clientPool.send(childrenKey, msg);
```

```
                    // Jump out the loop after sending the message successfully. 11/10/2014, Bing Li
                    break;
                  }
                  catch (IOException e)
                  {
                    /*
                     * The exception denotes that the remote end gets something wrong. It is required to
select another immediate child for the root from all of children of the immediate child of the root.
11/10/2014, Bing Li
                     */

                    // Remove the failed client from the pool. 11/10/2014, Bing Li
                    this.clientPool.removeClient(childrenKey);
                    // Select one new node from all of the children of the immediate node of the root.
11/10/2014, Bing Li
                    newParentNodeIndex = Rand.getRandom(allChildrenKeys.size());
                    // Get the new selected node key by its index. 11/10/2014, Bing Li
                    childrenKey = allChildrenKeys.get(newParentNodeIndex);
                    // Remove the newly selected parent node key from the children keys of the immediate
child of the root. 11/11/2014, Bing Li
                    allChildrenKeys.remove(newParentNodeIndex);
                    // Remove the new selected node key from the children's IPs of the immediate node of the
root. 11/10/2014, Bing Li
                    remoteNodeIPs.remove(childrenKey);
                    // Reset the updated the children's IPs in the message to be sent. 11/10/2014, Bing Li
                    msg.setChildrenNodes(remoteNodeIPs);
                  }
                }
              }
              else
              {
                /*
                 * When the line is executed, it indicates that the immediate child of the root has no children.
11/10/2014, Bing Li
                 */

                // If the instance of FreeClient is valid, a message can be created. Different from the above
one, the message does not contain children IPs of the immediate node of the root. 11/10/2014, Bing Li
                msg = this.messageCreator.createInstanceWithoutChildren(obj);
                try
                {
                  // Send the message to the immediate node of the root. 11/10/2014, Bing Li
                  this.clientPool.send(childrenKey, msg);
                }
                catch (IOException e)
                {
                  /*
                   * The exception denotes that the remote end gets something wrong. However, it does not
need to send the message since the immediate node has no children. 11/10/2014, Bing Li
                   */

                  // Remove the instance of FreeClient. 11/10/2014, Bing Li
                  this.clientPool.removeClient(childrenKey);
                }
              }
            }
          }
          else
          {
            /*
             * If the root has sufficient capacity to send the message concurrently, i.e., the root branch count
being greater than that of its immediate children, it is not necessary to construct a tree to lower the load.
11/10/2014, Bing Li
             */

            // Create the message without children's IPs. 11/10/2014, Bing Li
            msg = this.messageCreator.createInstanceWithoutChildren(obj);
            // Send the message one by one to the immediate nodes of the root. 11/10/2014, Bing Li
```

```
        for (String childClientKey : this.clientPool.getNodeKeys())
        {
          try
          {
            // Send the message to the immediate node of the root. 11/10/2014, Bing Li
            this.clientPool.send(childClientKey, msg);
          }
          catch (IOException e)
          {
            /*
             * The exception denotes that the remote end gets something wrong. However, it does not
need to send the message since the immediate node has no children. 11/10/2014, Bing Li
             */

            // Remove the instance of FreeClient. 11/10/2014, Bing Li
            this.clientPool.removeClient(childClientKey);
          }
        }
      }
    }

    /*
     * This method is similar to the above one. The difference is that the children nodes to receive the
multicast data are given by the caller rather than obtaining from the FreeClient pool. 11/10/2014, Bing Li
     */
    public void disseminate(ObjectedData obj, Set<String> clientKeys) throws InstantiationException,
IllegalAccessException, IOException, InterruptedException
    {
      // Check whether the children nodes are valid. 11/10/2014, Bing Li
      if (clientKeys != UtilConfig.NO_NODE_KEYS)
      {
        // Declare a message to contain the instance of ObjectedData, object. 11/10/2014, Bing Li
        Message msg;
        // Declare a tree to support the high efficiency multicasting. 11/10/2014, Bing Li
        Map<String, List<String>> tree;
        // Declare a list to take children keys. 11/10/2014, Bing Li
        List<String> allChildrenKeys;
        // Declare a list to take children keys. 11/10/2014, Bing Li
        HashMap<String, String> remoteNodeIPs;
        // An integer to keep the new parent node index. 11/10/2014, Bing Li
        int newParentNodeIndex;
        // Check whether the FreeClient pool has the count of nodes than the root capacity. 11/10/2014,
Bing Li
        if (clientKeys.size() > this.rootBranchCount)
        {
          // Construct a tree if the count of nodes is larger than the capacity of the root. Without the tree,
the root node has to send messages concurrently out of its capacity. To lower its load, the tree is
required. 11/10/2014, Bing Li
          tree = Tree.constructTree(UtilConfig.ROOT_KEY, new LinkedList<String>(clientKeys),
this.rootBranchCount, this.treeBranchCount);
          // After the tree is constructed, the root only needs to send messages to its immediate children
only. The loop does that by getting the root's children from the tree and sening the message one by one.
11/10/2014, Bing Li
          for (String childrenKey : tree.get(UtilConfig.ROOT_KEY))
          {
            // Get all of the children keys of the immediate child of the root. 11/10/2014, Bing Li
            allChildrenKeys = Tree.getAllChildrenKeys(tree, childrenKey);
            // Check if the children keys are valid. 11/10/2014, Bing Li
            if (allChildrenKeys != UtilConfig.NO_CHILDREN_KEYS)
            {
              // Initialize a map to keep the IPs of those children nodes of the immediate child of the root.
11/10/2014, Bing Li
              remoteNodeIPs = new HashMap<String, String>();
              // Retrieve the IP of each of the child node of the immediate child of the root and save the
IPs into the map. 11/10/2014, Bing Li
              for (String childrenKeyInTree : allChildrenKeys)
              {
                // Retrieve the IP of a child node of the immediate child of the root and save the IP into the
```

```
map. 11/10/2014, Bing Li
                remoteNodeIPs.put(childrenKeyInTree, this.clientPool.getIP(childrenKeyInTree));
              }
              // Create the message to the immediate child of the root. The message is created by
enclosing the object to be sent and the IPs of all of the children nodes of the immediate child of the root.
11/10/2014, Bing Li
              msg = this.messageCreator.createInstanceWithChildren(obj, remoteNodeIPs);
              // Check if the instance of FreeClient of the immediate child of the root is valid and all of the
children keys of the immediate child of the root are not empty. If both of the conditions are true, the loop
continues. 11/10/2014, Bing Li
              while (allChildrenKeys.size() > 0)
              {
                try
                {
                  // Send the message to the immediate child of the root. 11/10/2014, Bing Li
                  this.clientPool.send(childrenKey, msg);
                  // Jump out the loop after sending the message successfully. 11/10/2014, Bing Li
                  break;
                }
                catch (IOException e)
                {
                  /*
                   * The exception denotes that the remote end gets something wrong. It is required to
select another immediate child for the root from all of children of the immediate child of the root.
11/10/2014, Bing Li
                   */

                  // Remove the failed client from the pool. 11/10/2014, Bing Li
                  this.clientPool.removeClient(childrenKey);
                  // Select one new node from all of the children of the immediate node of the root.
11/10/2014, Bing Li
                  newParentNodeIndex = Rand.getRandom(allChildrenKeys.size());
                  // Get the new selected node key by its index. 11/10/2014, Bing Li
                  childrenKey = allChildrenKeys.get(newParentNodeIndex);
                  // Remove the newly selected parent node key from the children keys of the immediate
child of the root. 11/11/2014, Bing Li
                  allChildrenKeys.remove(newParentNodeIndex);
                  // Remove the new selected node key from the children's IPs of the immediate node of
the root. 11/10/2014, Bing Li
                  remoteNodeIPs.remove(childrenKey);
                  // Reset the updated the children's IPs in the message to be sent. 11/10/2014, Bing Li
                  msg.setChildrenNodes(remoteNodeIPs);
                }
              }
            }
            else
            {
              /*
               * When the line is executed, it indicates that the immediate child of the root has no children.
11/10/2014, Bing Li
               */

              // If the instance of FreeClient is valid, a message can be created. Different from the above
one, the message does not contain children IPs of the immediate node of the root. 11/10/2014, Bing Li
              msg = this.messageCreator.createInstanceWithoutChildren(obj);
              try
              {
                // Send the message to the immediate node of the root. 11/10/2014, Bing Li
                this.clientPool.send(childrenKey, msg);
              }
              catch (IOException e)
              {
                /*
                 * The exception denotes that the remote end gets something wrong. However, it does not
need to send the message since the immediate node has no children. 11/10/2014, Bing Li
                 */

                // Remove the instance of FreeClient. 11/10/2014, Bing Li
```

```
                    this.clientPool.removeClient(childrenKey);
                }
            }
        }
    }
    else
    {
        /*
         * If the root has sufficient capacity to send the message concurrently, i.e., the root branch count being greater than that of its immediate children, it is not necessary to construct a tree to lower the load. 11/10/2014, Bing Li
         */

        // Create the message without children's IPs. 11/10/2014, Bing Li
        msg = this.messageCreator.createInstanceWithoutChildren(obj);
        // Send the message one by one to the immediate nodes of the root. 11/10/2014, Bing Li
        for (String clientKey : clientKeys)
        {
            try
            {
                // Send the message to the immediate node of the root. 11/10/2014, Bing Li
                this.clientPool.send(clientKey, msg);
            }
            catch (IOException e)
            {
                /*
                 * The exception denotes that the remote end gets something wrong. However, it does not need to send the message since the immediate node has no children. 11/10/2014, Bing Li
                 */

                // Remove the instance of FreeClient. 11/10/2014, Bing Li
                this.clientPool.removeClient(clientKey);
            }
        }
    }
  }
}
```

- **RootMulticastorSource**

```
package com.greatfree.multicast;

import java.io.Serializable;

import com.greatfree.remote.FreeClientPool;

/*
 * The class contains all of the information that is required to create an instance fo
RootObjectMulticastor. It is needed in the relevant pool. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public class RootMulticastorSource<Source extends Serializable, MultiMessage extends
ServerMulticastMessage, MessageCreator extends ObjectMulticastCreatable<MultiMessage,
Source>>
{
  // The pool for the resources of FreeClient. 11/26/2014, Bing Li
  private FreeClientPool clientPool;
  // The size of the root branch. 11/26/2014, Bing Li
  private int rootBranchCount;
  // The size of the tree branch. 11/26/2014, Bing Li
  private int treeBranchCount;
  // The message creator that generates the multicast messages. 11/26/2014, Bing Li
  private MessageCreator creator;

  /*
   * Initialize the source to provide arguments to create multicastors. 11/26/2014, Bing Li
   */
  public RootMulticastorSource(FreeClientPool clientPool, int rootBranchCount, int treeBranchCount,
MessageCreator creator)
  {
    this.clientPool = clientPool;
    this.rootBranchCount = rootBranchCount;
    this.treeBranchCount = treeBranchCount;
    this.creator = creator;
  }

  /*
   * Expose the client pool. 11/26/2014, Bing Li
   */
  public FreeClientPool getClientPool()
  {
    return this.clientPool;
  }

  /*
   * Expose the root branch count. 11/26/2014, Bing Li
   */
  public int getRootBranchCount()
  {
    return this.rootBranchCount;
  }

  /*
   * Expose the tree branch count. 11/26/2014, Bing Li
   */
  public int getTreeBranchCount()
  {
    return this.treeBranchCount;
  }

  /*
   * Expose the message creator. 11/26/2014, Bing Li
   */
```

```
  public MessageCreator getCreator()
  {
    return this.creator;
  }
}
```

- **RootMessageCreatorGettable**

```
package com.greatfree.multicast;

import java.io.Serializable;


/*
 * The interface defines the method that returns the message creator to generate multicast messages. It
is used to define the instance of multicastor source. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public interface RootMessageCreatorGettable<Message extends ServerMulticastMessage,
ObjectData extends Serializable>
{
   public ObjectMulticastCreatable<Message, ObjectData> getMessageCreator();
}
```

- **ObjectMulticastCreatable**

```
package com.greatfree.multicast;

import java.io.Serializable;
import java.util.HashMap;


/*
 * The interface aims to define the methods to create multicast messages to be sent. 11/10/2014, Bing Li
 */

// Created: 11/10/2014, Bing Li
public interface ObjectMulticastCreatable<Message extends ServerMulticastMessage,
MessagedData extends Serializable>
{
   // The interface to create a multicastor with children information. It denotes the node receiving the
notification needs to forward the message to those children. 11/26/2014, Bing Li
   public Message createInstanceWithChildren(MessagedData message, HashMap<String, String>
childrenMap);
   // The interface to create a multicastor without children information. It represents that the multicasting
is ended in the node who receives the message. 11/26/2014, Bing Li
   public Message createInstanceWithoutChildren(MessagedData message);
}
```

- **ChildMulticastor**

```
package com.greatfree.multicast;

import java.io.IOException;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;

import com.greatfree.remote.FreeClientPool;
import com.greatfree.remote.IPPort;
import com.greatfree.util.HashFreeObject;
import com.greatfree.util.Rand;
import com.greatfree.util.UtilConfig;

/*
 * This is the multicasting code to run on a child node. 11/11/2014, Bing Li
 */

// Created: 11/10/2014, Bing Li
public class ChildMulticastor<Message extends ServerMulticastMessage, MessageCreator extends
ChildMulticastMessageCreatable<Message>> extends HashFreeObject
{
   // Declare an instance of FreeClient pool, which is used to manage instances of FreeClient to send
messages to relevant nodes. 11/11/2014, Bing Li
   private FreeClientPool clientPool;
   // Declare the node's branch count, which can be understood as the capacity of the node to send
messages concurrently. In this case, each node has the same capacity. 11/11/2014, Bing Li
   private int treeBranchCount;
   // Different from RootObjectMulticastor, the root node connects each node when the node is online.
The node within the multicast tree might not have connections to its children. If so, it is required to
provide the port to connect. 11/11/2014, Bing Li
   private int clusterNodePort;
   // The creator to generate multicast messages. 11/11/2014, Bing Li
   private MessageCreator messageCreator;

   /*
    * Initialize the multicastor. It is noted that the pool to manage instances of FreeClient is from the
outside of the multicastor. It denotes that the multicastor shares the pool with others. 11/11/2014, Bing Li
    */
   public ChildMulticastor(FreeClientPool clientPool, int treeBranchCount, int clusterServerPort,
MessageCreator messageCreator)
   {
      this.clientPool = clientPool;
      this.treeBranchCount = treeBranchCount;
      this.clusterNodePort = clusterServerPort;
      this.messageCreator = messageCreator;
   }

   /*
    * Since all of the resources are shared with others, it is unreasonable to dispose them inside the
multicastor. Just leave the method for future revisions. 11/11/2014, Bing li
    */
   public void dispose()
   {
   }

   /*
    * Disseminate the instance of Message. The message here is the one which is just received. It must
be forwarded by the local client. 11/11/2014, Bing Li
    */
   public void disseminate(Message message) throws InstantiationException, IllegalAccessException,
IOException, InterruptedException
   {
      // Declare a message to be forwarded, which contains data in the received message. 11/11/2014,
```

```
Bing Li
    Message forwardMessage;
    // Declare a tree to support the high efficient multicasting. 11/11/2014, Bing Li
    Map<String, List<String>> tree;
    // Declare a list to take children keys. 11/11/2014, Bing Li
    List<String> allChildrenKeys;
    // Declare a map to take remote nodes' IPs. 11/11/2014, Bing Li
    HashMap<String, String> remoteNodeIPs;
    // An integer to keep the new parent node index. 11/11/2014, Bing Li
    int newParentNodeIndex;
    // Check whether the message contains nodes which are waiting for the message. 11/11/2014, Bing
Li
    if (message.getChildrenNodes() != UtilConfig.NO_NODES)
    {
      // If the nodes exist, it needs to check whether the count of the nodes is greater than the branch
count. 11/11/2014, Bing Li
      if (message.getChildrenNodes().size() > this.treeBranchCount)
      {
        // If the count of the nodes is larger than that of the branch, it denotes that the load to forward the
message exceeds the capacity of the local node. Thus, it is required to construct a tree to lower the load.
11/11/2014, Bing Li
        tree = Tree.constructTree(UtilConfig.LOCAL_KEY, new
LinkedList<String>(message.getChildrenNodes().keySet()), this.treeBranchCount);
        // Forward the received message to the local node's immediate children one by one. 11/11/2014,
Bing Li
        for (String childrenKey : tree.get(UtilConfig.LOCAL_KEY))
        {
          // Get all of the children keys of the immediate child of the local node. 11/11/2014, Bing Li
          allChildrenKeys = Tree.getAllChildrenKeys(tree, childrenKey);
          // Check if the children keys are valid. 11/11/2014, Bing Li
          if (allChildrenKeys != UtilConfig.NO_CHILDREN_KEYS)
          {
            // Initialize a map to keep the IPs of those children nodes of the immediate child of the local
node. 11/11/2014, Bing Li
            remoteNodeIPs = new HashMap<String, String>();
            // Retrieve the IP of each of the child node of the immediate child of the local node and save
the IPs into the map. 11/11/2014, Bing Li
            for (String childrenKeyInTree : allChildrenKeys)
            {
              // Retrieve the IP of a child node of the immediate child of the local node and save the IP
into the map. 11/11/2014, Bing Li
              remoteNodeIPs.put(childrenKeyInTree, message.getChildrenNodeIP(childrenKeyInTree));
            }
            // Create the message to the immediate child of the local node. The message is created by
enclosing the object to be sent and the IPs of all of the children nodes of the immediate child of the local
node. 11/11/2014, Bing Li
            forwardMessage = this.messageCreator.createInstanceWithChildren(message,
remoteNodeIPs);
            // Check if the instance of FreeClient of the immediate child of the local node is valid and all
of the children keys of the immediate child of the local node are not empty. If both of the conditions are
true, the loop continues. 11/11/2014, Bing Li
            while (allChildrenKeys.size() > 0)
            {
              try
              {
                // Send the message to the immediate child of the local node. 11/11/2014, Bing Li
                this.clientPool.send(new IPPort(message.getChildrenNodeIP(childrenKey),
this.clusterNodePort), forwardMessage);
                // Jump out the loop after sending the message successfully. 11/11/2014, Bing Li
                break;
              }
              catch (IOException e)
              {
                /*
                 * The exception denotes that the remote end gets something wrong. It is required to
select another immediate child for the local node from all of children of the immediate child of the local
node. 11/11/2014, Bing Li
                 */
```

```
                    // Remove the failed client from the local node. 11/11/2014, Bing Li
                    this.clientPool.removeClient(childrenKey);
                    // Select one new node from all of the children of the immediate node of the local node.
11/11/2014, Bing Li
                    newParentNodeIndex = Rand.getRandom(allChildrenKeys.size());
                    // Get the new selected node key by its index. 11/11/2014, Bing Li
                    childrenKey = allChildrenKeys.get(newParentNodeIndex);
                    // Remove the newly selected parent node key from the children keys of the immediate
child of the local node. 11/11/2014, Bing Li
                    allChildrenKeys.remove(newParentNodeIndex);
                    // Remove the new selected node key from the children's IPs of the immediate node of
the local node. 11/11/2014, Bing Li
                    remoteNodeIPs.remove(childrenKey);
                    // Reset the updated the children's IPs in the message to be forwarded. 11/11/2014,
Bing Li
                    forwardMessage.setChildrenNodes(remoteNodeIPs);
                  }
                }
              }
              else
              {
                /*
                 * When the line is executed, it indicates that the immediate child of the local node has no
children. 11/11/2014, Bing Li
                 */

                // Check whether the instance of FreeClient is valid. 11/11/2014, Bing Li
                try
                {
                  // If the instance of FreeClient is valid, a message can be created. Different from the above
one, the message does not contain children IPs of the immediate node of the local node. 11/11/2014,
Bing Li
                  this.clientPool.send(new IPPort(message.getChildrenNodeIP(childrenKey),
this.clusterNodePort), message);
                }
                catch (IOException e)
                {
                  /*
                   * The exception denotes that the remote end gets something wrong. However, it does not
need to forward the message since the immediate node has no children. 11/11/2014, Bing Li
                   */

                  // Remove the instance of FreeClient. 11/11/2014, Bing Li
                  this.clientPool.removeClient(childrenKey);
                }
              }
            }
          }
          else
          {
            /*
             * If the local node has sufficient capacity to forward the message concurrently, i.e., the tree
branch count being greater than that of its immediate children, it is not necessary to construct a tree to
lower the load. 11/11/2014, Bing Li
             */

            // Create the message without children's IPs. 11/11/2014, Bing Li
            for (Map.Entry<String, String> serverAddressEntry : message.getChildrenNodes().entrySet())
            {
              try
              {
                // Send the message to the immediate node of the local node. 11/11/2014, Bing Li
                this.clientPool.send(new
IPPort(message.getChildrenNodeIP(serverAddressEntry.getValue()), this.clusterNodePort), message);
              }
              catch (IOException e)
              {
```

```
                /*
                 * The exception denotes that the remote end gets something wrong. However, it does not
need to send the message since the immediate node has no children. 11/11/2014, Bing Li
                 */

                // Remove the instance of FreeClient. 11/11/2014, Bing Li
                this.clientPool.removeClient(serverAddressEntry.getKey());
            }
        }
    }
  }
}
```

- **ChildMulticastorSource**

```
package com.greatfree.multicast;

import com.greatfree.remote.FreeClientPool;

/*
 * The class contains all of the data to create an instance of ChildMulticastor. That is why it is named ChildMulticastorSource. It is used by the resource pool to manage resources efficiently. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class ChildMulticastorSource<MultiMessage extends ServerMulticastMessage, MessageCreator extends ChildMulticastMessageCreatable<MultiMessage>>
{
  // The pool for the resources of FreeClient. 11/27/2014, Bing Li
  private FreeClientPool clientPool;
  // The size of the tree branch. 11/27/2014, Bing Li
  private int treeBranchCount;
  // The port number of the children nodes. 11/27/2014, Bing Li
  private int serverPort;
  // The message creator that generates the multicast messages. 11/27/2014, Bing Li
  private MessageCreator creator;

  /*
   * Initialize the source to provide arguments to create multicastors. 11/27/2014, Bing Li
   */
  public ChildMulticastorSource(FreeClientPool clientPool, int treeBranchCount, int serverPort, MessageCreator creator)
  {
    this.clientPool = clientPool;
    this.treeBranchCount = treeBranchCount;
    this.serverPort = serverPort;
    this.creator = creator;
  }

  /*
   * Expose the client pool. 11/27/2014, Bing Li
   */
  public FreeClientPool getClientPool()
  {
    return this.clientPool;
  }

  /*
   * Expose the tree branch count. 11/27/2014, Bing Li
   */
  public int getTreeBranchCount()
  {
    return this.treeBranchCount;
  }

  /*
   * Expose the children server port number. 11/27/2014, Bing Li
   */
  public int getServerPort()
  {
    return this.serverPort;
  }

  /*
   * Expose the message creator. 11/27/2014, Bing Li
   */
  public MessageCreator getMessageCreator()
  {
```

```
        return this.creator;
    }
}
```

- **ChildMulticastMessageCreatable**

```
package com.greatfree.multicast;

import java.util.HashMap;

/*
 * The interface defines the method to create a multicast message on a child node rather than the root
one. 11/10/2014, Bing Li
 */

// Created: 11/10/2014, Bing Li
public interface ChildMulticastMessageCreatable<Message extends ServerMulticastMessage>
{
   public Message createInstanceWithChildren(Message msg, HashMap<String, String> children);
}
```

- **ChildMessageCreatorGettable**

```
package com.greatfree.multicast;

/*
 * The interface defines the method that returns the message creator to generate multicast messages to children. It is used to define the instance of children multicastor source. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public interface ChildMessageCreatorGettable<Message extends ServerMulticastMessage>
{
   public ChildMulticastMessageCreatable<Message> getMessageCreator();
}
```

- **RootRequestBroadcastor**

```
package com.greatfree.multicast;

import java.io.IOException;
import java.io.Serializable;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;

import com.greatfree.concurrency.Collaborator;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.util.HashFreeObject;
import com.greatfree.util.Rand;
import com.greatfree.util.UtilConfig;

/*
 * This is the implementation to send a broadcast request to all of the nodes in a particular cluster to retrieve data on each of them. It is also required to collect the results and then form a response to return the root. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class RootRequestBroadcastor<MessagedRequestData extends Serializable, Request extends BroadcastRequest, Response extends BroadcastResponse, RequestCreator extends RootBroadcastRequestCreatable<Request, MessagedRequestData>> extends HashFreeObject
{
  // A TCP client pool to interact with remote nodes. 11/28/2014, Bing Li
  private FreeClientPool clientPool;
  // The root node's branch count, which can be understood as the capacity of the root node to send messages concurrently. The root is the broadcast request initial sender. 11/28/2014, Bing Li
  private int rootBranchCount;
  // The other node's branch count, which can be understood as the capacity of the nodes to send messages concurrently. In this case,  11//2014, Bing Li
  private int treeBranchCount;
  // The creator to generate broadcast requests. 11/28/2014, Bing Li
  private RequestCreator requestCreator;
  // All of the received responses from each of the node which is retrieved. 11/28/2014, Bing Li
  private Map<String, Response> responseMap;
  // The collaborator that synchronizes to collect the results and determine whether the request is responded sufficiently. 11/28/2014, Bing Li
  private Collaborator collaborator;
  // The time to wait for responses. If it lasts too long, it might get problems for the request processing. 11/28/2014, Bing Li
  private long waitTime;
  // The count of the total nodes in the broadcast. 11/28/2014, Bing Li
  private int nodeCount;

  /*
   * Initialize. 11/28/2014, Bing Li
   */
  public RootRequestBroadcastor(FreeClientPool clientPool, int rootBranchCount, int treeBranchCount, RequestCreator requestCreator, long waitTime)
  {
    this.clientPool = clientPool;
    this.rootBranchCount = rootBranchCount;
    this.treeBranchCount = treeBranchCount;
    this.requestCreator = requestCreator;
    this.responseMap = new ConcurrentHashMap<String, Response>();
    this.collaborator = new Collaborator();
    this.waitTime = waitTime;
  }
```

```
  /*
   * Dispose the broadcast requestor. 11/28/2014, Bing Li
   */
  public void dispose()
  {
    this.collaborator.setShutdown();
    this.collaborator.signalAll();
    this.responseMap.clear();
    this.responseMap = null;
  }

  /*
   * To reuse the broadcast to take another broadcast request, it must reset. 11/28/2014, Bing Li
   */
  public synchronized String resetBroadcast()
  {
    // Clear the existing responses. 11/28/2014, Bing Li
    this.responseMap.clear();
    // Reset the collaborator key and return it. The key is critical for the broadcast request to be
responded because it determines the procedure of synchronization during the procedure. 11/28/2014,
Bing Li
    return this.collaborator.resetKey();
  }

  /*
   * Save one particular response from the remote node. 11/28/2014, Bing Li
   */
  public void saveResponse(Response response)
  {
    // Check whether the response corresponds to the requestor. 11/29/2014, Bing Li
    if (this.collaborator.getKey().equals(response.getCollaboratorKey()))
    {
      // Put it into the collection. 11/28/2014, Bing Li
      this.responseMap.put(response.getKey(), response);
      // Check whether the count of the collected response exceeds the total node count. 11/28/2014,
Bing Li
      if (this.responseMap.size() >= this.nodeCount)
      {
        // If it does, it denotes that the request has been answered by all of the nodes. Notify the waiting
collaborator to end the request broadcast. 11/28/2014, Bing Li
        this.collaborator.signal();
      }
    }
  }

  /*
   * Send the request to the cluster of nodes and return the responses. 11/28/2014, Bing Li
   */
  public Map<String, Response> disseminate(MessagedRequestData messagedData) throws
InterruptedException, InstantiationException, IllegalAccessException, IOException
  {
    // The initial request to be sent. 11/28/2014, Bing Li
    Request requestToBeSent;
    // Declare a tree to support the high efficient multicasting. 11/28/2014, Bing Li
    Map<String, List<String>> tree;
    // Declare a list to take children keys. 11/28/2014, Bing Li
    List<String> allChildrenKeys;
    // Declare a map to take remote nodes' IPs. 11/28/2014, Bing Li
    HashMap<String, String> remoteServerIPs;
    // An integer to keep the new parent node index. 11/28/2014, Bing Li
    int newParentNodeIndex;
    // Keep the count of nodes that must respond the request. 11/28/2014, Bing Li
    this.nodeCount = this.clientPool.getClientSize();
    // Check whether the FreeClient pool has the count of nodes than the root capacity. 11/28/2014,
Bing Li
    if (this.clientPool.getClientSize() > this.rootBranchCount)
    {
      // Construct a tree if the count of nodes is larger than the capacity of the root. Without the tree, the
```

```
root node has to send requests concurrently out of its capacity. To lower its load, the tree is required. All the nodes to received the multicast data are from the FreeClient pool. 11/28/2014, Bing Li
        tree = Tree.constructTree(UtilConfig.ROOT_KEY, new LinkedList<String>(this.clientPool.getNodeKeys()), this.rootBranchCount, this.treeBranchCount);
        // After the tree is constructed, the root only needs to send requests to its immediate children only. The loop does that by getting the root's children from the tree and sending the request one by one. 11/28/2014, Bing Li
        for (String childrenKey : tree.get(UtilConfig.ROOT_KEY))
        {
          // Get all of the children keys of the immediate child of the root. 11/28/2014, Bing Li
          allChildrenKeys = Tree.getAllChildrenKeys(tree, childrenKey);
          // Check if the children keys are valid. 11/28/2014, Bing Li
          if (allChildrenKeys != UtilConfig.NO_CHILDREN_KEYS)
          {
            // Initialize a map to keep the IPs of those children nodes of the immediate child of the root. 11/28/2014, Bing Li
            remoteServerIPs = new HashMap<String, String>();
            // Retrieve the IP of each of the child node of the immediate child of the root and save the IPs into the map. 11/28/2014, Bing Li
            for (String childrenKeyInTree : allChildrenKeys)
            {
              // Retrieve the IP of a child node of the immediate child of the root and save the IP into the map. 11/28/2014, Bing Li
              remoteServerIPs.put(childrenKeyInTree, this.clientPool.getIP(childrenKeyInTree));
            }
            // Create the request to the immediate child of the root. The request is created by enclosing the object to be sent, the collaborator key and the IPs of all of the children nodes of the immediate child of the root. 11/28/2014, Bing Li
            requestToBeSent = this.requestCreator.createInstanceWithChildren(messagedData, this.collaborator.getKey(), remoteServerIPs);
            // Check if the instance of FreeClient of the immediate child of the root is valid and all of the children keys of the immediate child of the root are not empty. If both of the conditions are true, the loop continues. 11/28/2014, Bing Li
            while (allChildrenKeys.size() > 0)
            {
              try
              {
                // Send the request to the immediate child of the root. 11/28/2014, Bing Li
                this.clientPool.send(childrenKey, requestToBeSent);
                // Jump out the loop after sending the request successfully. 11/28/2014, Bing Li
                break;
              }
              catch (IOException e)
              {
                /*
                 * The exception denotes that the remote end gets something wrong. It is required to select another immediate child for the root from all of children of the immediate child of the root. 11/28/2014, Bing Li
                 */
                // Remove the failed child key. 11/28/2014, Bing Li
                this.clientPool.removeClient(childrenKey);
                // Select one new node from all of the children of the immediate node of the root. 11/28/2014, Bing Li
                newParentNodeIndex = Rand.getRandom(allChildrenKeys.size());
                // Get the new selected node key by its index. 11/28/2014, Bing Li
                childrenKey = allChildrenKeys.get(newParentNodeIndex);
                // Remove the newly selected parent node key from the children keys of the immediate child of the root. 11/11/2014, Bing Li
                allChildrenKeys.remove(newParentNodeIndex);
                // Remove the new selected node key from the children's IPs of the immediate node of the root. 11/28/2014, Bing Li
                remoteServerIPs.remove(childrenKey);
                // Reset the updated the children's IPs in the message to be sent. 11/28/2014, Bing Li
                requestToBeSent.setChildrenNodes(remoteServerIPs);
                // The count must be decremented for the failed node. 11/28/2014, Bing Li
                this.nodeCount--;
              }
            }
```

```
            }
            else
            {
               /*
                * When the line is executed, it indicates that the immediate child of the root has no children.
11/28/2014, Bing Li
                */

               // If the instance of FreeClient is valid, a message can be created. Different from the above
one, the message does not contain children IPs of the immediate node of the root. 11/28/2014, Bing Li
               requestToBeSent = this.requestCreator.createInstanceWithoutChildren(messagedData,
this.collaborator.getKey());
               try
               {
                  // Send the request to the immediate node of the root. 11/28/2014, Bing Li
                  this.clientPool.send(childrenKey, requestToBeSent);
               }
               catch (IOException e)
               {
                  /*
                   * The exception denotes that the remote end gets something wrong. However, it does not
need to send the message since the immediate node has no children. 11/28/2014, Bing Li
                   */

                  // Remove the instance of FreeClient. 11/28/2014, Bing Li
                  this.clientPool.removeClient(childrenKey);
                  // The count must be decremented for the failed node. 11/28/2014, Bing Li
                  this.nodeCount--;
               }
            }
         }
      }
      else
      {
         /*
          * If the root has sufficient capacity to send the request concurrently, i.e., the root branch count
being greater than that of its immediate children, it is not necessary to construct a tree to lower the load.
11/28/2014, Bing Li
          */

         // Create the request without children's IPs. 11/28/2014, Bing Li
         requestToBeSent = this.requestCreator.createInstanceWithoutChildren(messagedData,
this.collaborator.getKey());
         // Send the request one by one to the immediate nodes of the root. 11/28/2014, Bing Li
         for (String childClientKey : this.clientPool.getNodeKeys())
         {
            try
            {
               // Send the request to the immediate node of the root. 11/28/2014, Bing Li
               this.clientPool.send(childClientKey, requestToBeSent);
            }
            catch (IOException e)
            {
               /*
                * The exception denotes that the remote end gets something wrong. However, it does not
need to send the message since the immediate node has no children. 11/28/2014, Bing Li
                */

               // Remove the instance of FreeClient. 11/28/2014, Bing Li
               this.clientPool.removeClient(childClientKey);
               // The count must be decremented for the failed node. 11/28/2014, Bing Li
               this.nodeCount--;
            }
         }
      }
      // The requesting procedure is blocked until all of the responses are received or it has waited for
sufficiently long time. 11/28/2014, Bing Li
      this.collaborator.holdOn(this.waitTime);
```

```
    // Return the response collection. 11/28/2014, Bing Li
    return this.responseMap;
  }

  /*
   * Send the request to the cluster of nodes. Different from the above method, the one specifies the
exact node keys which must respond. The above one assumes that all of the nodes in the client pool
must respond. 11/28/2014, Bing Li
   */
  public Map<String, Response> disseminate(Set<String> clientKeys, MessagedRequestData
messagedData) throws InterruptedException, InstantiationException, IllegalAccessException,
IOException
  {
    // The initial request to be sent. 11/28/2014, Bing Li
    Request requestToBeSent;
    // Declare a tree to support the high efficient multicasting. 11/28/2014, Bing Li
    Map<String, List<String>> tree;
    // Declare a list to take children keys. 11/28/2014, Bing Li
    List<String> allChildrenKeys;
    // Declare a map to take remote nodes' IPs. 11/28/2014, Bing Li
    HashMap<String, String> remoteServerIPs;
    // An integer to keep the new parent node index. 11/28/2014, Bing Li
    int newParentNodeIndex;
    // Keep the count of nodes that must respond the request. 11/28/2014, Bing Li
    this.nodeCount = clientKeys.size();
    // Check whether the FreeClient pool has the count of nodes than the root capacity. 11/28/2014,
Bing Li
    if (this.nodeCount > this.rootBranchCount)
    {
      // Construct a tree if the count of nodes is larger than the capacity of the root. Without the tree, the
root node has to send requests concurrently out of its capacity. To lower its load, the tree is required. All
the nodes to received the multicast data are from the FreeClient pool. 11/28/2014, Bing Li
      tree = Tree.constructTree(UtilConfig.ROOT_KEY, new LinkedList<String>(clientKeys),
this.rootBranchCount, this.treeBranchCount);
      // After the tree is constructed, the root only needs to send requests to its immediate children only.
The loop does that by getting the root's children from the tree and sending the request one by one.
11/28/2014, Bing Li
      for (String childrenKey : tree.get(UtilConfig.ROOT_KEY))
      {
        // Get all of the children keys of the immediate child of the root. 11/28/2014, Bing Li
        allChildrenKeys = Tree.getAllChildrenKeys(tree, childrenKey);
        // Check if the children keys are valid. 11/28/2014, Bing Li
        if (allChildrenKeys != UtilConfig.NO_CHILDREN_KEYS)
        {
          // Initialize a map to keep the IPs of those children nodes of the immediate child of the root.
11/28/2014, Bing Li
          remoteServerIPs = new HashMap<String, String>();
          // Retrieve the IP of each of the child node of the immediate child of the root and save the IPs
into the map. 11/28/2014, Bing Li
          for (String childrenKeyInTree : allChildrenKeys)
          {
            // Retrieve the IP of a child node of the immediate child of the root and save the IP into the
map. 11/28/2014, Bing Li
            remoteServerIPs.put(childrenKeyInTree, this.clientPool.getIP(childrenKeyInTree));
          }
          // Create the request to the immediate child of the root. The request is created by enclosing
the object to be sent, the collaborator key and the IPs of all of the children nodes of the immediate child
of the root. 11/28/2014, Bing Li
          requestToBeSent = this.requestCreator.createInstanceWithChildren(messagedData,
this.collaborator.getKey(), remoteServerIPs);
          // Check if the instance of FreeClient of the immediate child of the root is valid and all of the
children keys of the immediate child of the root are not empty. If both of the conditions are true, the loop
continues. 11/28/2014, Bing Li
          while (allChildrenKeys.size() > 0)
          {
            try
            {
              // Send the request to the immediate child of the root. 11/28/2014, Bing Li
```

```
                    this.clientPool.send(childrenKey, requestToBeSent);
                    // Jump out the loop after sending the request successfully. 11/28/2014, Bing Li
                    break;
                  }
                  catch (IOException e)
                  {
                    /*
                     * The exception denotes that the remote end gets something wrong. It is required to
select another immediate child for the root from all of children of the immediate child of the root.
11/28/2014, Bing Li
                     */
                    // Remove the failed child key. 11/28/2014, Bing Li
                    this.clientPool.removeClient(childrenKey);
                    // Select one new node from all of the children of the immediate node of the root.
11/28/2014, Bing Li
                    newParentNodeIndex = Rand.getRandom(allChildrenKeys.size());
                    // Get the new selected node key by its index. 11/28/2014, Bing Li
                    childrenKey = allChildrenKeys.get(newParentNodeIndex);
                    // Remove the newly selected parent node key from the children keys of the immediate
child of the root. 11/11/2014, Bing Li
                    allChildrenKeys.remove(newParentNodeIndex);
                    // Remove the new selected node key from the children's IPs of the immediate node of the
root. 11/28/2014, Bing Li
                    remoteServerIPs.remove(childrenKey);
                    // Reset the updated the children's IPs in the message to be sent. 11/28/2014, Bing Li
                    requestToBeSent.setChildrenNodes(remoteServerIPs);
                    // The count must be decremented for the failed node. 11/28/2014, Bing Li
                    this.nodeCount--;
                  }
                }
              }
              else
              {
                /*
                 * When the line is executed, it indicates that the immediate child of the root has no children.
11/28/2014, Bing Li
                 */

                // If the instance of FreeClient is valid, a message can be created. Different from the above
one, the message does not contain children IPs of the immediate node of the root. 11/28/2014, Bing Li
                requestToBeSent = this.requestCreator.createInstanceWithoutChildren(messagedData,
this.collaborator.getKey());
                try
                {
                  // Send the request to the immediate node of the root. 11/28/2014, Bing Li
                  this.clientPool.send(childrenKey, requestToBeSent);
                }
                catch (IOException e)
                {
                  /*
                   * The exception denotes that the remote end gets something wrong. However, it does not
need to send the message since the immediate node has no children. 11/28/2014, Bing Li
                   */

                  // Remove the instance of FreeClient. 11/28/2014, Bing Li
                  this.clientPool.removeClient(childrenKey);
                  // The count must be decremented for the failed node. 11/28/2014, Bing Li
                  this.nodeCount--;
                }
              }
            }
          }
          else
          {
            /*
             * If the root has sufficient capacity to send the request concurrently, i.e., the root branch count
being greater than that of its immediate children, it is not necessary to construct a tree to lower the load.
11/28/2014, Bing Li
```

```
         */

        // Create the request without children's IPs. 11/28/2014, Bing Li
        requestToBeSent = this.requestCreator.createInstanceWithoutChildren(messagedData,
this.collaborator.getKey());
        // Send the request one by one to the immediate nodes of the root. 11/28/2014, Bing Li
        for (String childClientKey : clientKeys)
        {
          try
          {
            // Send the request to the immediate node of the root. 11/28/2014, Bing Li
            this.clientPool.send(childClientKey, requestToBeSent);
          }
          catch (IOException e)
          {
            /*
             * The exception denotes that the remote end gets something wrong. However, it does not
need to send the message since the immediate node has no children. 11/28/2014, Bing Li
             */

            // Remove the instance of FreeClient. 11/28/2014, Bing Li
            this.clientPool.removeClient(childClientKey);
            // The count must be decremented for the failed node. 11/28/2014, Bing Li
            this.nodeCount--;
          }
        }
      }
      // The requesting procedure is blocked until all of the responses are received or it has waited for
sufficiently long time. 11/28/2014, Bing Li
      this.collaborator.holdOn(this.waitTime);
      // Return the response collection. 11/28/2014, Bing Li
      return this.responseMap;
    }
}
```

- **RootBroadcastReaderSource**

```
package com.greatfree.multicast;

import java.io.Serializable;

import com.greatfree.remote.FreeClientPool;

/*
 * This class assists the resource pool to create instances of broadcast readers. Thus, it contains all of
required arguments to do that. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class RootBroadcastReaderSource<Source extends Serializable, BroadcastRequestMessage
extends BroadcastRequest, MessageCreator extends
RootBroadcastRequestCreatable<BroadcastRequestMessage, Source>>
{
  // A TCP client pool to interact with remote nodes. 11/29/2014, Bing Li
  private FreeClientPool clientPool;
  // The root node's branch count, which can be understood as the capacity of the root node to send
messages concurrently. The root is the broadcast request initial sender. 11/29/2014, Bing Li
  private int rootBranchCount;
  // The other node's branch count, which can be understood as the capacity of the nodes to send
messages concurrently. In this case,  11/29/2014, Bing Li
  private int treeBranchCount;
  // The creator to generate anycast requests. 11/29/2014, Bing Li
  private MessageCreator creator;

  /*
   * Initialize. 11/29/2014, Bing Li
   */
  public RootBroadcastReaderSource(FreeClientPool clientPool, int rootBranchCount, int
treeBranchCount, MessageCreator creator)
  {
    this.clientPool = clientPool;
    this.rootBranchCount = rootBranchCount;
    this.treeBranchCount = treeBranchCount;
    this.creator = creator;
  }

  /*
   * Expose the client pool. 11/29/2014, Bing Li
   */
  public FreeClientPool getClientPool()
  {
    return this.clientPool;
  }

  /*
   * Expose the root branch count. 11/29/2014, Bing Li
   */
  public int getRootBranchCount()
  {
    return this.rootBranchCount;
  }

  /*
   * Expose the tree branch count. 11/29/2014, Bing Li
   */
  public int getTreeBranchCount()
  {
    return this.treeBranchCount;
  }

  /*
```

```
    * Expose the request creator. 11/29/2014, Bing Li
    */
   public MessageCreator getCreator()
   {
      return this.creator;
   }
}
```

- **RootBroadcastRequestCreatable**

```
package com.greatfree.multicast;

import java.io.Serializable;
import java.util.HashMap;

/*
 * The interface defines the methods to create request in the broadcast requestor. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public interface RootBroadcastRequestCreatable<Request extends BroadcastRequest,
ObjectedData extends Serializable>
{
   // Create the request for the requestor that have children nodes. 11/29/2014, Bing Li
   public Request createInstanceWithChildren(ObjectedData t, String collaboratorKey,
HashMap<String, String> childrenMap);
   // Create the request for the requestor that have no children nodes. 11/29/2014, Bing Li
   public Request createInstanceWithoutChildren(ObjectedData t, String collaboratorKey);
}
```

- **RootBroadcastRequestCreatorGettable**

```
package com.greatfree.multicast;

import java.io.Serializable;

/*
 * The interface returns the broadcast request creator. It is used to constrain the source to provide the method for the resource pool to manage broadcastors. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public interface RootBroadcastRequestCreatorGettable<Request extends BroadcastRequest, MessagedData extends Serializable>
{
   public RootBroadcastRequestCreatable<Request, MessagedData> getRequestCreator();
}
```

- **RootRequestAnycastor**

```
package com.greatfree.multicast;

import java.io.IOException;
import java.io.Serializable;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;

import com.greatfree.concurrency.Collaborator;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.util.HashFreeObject;
import com.greatfree.util.Rand;
import com.greatfree.util.UtilConfig;

/*
 * This is the implementation to send an anycast request to all of the nodes in a particular cluster to retrieve data on each of them. It is also required to collect the results and then form a response to return the root. However, only one response is good enough. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class RootRequestAnycastor<MessagedData extends Serializable, Request extends AnycastRequest, Response extends AnycastResponse, RequestCreator extends RootAnycastRequestCreatable<Request, MessagedData>> extends HashFreeObject
{
  // A TCP client pool to interact with remote nodes. 11/29/2014, Bing Li
  private FreeClientPool clientPool;
  // The root node's branch count, which can be understood as the capacity of the root node to send messages concurrently. The root is the broadcast request initial sender. 11/29/2014, Bing Li
  private int rootBranchCount;
  // The other node's branch count, which can be understood as the capacity of the nodes to send messages concurrently. In this case,  11/29/2014, Bing Li
  private int treeBranchCount;
  // The creator to generate anycast requests. 11/29/2014, Bing Li
  private RequestCreator requestCreator;
  // The collaborator that synchronizes to collect the results and determine whether the request is responded sufficiently. 11/29/2014, Bing Li
  private Collaborator collaborator;
  // The time to wait for responses. If it lasts too long, it might get problems for the request processing. 11/29/2014, Bing Li
  private long waitTime;
  // Since one response is enough, only one is declared here. 11/29/2014, Bing Li
  private Response response;

  /*
   * Initialize. 11/29/2014, Bing Li
   */
  public RootRequestAnycastor(FreeClientPool clientPool, int rootBranchCount, int treeBranchCount, RequestCreator requestCreator, long waitTime)
  {
    this.clientPool = clientPool;
    this.rootBranchCount = rootBranchCount;
    this.treeBranchCount = treeBranchCount;
    this.requestCreator = requestCreator;
    this.collaborator = new Collaborator();
    this.waitTime = waitTime;
    this.response = null;
  }

  /*
   * Dispose the broadcast requestor. 11/29/2014, Bing Li
   */
  public void dispose()
```

```
  {
    this.collaborator.setShutdown();
    this.collaborator.signalAll();
  }

  /*
   * To reuse the broadcast to take another anycast request, it must reset. 11/29/2014, Bing Li
   */
  public synchronized String resetAnycast()
  {
    // Clear the response. 11/29/2014, Bing Li
    this.response = null;
    // Reset the collaborator key and return it. The key is critical for the anycast request to be responded
because it determines the procedure of synchronization during the procedure. 11/29/2014, Bing Li
    return this.collaborator.resetKey();
  }

  /*
   * Save one particular response from the remote node. 11/29/2014, Bing Li
   */
  public void notify(Response response)
  {
    // Check whether the response corresponds to the requestor. 11/29/2014, Bing Li
    if (this.collaborator.getKey().equals(response.getCollaboratorKey()))
    {
      // Assign the response. 11/29/2014, Bing Li
      this.response = response;
      // Notify the waiting collaborator to end the request broadcast. 11/29/2014, Bing Li
      this.collaborator.signal();
    }
  }

  /*
   * Send the request to the cluster of nodes and return the response. 11/29/2014, Bing Li
   */
  public Response disseminate(MessagedData messagedData) throws InterruptedException,
InstantiationException, IllegalAccessException, IOException
  {
    // The initial request to be sent. 11/29/2014, Bing Li
    Request requestToBeSent;
    // Declare a tree to support the high efficient multicasting. 11/29/2014, Bing Li
    Map<String, List<String>> tree;
    // Declare a list to take children keys. 11/29/2014, Bing Li
    List<String> allChildrenKeys;
    // Declare a map to take remote nodes' IPs. 11/29/2014, Bing Li
    HashMap<String, String> remoteServerIPs;
    // An integer to keep the new parent node index. 11/29/2014, Bing Li
    int newParentNodeIndex;
    // Check whether the FreeClient pool has the count of nodes than the root capacity. 11/29/2014,
Bing Li
    if (this.clientPool.getClientSize() > this.rootBranchCount)
    {
      // Construct a tree if the count of nodes is larger than the capacity of the root. Without the tree, the
root node has to send requests concurrently out of its capacity. To lower its load, the tree is required. All
the nodes to received the multicast data are from the FreeClient pool. 11/29/2014, Bing Li
      tree = Tree.constructTree(UtilConfig.ROOT_KEY, new
LinkedList<String>(this.clientPool.getNodeKeys()), this.rootBranchCount, this.treeBranchCount);
      // After the tree is constructed, the root only needs to send requests to its immediate children only.
The loop does that by getting the root's children from the tree and sending the request one by one.
11/29/2014, Bing Li
      for (String childrenKey : tree.get(UtilConfig.ROOT_KEY))
      {
        // Get all of the children keys of the immediate child of the root. 11/29/2014, Bing Li
        allChildrenKeys = Tree.getAllChildrenKeys(tree, childrenKey);
        // Check if the children keys are valid. 11/29/2014, Bing Li
        if (allChildrenKeys != UtilConfig.NO_CHILDREN_KEYS)
        {
          // Initialize a map to keep the IPs of those children nodes of the immediate child of the root.
```

```
11/29/2014, Bing Li
            remoteServerIPs = new HashMap<String, String>();
            // Retrieve the IP of each of the child node of the immediate child of the root and save the IPs
into the map. 11/29/2014, Bing Li
            for (String childrenKeyInTree : allChildrenKeys)
            {
              // Retrieve the IP of a child node of the immediate child of the root and save the IP into the
map. 11/29/2014, Bing Li
              remoteServerIPs.put(childrenKeyInTree, this.clientPool.getIP(childrenKeyInTree));
            }
            // Create the request to the immediate child of the root. The request is created by enclosing
the object to be sent, the collaborator key and the IPs of all of the children nodes of the immediate child
of the root. 11/29/2014, Bing Li
            requestToBeSent = this.requestCreator.createInstanceWithChildren(messagedData,
this.collaborator.getKey(), remoteServerIPs);
            // Check if the instance of FreeClient of the immediate child of the root is valid and all of the
children keys of the immediate child of the root are not empty. If both of the conditions are true, the loop
continues. 11/29/2014, Bing Li
            while (allChildrenKeys.size() > 0)
            {
              try
              {
                // Send the request to the immediate child of the root. 11/29/2014, Bing Li
                this.clientPool.send(childrenKey, requestToBeSent);
                // Jump out the loop after sending the request successfully. 11/29/2014, Bing Li
                break;
              }
              catch (IOException e)
              {
                /*
                 * The exception denotes that the remote end gets something wrong. It is required to
select another immediate child for the root from all of children of the immediate child of the root.
11/29/2014, Bing Li
                 */
                // Remove the failed child key. 11/29/2014, Bing Li
                this.clientPool.removeClient(childrenKey);
                // Select one new node from all of the children of the immediate node of the root.
11/29/2014, Bing Li
                newParentNodeIndex = Rand.getRandom(allChildrenKeys.size());
                // Get the new selected node key by its index. 11/29/2014, Bing Li
                childrenKey = allChildrenKeys.get(newParentNodeIndex);
                // Remove the newly selected parent node key from the children keys of the immediate
child of the root. 11/11/2014, Bing Li
                allChildrenKeys.remove(newParentNodeIndex);
                // Remove the new selected node key from the children's IPs of the immediate node of the
root. 11/29/2014, Bing Li
                remoteServerIPs.remove(childrenKey);
                // Reset the updated the children's IPs in the message to be sent. 11/29/2014, Bing Li
                requestToBeSent.setChildrenNodes(remoteServerIPs);
              }
            }
          }
          else
          {
            /*
             * When the line is executed, it indicates that the immediate child of the root has no children.
11/29/2014, Bing Li
             */

            // If the instance of FreeClient is valid, a message can be created. Different from the above
one, the message does not contain children IPs of the immediate node of the root. 11/29/2014, Bing Li
            requestToBeSent = this.requestCreator.createInstanceWithoutChildren(messagedData,
this.collaborator.getKey());
            try
            {
              // Send the request to the immediate node of the root. 11/29/2014, Bing Li
              this.clientPool.send(childrenKey, requestToBeSent);
            }
```

```
            catch (IOException e)
            {
              /*
               * The exception denotes that the remote end gets something wrong. However, it does not
need to send the message since the immediate node has no children. 11/29/2014, Bing Li
               */

              // Remove the instance of FreeClient. 11/29/2014, Bing Li
              this.clientPool.removeClient(childrenKey);
            }
          }
        }
      }
      else
      {
        /*
         * If the root has sufficient capacity to send the request concurrently, i.e., the root branch count
being greater than that of its immediate children, it is not necessary to construct a tree to lower the load.
11/29/2014, Bing Li
         */

        // Create the request without children's IPs. 11/29/2014, Bing Li
        requestToBeSent = this.requestCreator.createInstanceWithoutChildren(messagedData,
this.collaborator.getKey());
        // Send the request one by one to the immediate nodes of the root. 11/29/2014, Bing Li
        for (String childClientKey : this.clientPool.getNodeKeys())
        {
          try
          {
            // Send the request to the immediate node of the root. 11/29/2014, Bing Li
            this.clientPool.send(childClientKey, requestToBeSent);
          }
          catch (IOException e)
          {
            /*
             * The exception denotes that the remote end gets something wrong. However, it does not
need to send the message since the immediate node has no children. 11/29/2014, Bing Li
             */

            // Remove the instance of FreeClient. 11/29/2014, Bing Li
            this.clientPool.removeClient(childClientKey);
          }
        }
      }
      // The requesting procedure is blocked until at least one response is received or it has waited for
sufficiently long time. 11/29/2014, Bing Li
      this.collaborator.holdOn(this.waitTime);
      // Return the response collection. 11/29/2014, Bing Li
      return this.response;
    }
}
```

- **RootAnycastReaderSource**

```
package com.greatfree.multicast;

import java.io.Serializable;

import com.greatfree.remote.FreeClientPool;

/*
 * This class assists the resource pool to create instances of anycast readers. Thus, it contains all of
required arguments to do that. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class RootAnycastReaderSource<Source extends Serializable, AnycastRequestMessage
extends AnycastRequest, RequestCreator extends
RootAnycastRequestCreatable<AnycastRequestMessage, Source>>
{
   // A TCP client pool to interact with remote nodes. 11/29/2014, Bing Li
   private FreeClientPool clientPool;
   // The root node's branch count, which can be understood as the capacity of the root node to send
messages concurrently. The root is the broadcast request initial sender. 11/29/2014, Bing Li
   private int rootBranchCount;
   // The other node's branch count, which can be understood as the capacity of the nodes to send
messages concurrently. In this case,  11/29/2014, Bing Li
   private int treeBranchCount;
   // The creator to generate anycast requests. 11/29/2014, Bing Li
   private RequestCreator creator;

   /*
    * Initialize. 11/29/2014, Bing Li
    */
   public RootAnycastReaderSource(FreeClientPool clientPool, int rootBranchCount, int
treeBranchCount, RequestCreator creator)
   {
      this.clientPool = clientPool;
      this.rootBranchCount = rootBranchCount;
      this.treeBranchCount = treeBranchCount;
      this.creator = creator;
   }

   /*
    * Expose the client pool. 11/29/2014, Bing Li
    */
   public FreeClientPool getClientPool()
   {
      return this.clientPool;
   }

   /*
    * Expose the root branch count. 11/29/2014, Bing Li
    */
   public int getRootBranchCount()
   {
      return this.rootBranchCount;
   }

   /*
    * Expose the tree branch count. 11/29/2014, Bing Li
    */
   public int getTreeBranchCount()
   {
      return this.treeBranchCount;
   }

   /*
```

```
    * Expose the request creator. 11/29/2014, Bing Li
    */
   public RequestCreator getCreator()
   {
      return this.creator;
   }
}
```

- **RootAnycastRequestCreatable**

```
package com.greatfree.multicast;

import java.io.Serializable;
import java.util.HashMap;

/*
 * The interface defines the methods to create requests in the anycastor. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public interface RootAnycastRequestCreatable<Request extends AnycastRequest, MessagedData
extends Serializable>
{
   // Create the request for the requestor that have children nodes. 11/29/2014, Bing Li
   public Request createInstanceWithChildren(MessagedData t, String collaboratorKey,
HashMap<String, String> childrenMap);
   // Create the request for the requestor that have no children nodes. 11/29/2014, Bing Li
   public Request createInstanceWithoutChildren(MessagedData t, String collaboratorKey);
}
```

- **RootAnycastRequestCreatorGettable**

```
package com.greatfree.multicast;

import java.io.Serializable;

/*
 * The interface returns the anycast request creator. It is used to constrain the source to provide the
method for the resource pool to manage anycastors. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public interface RootAnycastRequestCreatorGettable<Request extends AnycastRequest,
MessagedData extends Serializable>
{
   public RootAnycastRequestCreatable<Request, MessagedData> getRequestCreator();
}
```

2.5 Utilities

- **Tools**

```
package com.greatfree.util;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.security.Key;
import javax.crypto.spec.SecretKeySpec;

import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.UUID;

import javax.crypto.Mac;

/*
 * The class contains some methods that provide other classes with some generic services. 07/30/2014,
Bing Li
 */

// Created: 07/17/2014, Bing Li
public class Tools
{
  /*
   * Create a unique key. 07/30/2014, Bing Li
   */
  public static String generateUniqueKey()
  {
    return UUID.randomUUID().toString();
  }

  /*
   * Create a hash string upon the input string. 07/30/2014, Bing Li
   */
  public static String getHash(String input)
  {
    try
    {
      byte[] keyBytes = UtilConfig.PRIVATE_KEY.getBytes();
      Key key = new SecretKeySpec(keyBytes, 0, keyBytes.length, UtilConfig.HMAC_MD5);
      Mac mac = Mac.getInstance(UtilConfig.HMAC_MD5);
      mac.init(key);
      return UtilConfig.A + byteArrayToHex(mac.doFinal(input.getBytes()));
    }
    catch (Exception e)
    {
      e.printStackTrace();
      return UtilConfig.EMPTY_STRING;
    }
  }

  protected static String byteArrayToHex(byte [] a)
  {
    int hn, ln, cx;
    StringBuffer buf = new StringBuffer(a.length * 2);
    for(cx = 0; cx < a.length; cx++)
    {
      hn = ((int)(a[cx]) & 0x00ff) / 16;
      ln = ((int)(a[cx]) & 0x000f);
      buf.append(UtilConfig.HEX_DIGIT_CHARS.charAt(hn));
      buf.append(UtilConfig.HEX_DIGIT_CHARS.charAt(ln));
```

```
    }
    return buf.toString();
  }

  /*
   * Create a hash string upon TCP Socket. 07/30/2014, Bing Li
   */
  public static String getClientIPPortKey(Socket clientSocket)
  {
    return Tools.getHash(clientSocket.getRemoteSocketAddress().toString());
  }

  /*
   * Get the IP address upon a Socket. 07/30/2014, Bing Li
   */
  public static String getClientIPAddress(Socket clientSocket)
  {
    String ipAddress = clientSocket.getRemoteSocketAddress().toString();
    ipAddress = ipAddress.substring(0, ipAddress.indexOf(Symbols.COLON));
    ipAddress = ipAddress.substring(ipAddress.indexOf(Symbols.FORWARD_SLASH) + 1);
    return ipAddress;
  }

  /*
   * Get the port number upon a Socket. 07/30/2014, Bing Li
   */
  public static int getClientIPPort(Socket clientSocket)
  {
    String ipPort = clientSocket.getRemoteSocketAddress().toString();
    ipPort = ipPort.substring(ipPort.indexOf(Symbols.COLON) + 1);
    return (new Integer(ipPort)).intValue();
  }

  /*
   * Create a hash string upon the IP address and the port number. 07/30/2014, Bing Li
   */
  public static String getKeyOfFreeClient(String ip, int port)
  {
    return Tools.getHash(ip + port);
  }

  /*
   * Estimate the byte size of an object. 11/25/2014, Bing Li
   */
  public static long sizeOf(Object original) throws IOException
  {
    ObjectOutputStream oos;
    ByteArrayOutputStream baos;
    baos = new ByteArrayOutputStream();
    oos = new ObjectOutputStream(baos);
    oos.writeObject(original);
    oos.close();
    return baos.toByteArray().length;
  }

  /*
   * Select the key from the set of keys that is most closed to the source key in terms of the their key
distance. 11/28/2014, Bing Li
   */
  public static String getClosestKey(String sourceKey, Set<String> keys)
  {
    // Initialize a collection. 11/28/2014, Bing Li
    Map<String, StringObj> keyMap = new HashMap<String, StringObj>();
    // Put the source key and its instance of StringObj into the collection. 11/28/2014, Bing Li
    keyMap.put(sourceKey, new StringObj(sourceKey));
    // Put each of the keys and its instance of StringObj  into the same collection. 11/28/2014, Bing Li
    for (String key : keys)
    {
```

```
            keyMap.put(key, new StringObj(key));
        }
        // Sort the collection in an ascendant order by their keys. 11/28/2014, Bing Li
        keyMap = CollectionSorter.sortByValue(keyMap);
        // Put the sorted keys into a list. 11/28/2014, Bing Li
        List<String> keyList = new LinkedList<String>(keyMap.keySet());
        // Get the index of the source key. 11/28/2014, Bing Li
        int index = keyList.indexOf(sourceKey);
        // Check whether the index is lower than the last one of the list, i.e., check whether the index of the
source key is the last one of the list. 11/28/2014, Bing Li
        if (index < keyList.size() - 1)
        {
            // If the index of the source key is not the last one, then the one that is immediately greater than
the source key is believed to be the one that is the most closed to the source key. 11/28/2014, Bing Li
            return keyList.get(index + 1);
        }
        else
        {
            // If the index of the source key is the last one, then the one that is immediately less than the
source key is believed to be the one that is the most closed to the source key. 11/28/2014, Bing Li
            return keyList.get(index - 1);
        }
    }

    /*
     * Select the key from the list of keys that is most closed to the source key in terms of the their key
distance. 11/28/2014, Bing Li
     */
    public static String getClosestKey(String sourceKey, List<String> keys)
    {
        // Initialize a collection. 11/28/2014, Bing Li
        Map<String, StringObj> keyMap = new HashMap<String, StringObj>();
        // Put the source key and its instance of StringObj into the collection. 11/28/2014, Bing Li
        keyMap.put(sourceKey, new StringObj(sourceKey));
        // Put each of the keys and its instance of StringObj  into the same collection. 11/28/2014, Bing Li
        for (String key : keys)
        {
            keyMap.put(key, new StringObj(key));
        }
        // Sort the collection in an ascendant order by their keys. 11/28/2014, Bing Li
        keyMap = CollectionSorter.sortByValue(keyMap);
        // Put the sorted keys into a list. 11/28/2014, Bing Li
        List<String> keyList = new LinkedList<String>(keyMap.keySet());
        // Get the index of the source key. 11/28/2014, Bing Li
        int index = keyList.indexOf(sourceKey);
        // Check whether the index is lower than the last one of the list, i.e., check whether the index of the
source key is the last one of the list. 11/28/2014, Bing Li
        if (index < keyList.size() - 1)
        {
            // If the index of the source key is not the last one, then the one that is immediately greater than
the source key is believed to be the one that is the most closed to the source key. 11/28/2014, Bing Li
            return keyList.get(index + 1);
        }
        else
        {
            // If the index of the source key is the last one, then the one that is immediately less than the
source key is believed to be the one that is the most closed to the source key. 11/28/2014, Bing Li
            return keyList.get(index - 1);
        }
    }
}
```

- **FileManager**

```
package com.greatfree.util;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.Writer;

/*
 * The class provides some fundamental file operations based on the API from File of JDK. 11/03/2014, Bing Li
 */

// Created: 11/03/2014, Bing Li
public class FileManager
{
    /*
     * Detect whether a path or a directory exists in a specific file system, such as Windows, Linux or Unix. 11/03/2014, Bing Li
     */
    public static boolean isDirExisted(String directory)
    {
        File d = new File(directory);
        return d.exists();
    }

    /*
     * Create a directory in a file system. 11/03/2014, Bing Li
     */
    public static boolean makeDir(String directory)
    {
        try
        {
            String parentDir = getParentDir(directory);
            if (!isDirExisted(parentDir))
            {
                makeDir(parentDir);
            }
            return (new File(directory)).mkdir();
        }
        catch (Exception e)
        {
            e.printStackTrace();
            return false;
        }
    }

    /*
     * Get the parent directory of the input one. 11/03/2014, Bing Li
     */
    private static String getParentDir(String directory)
    {
        int index = directory.lastIndexOf(Symbols.FORWARD_SLASH);
        if (index > 0)
        {
            return directory.substring(0, index);
        }
        return UtilConfig.NO_DIR;
    }

    /*
```

```
    * Remote a directory. 11/04/2014, Bing Li
    */
   public static void removeFiles(String path)
   {
      File directory = new File(path);
      if (directory.exists())
      {
         File[] files = directory.listFiles();
         if (files != null)
         {
            for (File file : files)
            {
               file.delete();
            }
         }
      }
   }

   /*
    * Create a text file that is made up with the text. 11/23/2014, Bing Li
    */
   public static void createTextFile(String fileName, String text) throws IOException
   {
      Writer writer = null;
      try
      {
         writer = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(fileName),
UtilConfig.UTF_8));
         writer.write(text);
      }
      finally
      {
         writer.close();
      }
   }

   /*
    * Load a text file into the memory. 11/25/2014, Bing Li
    */
   public static String loadText(String fileName) throws IOException
   {
      BufferedReader br = new BufferedReader(new FileReader(fileName));
      try
      {
         StringBuilder sb = new StringBuilder();
         String line = br.readLine();
         while (line != null)
         {
            sb.append(line);
            sb.append("\n");
            line = br.readLine();
         }
         return sb.toString();
      }
      finally
      {
         br.close();
      }
   }
}
```

- **FreeObject**

```
package com.greatfree.util;

import java.util.Calendar;
import java.util.Date;

/*
 * The class is designed in the system to fit object reusing, caching and so on. 07/30/2014, Bing Li
 */

// Created: 07/17/2014, Bing Li
public class FreeObject implements Comparable<FreeObject>
{
   // The object key is created uniquely with respect to a unique character of the instance of the class. 08/10/2014, Bing Li
   private String objectKey;
   // The hash key is created uniquely arbitrarily. 08/10/2014, Bing Li
   private String hashKey;
   // An integer that represents the type of the class. 08/10/2014, Bing Li
   public int type;
   // The read or write accessed time stamp, which assists the caching technique to differentiate frequently-used objects from seldom-used ones. 08/10/2014, Bing Li
   private Date accessedTime;

   /*
    * Initialize the object with an explicit type. 08/10/2014, Bing Li
    */
   public FreeObject(int type, String objectKey)
   {
      this.objectKey = objectKey;
      this.hashKey = Tools.generateUniqueKey();
      this.type = type;
      this.accessedTime = Calendar.getInstance().getTime();
   }

   /*
    * Initialize the object. When initializing the object with the constructor, the type is ignored. 08/10/2014, Bing Li
    */
   public FreeObject(String objectKey)
   {
      this.objectKey = objectKey;
      this.hashKey = Tools.generateUniqueKey();
      this.type = UtilConfig.NO_TYPE;
      this.accessedTime = Calendar.getInstance().getTime();
   }

   /*
    * Initialize the object. The constructor cares about neither the object key nor the type. 11/10/2014, Bing Li
    */
   public FreeObject()
   {
      this.objectKey = UtilConfig.NO_KEY;
      this.hashKey = Tools.generateUniqueKey();
      this.type = UtilConfig.NO_TYPE;
      this.accessedTime = Calendar.getInstance().getTime();
   }

   /*
    * Expose the hash key. 08/10/2014, Bing Li
    */
   public String getHashKey()
   {
      return this.hashKey;
```

```
  }

  /*
   * Expose the object key. 08/10/2014, Bing Li
   */
  public String getObjectKey()
  {
    return this.objectKey;
  }

  /*
   * Set the object key. 08/10/2014, Bing Li
   */
  public void setObjectKey(String objectKey)
  {
    this.objectKey = objectKey;
  }

  /*
   * Get the type of the object. 08/10/2014, Bing Li
   */
  public int getType()
  {
    return this.type;
  }

  /*
   * Get the last accessed time of the object. 08/10/2014, Bing Li
   */
  public Date getAccessedTime()
  {
    return this.accessedTime;
  }

  /*
   * Set the accessed time of the object. Usually, it is performed when the object is read and written.
08/10/2014, Bing Li
   */
  public void setAccessedTime()
  {
    this.accessedTime = Calendar.getInstance().getTime();
  }

  /*
   * The method achieves the goal that objects that extend the class can be compared upon their
accessed times. 08/10/2014, Bing Li
   */
  @Override
  public int compareTo(FreeObject obj)
  {
    if (obj != null)
    {
      if (this.accessedTime.equals(obj.getAccessedTime()))
      {
        return 0;
      }
      else if (this.accessedTime.after(obj.getAccessedTime()))
      {
        return 1;
      }
      else
      {
        return -1;
      }
    }
    else
    {
      return 1;
```

```
    }
  }
}
```

- **HashFreeObject**

```
package com.greatfree.util;

import java.util.Calendar;
import java.util.Date;

/*
 * This is another general object that defines some fundamental information that is required to manage in a pool. Different from the one, FreeObject, this object is managed by the hash key rather than the object key. 11/26/2014, Bing Li
 *
 * A hash key means that each object has a unique key for any instances without taking care of their types. Well, the object key is assigned to those objects which are classified in the same type. 11/26/2014, Bing Li
 *
 */

// Created: 11/26/2014, Bing Li
public class HashFreeObject implements Comparable<HashFreeObject>
{
  // The hash key is created uniquely arbitrarily. 11/26/2014, Bing Li
  private String hashKey;
  // An integer that represents the type of the class. 11/26/2014, Bing Li
  private int type;
  // The read or write accessed time stamp, which assists the caching technique to differentiate frequently-used objects from seldom-used ones. 11/26/2014, Bing Li
  private Date accessedTime;

  /*
   * Initialize the object. 11/26/2014, Bing Li
   */
  public HashFreeObject()
  {
    this.hashKey = Tools.generateUniqueKey();
    this.accessedTime = Calendar.getInstance().getTime();
  }

  /*
   * Dispose. 11/26/2014, Bing Li
   */
  public void dispose()
  {
  }

  /*
   * Expose the hash key. 11/26/2014, Bing Li
   */
  public String getHashKey()
  {
    return this.hashKey;
  }

  /*
   * Get the type of the object. 11/26/2014, Bing Li
   */
  public int getType()
  {
    return this.type;
  }

  /*
   * Get the last accessed time of the object. 11/26/2014, Bing Li
   */
  public Date getAccessedTime()
  {
```

```
    return this.accessedTime;
  }

  /*
   * Set the accessed time of the object. Usually, it is performed when the object is read and written.
11/26/2014, Bing Li
   */
  public void setAccessedTime()
  {
    this.accessedTime = Calendar.getInstance().getTime();
  }

  /*
   * The method achieves the goal that objects that extend the class can be compared upon their
accessed times. 11/26/2014, Bing Li
   */
  @Override
  public int compareTo(HashFreeObject obj)
  {
    if (obj != null)
    {
      if (this.accessedTime.equals(obj.getAccessedTime()))
      {
        return 0;
      }
      else if (this.accessedTime.after(obj.getAccessedTime()))
      {
        return 1;
      }
      else
      {
        return -1;
      }
    }
    else
    {
      return 1;
    }
  }
}
```

- **NullObject**

```
package com.greatfree.util;

import java.io.Serializable;

/*
 * The class represents nothing. It is used when an object needs to fill the placeholder of generics, but it does not matter what should be put there. 11/20/2014, Bing Li
 */

// Created: 11/20/2014, Bing Li
public class NullObject implements Serializable
{
   private static final long serialVersionUID = 6111364234809714519L;
}
```

- **StringObj**

```
package com.greatfree.util;

/*
 * This is an object that can be compared by its key. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class StringObj implements Comparable<StringObj>
{
  // The key of the object. 11/28/2014, Bing Li
  private String key;

  /*
   * Initialize. 11/28/2014, Bing Li
   */
  public StringObj(String key)
  {
    this.key = key;
  }

  /*
   * Expose the key. 11/28/2014, Bing Li
   */
  public String getKey()
  {
    return this.key;
  }

  /*
   * Compare the object with another one. 11/28/2014, Bing Li
   */
  @Override
  public int compareTo(StringObj remoteObj)
  {
    if (remoteObj != null)
    {
      return this.key.compareTo(remoteObj.getKey());
    }
    else
    {
      return 1;
    }
  }
}
```

- **Time**

```
package com.greatfree.util;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;

/*
 * The class consists of some common constant and methods to process time. 08/26/2014, Bing Li
 */

// Created: 08/26/2014, Bing Li
public class Time
{
   // The date format in the Chinese convention. 08/26/2014, Bing Li
   public static final DateFormat CHINA_DATE_FORMAT = new SimpleDateFormat("yyyy-MM-dd
EEE HH:mm:ss", Locale.CHINA);
   // The date format in the US convention. 08/26/2014, Bing Li
   public static final DateFormat US_DATE_FORMAT = new SimpleDateFormat("MM/dd/yyyy EEE
hh:mm:ss a", Locale.US);
   // The initial time in the type of String defined in the sample code. Readers can select another time to
fit your requirements. 08/26/2014, Bing Li
   public static final String INIT_TIME_US = "06/04/1989 SUN 00:00:00 AM";
   // A constant to represent the null value of Date. 08/26/2014, Bing Li
   public static final Date NO_TIME = null;
   // The initial time in the type of Date defined in the sample code. Readers can select another time to fit
your requirements. 08/26/2014, Bing Li
   public static final Date INIT_TIME = Time.convertUSTimeToDate(Time.INIT_TIME_US);

   /*
    * Calculate the time span in the unit of millisecond between two time moments. The argument,
endTime, is the later moment and the one of beginTime is the earlier moment. 08/26/2014, Bing Li
    */
   public static long getTimespanInMilliSecond(Date endTime, Date beginTime)
   {
      return (endTime.getTime() - beginTime.getTime());
   }

   /*
    * Calculate the time span in the unit of second between two time moments. The argument, endTime,
is the later moment and the one of beginTime is the earlier moment. 08/26/2014, Bing Li
    */
   public static long getTimeSpanInSecond(Date endTime, Date beginTime)
   {
      return (endTime.getTime() - beginTime.getTime()) / 1000;
   }

   /*
    * Convent the US time in the type of String to the type of Date. 08/26/2014, Bing Li
    */
   public synchronized static Date convertUSTimeToDate(String usTimeString)
   {
      try
      {
         return Time.US_DATE_FORMAT.parse(usTimeString);
      }
      catch (ParseException e)
      {
         e.printStackTrace();
         return UtilConfig.NO_TIME;
      }
   }
```

```
  /*
   * Convent the Chinese time in the type of String to the type of Date. 08/26/2014, Bing Li
   */
  public synchronized static Date convertChinaTimeToDate(String chinaTimeString)
  {
    try
    {
      return Time.CHINA_DATE_FORMAT.parse(chinaTimeString);
    }
    catch (ParseException e)
    {
      e.printStackTrace();
      return UtilConfig.NO_TIME;
    }
  }

  /*
   * Pause for a moment. The time of pausing is equal to the argument of time in the unit of millisecond.
08/26/2014, Bing Li
   */
  public static void pause(long time)
  {
    try
    {
      Thread.sleep(time);
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
  }
}
```

- **NodeID**

```
package com.greatfree.util;

/*
 * This singleton is used to save a node's unique ID only. 11/07/2014, Bing Li
 */

// Created: 11/07/2014, Bing Li
public class NodeID
{
  // The unique key. 11/07/2014, Bing Li
  private String key;

  private NodeID()
  {
  }

  // A singleton implementation. 11/07/2014, Bing Li
  private static NodeID instance = new NodeID();

  public static NodeID DISTRIBUTED()
  {
    if (instance == null)
    {
      instance = new NodeID();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Set the key. 11/07/2014, Bing Li
   */
  public void setKey(String clientKey)
  {
    this.key = clientKey;
  }

  /*
   * Expose the key. 11/07/2014, Bing Li
   */
  public String getKey()
  {
    return this.key;
  }
}
```

- **Rand**

```
package com.greatfree.util;

import java.util.Random;

/*
 * This code is used to generate different random number in integer, float and double by enclosing the
one, Random, in JDK. 11/10/2014, Bing Li
 */

// Created: 11/10/2014, Bing Li
public class Rand
{
  // Initialize an instance of Random. 11/10/2014, Bing Li
  private static Random random = new Random();

  /*
   * Get a random integer which is less than the value of max. 11/10/2014, Bing Li
   */
  public static int getRandom(int max)
  {
    return random.nextInt(max);
  }

  /*
   * Get a random integer between the value of startInt and that of endInt. 11/10/2014, Bing Li
   */
  public static int getRandom(int startInt, int endInt)
  {
    // Check whether startInt is greater than endInt. 11/10/2014, Bing Li
    if (startInt > endInt)
    {
      // If so, it does not make sense. Return -1. 11/10/2014, Bing Li
      return -1;
    }
    // Generate the random integer. 11/10/2014, Bing Li
    return startInt + (int)((endInt - startInt) * random.nextDouble());
  }

  /*
   * Get a random double. 11/10/2014, Bing Li
   */
  public static double getDRandom()
  {
    return random.nextDouble();
  }

  /*
   * Get a random float. 11/10/2014, Bing Li
   */
  public static float getFRandom()
  {
    return random.nextFloat();
  }
}
```

- **CollectionSorter**

```
package com.greatfree.util;

import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedHashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;

/*
 * The class aims to sort a collection in the ascending or descending manner. It is also able to select the maximum or minimum value from the collection. 08/26/2014, Bing Li
 */

// Created: 08/26/2014, Bing Li
public class CollectionSorter
{
  /*
   * Sort a map in the ascending order. 08/26/2014, Bing Li
   */
  public static <K, V extends Comparable<? super V>> Map<K, V> sortByValue(Map<K, V> map)
  {
    List<Map.Entry<K, V>> list = new LinkedList<Map.Entry<K, V>>(map.entrySet());
    Collections.sort(list, new Comparator<Map.Entry<K, V>>()
    {
      public int compare(Map.Entry<K, V> o1, Map.Entry<K, V> o2)
      {
        return (o1.getValue()).compareTo(o2.getValue());
      }
    });

    Map<K, V> result = new LinkedHashMap<K, V>();
    for (Map.Entry<K, V> entry : list)
    {
      result.put(entry.getKey(), entry.getValue());
    }
    return result;
  }

  /*
   * Sort a map in the ascending order and only the top ranked are returned. The returned count is equal to topCount. 08/26/2014, Bing Li
   */
  public static <K, V extends Comparable<? super V>> Map<K, V> sortByValue(Map<K, V> map,
double topCount)
  {
    List<Map.Entry<K, V>> list = new LinkedList<Map.Entry<K, V>>(map.entrySet());
    Collections.sort(list, new Comparator<Map.Entry<K, V>>()
    {
      public int compare(Map.Entry<K, V> o1, Map.Entry<K, V> o2)
      {
        return (o1.getValue()).compareTo(o2.getValue());
      }
    });

    Map<K, V> result = new LinkedHashMap<K, V>();
    for (Map.Entry<K, V> entry : list)
    {
      result.put(entry.getKey(), entry.getValue());
      if (result.size() >= topCount)
      {
        return result;
      }
    }
```

```
    return result;
  }

  /*
   * Retrieve the minimum value from a map. 08/26/2014, Bing Li
   */
  public static <K, V extends Comparable<? super V>> K minValueKey(Map<K, V> map)
  {
    if (map.size() > 0)
    {
      List<Map.Entry<K, V>> list = new LinkedList<Map.Entry<K, V>>(map.entrySet());
      Collections.sort(list, new Comparator<Map.Entry<K, V>>()
      {
        public int compare(Map.Entry<K, V> o1, Map.Entry<K, V> o2)
        {
          return (o1.getValue()).compareTo(o2.getValue());
        }
      });

      return list.get(0).getKey();
    }
    else
    {
      return null;
    }
  }

  /*
   * Sort a map in the descending order. 08/26/2014, Bing Li
   */
  public static <K, V extends Comparable<? super V>> Map<K, V>
sortDescendantByValue(Map<K, V> map)
  {
    List<Map.Entry<K, V>> list = new LinkedList<Map.Entry<K, V>>(map.entrySet());
    Collections.sort(list, new Comparator<Map.Entry<K, V>>()
    {
      public int compare(Map.Entry<K, V> o1, Map.Entry<K, V> o2)
      {
        return (o2.getValue()).compareTo(o1.getValue());
      }
    });

    Map<K, V> result = new LinkedHashMap<K, V>();
    for (Map.Entry<K, V> entry : list)
    {
      result.put(entry.getKey(), entry.getValue());
    }
    return result;
  }

  /*
   * Sort a map in the descending order and only the top ranked are returned. The returned count is
equal to topCount. 08/26/2014, Bing Li
   */
  public static <K, V extends Comparable<? super V>> Map<K, V>
sortDescendantByValue(Map<K, V> map, double topCount)
  {
    List<Map.Entry<K, V>> list = new LinkedList<Map.Entry<K, V>>(map.entrySet());
    Collections.sort(list, new Comparator<Map.Entry<K, V>>()
    {
      public int compare(Map.Entry<K, V> o1, Map.Entry<K, V> o2)
      {
        return (o2.getValue()).compareTo(o1.getValue());
      }
    });

    Map<K, V> result = new LinkedHashMap<K, V>();
    for (Map.Entry<K, V> entry : list)
```

```
      {
        result.put(entry.getKey(), entry.getValue());
        if (result.size() >= topCount)
        {
          return result;
        }
      }
      return result;
    }

    /*
     * Retrieve the maximum value from a map. 08/26/2014, Bing Li
     */
    public static <K, V extends Comparable<? super V>> K maxValueKey(Map<K, V> map)
    {
      if (map.size() > 0)
      {
        List<Map.Entry<K, V>> list = new LinkedList<Map.Entry<K, V>>(map.entrySet());
        Collections.sort(list, new Comparator<Map.Entry<K, V>>()
        {
          public int compare(Map.Entry<K, V> o1, Map.Entry<K, V> o2)
          {
            return (o2.getValue()).compareTo(o1.getValue());
          }
        });

        return list.get(0).getKey();
      }
      else
      {
        return null;
      }
    }
}
```

- **XMLReader**

```
package com.greatfree.util;

import java.io.IOException;
import java.io.StringReader;
import java.util.LinkedList;
import java.util.List;

import javax.xml.namespace.QName;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.xpath.XPathFactory;

import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;

/*
 * This is a tool to read information from an XML, which is usually managed by administrators manually.
11/25/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class XMLReader
{
  private String xmlResource;
  private Document xmlDocument;
  private XPath xPath;
  private DocumentBuilder xmlDocumentBuilder;
  private InputSource xmlInMemorySource;

  /*
   * Initialize. 11/25/2014, Bing Li
   */
  public XMLReader(String xmlFile, boolean isXMLInMemory)
  {
    if (!isXMLInMemory)
    {
      this.xmlResource = xmlFile;
      this.initObjects();
    }
    else
    {
      try
      {
        this.xmlResource = FileManager.loadText(xmlFile);
        this.xmlResource = this.trim(this.xmlResource);
        this.initObjectsFromMemory();
      }
      catch (IOException e)
      {
        e.printStackTrace();
      }
    }
  }

  /*
   * Close the reader. 11/25/2014, Bing Li
   */
  public void close()
```

```
{
  this.xmlResource = null;
  this.xmlDocument = null;
  this.xmlDocumentBuilder = null;
  this.xmlInMemorySource = null;
  this.xPath = null;
}

/*
 * Reset the XML to be parsed. 11/25/2014, Bing Li
 */
public boolean reset(String xmlResource, boolean isXMLInMemory)
{
  this.xmlResource = xmlResource;
  if (!isXMLInMemory)
  {
    return this.resetObjects();
  }
  else
  {
    this.xmlResource = this.trim(this.xmlResource);
    return this.resetObjectsFromMemory();
  }
}

/*
 * Parse the objects upon XML on the disk. 11/25/2014, Bing Li
 */
private boolean initObjects()
{
  try
  {
    this.xmlDocumentBuilder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
    this.xmlDocument = this.xmlDocumentBuilder.parse(this.xmlResource);
    this.xPath = XPathFactory.newInstance().newXPath();
    return true;
  }
  catch (IOException ex)
  {
    ex.printStackTrace();
    return false;
  }
  catch (SAXException ex)
  {
    ex.printStackTrace();
    return false;
  }
  catch (ParserConfigurationException ex)
  {
    ex.printStackTrace();
    return false;
  }
}

/*
 * Parse the objects upon XML in the memory. 11/25/2014, Bing Li
 */
private boolean initObjectsFromMemory()
{
  this.xmlInMemorySource = new InputSource();
  this.xmlInMemorySource.setCharacterStream(new StringReader(this.xmlResource));
  try
  {
    this.xmlDocumentBuilder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
    this.xmlDocument = this.xmlDocumentBuilder.parse(this.xmlInMemorySource);
    this.xPath =  XPathFactory.newInstance().newXPath();
    return true;
  }
```

```
  catch (IOException ex)
  {
    ex.printStackTrace();
    return false;
  }
  catch (SAXException ex)
  {
    ex.printStackTrace();
    return false;
  }
  catch (ParserConfigurationException ex)
  {
    ex.printStackTrace();
    return false;
  }
}

/*
 * Parse again from the XML on the disk. 11/25/2014, Bing Li
 */
private boolean resetObjects()
{
  try
  {
    this.xmlDocument = this.xmlDocumentBuilder.parse(this.xmlResource);
    return true;
  }
  catch (IOException ex)
  {
    ex.printStackTrace();
    return false;
  }
  catch (SAXException ex)
  {
    ex.printStackTrace();
    return false;
  }
}

/*
 * Parse again from the XML in the memory. 11/25/2014, Bing Li
 */
private boolean resetObjectsFromMemory()
{
  this.xmlInMemorySource.setCharacterStream(new StringReader(this.xmlResource));
  try
  {
    this.xmlDocument = this.xmlDocumentBuilder.parse(this.xmlInMemorySource);
    return true;
  }
  catch (IOException ex)
  {
    ex.printStackTrace();
    return false;
  }
  catch (SAXException ex)
  {
    ex.printStackTrace();
    return false;
  }
}

/*
 * Read an object from the XML. 11/25/2014, Bing Li
 */
public Object read(String expression, QName returnType)
{
  try
```

```
    {
      return this.xPath.compile(expression).evaluate(this.xmlDocument, returnType);
    }
    catch (XPathExpressionException ex)
    {
      ex.printStackTrace();
      return null;
    }
  }

  /*
   * Read a value encoded in the form of ASCII from the XML. 11/25/2014, Bing Li
   */
  public String read(String expression)
  {
    try
    {
      return (String)this.xPath.compile(expression).evaluate(this.xmlDocument,
XPathConstants.STRING);
    }
    catch (XPathExpressionException ex)
    {
      ex.printStackTrace();
      return UtilConfig.EMPTY_STRING;
    }
  }

  /*
   * Retrieve information from the XML. Since the result must not be a unique, it is necessary to save
them into a list. 11/25/2014, Bing Li
   */
  public NodeList readMany(String expression)
  {
    try
    {
      return (NodeList)this.xPath.compile(expression).evaluate(this.xmlDocument,
XPathConstants.NODESET);
    }
    catch (XPathExpressionException ex)
    {
      ex.printStackTrace();
      return UtilConfig.NO_MULTI_RESULTS;
    }
  }

  /*
   * Retrieve information from the XML. Since the result must not be a unique, it is necessary to save
them in the encoded ASCII into a list. 11/25/2014, Bing Li
   */
  public List<String> readStrings(String expression)
  {
    try
    {
      NodeList nodes = (NodeList)this.xPath.compile(expression).evaluate(this.xmlDocument,
XPathConstants.NODESET);
      List<String> nodeList = new LinkedList<String>();
      for (int i = 0; i < nodes.getLength(); i ++)
      {
        nodeList.add(nodes.item(i).getNodeValue());
      }
      return nodeList;
    }
    catch (XPathExpressionException ex)
    {
      ex.printStackTrace();
      return UtilConfig.NO_STRINGS;
    }
  }
```

```
    /*
     * Remove the format of the XML. 11/25/2014, Bing Li
     */
    private String trim(String xml)
    {
      return xml.trim().replaceFirst(UtilConfig.TRIM_EXPRESSION, UtilConfig.LESS_THAN);
    }
}
```

- **Symbols**

```
package com.greatfree.util;

/*
 * The class defines some frequently-used symbols. 07/30/2014, Bing Li
 */

// Created: 07/30/2014, Bing Li
public class Symbols
{
   public static final String LINE_FEED = "\n";
   public static final String EMPTY_STRING = "";
   public static final String UTF_8 = "UTF-8";
   public static final String COLON = ":";
   public static final String FORWARD_SLASH = "/";
   public static final String BACK_SLASH = "\\";
}
```

- **UtilConfig**

```
package com.greatfree.util;

import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.util.Set;
import java.util.Timer;

import org.w3c.dom.NodeList;

import com.greatfree.remote.FreeClient;
import com.greatfree.remote.IPPort;

/*
 * The class keeps relevant configurations and constants of the solution. 07/30/2014, Bing Li
 */

// Created: 07/17/2014, Bing Li
public class UtilConfig
{
  public final static String A = "a";
  public final static String PRIVATE_KEY = "06/04/1989";
  public final static int ADDTIONAL_THREAD_POOL_SIZE = 0;
  public final static String HMAC_MD5 = "HmacMD5";
  public final static String EMPTY_STRING = "";
  public final static String HEX_DIGIT_CHARS = "0123456789abcdef";

  public final static int NO_TYPE = 0;
  public final static String NO_IP = "";

  public final static Timer NO_TIMER = null;
  public final static Date NO_TIME = null;

  public final static String NO_KEY = "";

  public final static long ONE_SECOND = 1000;

  public final static String NO_DIR = "";

  public final static FreeClient NO_CLIENT = null;

  public final static String NO_QUEUE_KEY = "";
  public final static int NO_QUEUE_SIZE = -1;

  public final static IPPort NO_IPPORT = null;

  public final static long INIT_READ_WAIT_TIME = 2000;

  public final static String MERGE_SORT = "java.util.Arrays.useLegacyMergeSort";
  public final static String TRUE = "true";

  public final static HashMap<String, String> NO_NODES = null;

  public final static List<String> NO_CHILDREN_KEYS = null;
  public final static Set<String> NO_NODE_KEYS = null;
  public final static String ROOT_KEY = "RootKey";
  public final static String LOCAL_KEY = "LocalKey";

  public static final String UTF_8 = "UTF-8";

  public static final NodeList NO_MULTI_RESULTS = null;
  public static final List<String> NO_STRINGS = null;

  public final static String TRIM_EXPRESSION = "^([\\W]+)<";
```

```
    public final static String LESS_THAN = "<";
}
```

- **TerminateSignal**

```
package com.greatfree.util;

/*
 * The class is a flag that represents whether the client process is set to be terminated or not. For some long running threads, they can check the flag to stop their tasks immediately. 09/21/2014, Bing Li
 *
 * Since a process being terminated is its unique state. The class is implemented in the pattern of singleton. 09/21//2014, Bing Li
 */

// Created: 09/21/2014, Bing Li
public class TerminateSignal
{
  // The flag to indicate whether the client process is set to be terminated or not. 09/21/2014, Bing Li
  private boolean isTerminated;

  /*
   * Initialize. 09/21/2014, Bing Li
   */
  private TerminateSignal()
  {
    this.isTerminated = false;
  }

  // Implement it as a singleton. 09/21/2014, Bing Li
  private static TerminateSignal instance = new TerminateSignal();

  public static TerminateSignal SIGNAL()
  {
    if (instance == null)
    {
      instance = new TerminateSignal();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  public boolean isTerminated()
  {
    return this.isTerminated;
  }

  public void setTerminated()
  {
    this.isTerminated = true;
  }
}
```

3. Applications

3.1 Data

- **Constants**

```
package com.greatfree.testing.data;

/*
 * It contains some common constants in the samples. 11/30/2014, Bing Li
 */

// Created: 11/23/2014, Bing Li
public class Constants
{
   public final static String NO_URL_KEY = "";
   public final static String NO_URL = "";
   public final static long NO_UPDATING_PERIOD = 0;
   public static final int INIT_UPDATING_PERIOD = 0;

   public final static long MEMORY_BATCH_LOAD = 10000000;
}
```

- **ClientConfig**

```
package com.greatfree.testing.data;

/*
 * The class contains all of constants at the client end. 09/21/2014, Bing Li
 */

// Created: 09/21/2014, Bing Li
public class ClientConfig
{
   public final static int CLIENT_POOL_SIZE = 500;
   public final static String USERNAME = "greatfree";
   public final static String PASSWORD = "19890604";
   public final static int CLIENT_LISTENER_THREAD_POOL_SIZE = 80;
   public final static long CLIENT_LISTENER_THREAD_ALIVE_TIME = 10000;

   public final static long CLIENT_IDLE_CHECK_DELAY = 3000;
   public final static long CLIENT_IDLE_CHECK_PERIOD = 3000;
   public final static long CLIENT_MAX_IDLE_TIME = 3000;

   public final static int NOTIFICATION_DISPATCHER_POOL_SIZE = 30;
   public final static long NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME = 2000;

   public final static int MAX_NOTIFICATION_TASK_SIZE = 500;
   public final static int MAX_NOTIFICATION_THREAD_SIZE = 10;

   public final static long NOTIFICATION_DISPATCHER_WAIT_TIME = 1000;
   public final static long NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY = 3000;
   public final static long NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD = 3000;

   public final static int TIME_TO_WAIT_FOR_THREAD_TO_DIE = 10;

   public final static int SERVER_CLIENT_POOL_SIZE = 10;

   public final static int EVENTER_THREAD_POOL_SIZE = 10;
   public final static long EVENTER_THREAD_POOL_ALIVE_TIME = 10000;

   public final static int SERVER_IO_POOL_SIZE = 100;

   public final static int MULTICASTOR_POOL_SIZE = 100;
   public final static long MULTICASTOR_WAIT_TIME = 1000;
   public static final int MULTICAST_BRANCH_COUNT = 16;

   public final static int EVENT_QUEUE_SIZE = 100;
   public final static int EVENTER_SIZE = 10;
   public final static int EVENTING_WAIT_TIME = 1000;
   public final static int EVENTER_WAIT_TIME = 1000;
   public final static long EVENT_IDLE_CHECK_DELAY = 3000;
   public final static long EVENT_IDLE_CHECK_PERIOD = 3000;
}
```

- **ServerConfig**

```
package com.greatfree.testing.data;

import com.greatfree.remote.FreeClient;

/*
 * The class keeps constants of the testing sample code. 08/04/2014, Bing Li
 */

// Created: 08/04/2014, Bing Li
public class ServerConfig
{
  public final static String SERVER_IP = "192.168.1.106";
  public final static int SERVER_PORT = 8964;
  public final static int CLIENT_PORT = 8949;

  public final static int DISPATCHER_POOL_SIZE = 500;
  public final static long DISPATCHER_POOL_THREAD_POOL_ALIVE_TIME = 2000;

  public final static int LISTENING_THREAD_COUNT = 5;
  public final static int LISTENER_THREAD_POOL_SIZE = 50;
  public final static long LISTENER_THREAD_ALIVE_TIME = 10000;

  public final static int MAX_SERVER_IO_COUNT = 500;

  public final static long TERMINATE_SLEEP = 2000;

  public final static int REMOTE_SERVER_PORT = 8962;

  public final static FreeClient NO_CLIENT = null;

  public final static int MY_SERVER = 1;

  public final static int MAX_CLIENT_LISTEN_THREAD_COUNT = 5;

  public final static String ROOT_PATH = "/home/libing/GreatFreeLabs/";
  public final static String CONFIG_HOME = ServerConfig.ROOT_PATH + "Config/";

  public final static String COORDINATOR_ADDRESS = "192.168.1.113";
  public final static int COORDINATOR_PORT_FOR_CRAWLER = 8963;
  public final static int COORDINATOR_PORT_FOR_MEMORY = 8965;
  public final static int COORDINATOR_PORT_FOR_ADMIN = 8951;
  public final static int COORDINATOR_PORT_FOR_SEARCH = 8950;
  public final static int CRAWL_SERVER_PORT = 8960;
  public final static int MEMORY_SERVER_PORT = 8961;
  public final static int SEARCH_CLIENT_PORT = 8948;

  public final static long RETRIEVE_THREAD_WAIT_TIME = 1000;
  public final static long NOTIFICATION_THREAD_WAIT_TIME = 1000;

  public final static int REQUEST_DISPATCHER_POOL_SIZE = 50;
  public final static long REQUEST_DISPATCHER_THREAD_ALIVE_TIME = 500;
  public final static int MAX_REQUEST_TASK_SIZE = 500;
  public final static int MAX_REQUEST_THREAD_SIZE = 50;
  public final static long REQUEST_DISPATCHER_WAIT_TIME = 1000;
  public final static long REQUEST_DISPATCHER_IDLE_CHECK_DELAY = 2000;
  public final static long REQUEST_DISPATCHER_IDLE_CHECK_PERIOD = 2000;

  public final static int NOTIFICATION_DISPATCHER_POOL_SIZE = 30;
  public final static long NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME = 2000;

  public final static int MAX_NOTIFICATION_TASK_SIZE = 500;
  public final static int MAX_NOTIFICATION_THREAD_SIZE = 10;

  public final static long NOTIFICATION_DISPATCHER_WAIT_TIME = 1000;
```

```
    public final static long NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY = 3000;
    public final static long NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD = 3000;

    public final static int CLIENT_POOL_SIZE = 500;
    public final static long CLIENT_IDLE_CHECK_DELAY = 3000;
    public final static long CLIENT_IDLE_CHECK_PERIOD = 3000;
    public final static long CLIENT_MAX_IDLE_TIME = 3000;

    public final static long DISTRIBUTE_DATA_WAIT_TIME = 2000;
    public final static int MULTICASTOR_POOL_SIZE = 100;
    public final static long MULTICASTOR_POOL_WAIT_TIME = 1000;

    public final static int ROOT_MULTICAST_BRANCH_COUNT = 100;
    public final static int MULTICAST_BRANCH_COUNT = 16;

}
```

- **CrawledLink**

```
package com.greatfree.testing.data;

import java.io.Serializable;

/*
 * The class that contains the crawling results, the link and the text of the link. 11/23/2014, Bing Li
 */

// Created: 11/23/2014, Bing Li
public class CrawledLink implements Serializable
{
  private static final long serialVersionUID = -833353988970909339L;

  // The key of the link, which is created by the method, Tools.getAHash(). 11/23/2014, Bing Li
  private String key;
  // The link crawled from the hub URL. 11/23/2014, Bing Li
  private String link;
  // The text associated with the crawled link, such as the title of the link. 11/23/2014, Bing Li
  private String text;
  // The key of the hub URL being crawled. 11/23/2014, Bing Li
  private String hubURLKey;

  /*
   * Initialize. 11/23/2014, Bing Li
   */
  public CrawledLink(String key, String link, String text, String hubURLKey)
  {
    this.key = key;
    this.link = link;
    this.text = text;
    this.hubURLKey = hubURLKey;
  }

  /*
   * Expose the key. 11/23/2014, Bing Li
   */
  public String getKey()
  {
    return this.key;
  }

  /*
   * Expose the link. 11/23/2014, Bing Li
   */
  public String getLink()
  {
    return this.link;
  }

  /*
   * Expose the text. 11/23/2014, Bing Li
   */
  public String getText()
  {
    return this.text;
  }

  /*
   * Expose the key of the hub URL being crawled. 11/23/2014, Bing Li
   */
  public String getHubURLKey()
  {
    return this.hubURLKey;
  }
```

```
}
```

- **URLValue**

```
package com.greatfree.testing.data;

import java.io.Serializable;

/*
 * The class contains the URL to be crawled. Meanwhile, the class can be transmitted between the
coordinator and the crawler. Thus, it must implement the interface, Serializable. 11/28/2014, Bing Li
 */

// Created: 11/23/2014, Bing Li
public class URLValue implements Serializable
{
  private static final long serialVersionUID = -8117874901342689273L;

  // The key of the URL. 11/28/2014, Bing Li
  private String key;
  // The URL to be crawled. 11/28/2014, Bing Li
  private String url;
  // The updating period of the URL. According to it, the crawler can determine the moment to crawl it.
11/28/2014, Bing Li
  private long updatingPeriod;

  /*
   * Initialize. 11/28/2014, Bing Li
   */
  public URLValue()
  {
    this.key = Constants.NO_URL_KEY;
    this.url = Constants.NO_URL;
    this.updatingPeriod = Constants.NO_UPDATING_PERIOD;
  }

  /*
   * Initialize. 11/28/2014, Bing Li
   */
  public URLValue(String key, String url, long updatingPeriod)
  {
    this.key = key;
    this.url = url;
    this.updatingPeriod = updatingPeriod;
  }

  /*
   * Expose the key of the URL. 11/28/2014, Bing Li
   */
  public String getKey()
  {
    return this.key;
  }

  /*
   * Expose the URL. 11/28/2014, Bing Li
   */
  public String getURL()
  {
    return this.url;
  }

  /*
   * Expose the updating period. 11/28/2014, Bing Li
   */
  public long getUpdatingPeriod()
  {
    return this.updatingPeriod;
```

```
    }
}
```

3.2 Databases

- **DBConfig**

```
package com.greatfree.testing.db;

import com.greatfree.testing.data.ServerConfig;

/*
 * It keeps some configuration constants for the object-oriented database, the Berkeley DB. 11/03/2014,
Bing Li
 */

// Created: 11/03/2014, Bing Li
public class DBConfig
{
  public static final String DB_HOME = ServerConfig.ROOT_PATH + "DB/";

  public final static long DB_CACHE_SIZE = 1000000;
  public final static long LOCK_TIME_OUT = 0;
  public final static int DB_POOL_SIZE = 100;

  public final static String NODE_STORE = "NodeStore";
  public final static NodeDB NO_NODE_DB = null;
  public final static String NODE_DB_PATH = DBConfig.DB_HOME + "NodeDB/";

  public final static String URL_STORE = "URLStore";
  public final static URLDB NO_URL_DB = null;
  public final static String URL_DB_PATH = DBConfig.DB_HOME + "URLDB/";
}
```

- **DBEnv**

```
package com.greatfree.testing.db;

import java.io.File;
import java.util.concurrent.TimeUnit;

import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import com.sleepycat.persist.EntityStore;
import com.sleepycat.persist.StoreConfig;

/*
 * The class keeps some required configurations to set up the object-oriented database, the Berkeley
DB. 11/03/2014, Bing Li
 */

// Created: 11/03/2014, Bing Li
public class DBEnv
{
  // Declare the parameters that are required to set up the object-oriented database. 11/03/2014, Bing Li
  private Environment env;
  private EntityStore store;

  /*
   * Initialize. 11/03/2014, Bing Li
   */
  public DBEnv(File envHome, boolean readOnly, long cacheSize, long timeout, String storeID)
  {
    EnvironmentConfig envConfig = new EnvironmentConfig();
    envConfig.setCacheSize(cacheSize);

    envConfig.setLockTimeout(timeout, TimeUnit.MILLISECONDS);
    StoreConfig storeConfig = new StoreConfig();

    envConfig.setReadOnly(readOnly);
    storeConfig.setReadOnly(readOnly);

    envConfig.setAllowCreate(!readOnly);
    storeConfig.setAllowCreate(!readOnly);

    this.env = new Environment(envHome, envConfig);
    this.store = new EntityStore(this.env, storeID, storeConfig);
  }

  /*
   * Expose the relevant attribute. 11/03/2014, Bing Li
   */
  public EntityStore getEntityStore()
  {
    return this.store;
  }

  /*
   * Expose the relevant attribute. 11/03/2014, Bing Li
   */
  public Environment getEnv()
  {
    return this.env;
  }

  /*
   * Close the environment. 11/03/2014, Bing Li
   */
  public void close()
```

```
  {
    if (this.store != null)
    {
      try
      {
        this.store.close();
      }
      catch (DatabaseException dbe)
      {
        dbe.printStackTrace();
      }
    }

    if (this.env != null)
    {
      try
      {
        this.env.close();
      }
      catch (DatabaseException dbe)
      {
        dbe.printStackTrace();
      }
    }
  }
}
```

- **DBPoolable**

```
package com.greatfree.testing.db;

import com.greatfree.util.FreeObject;

/*
 * The interface consists of methods that are required to implement a singleton of a database pool based
on QueuedPool. 11/04/2014, Bing Li
 */

// Created: 11/04/2014, Bing Li
public interface DBPoolable<DB extends FreeObject>
{
   // Remove a database. 11/04/2014, Bing Li
   public void removeDB(String path);
   // Shutdown a database. 11/04/2014, Bing Li
   public void shutdown();
   // Dispose a database. 11/04/2014, Bing Li
   public void dispose(DB db);
   // Create a new database. 11/04/2014, Bing Li
   public DB create(String path);
   // Set the parameters for the idle checking. 11/04/2014, Bing Li
   public void setIdleChecker(long idleCheckDelay, long idleCheckPeriod, long maxIdleTime);
   // Collect a database for reuse. 11/04/2014, Bing Li
   public void collectDB(DB db);
   // Get a database from the pool. 11/04/2014, Bing Li
   public DB getDB(String path);
}
```

- **NodeAccessor**

```
package com.greatfree.testing.db;

import com.sleepycat.persist.EntityStore;
import com.sleepycat.persist.PrimaryIndex;

/*
 * This class is used to save NodeEntity by its primary index. 10/03/2014, Bing Li
 */

// Created: 10/03/2014, Bing Li
public class NodeAccessor
{
  // Define the primary index. 10/08/2014, Bing Li
  private PrimaryIndex<String, NodeEntity> primaryIndex;

  /*
   * Initialize. 10/08/2014, Bing Li
   */
  public NodeAccessor(EntityStore store)
  {
    this.primaryIndex = store.getPrimaryIndex(String.class, NodeEntity.class);
  }

  /*
   * Dispose. 10/08/2014, Bing Li
   */
  public void dispose()
  {
    this.primaryIndex = null;
  }

  /*
   * Expose the primary index. 10/08/2014, Bing Li
   */
  public PrimaryIndex<String, NodeEntity> getPrimaryIndex()
  {
    return this.primaryIndex;
  }
}
```

- **NodeDB**

```
package com.greatfree.testing.db;

import java.io.File;
import java.util.HashMap;
import java.util.Map;

import com.greatfree.testing.server.Node;
import com.greatfree.util.FileManager;
import com.greatfree.util.FreeObject;
import com.greatfree.util.Tools;
import com.sleepycat.persist.EntityCursor;

/*
 * The class implements the manipulations on the object, NodeEntity, to save and retrieve in the way supported by the Berkeley DB. It derives from FreeObject such that its instance can be managed by a QueuedPool. 11/03/2014, Bing Li
 */

// Created: 10/08/2014, Bing Li
public class NodeDB extends FreeObject
{
  // Declare the instance of File since operations on a file system is required. 11/03/2014, Bing Li
  private File envPath;
  // Declare the instance of DBEnv to set up the required environment. 11/03/2014, Bing Li
  private DBEnv env;
  // Declare the instance of NodeAccessor to manipulate objects. 11/03/2014, Bing Li
  private NodeAccessor accessor;

  /*
   * Initialize. 11/03/2014, Bing Li
   */
  public NodeDB(String path)
  {
    super(Tools.getHash(path));
    if (!FileManager.isDirExisted(path))
    {
      FileManager.makeDir(path);
    }
    this.envPath = new File(path);
    this.env = new DBEnv(this.envPath, false, DBConfig.DB_CACHE_SIZE, DBConfig.LOCK_TIME_OUT, DBConfig.NODE_STORE);
    this.accessor = new NodeAccessor(this.env.getEntityStore());
  }

  /*
   * Dispose the database. 11/03/2014, Bing Li
   */
  public void dispose()
  {
    this.accessor.dispose();
    this.env.close();
  }

  /*
   * Load all of the persisted nodes from the database. 11/03/2014, Bing Li
   */
  public Map<String, Node> loadAllNodes()
  {
    EntityCursor<NodeEntity> results = this.env.getEntityStore().getPrimaryIndex(String.class, NodeEntity.class).entities();
    Map<String, Node> nodes = new HashMap<String, Node>();
    Node nodeValue;
    for (NodeEntity node : results)
    {
```

```
            nodeValue = new Node(node.getKey(), node.getUserName(), node.getPassword());
            nodes.put(nodeValue.getKey(), nodeValue);
        }
        results.close();
        return nodes;
    }

    /*
     * Persist a collection of nodes. 11/03/2014, Bing Li
     */
    public void saveNodes(Map<String, Node> nodes)
    {
        for (Node node : nodes.values())
        {
            this.saveNode(node);
        }
    }

    /*
     * Persist a single node. 11/03/2014, Bing Li
     */
    public void saveNode(Node node)
    {
        this.accessor.getPrimaryIndex().put(new NodeEntity(node.getKey(), node.getUsername(),
node.getPassword()));
    }
}
```

- **NodeDBCreator**

```
package com.greatfree.testing.db;

import java.io.IOException;

import com.greatfree.reuse.Creatable;

/*
 * A creator that initializes instances of NodeDB. 11/04/2014, Bing Li
 */

// Created: 11/04/2014, Bing Li
public class NodeDBCreator implements Creatable<String, NodeDB>
{
  @Override
  public NodeDB createResourceInstance(String source) throws IOException
  {
    return new NodeDB(source);
  }
}
```

- **NodeDBDisposer**

```
package com.greatfree.testing.db;

import com.greatfree.reuse.Disposable;

/*
 * A disposer to dispose instances of NodeDB. 11/04/2014, Bing Li
 */

// Created: 11/04/2014, Bing Li
public class NodeDBDisposer implements Disposable<NodeDB>
{
   @Override
   public void dispose(NodeDB rsc)
   {
      rsc.dispose();
   }
}
```

- **NodeDBPool**

```
package com.greatfree.testing.db;

import java.io.IOException;

import com.greatfree.reuse.QueuedPool;
import com.greatfree.util.FileManager;

/*
 * A singleton of a database pool for NodeDB upon QueuedPool. 11/04/2014, Bing Li
 */

// Created: 11/04/2014, Bing Li
public class NodeDBPool implements DBPoolable<NodeDB>
{
  // Declare a database pool. 11/04/2014, Bing Li
  private QueuedPool<NodeDB, NodeDBCreator, NodeDBDisposer> pool;

  // Initialize. 11/04/2014, Bing Li
  private NodeDBPool()
  {
    this.pool = new QueuedPool<NodeDB, NodeDBCreator,
NodeDBDisposer>(DBConfig.DB_POOL_SIZE, new NodeDBCreator(), new NodeDBDisposer());
  }

  // Define a singleton. 11/04/2014, Bing Li
  private static NodeDBPool instance = new NodeDBPool();

  public static NodeDBPool PERSISTENT()
  {
    if (instance == null)
    {
      instance = new NodeDBPool();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Remove the database from the file system. 11/04/2014, Bing Li
   */
  @Override
  public void removeDB(String path)
  {
    try
    {
      // Shutdown the database. 11/04/2014, Bing Li
      this.pool.shutdown();
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
    // Remove the directory of the database from the file system. 11/04/2014, Bing Li
    FileManager.removeFiles(path);
  }

  /*
   * Shutdown the database. 11/04/2014, Bing Li
   */
  @Override
  public void shutdown()
```

```
{
  try
  {
    this.pool.shutdown();
  }
  catch (IOException e)
  {
    e.printStackTrace();
  }
}

/*
 * Dispose an instance of a database. 11/04/2014, Bing Li
 */
@Override
public void dispose(NodeDB db)
{
  try
  {
    this.pool.dispose(db);
  }
  catch (IOException e)
  {
    e.printStackTrace();
  }
}

/*
 * Create a new instance of NodeDB, which is not managed in the pool. 11/04/2014, Bing Li
 */
@Override
public NodeDB create(String path)
{
  try
  {
    return this.pool.create(path);
  }
  catch (IOException e)
  {
    e.printStackTrace();
  }
  catch (InterruptedException e)
  {
    e.printStackTrace();
  }
  return DBConfig.NO_NODE_DB;
}

/*
 * Set the idle checking parameters. 11/04/2014, Bing Li
 */
@Override
public void setIdleChecker(long idleCheckDelay, long idleCheckPeriod, long maxIdleTime)
{
  this.pool.setIdleChecker(idleCheckDelay, idleCheckPeriod, maxIdleTime);
}

/*
 * Collect an instance of NodeDB though the pool. 11/04/2014, Bing Li
 */
@Override
public void collectDB(NodeDB db)
{
  this.pool.collect(db);
}

/*
 * Get an instance of NodeDB from the pool. 11/04/2014, Bing Li
```

```
     */
    @Override
    public NodeDB getDB(String path)
    {
      try
      {
        return this.pool.get(path);
      }
      catch (InstantiationException e)
      {
        e.printStackTrace();
      }
      catch (IllegalAccessException e)
      {
        e.printStackTrace();
      }
      catch (IOException e)
      {
        e.printStackTrace();
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
      return DBConfig.NO_NODE_DB;
    }
}
```

- **NodeEntity**

```
package com.greatfree.testing.db;

import com.sleepycat.persist.model.Entity;
import com.sleepycat.persist.model.PrimaryKey;

/*
 * This is an example to create an object that can be persisted in the object-oriented database, the
Berkeley DB. 09/26/2014, Bing Li
 */

// Created: 09/26/2014, Bing Li
@Entity
public class NodeEntity
{
   // The unique key that distinguishes from others. For an object to be persisted, it is required to design
such a key. 09/26/2014, Bing Li
   @PrimaryKey
   private String key;

   // A field, userName, to be saved. 09/26/2014, Bing Li
   private String username;
   // A field, password, to be saved. 09/26/2014, Bing Li
   private String password;

   /*
    * The constructor is required by the database, the Berkeley DB. 09/26/2014, Bing Li
    */
   public NodeEntity()
   {
   }

   /*
    * An constructor initializes the class to set values for all of the properties. 10/03/2014, Bing Li
    */
   public NodeEntity(String key, String username, String password)
   {
      this.key = key;
      this.username = username;
      this.password = password;
   }

   /*
    * The setter for the attribute of key. It is required by the Berkeley DB. 10/03/2014, Bing Li
    */
   public void setKey(String key)
   {
      this.key = key;
   }

   /*
    * The getter for the attribute of key. It is required by the Berkeley DB. 10/03/2014, Bing Li
    */
   public String getKey()
   {
      return this.key;
   }

   /*
    * The setter for the attribute of userName. It is required by the Berkeley DB. 10/03/2014, Bing Li
    */
   public void setUserName(String username)
   {
      this.username = username;
   }
```

```
    /*
     * The getter for the attribute of userName. It is required by the Berkeley DB. 10/03/2014, Bing Li
     */
    public String getUserName()
    {
      return this.username;
    }

    /*
     * The setter for the attribute of password. It is required by the Berkeley DB. 10/03/2014, Bing Li
     */
    public void setPassword(String password)
    {
      this.password = password;
    }

    /*
     * The getter for the attribute of password. It is required by the Berkeley DB. 10/03/2014, Bing Li
     */
    public String getPassword()
    {
      return this.password;
    }
}
```

- **URLAccessor**

```
package com.greatfree.testing.db;

import com.sleepycat.persist.EntityStore;
import com.sleepycat.persist.PrimaryIndex;

/*
 * This class is used to save URLEntity by its primary index. 11/25/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class URLAccessor
{
  // Define the primary index. 11/25/2014, Bing Li
  private PrimaryIndex<String, URLEntity> primaryIndex;

  /*
   * Initialize. 11/25/2014, Bing Li
   */
  public URLAccessor(EntityStore store)
  {
    this.primaryIndex = store.getPrimaryIndex(String.class, URLEntity.class);
  }

  /*
   * Dispose. 11/25/2014, Bing Li
   */
  public void dispose()
  {
    this.primaryIndex = null;
  }

  /*
   * Expose the primary index. 11/25/2014, Bing Li
   */
  public PrimaryIndex<String, URLEntity> getPrimaryIndex()
  {
    return this.primaryIndex;
  }
}
```

- **URLDB**

```
package com.greatfree.testing.db;

import java.io.File;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

import com.google.common.collect.Sets;
import com.greatfree.testing.data.URLValue;
import com.greatfree.util.FileManager;
import com.greatfree.util.FreeObject;
import com.greatfree.util.Tools;
import com.sleepycat.persist.EntityCursor;

/*
 * The class implements the manipulations on the object, URLEntity, to save and retrieve in the way supported by the Berkeley DB. It derives from FreeObject such that its instance can be managed by a QueuedPool. 11/25/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class URLDB extends FreeObject
{
  // Declare the instance of File since operations on a file system is required. 11/25/2014, Bing Li
  private File envPath;
  // Declare the instance of DBEnv to set up the required environment. 11/25/2014, Bing Li
  private DBEnv env;
  // Declare the instance of NodeAccessor to manipulate objects. 11/25/2014, Bing Li
  private URLAccessor accessor;

  /*
   * Initialize the DB. 11/25/2014, Bing Li
   */
  public URLDB(String path)
  {
    super(Tools.getHash(path));
    if (!FileManager.isDirExisted(path))
    {
      FileManager.makeDir(path);
    }
    this.envPath = new File(path);
    this.env = new DBEnv(this.envPath, false, DBConfig.DB_CACHE_SIZE, DBConfig.LOCK_TIME_OUT, DBConfig.URL_STORE);
    this.accessor = new URLAccessor(this.env.getEntityStore());
  }

  /*
   * Dispose the DB. 11/25/2014, Bing Li
   */
  public void dispose()
  {
    this.accessor.dispose();
    this.env.close();
  }

  /*
   * Load all of the persisted URLs from the database. 11/25/2014, Bing Li
   */
  public Map<String, URLValue> loadAllURLs()
  {
    EntityCursor<URLEntity> results = this.env.getEntityStore().getPrimaryIndex(String.class, URLEntity.class).entities();
    Map<String, URLValue> urls = new HashMap<String, URLValue>();
    URLValue url;
```

```
    for (URLEntity entity : results)
    {
      url = new URLValue(entity.getKey(), entity.getURL(), entity.getUpdatingPeriod());
      urls.put(url.getKey(), url);
    }
    results.close();
    return urls;
  }

  /*
   * Load the keys of all of the persisted URLs from the database. 11/25/2014, Bing Li
   */
  public Set<String> loadAllURLKeys()
  {
    EntityCursor<URLEntity> results = this.env.getEntityStore().getPrimaryIndex(String.class,
URLEntity.class).entities();
    Set<String> urlKeys = Sets.newHashSet();
    for (URLEntity entity : results)
    {
      urlKeys.add(entity.getKey());
    }
    results.close();
    return urlKeys;
  }

  /*
   * Load the count of all of the persisted URLs. 11/25/2014, Bing Li
   */
  public long loadAllURLCount()
  {
    return this.env.getEntityStore().getPrimaryIndex(String.class, URLEntity.class).entities().count();
  }

  /*
   * Persist a collection of URLs. 11/25/2014, Bing Li
   */
  public void saveURLs(Map<String, URLValue> urls)
  {
    for (URLValue url : urls.values())
    {
      this.saveURL(url);
    }
  }

  /*
   * Persist a single URL. 11/25/2014, Bing Li
   */
  public void saveURL(URLValue url)
  {
    this.accessor.getPrimaryIndex().put(new URLEntity(url.getKey(), url.getURL(),
url.getUpdatingPeriod()));
  }
}
```

- **URLDBCreator**

```
package com.greatfree.testing.db;

import java.io.IOException;

import com.greatfree.reuse.Creatable;

/*
 * A creator that initializes instances of URLDB. It works with the URLDBPool to manage the instances of
URLDB. 11/25/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class URLDBCreator implements Creatable<String, URLDB>
{
  @Override
  public URLDB createResourceInstance(String source) throws IOException
  {
    return new URLDB(source);
  }
}
```

- **URLDBDisposer**

```
package com.greatfree.testing.db;

import com.greatfree.reuse.Disposable;

/*
 * A disposer that collects instances of URLDB. It works with the URLDBPool to manage the instances of
URLDB. 11/25/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class URLDBDisposer implements Disposable<URLDB>
{
   @Override
   public void dispose(URLDB rsc)
   {
      rsc.dispose();
   }
}
```

- **URLDBPool**

```
package com.greatfree.testing.db;

import java.io.IOException;

import com.greatfree.reuse.QueuedPool;
import com.greatfree.util.FileManager;

/*
 * A singleton of a database pool for URLDB upon QueuedPool. 11/25/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class URLDBPool implements DBPoolable<URLDB>
{
  // Declare a database pool. 11/25/2014, Bing Li
  private QueuedPool<URLDB, URLDBCreator, URLDBDisposer> pool;

  // Initialize. 11/25/2014, Bing Li
  private URLDBPool()
  {
    this.pool = new QueuedPool<URLDB, URLDBCreator,
URLDBDisposer>(DBConfig.DB_POOL_SIZE, new URLDBCreator(), new URLDBDisposer());
  }

  // Define a singleton. 11/25/2014, Bing Li
  private static URLDBPool instance = new URLDBPool();

  public static URLDBPool PERSISTENT()
  {
    if (instance == null)
    {
      instance = new URLDBPool();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Remove the database from the file system. 11/25/2014, Bing Li
   */
  @Override
  public void removeDB(String path)
  {
    try
    {
      // Shutdown the database. 11/25/2014, Bing Li
      this.pool.shutdown();
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
    // Remove the directory of the database from the file system. 11/04/2014, Bing Li
    FileManager.removeFiles(path);
  }

  /*
   * Shutdown the database. 11/25/2014, Bing Li
   */
  @Override
  public void shutdown()
```

```
{
  try
  {
    this.pool.shutdown();
  }
  catch (IOException e)
  {
    e.printStackTrace();
  }
}

/*
 * Dispose an instance of a database. 11/25/2014, Bing Li
 */
@Override
public void dispose(URLDB db)
{
  try
  {
    this.pool.dispose(db);
  }
  catch (IOException e)
  {
    e.printStackTrace();
  }
}

/*
 * Create a new instance of URLDB, which is not managed in the pool. 11/25/2014, Bing Li
 */
@Override
public URLDB create(String path)
{
  try
  {
    return this.pool.create(path);
  }
  catch (IOException e)
  {
    e.printStackTrace();
  }
  catch (InterruptedException e)
  {
    e.printStackTrace();
  }
  return DBConfig.NO_URL_DB;
}

/*
 * Set the idle checking parameters. 11/25/2014, Bing Li
 */
@Override
public void setIdleChecker(long idleCheckDelay, long idleCheckPeriod, long maxIdleTime)
{
  this.pool.setIdleChecker(idleCheckDelay, idleCheckPeriod, maxIdleTime);
}

/*
 * Collect an instance of URLDB though the pool. 11/25/2014, Bing Li
 */
@Override
public void collectDB(URLDB db)
{
  this.pool.collect(db);
}

/*
 * Get an instance of URLDB from the pool. 11/25/2014, Bing Li
```

```
     */
    @Override
    public URLDB getDB(String path)
    {
      try
      {
        return this.pool.get(path);
      }
      catch (InstantiationException e)
      {
        e.printStackTrace();
      }
      catch (IllegalAccessException e)
      {
        e.printStackTrace();
      }
      catch (IOException e)
      {
        e.printStackTrace();
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
      return DBConfig.NO_URL_DB;
    }
}
```

- **URLEntity**

```
package com.greatfree.testing.db;

import com.greatfree.testing.data.Constants;
import com.sleepycat.persist.model.Entity;
import com.sleepycat.persist.model.PrimaryKey;

/*
 * This is an entity to save URL in the object-oriented database, the Berkeley DB. 11/25/2014, Bing Li
 */

// Created: 11/23/2014, Bing Li
@Entity
public class URLEntity
{
   // The unique key that distinguishes from others. For an object to be persisted, it is required to design such a key. 11/25/2014, Bing Li
   @PrimaryKey
   private String key;

   // The URL. 11/25/2014, Bing Li
   private String url;
   // The updating period of the URL. 11/25/2014, Bing Li
   private long updatingPeriod;

   /*
    * The empty constructor is required by the database, the Berkeley DB. 11/25/2014, Bing Li
    */
   public URLEntity()
   {
      this.key = Constants.NO_URL_KEY;
      this.url = Constants.NO_URL;
      this.updatingPeriod = Constants.NO_UPDATING_PERIOD;
   }

   /*
    * Initialize the entity. 11/25/2014, Bing Li
    */
   public URLEntity(String key, String url, long updatingPeriod)
   {
      this.key = key;
      this.url = url;
      this.updatingPeriod = updatingPeriod;
   }

   /*
    * The getter for the attribute of key. It is required by the Berkeley DB. 10/03/2014, Bing Li
    */
   public String getKey()
   {
      return this.key;
   }

   /*
    * The setter for the attribute of key. It is required by the Berkeley DB. 11/25/2014, Bing Li
    */
   public void setKey(String key)
   {
      this.key = key;
   }

   /*
    * The getter for the attribute of URL. It is required by the Berkeley DB. 11/25/2014, Bing Li
    */
   public String getURL()
```

```
    {
      return this.url;
    }

    /*
     * The setter for the attribute of URL. It is required by the Berkeley DB. 11/25/2014, Bing Li
     */
    public void setURL(String url)
    {
      this.url = url;
    }

    /*
     * The getter for the attribute of updating period. It is required by the Berkeley DB. 11/25/2014, Bing Li
     */
    public long getUpdatingPeriod()
    {
      return this.updatingPeriod;
    }

    /*
     * The setter for the attribute of updating period. It is required by the Berkeley DB. 11/25/2014, Bing Li
     */
    public void setUpdatingPeriod(long updatingPeriod)
    {
      this.updatingPeriod = updatingPeriod;
    }
}
```

3.3 Messages

- **MessageType**

```
package com.greatfree.testing.message;

/*
 * The class contains the identification of each type of messages. 11/28/2014, Bing Li
 */

// Created: 09/20/2014, Bing Li
public class MessageType
{
   public final static int NODE_KEY_NOTIFICATION = 0;
   public final static int SIGN_UP_REQUEST = 1;
   public final static int SIGN_UP_RESPONSE = 2;
   public final static int INIT_READ_NOTIFICATION = 3;
   public final static int INIT_READ_FEEDBACK_NOTIFICATION = 4;
   public final static int ONLINE_NOTIFICATION = 5;
   public final static int REGISTER_CLIENT_NOTIFICATION = 6;
   public final static int UNREGISTER_CLIENT_NOTIFICATION = 7;
   public final static int CRAWLED_LINKS_NOTIFICATION = 8;
   public final static int REGISTER_CRAWL_SERVER_NOTIFICATION = 9;
   public final static int UNREGISTER_CRAWL_SERVER_NOTIFICATION = 10;
   public final static int CRAWL_LOAD_NOTIFICATION = 11;
   public final static int START_CRAWL_MULTI_NOTIFICATION = 12;
   public final static int SHUTDOWN_CRAWL_SERVER_NOTIFICATION = 13;
   public final static int SHUTDOWN_MEMORY_SERVER_NOTIFICATION = 14;
   public final static int SHUTDOWN_COORDINATOR_SERVER_NOTIFICATION = 15;
   public final static int STOP_CRAWL_MULTI_NOTIFICATION = 16;
   public final static int STOP_MEMORY_SERVER_NOTIFICATION = 17;
   public final static int REGISTER_MEMORY_SERVER_NOTIFICATION = 18;
   public final static int UNREGISTER_MEMORY_SERVER_NOTIFICATION = 19;
   public final static int ADD_CRAWLED_LINK_NOTIFICATION = 20;
   public final static int IS_PUBLISHER_EXISTED_REQUEST = 21;
   public final static int IS_PUBLISHER_EXISTED_RESPONSE = 22;
   public final static int SEARCH_KEYWORD_REQUEST = 23;
   public final static int SEARCH_KEYWORD_RESPONSE = 24;
   public final static int IS_PUBLISHER_EXISTED_ANYCAST_REQUEST = 25;
   public final static int IS_PUBLISHER_EXISTED_ANYCAST_RESPONSE = 26;
   public final static int SEARCH_KEYWORD_BROADCAST_REQUEST = 27;
   public final static int SEARCH_KEYWORD_BROADCAST_RESPONSE = 28;
}
```

- **MessageConfig**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * The class contains all of the constants related to messages transmitted between remote nodes.
09/21/2014, Bing Li
 */

// Created: 09/21/2014, Bing Li
public class MessageConfig
{
   public final static ServerMessage NO_MESSAGE = null;
   public final static SignUpResponse NO_SIGN_UP_RESPONSE = null;
}
```

3.3.1 Notifications

- **AddCrawledLinkNotification**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;
import com.greatfree.testing.data.CrawledLink;

/*
 * The notification contains one crawled link. It is sent to one distributed memory node for storage.
11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class AddCrawledLinkNotification extends ServerMessage
{
  private static final long serialVersionUID = 2093100652281316873L;

  // The memory server key. 11/28/2014, Bing Li
  private String dcKey;
  // The crawled link. 11/28/2014, Bing Li
  private CrawledLink link;

  /*
   * Initialize. 11/28/2014, Bing Li
   */
  public AddCrawledLinkNotification(String dcKey, CrawledLink link)
  {
    super(MessageType.ADD_CRAWLED_LINK_NOTIFICATION);
    this.dcKey = dcKey;
    this.link = link;
  }

  /*
   * Expose the memory server key. 11/28/2014, Bing Li
   */
  public String getDCKey()
  {
    return this.dcKey;
  }

  /*
   * Expose the crawled link. 11/28/2014, Bing Li
   */
  public CrawledLink getLink()
  {
    return this.link;
  }
}
```

- **CrawledLinksNotification**

```
package com.greatfree.testing.message;

import java.util.Set;

import com.greatfree.multicast.ServerMessage;
import com.greatfree.testing.data.CrawledLink;

/*
 * The notification contains the crawled links to be sent to the coordinator from the crawler. 11/23/2014,
Bing Li
 */

// Created: 11/23/2014, Bing Li
public class CrawledLinksNotification extends ServerMessage
{
  private static final long serialVersionUID = -2613694876747449900L;

  // The crawled links. 11/23/2014, Bing Li
  private Set<CrawledLink> links;

  /*
   * Initialize the crawled links. 11/23/2014, Bing Li
   */
  public CrawledLinksNotification(Set<CrawledLink> links)
  {
    super(MessageType.CRAWLED_LINKS_NOTIFICATION);
    this.links = links;
  }

  /*
   * Expose the crawled links. 11/23/2014, Bing Li
   */
  public Set<CrawledLink> getLinks()
  {
    return this.links;
  }
}
```

- **CrawlLoadNotification**

```
package com.greatfree.testing.message;

import java.util.HashMap;
import java.util.Map;

import com.greatfree.multicast.ServerMessage;
import com.greatfree.testing.data.URLValue;

/*
 * This is a notification sent to a crawler to assign the URLs to it for crawling. 11/25/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class CrawlLoadNotification extends ServerMessage
{
  private static final long serialVersionUID = 2075087040783551384L;

  // The crawler key to be sent to. 11/25/2014, Bing Li
  private String dcKey;
  // The URLs to be crawled by the crawler. 11/25/2014, Bing Li
  private Map<String, URLValue> urls;
  // Since the load might be high, the load might be sent to the crawler in a piece of notifications. The flag denotes which one is the first. 11/25/2014, Bing Li
  private boolean isFirst;
  // Since the load might be high, the load might be sent to the crawler in a piece of notifications. The flag denotes which one is the last one. 11/25/2014, Bing Li
  private boolean isDone;

  /*
   * Initialize the notification. 11/25/2014, Bing Li
   */
  public CrawlLoadNotification(String dcKey, boolean isFirst)
  {
    super(MessageType.CRAWL_LOAD_NOTIFICATION);
    this.dcKey = dcKey;
    this.urls = new HashMap<String, URLValue>();
    this.isFirst = isFirst;
    this.isDone = false;
  }

  /*
   * Expose the crawler key. 11/25/2014, Bing Li
   */
  public String getDCKey()
  {
    return this.dcKey;
  }

  /*
   * Expose the URLs to be assigned. 11/25/2014, Bing Li
   */
  public Map<String, URLValue> getURLs()
  {
    return this.urls;
  }

  /*
   * Set the URLs to be assigned. 11/25/2014, Bing Li
   */
  public void setURLs(Map<String, URLValue> urls)
  {
    this.urls = urls;
  }
```

```
  /*
   * Expose the flag of whether the notification is the first one to assign the load to the crawler.
11/25/2014, Bing Li
   */
  public boolean isFirst()
  {
    return this.isFirst;
  }

  /*
   * Expose the flag of whether the notification is the last one to assign the load to the crawler.
11/25/2014, Bing Li
   */
  public boolean isDone()
  {
    return this.isDone;
  }

  /*
   * Set the done flag. 11/25/2014, Bing Li
   */
  public void setDone()
  {
    this.isDone = true;
  }
}
```

- **InitReadFeedbackNotification**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 *
 * The message is sent as a notification by a server after its corresponding ObjectOutputStream is
initialized. It notifies the cl
 */

// Created: 11/07/2014, Bing Li
public class InitReadFeedbackNotification extends ServerMessage
{
   private static final long serialVersionUID = 4474239685077563637L;

   /*
    * Initialize. It is just a notification without additional information. 11/07/2014, Bing Li
    */
   public InitReadFeedbackNotification()
   {
      super(MessageType.INIT_READ_FEEDBACK_NOTIFICATION);
   }
}
```

- **InitReadNotification**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * The message is sent to the remote server to ensure a relevant instance of ObjectOutputStream is initialized on the server. If so, it means that the local ObjectInputStream can be initialized. Otherwise, the client must get stuck. To avoid the problem, the message is required to be sent if the local client needs to receive data from the server. 11/03/2014, Bing Li
 */

// Created: 11/03/2014, Bing Li
public class InitReadNotification extends ServerMessage
{
  private static final long serialVersionUID = -559357101111103183L;

  // It is required to send the client key to the remote server such that the server is able to retrieve the client socket on the server to notify the local client to initialize ObjectInputStream. 11/03/2014, Bing Li
  private String clientKey;

  /*
   * Initialize. 11/03/2014, Bing Li
   */
  public InitReadNotification(String clientKey)
  {
    super(MessageType.INIT_READ_NOTIFICATION);
    this.clientKey = clientKey;
  }

  /*
   * Expose the client key. 11/03/2014, Bing Li
   */
  public String getClientKey()
  {
    return this.clientKey;
  }
}
```

- **NodeKeyNotification**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * This message sends to a client when it is online initially. A unique key is assigned to the client to
identify each of them. 11/09/2014, Bing Li
 */

// Created: 09/19/2014, Bing Li
public class NodeKeyNotification extends ServerMessage
{
  private static final long serialVersionUID = -6132975872385412676L;

  // The key of the node. 11/09/2014, Bing Li
  private String nodeKey;

  /*
   * Initialize the node key. 11/09/2014, Bing Li
   */
  public NodeKeyNotification(String nodeKey)
  {
    super(MessageType.NODE_KEY_NOTIFICATION);
    this.nodeKey = nodeKey;
  }

  /*
   * Expose the node key. 11/09/2014, Bing Li
   */
  public String getNodeKey()
  {
    return this.nodeKey;
  }
}
```

- **OnlineNotification**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * This notification is sent to a remote server that the client expects to connect. Its goal is to establish a connection. Moreover, if a peer-to-peer architecture needs to be constructed between them, the server starts to connect to the client after receiving the notification. The notification works like a stimulus.
11/07/2014, Bing Li
 */

// Created: 11/07/2014, Bing Li
public class OnlineNotification extends ServerMessage
{
  private static final long serialVersionUID = -8946571501653241937L;

  public OnlineNotification()
  {
    super(MessageType.ONLINE_NOTIFICATION);
  }
}
```

- **RegisterClientNotification**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * This notification sends its unique key to a remote server to register its necessary information on the server. 11/07/2014, Bing Li
 */

// Created: 11/07/2014, Bing Li
public class RegisterClientNotification extends ServerMessage
{
  private static final long serialVersionUID = -6716159464958246998L;

  // The unique key of a client. 11/07/2014, Bing Li
  private String clientKey;

  /*
   * Initialize. 11/07/2014, Bing Li
   */
  public RegisterClientNotification(String clientKey)
  {
    super(MessageType.REGISTER_CLIENT_NOTIFICATION);
    this.clientKey = clientKey;
  }

  /*
   * Expose the client key. 11/07/2014, Bing Li
   */
  public String getClientKey()
  {
    return this.clientKey;
  }
}
```

- **RegisterCrawlServerNotification**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * This is the notification for a crawler to register on the coordinator. 11/23/2014, Bing Li
 */

// Created: 11/23/2014, Bing Li
public class RegisterCrawlServerNotification extends ServerMessage
{
  private static final long serialVersionUID = 4999830006599286378L;

  // The key of the crawler server. Here, DC stands for the term, distributed component. 11/23/2014,
Bing Li
  private String dcKey;
  // The count of URLs the crawler server needs to crawl. 11/23/2014, Bing Li
  private long urlCount;

  /*
   * Initialize. 11/23/2014, Bing Li
   */
  public RegisterCrawlServerNotification(String dcKey, long urlCount)
  {
    super(MessageType.REGISTER_CRAWL_SERVER_NOTIFICATION);
    this.dcKey = dcKey;
    this.urlCount = urlCount;
  }

  /*
   * Expose the key of the crawler. 11/23/2014, Bing Li
   */
  public String getDCKey()
  {
    return this.dcKey;
  }

  /*
   * Expose the URL count. 11/23/2014, Bing Li
   */
  public long getURLCount()
  {
    return this.urlCount;
  }
}
```

- **RegisterMemoryServerNotification**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * This is the notification for a memory server to register on the coordinator. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class RegisterMemoryServerNotification extends ServerMessage
{
  private static final long serialVersionUID = -6985005756162124112L;

  // The key of the memory server. Here, DC stands for the term, distributed component. 11/28/2014,
Bing Li
  private String dcKey;

  /*
   * Initialize. 11/28/2014, Bing Li
   */
  public RegisterMemoryServerNotification(String dcKey)
  {
    super(MessageType.REGISTER_MEMORY_SERVER_NOTIFICATION);
    this.dcKey = dcKey;
  }

  /*
   * Expose the key of the memory. 11/28/2014, Bing Li
   */
  public String getDCKey()
  {
    return this.dcKey;
  }
}
```

- **ShutdownCoordinatorServerNotification**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * The notification is raised by the administrator and then sent to the coordinator. In the sample, the coordinator is a standalone machine such that it can be terminated when receiving the notification. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class ShutdownCoordinatorServerNotification extends ServerMessage
{
	private static final long serialVersionUID = 9220677303206367246L;

	/*
	 * Initialize. No any arguments are needed. 11/27/2014, Bing Li
	 */
	public ShutdownCoordinatorServerNotification()
	{
		super(MessageType.SHUTDOWN_COORDINATOR_SERVER_NOTIFICATION);
	}
}
```

- **ShutdownCrawlServerNotification**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * The notification is raised by the administrator and then sent to the coordinator. Through the
coordinator, the relevant notification is multicast to all of the crawlers to stop the crawling process.
11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class ShutdownCrawlServerNotification extends ServerMessage
{
  private static final long serialVersionUID = 3492103470869232426L;

  /*
   * Initialize. No any arguments are needed. 11/27/2014, Bing Li
   */
  public ShutdownCrawlServerNotification()
  {
    super(MessageType.SHUTDOWN_CRAWL_SERVER_NOTIFICATION);
  }
}
```

- **ShutdownMemoryServerNotification**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * The notification is raised by the administrator and then sent to the coordinator. Through the
coordinator, the relevant notification is multicast to all of the memory nodes to stop working. 11/27/2014,
Bing Li
 */

// Created: 11/27/2014, Bing Li
public class ShutdownMemoryServerNotification extends ServerMessage
{
  private static final long serialVersionUID = 4490404955904893382L;

  /*
   * Initialize. No any arguments are needed. 11/27/2014, Bing Li
   */
  public ShutdownMemoryServerNotification()
  {
    super(MessageType.SHUTDOWN_MEMORY_SERVER_NOTIFICATION);
  }
}
```

- **StartCrawlMultiNotification**

```
package com.greatfree.testing.message;

import java.util.HashMap;

import com.greatfree.multicast.ServerMulticastMessage;

/*
 * This notification is multicast to all of the crawlers to start the crawling. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public class StartCrawlMultiNotification extends ServerMulticastMessage
{
  private static final long serialVersionUID = -2832937472197546783L;

  /*
   * Initialize. 11/26/2014, Bing Li
   *
   * The key is used for binding multiple threads for synchronization management when all of them need to process the notification.
   */
  public StartCrawlMultiNotification(String key)
  {
    super(MessageType.START_CRAWL_MULTI_NOTIFICATION, key);
  }

  /*
   * Initialize. 11/26/2014, Bing Li
   *
   * The key is used for binding multiple threads for synchronization management when all of them need to process the notification.
   *
   * The collection, childrenServers, keeps all of the children keys and IPs. Through it, the node that receives the notification can send the notification to those nodes, the children.
   *
   */
  public StartCrawlMultiNotification(String key, HashMap<String, String> childrenServers)
  {
    super(MessageType.START_CRAWL_MULTI_NOTIFICATION, key, childrenServers);
  }
}
```

- **StopCrawlMultiNotification**

```
package com.greatfree.testing.message;

import java.util.HashMap;

import com.greatfree.multicast.ServerMulticastMessage;

/*
 * This notification is multicast to all of the crawlers to stop the crawling. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StopCrawlMultiNotification extends ServerMulticastMessage
{
  private static final long serialVersionUID = -1784254133690452874L;

  /*
   * Initialize. 11/27/2014, Bing Li
   *
   * The key is used for binding multiple threads for synchronization management when all of them need to process the notification.
   */
  public StopCrawlMultiNotification(String key)
  {
    super(MessageType.STOP_CRAWL_MULTI_NOTIFICATION, key);
  }

  /*
   * Initialize. 11/27/2014, Bing Li
   *
   * The key is used for binding multiple threads for synchronization management when all of them need to process the notification.
   *
   * The collection, childrenServers, keeps all of the children keys and IPs. Through it, the node that receives the notification can send the notification to those nodes, the children.
   *
   */
  public StopCrawlMultiNotification(String key, HashMap<String, String> childrenMap)
  {
    super(MessageType.STOP_CRAWL_MULTI_NOTIFICATION, key, childrenMap);
  }
}
```

- **UnregisterClientNotification**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * When a client shuts down, it needs to send the notification to its connected remote server. So, that server can remove relevant information and collect corresponding resources. 11/07/2014, Bing Li
 */

// Created: 11/07/2014, Bing Li
public class UnregisterClientNotification extends ServerMessage
{
	private static final long serialVersionUID = -2397160617314342013L;

	// The client key to be sent to the remote server. 11/07/2014 Bing Li
	private String clientKey;

	/*
	 * Initialize. 11/07/2014, Bing Li
	 */
	public UnregisterClientNotification(String clientKey)
	{
		super(MessageType.UNREGISTER_CLIENT_NOTIFICATION);
		this.clientKey = clientKey;
	}

	/*
	 * Expose the client key. 11/07/2014, Bing Li
	 */
	public String getClientKey()
	{
		return this.clientKey;
	}
}
```

- **UnregisterCrawlServerNotification**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * This is the notification for a crawler to unregister on the coordinator. 11/23/2014, Bing Li
 */

// Created: 11/23/2014, Bing Li
public class UnregisterCrawlServerNotification extends ServerMessage
{
  private static final long serialVersionUID = -2742181590283782035L;

  // The key of the crawler server. Here, DC stands for the term, distributed component. 11/23/2014,
Bing Li
  private String dcKey;

  /*
   * Initialize. 11/23/2014, Bing Li
   */
  public UnregisterCrawlServerNotification(String dcKey)
  {
    super(MessageType.UNREGISTER_CRAWL_SERVER_NOTIFICATION);
    this.dcKey = dcKey;
  }

  /*
   * Expose the key of the crawler. 11/23/2014, Bing Li
   */
  public String getDCKey()
  {
    return this.dcKey;
  }
}
```

- **UnregisterMemoryServerNotification**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * This is the notification for a memory server to unregister on the coordinator. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class UnregisterMemoryServerNotification extends ServerMessage
{
  private static final long serialVersionUID = -1541793327700936820L;

  // The key of the memory server. Here, DC stands for the term, distributed component. 11/28/2014, Bing Li
  private String dcKey;

  /*
   * Initialize. 11/28/2014, Bing Li
   */
  public UnregisterMemoryServerNotification(String dcKey)
  {
    super(MessageType.UNREGISTER_MEMORY_SERVER_NOTIFICATION);
    this.dcKey = dcKey;
  }

  /*
   * Expose the key of the memory server. 11/28/2014, Bing Li
   */
  public String getDCKey()
  {
    return this.dcKey;
  }
}
```

3.3.2 Requests/Responses

- **IsPublisherExistedAnycastRequest**

```
package com.greatfree.testing.message;

import java.util.HashMap;

import com.greatfree.multicast.AnycastRequest;

/*
 * This is an anycast request to raise the system to retrieve URL among the cluster of memory nodes in an anycast way. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedAnycastRequest extends AnycastRequest
{
  private static final long serialVersionUID = -2127692519947528923L;

  // The URL to be retrieved. 11/29/2014, Bing Li
  private String url;

  /*
   * Initialize. This is initialized for the node which has no children. 11/29/2014, Bing Li
   */
  public IsPublisherExistedAnycastRequest(String url, String key, String collaboratorKey)
  {
    super(MessageType.IS_PUBLISHER_EXISTED_ANYCAST_REQUEST, key, collaboratorKey);
    this.url = url;
  }

  /*
   * Initialize. This is initialized for the node which has children. 11/29/2014, Bing Li
   */
  public IsPublisherExistedAnycastRequest(String url, String key, String collaboratorKey, HashMap<String, String> childrenServerMap)
  {
    super(MessageType.IS_PUBLISHER_EXISTED_ANYCAST_REQUEST, key, collaboratorKey, childrenServerMap);
    this.url = url;
  }

  /*
   * Expose the URL. 11/29/2014, Bing Li
   */
  public String getURL()
  {
    return this.url;
  }
}
```

- **IsPublisherExistedAnycastResponse**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.AnycastResponse;

/*
 * This is a response to the anycast request, IsPublisherExistedAnycastRequest. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedAnycastResponse extends AnycastResponse
{
  private static final long serialVersionUID = -1149369827314333148L;

  // The flag to indicate whether the URL is existed. 11/29/2014, Bing Li
  private boolean isExisted;

  /*
   * Initialize. 11/29/2014, Bing Li
   */
  public IsPublisherExistedAnycastResponse(boolean isExisted, String collaboratorKey)
  {
    super(MessageType.IS_PUBLISHER_EXISTED_ANYCAST_RESPONSE, collaboratorKey);
    this.isExisted = isExisted;
  }

  /*
   * Expose the flag. 11/29/2014, Bing Li
   */
  public boolean isExisted()
  {
    return this.isExisted;
  }
}
```

- **IsPublisherExistedRequest**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * The request is sent by the searcher to the coordinator to retrieve whether a publisher, represented in the form of URL, is existed. It must raise an anycast retrieval on the cluster controlled by the coordinator.
11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedRequest extends ServerMessage
{
  private static final long serialVersionUID = 7791005060878729509L;

  // The URL to be retrieved. 11/29/2014, Bing Li
  private String url;

  /*
   * Initialize. 11/29/2014, Bing Li
   */
  public IsPublisherExistedRequest(String url)
  {
    super(MessageType.IS_PUBLISHER_EXISTED_REQUEST);
    this.url = url;
  }

  /*
   * Expose the URL. 11/29/2014, Bing Li
   */
  public String getURL()
  {
    return this.url;
  }
}
```

- **IsPublisherExistedResponse**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * This is the response from the coordinator to the request, IsPublisherExistedRequest. 11/29/2014, Bing
Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedResponse extends ServerMessage
{
  private static final long serialVersionUID = -4476671398966834594L;

  // The flag to indicate whether the publisher is existed. 11/29/2014, Bing Li
  private boolean isExisted;

  public IsPublisherExistedResponse(boolean isExisted)
  {
    super(MessageType.IS_PUBLISHER_EXISTED_RESPONSE);
    this.isExisted = isExisted;
  }

  public boolean isExisted()
  {
    return this.isExisted;
  }
}
```

- **IsPublisherExistedStream**

```
package com.greatfree.testing.message;

import java.io.ObjectOutputStream;
import java.util.concurrent.locks.Lock;

import com.greatfree.remote.OutMessageStream;

/*
 * The class is derived from OutMessageStream. It contains the received request, its associated output stream and the lock that keeps the responding operations atomic. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedStream extends OutMessageStream<IsPublisherExistedRequest>
{
  public IsPublisherExistedStream(ObjectOutputStream out, Lock lock, IsPublisherExistedRequest message)
  {
    super(out, lock, message);
  }
}
```

- **SearchKeywordBroadcastRequest**

```
package com.greatfree.testing.message;

import java.util.HashMap;

import com.greatfree.multicast.BroadcastRequest;

/*
 * This is a broadcast request to raise the system to retrieve keyword among the cluster of memory
nodes in a broadcast way. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordBroadcastRequest extends BroadcastRequest
{
  private static final long serialVersionUID = -5910485952080467801L;

  // The keyword to be retrieved. 11/29/2014, Bing Li
  private String keyword;

  /*
   * Initialize. This is initialized for the node which has no children. 11/29/2014, Bing Li
   */
  public SearchKeywordBroadcastRequest(String keyword, String key, String collaboratorKey)
  {
    super(MessageType.SEARCH_KEYWORD_BROADCAST_REQUEST, key, collaboratorKey);
    this.keyword = keyword;
  }

  /*
   * Initialize. This is initialized for the node which has children. 11/29/2014, Bing Li
   */
  public SearchKeywordBroadcastRequest(String keyword, String key, String collaboratorKey,
HashMap<String, String> children)
  {
    super(MessageType.SEARCH_KEYWORD_BROADCAST_REQUEST, key, collaboratorKey,
children);
    this.keyword = keyword;
  }

  /*
   * Expose the keyword. 11/29/2014, Bing Li
   */
  public String getKeyword()
  {
    return this.keyword;
  }
}
```

- **SearchKeywordBroadcastResponse**

```
package com.greatfree.testing.message;

import java.util.Set;

import com.greatfree.multicast.BroadcastResponse;

/*
 * This is a response to the broadcast request, SearchKeywordBroadcastRequest. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordBroadcastResponse extends BroadcastResponse
{
  private static final long serialVersionUID = 727494758786064640L;

  // The retrieved results, hyperlinks. 11/29/2014, Bing Li
  private Set<String> links;

  /*
   * Initialize. 11/29/2014, Bing Li
   */
  public SearchKeywordBroadcastResponse(Set<String> links, String key, String collaboratorKey)
  {
    super(MessageType.SEARCH_KEYWORD_BROADCAST_RESPONSE, key, collaboratorKey);
    this.links = links;
  }

  /*
   * Expose the links. 11/29/2014, Bing Li
   */
  public Set<String> getLinks()
  {
    return this.links;
  }
}
```

- **SearchKeywordRequest**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * This is a search request to be sent to the coordinator. It must raise a broadcast retrieval within the cluster under the control of the coordinator. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordRequest extends ServerMessage
{
	private static final long serialVersionUID = 98392936894809O771L;

	// The keyword to be retrieved. 11/29/2014, Bing Li
	private String keyword;

	/*
	 * Initialize the request. 11/29/2014, Bing Li
	 */
	public SearchKeywordRequest(String keyword)
	{
		super(MessageType.SEARCH_KEYWORD_REQUEST);
		this.keyword = keyword;
	}

	/*
	 * Expose the keyword. 11/29/2014, Bing Li
	 */
	public String getKeyword()
	{
		return this.keyword;
	}
}
```

- **SearchKeywordResponse**

```
package com.greatfree.testing.message;

import java.util.Set;

import com.greatfree.multicast.ServerMessage;

/*
 * This is a response to the request, SearchKeywordResponse. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordResponse extends ServerMessage
{
  private static final long serialVersionUID = -341670699036102757L;

  // The keyword retrieved. 11/29/2014, Bing Li
  private String keyword;
  // The links which include the keyword. 11/29/2014, Bing Li
  private Set<String> links;

  /*
   * Initialize. 11/29/2014, Bing Li
   */
  public SearchKeywordResponse(String keyword, Set<String> links)
  {
    super(MessageType.SEARCH_KEYWORD_RESPONSE);
    this.keyword = keyword;
    this.links = links;
  }

  /*
   * Expose the keyword. 11/29/2014, Bing Li
   */
  public String getKeyword()
  {
    return this.keyword;
  }

  /*
   * Expose the links. 11/29/2014, Bing Li
   */
  public Set<String> getLinks()
  {
    return this.links;
  }
}
```

- **SearchKeywordStream**

```
package com.greatfree.testing.message;

import java.io.ObjectOutputStream;
import java.util.concurrent.locks.Lock;

import com.greatfree.remote.OutMessageStream;

/*
 * The class is derived from OutMessageStream. It contains the received request, its associated output stream and the lock that keeps the responding operations atomic. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordStream extends OutMessageStream<SearchKeywordRequest>
{
  public SearchKeywordStream(ObjectOutputStream out, Lock lock, SearchKeywordRequest message)
  {
    super(out, lock, message);
  }
}
```

- **SignUpRequest**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * The class is an example of a request for signing up. It must derive from the base class,
ServerMessage. 09/21/2014, Bing Li
 */

// Created: 09/21/2014, Bing Li
public class SignUpRequest extends ServerMessage
{
  private static final long serialVersionUID = -4781998022159903485L;

  // Data in the request to be sent to the polling server. 09/21/2014, Bing Li
  private String userName;
  // Data in the request to be sent to the polling server. 09/21/2014, Bing Li
  private String password;

  /*
   * Initialize. 09/21/2014, Bing Li
   */
  public SignUpRequest(String userName, String password)
  {
    super(MessageType.SIGN_UP_REQUEST);
    this.userName = userName;
    this.password = password;
  }

  /*
   * Expose the userName. 09/21/2014, Bing Li
   */
  public String getUserName()
  {
    return this.userName;
  }

  /*
   * Expose the password. 09/21/2014, Bing Li
   */
  public String getPassword()
  {
    return this.password;
  }
}
```

- **SignUpResponse**

```
package com.greatfree.testing.message;

import com.greatfree.multicast.ServerMessage;

/*
 * The class is an example of the response, which must derive from the base class, ServerMessage.
09/21/2014, Bing Li
 */

// Created: 09/21/2014, Bing Li
public class SignUpResponse extends ServerMessage
{
  private static final long serialVersionUID = -3347732482627702119L;

  // The data that must be transmitted from the polling server to the client. 09/21/2014, Bing Li
  private boolean isSucceeded;

  /*
   * Initialize. 09/21/2014, Bing Li
   */
  public SignUpResponse(boolean isSucceeded)
  {
    super(MessageType.SIGN_UP_RESPONSE);
    this.isSucceeded = isSucceeded;
  }

  /*
   * Expose the responded data. 09/21/2014, Bing Li
   */
  public boolean isSucceeded()
  {
    return this.isSucceeded;
  }
}
```

- **SignUpStream**

```
package com.greatfree.testing.message;

import java.io.ObjectOutputStream;
import java.util.concurrent.locks.Lock;

import com.greatfree.remote.OutMessageStream;

/*
 * The class is derived from OutMessageStream. It contains the received sign up request, its associated output stream and the lock that keeps the responding operations atomic. 09/22/2014, Bing Li
 */

// Created: 09/22/2014, Bing Li
public class SignUpStream extends OutMessageStream<SignUpRequest>
{
   // Initialize the instance of the request stream. 09/22/2014, Bing Li
   public SignUpStream(ObjectOutputStream out, Lock lock, SignUpRequest request)
   {
      super(out, lock, request);
   }
}
```

3.4 An Ordinary Client

- **StartClient**

```
package com.greatfree.testing.client;

import java.io.IOException;
import java.util.Scanner;

import com.greatfree.testing.data.ServerConfig;
import com.greatfree.util.TerminateSignal;

/*
 * This is a client that interacts with the polling server through the manner of requesting. The polling server can only respond to the client after it requests. 09/21/2014, Bing Li
 */

// Created: 09/21/2014, Bing Li
public class StartClient
{
  /*
   * The starting point of the client. 09/21/2014, Bing Li
   */
  public static void main(String[] args)
  {
    // Start the client server. 11/08/2014, Bing Li
    ClientServer.CLIENT().start(ServerConfig.CLIENT_PORT);

    // Initialize the option which represents a user's intents of operations. 09/21/2014, Bing Li
    int option = MenuOptions.NO_OPTION;

    // Initialize a command input console for users to interact with the system. 09/21/2014, Bing Li
    Scanner in = new Scanner(System.in);
    String optionStr;

    // Keep the loop running to interact with users until an end option is selected. 09/21/2014, Bing Li
    while (option != MenuOptions.END)
    {
      // Display the menu to users. 09/21/2014, Bing Li
      ClientUI.FACE().printMenu();
      // Input a string that represents users' intents. 09/21/2014, Bing Li
      optionStr = in.nextLine();
      try
      {
        // Convert the input string to integer. 09/21/2014, Bing Li
        option = Integer.parseInt(optionStr);
        System.out.println("Your choice: " + option);

        // Send the option to the polling server. 09/21/2014, Bing Li
        ClientUI.FACE().send(option);
      }
      catch (NumberFormatException e)
      {
        option = MenuOptions.NO_OPTION;
        System.out.println(ClientMenu.WRONG_OPTION);
      }
    }

    // Set the terminating flag to true. 09/21/2014, Bing Li
    TerminateSignal.SIGNAL().setTerminated();

    try
    {
      // Stop the client server. 11/08/2014, Bing Li
      ClientServer.CLIENT().stop();
    }
    catch (InterruptedException e)
    {
```

```
            e.printStackTrace();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

- **ClientServer**

```
package com.greatfree.testing.client;

import java.io.IOException;
import java.net.ServerSocket;
import java.util.ArrayList;
import java.util.List;

import com.greatfree.concurrency.Runner;
import com.greatfree.remote.RemoteReader;
import com.greatfree.testing.data.ClientConfig;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.util.UtilConfig;

/*
 * The server aims to form a peer-to-peer architecture with a remote server. Another general usage is to set up ObjectInputStream, which needs to be accomplished after receiving feedback from the remote end. 11/07/2014, Bing Li
 */

// Created: 11/07/2014, Bing Li
public class ClientServer
{
   // A socket that waits for connections from remote nodes. In this case, it is used to wait for connections from the server. 11/23/2014, Bing Li
   private ServerSocket serverSocket;
   // The port that is open to remote nodes. 11/23/2014, Bing Li
   private int port;
   // Multiple threads waiting for remote connections. 11/23/2014, Bing Li
   private List<Runner<ClientListener, ClientListenerDisposer>> listenerRunners;

   private ClientServer()
   {
   }

   /*
    * A singleton definition. 11/23/2014, Bing Li
    */
   private static ClientServer instance = new ClientServer();

   public static ClientServer CLIENT()
   {
      if (instance == null)
      {
         instance = new ClientServer();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Start the client. 11/23/2014, Bing Li
    */
   public void start(int port)
   {
      // On JD7, the sorting algorithm is replaced with TimSort rather than MargeSort. To run correctly, it is necessary to use the old one. the following line sets that up. 10/03/2014, Bing Li
      System.setProperty(UtilConfig.MERGE_SORT, UtilConfig.TRUE);

      // Set the port number. 11/23/2014, Bing Li
      this.port = port;
      // Initialize a disposer. 11/23/2014, Bing Li
```

```
    ClientListenerDisposer disposer = new ClientListenerDisposer();
    // Initialize a list to take all of the listeners to wait for remote connections concurrently. 11/23/2014,
Bing Li
    this.listenerRunners = new ArrayList<Runner<ClientListener, ClientListenerDisposer>>();
    // The runner is responsible for starting to wait for connections asynchronously. 11/23/2014, Bing Li
    Runner<ClientListener, ClientListenerDisposer> listenerRunner;
    try
    {
      // Initialize the socket to wait for remote connections. 11/23/2014, Bing Li
      this.serverSocket = new ServerSocket(this.port);
      // Start a bunch of threads to listen to connections. 11/23/2014, Bing Li
      for (int i = 0; i < ServerConfig.MAX_CLIENT_LISTEN_THREAD_COUNT; i++)
      {
        // Initialize the runner which contains the listener. 11/23/2014, Bing Li
        listenerRunner = new Runner<ClientListener, ClientListenerDisposer>(new
ClientListener(this.serverSocket), disposer, true);
        // Put the runner into a list for management. 11/23/2014, Bing Li
        this.listenerRunners.add(listenerRunner);
        // Start the runner. 11/23/2014, Bing Li
        listenerRunner.start();
      }
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }

    // Initialize the server IO registry. 11/23/2014, Bing Li
    ClientServerIORegistry.REGISTRY().init();
    // Initialize the client pool. 11/23/2014, Bing Li
    ClientPool.LOCAL().init();

    // Initialize the message producer to dispatcher messages. 11/23/2014, Bing Li
    ClientServerMessageProducer.CLIENT().init();

    // Initialize the eventer to notify the remote server. 11/23/2014, Bing Li
    ClientEventer.NOTIFY().init(ServerConfig.SERVER_IP, ServerConfig.SERVER_PORT);
    // Initialize the remote reader to send requests and receive responses from the remote server.
11/23/2014, Bing Li
    RemoteReader.REMOTE().init(ClientConfig.SERVER_CLIENT_POOL_SIZE);

    try
    {
      // The line tries to connect the CServer. Its IP and port number are saved on the server for that.
Then, the RetrievablePool can work on the NodeKey to retrieve the IP and the port number. 10/03/2014,
Bing Li
      ClientEventer.NOTIFY().notifyOnline();
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }

    // Register the client on the remote server. 11/23/2014, Bing Li
    ClientEventer.NOTIFY().register();
  }

  /*
   * Stop the client. 11/23/2014, Bing Li
   */
  public void stop() throws InterruptedException, IOException
  {
    // Close the listeners. 11/23/2014, Bing Li
    for (Runner<ClientListener, ClientListenerDisposer> runner : this.listenerRunners)
```

```
        {
            runner.stop(ClientConfig.TIME_TO_WAIT_FOR_THREAD_TO_DIE);
        }
        // Close the socket. 11/23/2014, Bing Li
        this.serverSocket.close();

        // Dispose the message producer. 11/23/2014, Bing Li
        ClientServerMessageProducer.CLIENT().dispose();

        // Dispose the client pool. 11/23/2014, Bing Li
        ClientPool.LOCAL().dispose();
        // Dispose the eventer. 11/23/2014, Bing Li
        ClientEventer.NOTIFY().dispose();
        // Shutdown the remote reader. 11/23/2014, Bing Li
        RemoteReader.REMOTE().shutdown();
        // Dispose the server IO registry. 11/23/2014, Bing Li
        ClientServerIORegistry.REGISTRY().dispose();
    }
}
```

- **ClientListener**

```
package com.greatfree.testing.client;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

import com.greatfree.remote.ServerListener;
import com.greatfree.testing.data.ClientConfig;

/*
 * The listener holds on waiting for connections from the remote end. 07/30/2014, Bing Li
 */

// Created: 11/07/2014, Bing Li
public class ClientListener extends ServerListener implements Runnable
{
  /*
   * Initialize the listener. The instance of ServerSocket is used to wait for connections with remote ends. 11/07/2014, Bing Li
   */
  public ClientListener(ServerSocket serverSocket)
  {
    super(serverSocket, ClientConfig.CLIENT_LISTENER_THREAD_POOL_SIZE, ClientConfig.CLIENT_LISTENER_THREAD_ALIVE_TIME);
  }

  /*
   * The connection waiting is executed asynchronously. 11/07/2014, Bing Li
   */
  @Override
  public void run()
  {
    // Declare the instance of client socket. 11/07/2014, Bing Li
    Socket clientSocket;
    // Declare the instance of ClientServerIO. 11/07/2014, Bing Li
    ClientServerIO serverIO;
    // The listener should be always keeping alive unless the client is shutdown. 11/07/2014, Bing Li
    while (!super.isShutdown())
    {
      try
      {
        // Wait for remote connections. 11/07/2014, Bing Li
        clientSocket = super.accept();
        // Check if the connections reach the upper limit. If the total count of connections exceeds the maximum count, it is required to wait until a connection is disconnected. It attempts to protect the resources from being used up. 11/07/2014, Bing Li
        if (ClientServerIORegistry.REGISTRY().getIOCount() >= ClientConfig.SERVER_IO_POOL_SIZE)
        {
          try
          {
            // Wait until a connection is disconnected. 11/07/2014, Bing Li
            super.holdOn();
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
        }
        // Once if a connection is constructed, initialize the instance of ClientServerIO with the instance of Socket. 11/07/2014, Bing Li
        serverIO = new ClientServerIO(clientSocket, super.getCollaborator());
        // Add the instance of ClientServerIO to the registry for management. 11/07/2014, Bing Li
        ClientServerIORegistry.REGISTRY().addIO(serverIO);
```

```
            // Start the instance of ClientServerIO asynchronously to wait for messages for further
processing. 11/07/2014, Bing Li
            super.execute(serverIO);
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
        }
      }
    }
```

- **ClientListenerDisposer**

```
package com.greatfree.testing.client;

import com.greatfree.reuse.RunDisposable;

/*
 * This is an implementation of RunDisposable interface to shutdown ClientListener. 11/07/2014, Bing Li
 */

// Created: 11/07/2014, Bing Li
public class ClientListenerDisposer implements RunDisposable<ClientListener>
{
  @Override
  public void dispose(ClientListener r)
  {
    r.shutdown();
  }

  @Override
  public void dispose(ClientListener r, long time)
  {
    r.shutdown();
  }
}
```

- **ClientServerIORegistry**

```
package com.greatfree.testing.client;

import java.io.IOException;
import java.util.Set;

import com.greatfree.remote.ServerIORegistry;

/*
 * The registry keeps all of the connections' server IOs. 11/30/2014, Bing Li
 */

// Created: 11/07/2014, Bing Li
public class ClientServerIORegistry
{
  // Declare an instance of ServerIORegistry for ClientServerIOs. 11/07/2014, Bing Li
  private ServerIORegistry<ClientServerIO> registry;

  /*
   * Initializing ... 11/07/2014, Bing Li
   */
  private ClientServerIORegistry()
  {
  }

  /*
   * Define a singleton for the registry. 11/07/2014, Bing Li
   */
  private static ClientServerIORegistry instance = new ClientServerIORegistry();

  public static ClientServerIORegistry REGISTRY()
  {
    if (instance == null)
    {
      instance = new ClientServerIORegistry();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the registry. 11/07/2014, Bing Li
   */
  public void dispose() throws IOException
  {
    this.registry.removeAllIOs();
  }

  /*
   * Initialize the registry. 11/07/2014, Bing Li
   */
  public void init()
  {
    this.registry = new ServerIORegistry<ClientServerIO>();
  }

  /*
   * Add a new instance of ClientServerIO to the registry. 11/07/2014, Bing Li
   */
  public void addIO(ClientServerIO io)
  {
    this.registry.addIO(io);
```

```
  }

  /*
   * Get all of the IPs of the connected clients from the corresponding ClientServerIOs. 11/07/2014, Bing
Li
   */
  public Set<String> getIPs()
  {
    return this.registry.getIPs();
  }

  /*
   * Get the count of the registered ClientServerIOs. 11/07/2014, Bing Li
   */
  public int getIOCount()
  {
    return this.registry.getIOCount();
  }

  /*
   * Remove or unregister an ClientServerIO. It is executed when a client is down or the connection gets
something wrong. 11/07/204, Bing Li
   */
  public void removeIO(ClientServerIO io) throws IOException
  {
    this.registry.removeIO(io);
  }

  /*
   * Remove or unregister all of the registered ClientServerIOs. It is executed when the server process is
shutdown. 11/07/2014, Bing Li
   */
  public void removeAllIOs() throws IOException
  {
    this.registry.removeAllIOs();
  }
}
```

- **ClientServerIO**

```
package com.greatfree.testing.client;

import java.io.IOException;
import java.net.Socket;
import java.net.SocketException;

import com.greatfree.concurrency.Collaborator;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.remote.ServerIO;

/*
 * This is an implementation of ServerIO to receive and even respond remote ends. In this case of the client, it only receives feedbacks (notifications) from a remote server to set up the local ObjectInputStream. 11/07/2014, Bing Li
 */

// Created: 11/07/2014, Bing Li
public class ClientServerIO extends ServerIO
{
  // Initialize the server IO. 11/07/2014, Bing Li
  public ClientServerIO(Socket clientSocket, Collaborator collaborator)
  {
    super(clientSocket, collaborator);
  }

  /*
   * A concurrent running thread to receive and respond the received messages asynchronously. 11/07/2014, Bing Li
   */
  public void run()
  {
    ServerMessage message;
    while (!super.isShutdown())
    {
      try
      {
        // Wait and read messages from a client. 11/07/2014, Bing Li
        message = (ServerMessage)super.read();
        // Convert the received message to OutMessageStream and put it into the relevant dispatcher for concurrent processing. 11/07/2014, Bing Li
        ClientServerMessageProducer.CLIENT().produceMessage(new OutMessageStream<ServerMessage>(super.getOutStream(), super.getLock(), message));
      }
      catch (SocketException e)
      {
        e.printStackTrace();
      }
      catch (IOException e)
      {
        e.printStackTrace();
      }
      catch (ClassNotFoundException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **ClientServerMessageProducer**

```
package com.greatfree.testing.client;

import com.greatfree.concurrency.MessageProducer;
import com.greatfree.concurrency.Threader;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.testing.data.ServerConfig;

/*
 * The class is a singleton to enclose the instances of MessageProducer. Each of the enclosed message producers serves for one particular server that connects to a respective port on the client. Usually, each port aims to provide one particular service. 11/07/2014, Bing Li
 *
 * The class is a wrapper that encloses all of the asynchronous message producers. It is responsible for assigning received messages to the corresponding producer in an asynchronous way. 11/07/2014, Bing Li
 */

// Created: 11/07/2014, Bing Li
public class ClientServerMessageProducer
{
   // The Threader aims to associate with the message producer to guarantee the producer can work concurrently. 11/07/2014, Bing Li
   private Threader<MessageProducer<ClientServerDispatcher>, ClientServerDispatcherDisposer> producerThreader;

   private ClientServerMessageProducer()
   {
   }

   /*
    * The class is required to be a singleton since it is nonsense to initiate it for the producers are unique. 11/07/2014, Bing Li
    */
   private static ClientServerMessageProducer instance = new ClientServerMessageProducer();

   public static ClientServerMessageProducer CLIENT()
   {
      if (instance == null)
      {
         instance = new ClientServerMessageProducer();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Dispose the producers when the process of the server is shutdown. 11/07/2014, Bing Li
    */
   public void dispose() throws InterruptedException
   {
      this.producerThreader.stop();
   }

   /*
    * Initialize the message producers. It is invoked when the connection modules of the server is started since clients can send requests or notifications only after it is started. 11/07/2014, Bing Li
    */
   public void init()
   {
      // Initialize the message producer. A threader is associated with the message producer such that the
```

```
producer is able to work in a concurrent way. 09/20/2014, Bing Li
      this.producerThreader = new Threader<MessageProducer<ClientServerDispatcher>,
ClientServerDispatcherDisposer>(new MessageProducer<ClientServerDispatcher>(new
ClientServerDispatcher(ServerConfig.DISPATCHER_POOL_SIZE,
ServerConfig.DISPATCHER_POOL_THREAD_POOL_ALIVE_TIME)), new
ClientServerDispatcherDisposer());
      // Start the associated thread for the message producer. 09/20/2014, Bing Li
      this.producerThreader.start();
    }

    /*
     * Assign messages, requests or notifications, to the bound message dispatcher such that they can be
responded or dealt with. 11/07/2014, Bing Li
     */
    public void produceMessage(OutMessageStream<ServerMessage> message)
    {
      this.producerThreader.getFunction().produce(message);
    }
}
```

- **ClientServerDispatcher**

```
package com.greatfree.testing.client;

import com.greatfree.concurrency.NotificationDispatcher;
import com.greatfree.concurrency.ServerMessageDispatcher;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.testing.data.ClientConfig;
import com.greatfree.testing.message.InitReadFeedbackNotification;
import com.greatfree.testing.message.MessageType;
import com.greatfree.testing.message.NodeKeyNotification;

/*
 * The server dispatcher resides on the client rather than on the classic server. It has two goals. First, if the client is a peer end, i.e., a server plus a client, the dispatcher is required. Second, it helps a client to initialize instances of FreeClient to read remote data. The 2nd goal is needed in most clients. This is what it is done in the sample. 11/07/2014, Bing Li
 */

// Created: 11/07/2014, Bing Li
public class ClientServerDispatcher extends ServerMessageDispatcher<ServerMessage>
{
  // Declare a instance of notification dispatcher to deal with received the notification that contains the node key. 11/09/2014, Bing Li
  private NotificationDispatcher<NodeKeyNotification, RegisterThread, RegisterThreadCreator> nodeKeyNotificationDispatcher;

  // Declare a instance of notification dispatcher to deal with received the feedback for ObjectInputStream. 11/07/2014, Bing Li
  private NotificationDispatcher<InitReadFeedbackNotification, SetInputStreamThread, SetInputStreamThreadCreator> setInputStreamNotificationDispatcher;

  /*
   * Initialize the dispatcher. 11/07/2014, Bing Li
   */
  public ClientServerDispatcher(int corePoolSize, long keepAliveTime)
  {
    // Initialize the parent class. 11/07/2014, Bing Li
    super(corePoolSize, keepAliveTime);

    // Initialize the notification dispatcher for the notification, NodeKeyNotification. 11/09/2014, Bing Li
    this.nodeKeyNotificationDispatcher = new NotificationDispatcher<NodeKeyNotification, RegisterThread, RegisterThreadCreator>(ClientConfig.NOTIFICATION_DISPATCHER_POOL_SIZE, ClientConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new RegisterThreadCreator(), ClientConfig.MAX_NOTIFICATION_TASK_SIZE, ClientConfig.MAX_NOTIFICATION_THREAD_SIZE, ClientConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
    // Set the idle checking of the nodeKeyNotificationDispatcher. 11/09/2014, Bing Li
  this.nodeKeyNotificationDispatcher.setIdleChecker(ClientConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY, ClientConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
    // Start the server dispatcher of the nodeKeyNotificationDispatcher. 11/09/2014, Bing Li
    super.execute(this.nodeKeyNotificationDispatcher);

    // Initialize the notification dispatcher for the notification, InitReadFeedbackNotification. 11/07/2014, Bing Li
    this.setInputStreamNotificationDispatcher = new NotificationDispatcher<InitReadFeedbackNotification, SetInputStreamThread, SetInputStreamThreadCreator>(ClientConfig.NOTIFICATION_DISPATCHER_POOL_SIZE, ClientConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new SetInputStreamThreadCreator(), ClientConfig.MAX_NOTIFICATION_TASK_SIZE, ClientConfig.MAX_NOTIFICATION_THREAD_SIZE, ClientConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
    // Set the idle checking of the setInputStreamNotificationDispatcher. 11/07/2014, Bing Li
  this.setInputStreamNotificationDispatcher.setIdleChecker(ClientConfig.NOTIFICATION_DISPATCHE
```

```
R_IDLE_CHECK_DELAY, ClientConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
      // Start the server dispatcher of the setInputStreamNotificationDispatcher. 11/07/2014, Bing Li
      super.execute(this.setInputStreamNotificationDispatcher);
    }

    /*
     * Shutdown the dispatcher. 11/07/2014, Bing Li
     */
    public void shutdown()
    {
      // Shutdown the notification dispatcher for setting the node key. 11/07/2014, Bing Li
      this.nodeKeyNotificationDispatcher.dispose();
      // Shutdown the notification dispatcher for initializing ObjectInputStream. 11/07/2014, Bing Li
      this.setInputStreamNotificationDispatcher.dispose();
      // Shutdown the parent dispatcher. 11/07/2014, Bing Li
      super.shutdown();
    }

    /*
     * Dispatch received messages to corresponding threads respectively for concurrent processing.
11/07/2014, Bing Li
     */
    public void consume(OutMessageStream<ServerMessage> message)
    {
      // Detect the message type. 11/07/2014, Bing Li
      switch (message.getMessage().getType())
      {
        // Process the notification of NodeKeyNotification. 11/09/2014, Bing Li
        case MessageType.NODE_KEY_NOTIFICATION:
          // Enqueue the notification into the notification dispatcher. The notifications are queued and
processed asynchronously. 11/09/2014, Bing Li
          this.nodeKeyNotificationDispatcher.enqueue((NodeKeyNotification)message.getMessage());
          break;

        // Process the notification of the type, InitReadFeedbackNotification. 11/07/2014, Bing Li
        case MessageType.INIT_READ_FEEDBACK_NOTIFICATION:
          // Enqueue the notification into the notification dispatcher. The notifications are queued and
processed asynchronously. 11/07/2014, Bing Li

    this.setInputStreamNotificationDispatcher.enqueue((InitReadFeedbackNotification)message.getMess
age());
          break;
      }
    }
}
```

- **ClientServerDispatcherDisposer**

```
package com.greatfree.testing.client;

import com.greatfree.concurrency.MessageProducer;
import com.greatfree.reuse.ThreadDisposable;

/*
 * This is the disposer to dispose the instance of ClientServerDispatcher. It is usually executed when the client is shutdown. 11/07/2014, Bing Li
 */

// Created: 11/07/2014, Bing Li
public class ClientServerDispatcherDisposer implements ThreadDisposable<MessageProducer<ClientServerDispatcher>>
{
	/*
	 * Dispose the message producer. 11/07/2014, Bing Li
	 */
	@Override
	public void dispose(MessageProducer<ClientServerDispatcher> r)
	{
		r.dispose();
	}

	/*
	 * The method does not make sense to the class of MessageProducer. Just leave it here. 11/07/2014, Bing Li
	 */
	@Override
	public void dispose(MessageProducer<ClientServerDispatcher> r, long time)
	{
		r.dispose();
	}
}
```

- **RegisterThread**

```
package com.greatfree.testing.client;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.NodeKeyNotification;
import com.greatfree.util.NodeID;

/*
 * The thread starts to run when a node key notification is received. It keeps its unique key and registers itself to the server with the key. 11/09/2014, Bing Li
 */

// Created: 11/09/2014, Bing Li
public class RegisterThread extends NotificationQueue<NodeKeyNotification>
{
  /*
   * Initialize the thread. The argument, taskSize, is used to limit the count of tasks to be queued. 11/09/2014, Bing Li
   */
  public RegisterThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * Once if a node key notification is received, it is processed concurrently as follows. 11/09/2014, Bing Li
   */
  public void run()
  {
    // Declare an instance of NodeKeyNotification. 11/09/2014, Bing Li
    NodeKeyNotification notification;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/09/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/09/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/09/2014, Bing Li
          notification = this.getNotification();
          // Set the client key. 11/09/2014, Bing Li
          NodeID.DISTRIBUTED().setKey(notification.getNodeKey());
          // Register the client after getting the key. 11/09/2014, Bing Li
          ClientEventer.NOTIFY().register();
          // Dispose the notification. 11/09/2014, Bing Li
          this.disposeMessage(notification);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      try
      {
        // Wait for a moment after all of the existing notifications are processed. 11/09/2014, Bing Li
        this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
```

```
    }
}
```

- **RegisterThreadCreator**

```
package com.greatfree.testing.client;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.NodeKeyNotification;

/*
 * The code here attempts to create instances of RegisterThread. It is used by the notification dispatcher.
11/09/2014, Bing Li
 */

// Created: 11/09/2014, Bing Li
public class RegisterThreadCreator implements NotificationThreadCreatable<NodeKeyNotification,
RegisterThread>
{
  // Create the instance of RegisterThread. 11/09/2014, Bing Li
  @Override
  public RegisterThread createNotificationThreadInstance(int taskSize)
  {
    return new RegisterThread(taskSize);
  }
}
```

- **SetInputStreamThread**

```
package com.greatfree.testing.client;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.remote.RemoteReader;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.InitReadFeedbackNotification;

/*
 * The notification thread intends to set up the ObjectInputStream of a FreeClient instance after receiving
the relevant feedback notification from the remote server. 11/07/2014, Bing Li
 */

// Created: 11/07/2014, Bing Li
public class SetInputStreamThread extends NotificationQueue<InitReadFeedbackNotification>
{
  // Initialize. 11/07/2014, Bing Li
  public SetInputStreamThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * The thread to process notifications asynchronously. 11/07/2014, Bing Li
   */
  public void run()
  {
    // Declare a notification instance. 11/07/2014, Bing Li
    InitReadFeedbackNotification notification;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/07/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/07/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/07/2014, Bing Li
          notification = this.getNotification();
          // Notify the instance of FreeClient that it is time to initialize the ObjectInputStream.
11/07/2014, Bing Li
          RemoteReader.REMOTE().notifyOutStreamDone();
          // Dispose the notification. 11/07/2014, Bing Li
          this.disposeMessage(notification);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      try
      {
        // Wait for a moment after all of the existing notifications are processed. 11/07/2014, Bing Li
        this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **SetInputStreamThreadCreator**

```
package com.greatfree.testing.client;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.InitReadFeedbackNotification;

/*
 * The code here attempts to create instances of SetInputStreamThread. 11/07/2014, Bing Li
 */

// Created: 11/07/2014, Bing Li
public class SetInputStreamThreadCreator implements
NotificationThreadCreatable<InitReadFeedbackNotification, SetInputStreamThread>
{
  // Create the instance of SetInputStreamThread. 11/09/2014, Bing Li
  @Override
  public SetInputStreamThread createNotificationThreadInstance(int taskSize)
  {
    return new SetInputStreamThread(taskSize);
  }
}
```

- **ClientPool**

```
package com.greatfree.testing.client;

import java.io.IOException;

import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.data.ClientConfig;

/*
 * This is a class that manages all of instances of FreeClient to connect to the remote server.
09/21/2014, Bing Li
 */

// Created: 09/21/2014, Bing Li
public class ClientPool
{
  // Define an instance of FreeClientPool. 09/21/2014, Bing Li
  private FreeClientPool pool;

  /*
   * Initialize. 09/21/2014, Bing Li
   */
  private ClientPool()
  {
  }

  /*
   * A singleton definition. 11/23/2014, Bing Li
   */
  private static ClientPool instance = new ClientPool();

  public static ClientPool LOCAL()
  {
    if (instance == null)
    {
      instance = new ClientPool();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the free client pool. 11/23/2014, Bing Li
   */
  public void dispose() throws IOException
  {
    this.pool.dispose();
  }

  /*
   * Initialize the client pool. The method is called when the client process is started. 09/17/2014, Bing Li
   */
  public void init()
  {
    // Initialize the client pool. 11/23/2014, Bing Li
    this.pool = new FreeClientPool(ClientConfig.CLIENT_POOL_SIZE);
    // Set idle checking for the client pool. 11/23/2014, Bing Li
    this.pool.setIdleChecker(ClientConfig.CLIENT_IDLE_CHECK_DELAY,
ClientConfig.CLIENT_IDLE_CHECK_PERIOD, ClientConfig.CLIENT_MAX_IDLE_TIME);
  }

  /*
```

```
    * Expose the client pool. 11/23/2014, Bing Li
    */
   public FreeClientPool getPool()
   {
      return this.pool;
   }
}
```

- **ClientEventer**

```
package com.greatfree.testing.client;

import java.io.IOException;

import com.greatfree.concurrency.ThreadPool;
import com.greatfree.remote.SyncRemoteEventer;
import com.greatfree.testing.data.ClientConfig;
import com.greatfree.testing.message.OnlineNotification;
import com.greatfree.testing.message.RegisterClientNotification;
import com.greatfree.testing.message.UnregisterClientNotification;
import com.greatfree.util.NodeID;

/*
 * The class is an example that applies SynchRemoteEventer and AsyncRemoteEventer. 11/05/2014,
Bing Li
 */

// Created: 11/05/2014, Bing Li
public class ClientEventer
{
  // Declare the ip of the remote server. 11/07/2014, Bing Li
  private String ip;
  // Declare the port of the remote server. 11/07/2014, Bing Li
  private int port;
  // The eventer to send the online notification. 11/07/2014, Bing Li
  private SyncRemoteEventer<OnlineNotification> onlineEventer;
  // The eventer to send the registering notification. 11/07/2014, Bing Li
  private SyncRemoteEventer<RegisterClientNotification> registerClientEventer;
  // The eventer to send the unregistering notification. 11/07/2014, Bing Li
  private SyncRemoteEventer<UnregisterClientNotification> unregisterClientEventer;

  // A thread pool to assist sending notification asynchronously. 11/07/2014, Bing Li
  private ThreadPool pool;

  /*
   * Initialize. 11/07/2014, Bing Li
   */
  private ClientEventer()
  {
  }

  /*
   * A singleton implementation. 11/07/2014, Bing Li
   */
  private static ClientEventer instance = new ClientEventer();

  public static ClientEventer NOTIFY()
  {
    if (instance == null)
    {
      instance = new ClientEventer();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the eventers. 11/07/2014, Bing Li
   */
  public void dispose()
  {
```

```
    this.onlineEventer.dispose();
    this.registerClientEventer.dispose();
    this.unregisterClientEventer.dispose();

    // Shutdown the thread pool. 11/07/2014, Bing Li
    this.pool.shutdown();
  }

  /*
   * Initialize the eventers. 11/07/2014, Bing Li
   */
  public void init(String ip, int port)
  {
    this.ip = ip;
    this.port = port;
    this.pool = new ThreadPool(ClientConfig.EVENTER_THREAD_POOL_SIZE,
ClientConfig.EVENTER_THREAD_POOL_ALIVE_TIME);
    this.onlineEventer = new SyncRemoteEventer<OnlineNotification>(ClientPool.LOCAL().getPool());
    this.registerClientEventer = new
SyncRemoteEventer<RegisterClientNotification>(ClientPool.LOCAL().getPool());
    this.unregisterClientEventer = new
SyncRemoteEventer<UnregisterClientNotification>(ClientPool.LOCAL().getPool());
  }

  /*
   * Send the online notification to the remote server. 11/07/2014, Bing Li
   */
  public void notifyOnline() throws IOException, InterruptedException
  {
    this.onlineEventer.notify(this.ip, this.port, new OnlineNotification());
  }

  /*
   * Send the registering notification to the remote server. 11/07/2014, Bing Li
   */
  public void register()
  {
    try
    {
      this.registerClientEventer.notify(this.ip, this.port, new
RegisterClientNotification(NodeID.DISTRIBUTED().getKey()));
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
  }

  /*
   * Send the unregistering notification to the remote server. 11/07/2014, Bing Li
   */
  public void unregister()
  {
    try
    {
      this.unregisterClientEventer.notify(this.ip, this.port, new
UnregisterClientNotification(NodeID.DISTRIBUTED().getKey()));
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
    catch (InterruptedException e)
    {
```

```
            e.printStackTrace();
        }
    }
}
```

- **ClientReader**

```
package com.greatfree.testing.client;

import java.io.IOException;

import com.greatfree.exceptions.RemoteReadException;
import com.greatfree.remote.RemoteReader;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.MessageConfig;
import com.greatfree.testing.message.SignUpRequest;
import com.greatfree.testing.message.SignUpResponse;
import com.greatfree.util.NodeID;

/*
 * The class wraps the class, RemoteReader, to send requests to the remote server and wait until
relevant responses are received. 09/22/2014, Bing Li
 */

// Created: 09/21/2014, Bing Li
public class ClientReader
{
   /*
    * Send the request of SignUpRequest to the polling server and wait for the response,
SignUpResponse. 09/22/2014, Bing Li
    */
   public static SignUpResponse signUp(String userName, String password)
   {
      // When the connection gets something wrong, a RemoteReadException is raised. 09/22/2014, Bing
Li
      try
      {
         return (SignUpResponse)(RemoteReader.REMOTE().read(NodeID.DISTRIBUTED().getKey(),
ServerConfig.SERVER_IP, ServerConfig.SERVER_PORT, new SignUpRequest(userName,
password)));
      }
      catch (RemoteReadException e)
      {
         e.printStackTrace();
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
      catch (ClassNotFoundException e)
      {
         e.printStackTrace();
      }
      // When reading gets something wrong, a null response is returned. 09/22/2014, Bing Li
      return MessageConfig.NO_SIGN_UP_RESPONSE;
   }
}
```

- **ClientUI**

```
package com.greatfree.testing.client;

import com.greatfree.testing.data.ClientConfig;
import com.greatfree.testing.message.SignUpResponse;

/*
 * The class aims to print a menu list on the screen for users to interact with the client and communicate with the polling server. The menu is unique in the client such that it is implemented in the pattern of a singleton. 09/21/2014, Bing Li
 */

// Created: 09/21/2014, Bing Li
public class ClientUI
{
  /*
   * Initialize. 09/21/2014, Bing Li
   */
  private ClientUI()
  {
  }

  /*
   * Initialize a singleton. 09/21/2014, Bing Li
   */
  private static ClientUI instance = new ClientUI();

  public static ClientUI FACE()
  {
    if (instance == null)
    {
      instance = new ClientUI();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Print the menu list on the screen. 09/21/2014, Bing Li
   */
  public void printMenu()
  {
    System.out.println(ClientMenu.MENU_HEAD);
    System.out.println(ClientMenu.SIGN_UP);
    System.out.println(ClientMenu.QUIT);
    System.out.println(ClientMenu.END);
    System.out.println(ClientMenu.MENU_TAIL);
    System.out.println(ClientMenu.INPUT_PROMPT);
  }

  /*
   * Send the users' option to the polling server. 09/21/2014, Bing Li
   */
  public void send(int option)
  {
    SignUpResponse signUpResponse;

    // Check the option to interact with the polling server. 09/21/2014, Bing Li
    switch (option)
    {
      // If the sign up option is selected, send the request message to the polling server. 09/21/2014, Bing Li
```

```
        case MenuOptions.SIGN_UP:
          signUpResponse = ClientReader.signUp(ClientConfig.USERNAME, ClientConfig.PASSWORD);
          System.out.println(signUpResponse.isSucceeded());
          break;

        // If the quit option is selected, send the notification message to the polling server. 09/21/2014,
Bing Li
        case MenuOptions.QUIT:
          break;
      }
    }
  }
```

- **MenuOptions**

```
package com.greatfree.testing.client;

/*
 * The class contains all of constants of menu options. 09/22/2014, Bing Li
 */

// Created: 09/21/2014, Bing Li
public class MenuOptions
{
   public final static int NO_OPTION = -1;
   public final static int END = 0;
   public final static int SIGN_UP = 1;
   public final static int QUIT = 2;
}
```

- **ClientMenu**

```
package com.greatfree.testing.client;

/*
 * This is a simple menu for the client which is operated by a human being. 11/30/2014, Bing Li
 */

// Created: 09/21/2014, Bing Li
public class ClientMenu
{
   public final static String TAB = "  ";
   public final static String MENU_HEAD = "\n========== Menu Head ==========";
   public final static String END = ClientMenu.TAB + "0) End";
   public final static String SIGN_UP = ClientMenu.TAB + "1) Sign Up";
   public final static String QUIT = ClientMenu.TAB + "2) Quit";
   public final static String MENU_TAIL = "========== Menu Tail ==========\n";
   public final static String INPUT_PROMPT = "Input an option:";

   public final static String WRONG_OPTION = "Wrong option!";
}
```

3.5 An Ordinary Server

- **StartServer**

```
package com.greatfree.testing.server;

import com.greatfree.testing.data.ServerConfig;
import com.greatfree.util.TerminateSignal;

/*
 * The class is the entry to the server process. 08/22/2014, Bing Li
 */

// Created: 07/17/2014, Bing Li
public class StartServer
{
  public static void main(String[] args)
  {
    // Start the server. 08/22/2014, Bing Li
    Server.FREE().start(ServerConfig.SERVER_PORT);

    // After the server is started, the loop check whether the flag of terminating is set. If the terminating flag is true, the process is ended. Otherwise, the process keeps running. 08/22/2014, Bing Li
    while (!TerminateSignal.SIGNAL().isTerminated())
    {
      try
      {
        // If the terminating flag is false, it is required to sleep for some time. Otherwise, it might cause the high CPU usage. 08/22/2014, Bing Li
        Thread.sleep(ServerConfig.TERMINATE_SLEEP);
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **Server**

```
package com.greatfree.testing.server;

import java.io.IOException;
import java.net.ServerSocket;
import java.util.ArrayList;
import java.util.List;

import com.greatfree.concurrency.Runner;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.util.TerminateSignal;

/*
 * The class is a testing case for relevant server classes. It is responsible for starting up and shutting down the server. Since it is the unique entry and exit of the server, it is implemented in the form of a singleton. 07/30/2014, Bing Li
 */

// Created: 07/17/2014, Bing Li
public class Server
{
   // The ServerSocket waits for clients' connecting. The socket serves the server in the sense that it not only responds to clients' requests but also notifies clients even without clients' requests. 08/10/2014, Bing Li
   private ServerSocket socket;
   // The port number for socket. 08/10/2014, Bing Li
   private int port;

   // The list keeps all of the threads that listen to connecting from clients of the server. When the server is shutdown, those threads can be killed to avoid possible missing. 08/10/2014, Bing Li
   private List<Runner<MyServerListener, MyServerListenerDisposer>> listenerRunnerList;

   /*
    * A singleton is designed for the server's startup and shutdown interface. 08/10/2014, Bing Li
    */
   private Server()
   {
   }

   private static Server instance = new Server();

   public static Server FREE()
   {
      if (instance == null)
      {
         instance = new Server();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Start the server and relevant listeners with concurrent threads for potential busy connecting. 08/10/2014, Bing Li
    */
   public void start(int port)
   {
      // Initialize and start the server sockets. 08/10/2014, Bing Li
      this.port = port;
      try
      {
         this.socket = new ServerSocket(this.port);
```

```
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }

    // Initialize a disposer which collects the server listener. 08/10/2014, Bing Li
    MyServerListenerDisposer disposer = new MyServerListenerDisposer();
    // Initialize the runner list. 11/25/2014, Bing Li
    this.listenerRunnerList = new ArrayList<Runner<MyServerListener,
MyServerListenerDisposer>>();

    // Start up the threads to listen to connecting from clients which send requests as well as receive
notifications. 08/10/2014, Bing Li
    Runner<MyServerListener, MyServerListenerDisposer> runner;
    for (int i = 0; i < ServerConfig.LISTENING_THREAD_COUNT; i++)
    {
      runner = new Runner<MyServerListener, MyServerListenerDisposer>(new
MyServerListener(this.socket, ServerConfig.LISTENER_THREAD_POOL_SIZE,
ServerConfig.LISTENER_THREAD_ALIVE_TIME), disposer, true);
      this.listenerRunnerList.add(runner);
      runner.start();
    }

    // Initialize the server IO registry. 11/07/2014, Bing Li
    MyServerIORegistry.REGISTRY().init();
    // Initialize a client pool, which is used by the server to connect to the remote end. 09/17/2014, Bing
Li
    ClientPool.SERVER().init();
  }

  /*
   * Shutdown the server. 08/10/2014, Bing Li
   */
  public void stop() throws IOException, InterruptedException
  {
    // Set the terminating signal. 11/25/2014, Bing Li
    TerminateSignal.SIGNAL().setTerminated();
    // Close the socket for the server. 08/10/2014, Bing Li
    this.socket.close();

    // Stop all of the threads that listen to clients' connecting to the server. 08/10/2014, Bing Li
    for (Runner<MyServerListener, MyServerListenerDisposer> runner : this.listenerRunnerList)
    {
      runner.stop();
    }

    // Shutdown the IO registry. 11/07/2014, Bing Li
    MyServerIORegistry.REGISTRY().dispose();

    // Shut down the client pool. 09/17/2014, Bing Li
    ClientPool.SERVER().dispose();
  }
}
```

- **MyServerListener**

```
package com.greatfree.testing.server;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

import com.greatfree.remote.IPPort;
import com.greatfree.remote.ServerListener;
import com.greatfree.testing.data.ServerConfig;

/*
 * The class is a server listener that not only responds to clients but also intends to connect and
communicate with the client. The class assists the local server to interact with clients in an eventing
manner other than waiting for requests and then responding only. 07/30/2014, Bing Li
 */

// Created: 07/30/2014, Bing Li
public class MyServerListener extends ServerListener implements Runnable
{
  // A thread to connect the remote client concurrently. 11/24/2014, Bing Li
  private ConnectClientThread connectThread;

  /*
   * Initialize the listener. 08/22/2014, Bing Li
   */
  public MyServerListener(ServerSocket serverSocket, int threadPoolSize, long keepAliveTime)
  {
    super(serverSocket, threadPoolSize, keepAliveTime);
  }

  /*
   * Shutdown the listener. 11/24/2014, Bing Li
   */
  public void shutdown()
  {
    // Dispose the connecting thread. 11/24/2014, Bing Li
    this.connectThread.dispose();
    // Shutdown the listener. 11/24/2014, Bing Li
    super.shutdown();
  }

  /*
   * The task that must be executed concurrently. 08/22/2014, Bing Li
   */
  @Override
  public void run()
  {
    Socket clientSocket;
    MyServerIO serverIO;

    // Detect whether the listener is shutdown. If not, it must be running all the time to wait for potential
connections from clients. 08/22/2014, Bing Li
    while (!super.isShutdown())
    {
      try
      {
        // Wait and accept a connecting from a possible client. 08/22/2014, Bing Li
        clientSocket = super.accept();
        // Check whether the connected server IOs exceed the upper limit. 08/22/2014, Bing Li
        if (MyServerIORegistry.REGISTRY().getIOCount() >=
ServerConfig.MAX_SERVER_IO_COUNT)
        {
          try
          {
```

```
                // If the upper limit is reached, the listener has to wait until an existing server IO is disposed. 08/22/2014, Bing Li
                super.holdOn();
              }
              catch (InterruptedException e)
              {
                e.printStackTrace();
              }
            }
            // If the upper limit of IOs is not reached, a server IO is initialized. A common Collaborator and the socket are the initial parameters. The shared common collaborator guarantees all of the server IOs from a certain client could notify with each other with the same lock. Then, the upper limit of server IOs is under the control. 08/22/2014, Bing Li
            serverIO = new MyServerIO(clientSocket, super.getCollaborator());

            /*
             * Since the listener servers need to form a peer-to-peer architecture, it is required to connect to the remote end. Thus, the local server can send messages without waiting for any requests from the remote end. 09/17/2014, Bing Li
             */

            // Check whether a client to the IP and the port number of the remote end is existed in the client pool. If such a client does not exist, it is required to connect the remote end by the thread concurrently. Doing that concurrently is to speed up the rate of responding to the client. 09/17/2014, Bing Li
            if (!ClientPool.SERVER().getPool().isClientExisted(serverIO.getIP(), ServerConfig.REMOTE_SERVER_PORT))
            {
              // Since the client does not exist in the pool, input the IP address and the port number to the thread. 09/17/2014, Bing Li
              this.connectThread.enqueue(new IPPort(serverIO.getIP(), ServerConfig.REMOTE_SERVER_PORT));
              // Execute the thread to connect to the remote end. 09/17/2014, Bing Li
              super.execute(this.connectThread);
            }

            // Add the new created server IO into the registry for further management. 08/22/2014, Bing Li
            MyServerIORegistry.REGISTRY().addIO(serverIO);
            // Execute the new created server IO concurrently to respond the client requests and notifications in an asynchronous manner. 08/22/2014, Bing Li
            super.execute(serverIO);
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
        }
      }
    }
```

- **MyServerListenerDisposer**

```
package com.greatfree.testing.server;

import com.greatfree.reuse.RunDisposable;

/*
 * The class is responsible for disposing the instance of MyServerListener by invoking its method of shutdown(). 09/20/2014, Bing Li
 */

// Created: 08/10/2014, Bing Li
public class MyServerListenerDisposer implements RunDisposable<MyServerListener>
{
  /*
   * Dispose the instance of MyServerListener. 09/20/2014, Bing Li
   */
  @Override
  public void dispose(MyServerListener r)
  {
    r.shutdown();
  }

  /*
   * Dispose the instance of MyServerListener. The method does not make sense to MyServerListener. Just leave it here for the requirement of the interface, RunDisposable<MyServerListener>. 09/20/2014, Bing Li
   */
  @Override
  public void dispose(MyServerListener r, long time)
  {
    r.shutdown();
  }
}
```

- **ConnectClientThread**

```
package com.greatfree.testing.server;

import java.io.IOException;
import java.util.Queue;
import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.remote.IPPort;
import com.greatfree.testing.message.NodeKeyNotification;

/*
 * The class is derived from Thread. It is responsible for connecting the remote client such that it is
feasible for the server to notify the client even without the client's request. 08/10/2014, Bing Li
 */

// Created: 08/10/2014, Bing Li
public class ConnectClientThread extends Thread
{
  // The queue keeps the clients' IPs which need to be connected to. 08/24/2014, Bing Li
  private Queue<IPPort> ipQueue;

  /*
   * Initialize. 09/19/2014, Bing Li
   */
  public ConnectClientThread()
  {
    this.ipQueue = new LinkedBlockingQueue<IPPort>();
  }

  /*
   * Dispose the resource of the class. 09/19/2014, Bing Li
   */
  public synchronized void dispose()
  {
    if (this.ipQueue != null)
    {
      this.ipQueue.clear();
      this.ipQueue = null;
    }
  }

  /*
   * Input the IP address and the port number into the queue. They are connected in the order of first-in-
first-out. The ClientPool is responsible for the connections. 09/19/2014, Bing Li
   */
  public void enqueue(IPPort ipPort)
  {
    this.ipQueue.add(ipPort);
  }

  /*
   * Connect the remote IP addresses and the associated port numbers concurrently. 09/19/2014, Bing
Li
   */
  public void run()
  {
    IPPort ipPort;
    // The thread keeps working until all of the IP addresses are connected. 09/19/2014, Bing Li
    while (this.ipQueue.size() > 0)
    {
      // Dequeue the IP addresses. 09/19/2014, Bing Li
      ipPort = this.ipQueue.poll();
      try
      {
        // Notify the remote node its unique ID. This is usually used to set up a multicasting cluster which
```

```
contains a bunch of nodes. Each of them is identified by the ID. 09/20/2014, Bing Li
            ClientPool.SERVER().getPool().send(ipPort, new NodeKeyNotification(ipPort.getObjectKey()));
        }
        catch (IOException e)
        {
          e.printStackTrace();
        }
      }
    }
}
```

- **MyServerIORegistry**

```
package com.greatfree.testing.server;

import java.io.IOException;
import java.util.Set;

import com.greatfree.remote.ServerIORegistry;

/*
 * The class keeps all of MyServerIOs. This is a singleton wrapper of ServerIORegistry. 08/22/2014,
Bing Li
 */

// Created: 08/22/2014, Bing Li
public class MyServerIORegistry
{
  // Declare an instance of ServerIORegistry for MyServerIOs. 08/22/2014, Bing Li
  private ServerIORegistry<MyServerIO> registry;

  /*
   * Initializing ... 08/22/2014, Bing Li
   */
  private MyServerIORegistry()
  {
  }

  private static MyServerIORegistry instance = new MyServerIORegistry();

  public static MyServerIORegistry REGISTRY()
  {
    if (instance == null)
    {
      instance = new MyServerIORegistry();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the registry. 08/22/2014, Bing Li
   */
  public void dispose() throws IOException
  {
    this.registry.removeAllIOs();
  }

  /*
   * Initialize the registry. 11/07/2014, Bing Li
   */
  public void init()
  {
    this.registry = new ServerIORegistry<MyServerIO>();
  }

  /*
   * Add a new instance of MyServerIO to the registry. 08/22/2014, Bing Li
   */
  public void addIO(MyServerIO io)
  {
    this.registry.addIO(io);
  }
```

```
  /*
   * Get all of the IPs of the connected clients from the corresponding MyServerIOs. 08/22/2014, Bing Li
   */
  public Set<String> getIPs()
  {
    return this.registry.getIPs();
  }

  /*
   * Get the count of the registered MyServerIOs. 08/22/2014, Bing Li
   */
  public int getIOCount()
  {
    return this.registry.getIOCount();
  }

  /*
   * Remove or unregister an MyServerIO. It is executed when a client is down or the connection gets
something wrong. 08/10/2014, Bing Li
   */
  public void removeIO(MyServerIO io) throws IOException
  {
    this.registry.removeIO(io);
  }

  /*
   * Remove or unregister all of the registered MyServerIOs. It is executed when the server process is
shutdown. 08/10/2014, Bing Li
   */
  public void removeAllIOs() throws IOException
  {
    this.registry.removeAllIOs();
  }
}
```

- **MyServerIO**

```
package com.greatfree.testing.server;

import java.io.IOException;
import java.net.Socket;
import java.net.SocketException;

import com.greatfree.concurrency.Collaborator;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.remote.ServerIO;

/*
 * The class is actually an implementation of ServerIO, which serves for clients which access the server. 07/30/2014, Bing Li
 */

// Created: 08/10/2014, Bing Li
public class MyServerIO extends ServerIO
{
  /*
   * Initialize the server IO. The socket is the connection between the client and the server. The collaborator is shared with other IOs to control the count of ServerIOs instances. 11/23/2014, Bing Li
   */
  public MyServerIO(Socket clientSocket, Collaborator collaborator)
  {
    super(clientSocket, collaborator);
  }

  /*
   * A concurrent method to respond the received messages asynchronously. 08/22/2014, Bing Li
   */
  public void run()
  {
    ServerMessage message;
    while (!super.isShutdown())
    {
      try
      {
        // Wait and read messages from a client. 08/22/2014, Bing Li
        message = (ServerMessage)super.read();
        // Convert the received message to OutMessageStream and put it into the relevant dispatcher for concurrent processing. 09/20/2014, Bing Li
        MyServerMessageProducer.SERVER().produceMessage(new OutMessageStream<ServerMessage>(super.getOutStream(), super.getLock(), message));
      }
      catch (SocketException e)
      {
        e.printStackTrace();
      }
      catch (IOException e)
      {
        e.printStackTrace();
      }
      catch (ClassNotFoundException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **MyServerMessageProducer**

```
package com.greatfree.testing.server;

import com.greatfree.concurrency.MessageProducer;
import com.greatfree.concurrency.Threader;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.testing.data.ServerConfig;

/*
 * The class is a singleton to enclose the instances of MessageProducer. Each of the enclosed message producers serves for one particular client that connects to a respective port on the server. Usually, each port aims to provide one particular service. 09/20/2014, Bing Li
 *
 * The class is a wrapper that encloses all of the asynchronous message producers. It is responsible for assigning received messages to the corresponding producer in an asynchronous way. 08/22/2014, Bing Li
 */

// Created: 08/04/2014, Bing Li
public class MyServerMessageProducer
{
   // The Threader aims to associate with the message producer to guarantee the producer can work concurrently. 09/20/2014, Bing Li
   private Threader<MessageProducer<MyServerDispatcher>, MyServerProducerDisposer> producerThreader;

   private MyServerMessageProducer()
   {
   }

   /*
    * The class is required to be a singleton since it is nonsense to initiate it for the producers are unique. 08/22/2014, Bing Li
    */
   private static MyServerMessageProducer instance = new MyServerMessageProducer();

   public static MyServerMessageProducer SERVER()
   {
      if (instance == null)
      {
         instance = new MyServerMessageProducer();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Dispose the producers when the process of the server is shutdown. 08/22/2014, Bing Li
    */
   public void dispose() throws InterruptedException
   {
      this.producerThreader.stop();
   }

   /*
    * Initialize the message producers. It is invoked when the connection modules of the server is started since clients can send requests or notifications only after it is started. 08/22/2014, Bing Li
    */
   public void init()
   {
      // Initialize the message producer. A threader is associated with the message producer such that the
```

```
producer is able to work in a concurrent way. 09/20/2014, Bing Li
      this.producerThreader = new Threader<MessageProducer<MyServerDispatcher>,
MyServerProducerDisposer>(new MessageProducer<MyServerDispatcher>(new
MyServerDispatcher(ServerConfig.DISPATCHER_POOL_SIZE,
ServerConfig.DISPATCHER_POOL_THREAD_POOL_ALIVE_TIME)), new
MyServerProducerDisposer());
      // Start the associated thread for the message producer. 09/20/2014, Bing Li
      this.producerThreader.start();
    }

    /*
     * Assign messages, requests or notifications, to the bound message dispatcher such that they can be
responded or dealt with. 09/20/2014, Bing Li
     */
    public void produceMessage(OutMessageStream<ServerMessage> message)
    {
      this.producerThreader.getFunction().produce(message);
    }
}
```

- **MyServerDispatcher**

```
package com.greatfree.testing.server;

import com.greatfree.concurrency.NotificationDispatcher;
import com.greatfree.concurrency.RequestDispatcher;
import com.greatfree.concurrency.ServerMessageDispatcher;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.InitReadNotification;
import com.greatfree.testing.message.MessageType;
import com.greatfree.testing.message.RegisterClientNotification;
import com.greatfree.testing.message.SignUpRequest;
import com.greatfree.testing.message.SignUpResponse;
import com.greatfree.testing.message.SignUpStream;

/*
 * This is an implementation of ServerMessageDispatcher. It contains the concurrency mechanism to
respond clients' requests and receive clients' notifications for the server. 09/20/2014, Bing Li
 */

// Created: 09/20/2014, Bing Li
public class MyServerDispatcher extends ServerMessageDispatcher<ServerMessage>
{
   // Declare a notification dispatcher to process the registration notification concurrently. 11/04/2014,
Bing Li
   private NotificationDispatcher<RegisterClientNotification, RegisterClientThread,
RegisterClientThreadCreator> registerClientNotificationDispatcher;
   // Declare a request dispatcher to respond users sign-up requests concurrently. 11/04/2014, Bing Li
   private RequestDispatcher<SignUpRequest, SignUpStream, SignUpResponse, SignUpThread,
SignUpThreadCreator> signUpRequestDispatcher;
   // Declare a notification dispatcher to deal with instances of InitReadNotification from a client
concurrently such that the client can initialize its ObjectInputStream. 11/09/2014, Bing Li
   private NotificationDispatcher<InitReadNotification, InitReadFeedbackThread,
InitReadFeedbackThreadCreator> initReadFeedbackNotificationDispatcher;

   /*
    * Initialize. 09/20/2014, Bing Li
    */
   public MyServerDispatcher(int corePoolSize, long keepAliveTime)
   {
      // Set the pool size and threads' alive time. 11/04/2014, Bing Li
      super(corePoolSize, keepAliveTime);

      // Initialize the notification dispatcher. 11/30/2014, Bing Li
      this.registerClientNotificationDispatcher = new NotificationDispatcher<RegisterClientNotification,
RegisterClientThread,
RegisterClientThreadCreator>(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new
RegisterClientThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
      // Set the idle checking. 11/30/2014, Bing Li

   this.registerClientNotificationDispatcher.setIdleChecker(ServerConfig.NOTIFICATION_DISPATCHE
R_IDLE_CHECK_DELAY, ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
      // Start the notification dispatcher. 11/30/2014, Bing Li
      super.execute(this.registerClientNotificationDispatcher);

      // Initialize the sign up dispatcher. 11/04/2014, Bing Li
      this.signUpRequestDispatcher = new RequestDispatcher<SignUpRequest, SignUpStream,
SignUpResponse, SignUpThread,
SignUpThreadCreator>(ServerConfig.REQUEST_DISPATCHER_POOL_SIZE,
ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME, new SignUpThreadCreator(),
ServerConfig.MAX_REQUEST_TASK_SIZE, ServerConfig.MAX_REQUEST_THREAD_SIZE,
```

```
ServerConfig.REQUEST_DISPATCHER_WAIT_TIME);
      // Set the parameters to check idle states of threads. 11/04/2014, Bing Li

    this.signUpRequestDispatcher.setIdleChecker(ServerConfig.REQUEST_DISPATCHER_IDLE_CHEC
K_DELAY, ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD);
      // Start the dispatcher. 11/04/2014, Bing Li
      super.execute(this.signUpRequestDispatcher);

      // Initialize the notification dispatcher. 11/30/2014, Bing Li
      this.initReadFeedbackNotificationDispatcher = new NotificationDispatcher<InitReadNotification,
InitReadFeedbackThread,
InitReadFeedbackThreadCreator>(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new
InitReadFeedbackThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
      // Set the idle checking. 11/30/2014, Bing Li

    this.initReadFeedbackNotificationDispatcher.setIdleChecker(ServerConfig.NOTIFICATION_DISPAT
CHER_IDLE_CHECK_DELAY, ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
      // Start the notification dispatcher. 11/30/2014, Bing Li
      super.execute(this.initReadFeedbackNotificationDispatcher);
    }

    /*
     * Shut down the server message dispatcher. 09/20/2014, Bing Li
     */
    public void shutdown()
    {
      // Dispose the sign-up dispatcher. 11/04/2014, Bing Li
      this.signUpRequestDispatcher.dispose();
      // Dispose the dispatcher for initializing reading feedback. 11/09/2014, Bing Li
      this.initReadFeedbackNotificationDispatcher.dispose();
      // Shutdown the derived server dispatcher. 11/04/2014, Bing Li
      super.shutdown();
    }

    /*
     * Process the available messages in a concurrent way. 09/20/2014, Bing Li
     */
    public void consume(OutMessageStream<ServerMessage> message)
    {
      // Check the types of received messages. 11/09/2014, Bing Li
      switch (message.getMessage().getType())
      {
        // If the message is the one of sign-up requests. 11/09/2014, Bing Li
        case MessageType.SIGN_UP_REQUEST:
          // Enqueue the request into the dispatcher for concurrent responding. 11/09/2014, Bing Li
          this.signUpRequestDispatcher.enqueue(new SignUpStream(message.getOutStream(),
message.getLock(), (SignUpRequest)message.getMessage()));
          break;

        // If the message is the one of initializing notification. 11/09/2014, Bing Li
        case MessageType.INIT_READ_NOTIFICATION:
          // Enqueue the notification into the dispatcher for concurrent feedback. 11/09/2014, Bing Li

    this.initReadFeedbackNotificationDispatcher.enqueue((InitReadNotification)message.getMessage());
          break;
      }
    }
}
```

- **MyServerProducerDisposer**

```
package com.greatfree.testing.server;

import com.greatfree.concurrency.MessageProducer;
import com.greatfree.reuse.ThreadDisposable;

/*
 * The class is responsible for disposing the message producer of the server. 09/20/2014, Bing Li
 */

// Created: 09/20/2014, Bing Li
public class MyServerProducerDisposer implements
ThreadDisposable<MessageProducer<MyServerDispatcher>>
{
   /*
    * Dispose the message producer. 09/20/2014, Bing Li
    */
   @Override
   public void dispose(MessageProducer<MyServerDispatcher> r)
   {
      r.dispose();
   }

   /*
    * The method does not make sense to the class of MessageProducer. Just leave it here. 09/20/2014,
Bing Li
    */
   @Override
   public void dispose(MessageProducer<MyServerDispatcher> r, long time)
   {
      r.dispose();
   }
}
```

- **RegisterClientThread**

```
package com.greatfree.testing.server;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.RegisterClientNotification;

/*
 * The thread receives registration notification from clients and keep their keys in the registry.
11/09/2014, Bing Li
 */

// Created: 11/08/2014, Bing Li
public class RegisterClientThread extends NotificationQueue<RegisterClientNotification>
{
  /*
   * Initialize the thread. The argument, taskSize, is used to limit the count of tasks to be queued.
11/09/2014, Bing Li
   */
  public RegisterClientThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * Once if a client registration notification is received, it is processed concurrently as follows.
11/09/2014, Bing Li
   */
  public void run()
  {
    // Declare an instance of RegisterClientNotification. 11/09/2014, Bing Li
    RegisterClientNotification notification;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/09/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/09/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/09/2014, Bing Li
          notification = this.getNotification();
          // Register the client. 11/09/2014, Bing Li
          ClientRegistry.MANAGEMENT().register(notification.getClientKey());
          // Dispose the notification. 11/09/2014, Bing Li
          this.disposeMessage(notification);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      try
      {
        // Wait for a moment after all of the existing notifications are processed. 11/09/2014, Bing Li
        this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **RegisterClientThreadCreator**

```
package com.greatfree.testing.server;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.RegisterClientNotification;

/*
 * The code here attempts to create instances of SetInputStreamThread. 11/30/2014, Bing Li
 */

// Created: 11/30/2014, Bing Li
public class RegisterClientThreadCreator implements
NotificationThreadCreatable<RegisterClientNotification, RegisterClientThread>
{
  @Override
  public RegisterClientThread createNotificationThreadInstance(int taskSize)
  {
    return new RegisterClientThread(taskSize);
  }
}
```

- **SignUpThread**

```
package com.greatfree.testing.server;

import java.io.IOException;

import com.greatfree.concurrency.RequestQueue;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.db.DBConfig;
import com.greatfree.testing.db.NodeDB;
import com.greatfree.testing.db.NodeDBPool;
import com.greatfree.testing.message.SignUpRequest;
import com.greatfree.testing.message.SignUpResponse;
import com.greatfree.testing.message.SignUpStream;
import com.greatfree.util.Tools;

/*
 * This is an example to use RequestQueue, which receives users' requests and responds concurrently.
11/04/2014, Bing Li
 */

// Created: 09/22/2014, Bing Li
public class SignUpThread extends RequestQueue<SignUpRequest, SignUpStream,
SignUpResponse>
{
   /*
    * Initialize the thread. The value of maxTaskSize is the length of the queue to take the count of
requests. 11/04/2014, Bing Li
    */
   public SignUpThread(int maxTaskSize)
   {
      super(maxTaskSize);
   }

   /*
    * Respond users' requests concurrently. 11/04/2014, Bing Li
    */
   public void run()
   {
      // Declare the request stream. 11/04/2014, Bing Li
      SignUpStream request;
      // Declare the response. 11/04/2014, Bing Li
      SignUpResponse response;
      // Get an instance of NodeDB to save new nodes. 11/04/2014, Bing Li
      NodeDB db = NodeDBPool.PERSISTENT().getDB(DBConfig.NODE_DB_PATH);
      // The thread is shutdown when it is idle long enough. Before that, the thread keeps alive. It is
necessary to detect whether it is time to end the task. 11/04/2014, Bing Li
      while (!this.isShutdown())
      {
         // The loop detects whether the queue is empty or not. 11/04/2014, Bing Li
         while (!this.isEmpty())
         {
            // Dequeue a request. 11/04/2014, Bing Li
            request = this.getRequest();
            // Persist the node locally. 11/04/2014, Bing Li
            db.saveNode(new Node(Tools.getHash(request.getMessage().getUserName()),
request.getMessage().getUserName(), request.getMessage().getPassword()));
            // Initialize a new response. 11/04/2014, Bing Li
            response = new SignUpResponse(true);
            try
            {
               // Respond to the client. 11/04/2014, Bing Li
               this.respond(request.getOutStream(), request.getLock(), response);
            }
            catch (IOException e)
            {
```

```
            e.printStackTrace();
          }
          // Dispose the request and the response. 11/04/2014, Bing Li
          this.disposeMessage(request, response);
        }
        try
        {
          // Wait for some time when the queue is empty. During the period and before the thread is killed,
some new requests might be received. If so, the thread can keep working. 11/04/2014, Bing Li
          this.holdOn(ServerConfig.RETRIEVE_THREAD_WAIT_TIME);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      // Collect the instance of the database after using it. 11/04/2014, Bing Li
      NodeDBPool.PERSISTENT().collectDB(db);
    }
  }
```

- **SignUpThreadCreator**

```
package com.greatfree.testing.server;

import com.greatfree.concurrency.RequestThreadCreatable;
import com.greatfree.testing.message.SignUpRequest;
import com.greatfree.testing.message.SignUpResponse;
import com.greatfree.testing.message.SignUpStream;

/*
 * A creator to initialize instances of SignUpThread. It is used in the instance of RequestDispatcher.
11/04/2014, Bing Li
 */

// Created: 11/04/2014, Bing Li
public class SignUpThreadCreator implements RequestThreadCreatable<SignUpRequest,
SignUpStream, SignUpResponse, SignUpThread>
{
  @Override
  public SignUpThread createRequestThreadInstance(int taskSize)
  {
    return new SignUpThread(taskSize);
  }
}
```

- **InitReadFeedbackThread**

```
package com.greatfree.testing.server;

import java.io.IOException;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.InitReadFeedbackNotification;
import com.greatfree.testing.message.InitReadNotification;

/*
 * This is an important thread since it ensure the local ObjectOutputStream is initialized and it notifies to
the relevant remote ObjectInputStream can be initialized. 11/09/2014, Bing Li
 */

// Created: 11/09/2014, Bing Li
public class InitReadFeedbackThread extends NotificationQueue<InitReadNotification>
{
  /*
   * Initialize the thread. The argument, taskSize, is used to limit the count of tasks to be queued.
11/09/2014, Bing Li
   */
  public InitReadFeedbackThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * Once if a node key notification is received, it is processed concurrently as follows. 11/09/2014, Bing
Li
   */
  public void run()
  {
    // Declare an instance of InitReadNotification. 11/09/2014, Bing Li
    InitReadNotification notification;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/09/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/09/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/09/2014, Bing Li
          notification = this.getNotification();
          try
          {
            // Send the instance of InitReadFeedbackNotification to the client which needs to initialize
the ObjectInputStream of an instance of FreeClient. 11/09/2014, Bing Li
            ClientPool.SERVER().getPool().send(notification.getClientKey(), new
InitReadFeedbackNotification());
            this.disposeMessage(notification);
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      try
      {
```

```
            // Wait for a moment after all of the existing notifications are processed. 11/09/2014, Bing Li
            this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
        }
      }
    }
```

- **InitReadFeedbackThreadCreator**

```
package com.greatfree.testing.server;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.InitReadNotification;

/*
 * The code here attempts to create instances of SetInputStreamThread. 11/07/2014, Bing Li
 */

// Created: 11/09/2014, Bing Li
public class InitReadFeedbackThreadCreator implements
NotificationThreadCreatable<InitReadNotification, InitReadFeedbackThread>
{
  @Override
  public InitReadFeedbackThread createNotificationThreadInstance(int taskSize)
  {
    return new InitReadFeedbackThread(taskSize);
  }
}
```

- **ClientPool**

```
package com.greatfree.testing.server;

import java.io.IOException;

import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.data.ServerConfig;

/*
 * The pool is responsible for creating and managing instance of FreeClient to achieve the goal of using as small number of instances of FreeClient to send messages to the client in a high performance. 11/24/2014, Bing Li
 */

// Created: 09/17/2014, Bing Li
public class ClientPool
{
   // An instance of FreeClientPool is defined to interact with the client. 11/24/2014, Bing Li
   private FreeClientPool pool;

   /*
    * Define the singleton wrapper. 09/17/2014, Bing Li
    */
   private ClientPool()
   {
   }

   /*
    * Define the singleton wrapper. The static method to access the instance of the client pool is named P2P, which represents the peer-to-peer. Since the pool is usually used by an eventing server which connects to the remote end, a P2P architecture is formed. So the method is named like this. 09/17/2014, Bing Li
    */
   private static ClientPool instance = new ClientPool();

   public static ClientPool SERVER()
   {
      if (instance == null)
      {
         instance = new ClientPool();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Dispose the free client pool. 11/24/2014, Bing Li
    */
   public void dispose() throws IOException
   {
      this.pool.dispose();
   }

   /*
    * Initialize the client pool. The method is called when the crawler process is started. 11/24/2014, Bing Li
    */
   public void init()
   {
      // Initialize the client pool. 11/24/2014, Bing Li
      this.pool = new FreeClientPool(ServerConfig.CLIENT_POOL_SIZE);
      // Set idle checking for the client pool. 11/24/2014, Bing Li
```

```
        this.pool.setIdleChecker(ServerConfig.CLIENT_IDLE_CHECK_DELAY,
ServerConfig.CLIENT_IDLE_CHECK_PERIOD, ServerConfig.CLIENT_MAX_IDLE_TIME);
    }

    /*
     * Expose the client pool. 11/24/2014, Bing Li
     */
    public FreeClientPool getPool()
    {
        return this.pool;
    }
}
```

- **ClientRegistry**

```
package com.greatfree.testing.server;

import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

/*
 * This is a collection to keep all of the keys of online clients. It is used to manage clients, such as
multicasting. 11/09/2014, Bing Li
 */

// Created: 11/09/2014, Bing Li
public class ClientRegistry
{
  // Declare a list to keep all of the keys. 11/09/2014, Bing Li
  private List<String> clientKeys;

  /*
   * Initialize. 11/09/2014, Bing Li
   */
  private ClientRegistry()
  {
    // Define a synchronized list for the consideration of consistency. 11/09/2014, Bing Li
    this.clientKeys = new CopyOnWriteArrayList<String>();
  }

  /*
   * A singleton implementation. 11/09/2014, Bing Li
   */
  private static ClientRegistry instance = new ClientRegistry();

  public static ClientRegistry MANAGEMENT()
  {
    if (instance == null)
    {
      instance = new ClientRegistry();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the registry. 11/09/2014, Bing Li
   */
  public void dispose()
  {
    this.clientKeys.clear();
    this.clientKeys = null;
  }

  /*
   * Register the client key. 11/09/2014, Bing Li
   */
  public void register(String clientKey)
  {
    // Detect if the client key exists. 11/09/2014, Bing Li
    if (!this.clientKeys.contains(clientKey))
    {
      // Add the client key if it does not exist. 11/09/2014, Bing Li
      this.clientKeys.add(clientKey);
    }
  }
```

```
  /*
   * Unregister the client key. 11/09/2014, Bing Li
   */
  public void unregister(String clientKey)
  {
    // Check if the client key exists. 11/09/2014, Bing Li
    if (this.clientKeys.contains(clientKey))
    {
      // Remove the client key if it exists. 11/09/2014, Bing Li
      this.clientKeys.remove(clientKey);
    }
  }

  /*
   * Return the count of all of the registered clients. 11/09/2014, Bing Li
   */
  public int getClientCount()
  {
    return this.clientKeys.size();
  }

  /*
   * Return all of the registered client keys. 11/09/2014, Bing Li
   */
  public List<String> getClientKeys()
  {
    return this.clientKeys;
  }
}
```

- **Node**

```
package com.greatfree.testing.server;

/*
 * The class keeps the information to define a sign-up or sign-in client. It is usually different from NoteEntity, which consists of data for a node only without relevant manipulations. For the testing case, they are almost the same since the scenarios are simple. In practice, it is recommended to differentiate them like the samples. 11/03/2014, Bing Li
 */

// Created: 11/03/2014, Bing Li
public class Node
{
  private String key;
  private String username;
  private String password;

  /*
   * Initialize. 11/03/2014, Bing Li
   */
  public Node(String key, String username, String password)
  {
    this.key = key;
    this.username = username;
    this.password = password;
  }

  /*
   * Expose the key. 11/03/2014, Bing Li
   */
  public String getKey()
  {
    return this.key;
  }

  /*
   * Expose the username. 11/03/2014, Bing Li
   */
  public String getUsername()
  {
    return this.username;
  }

  /*
   * Expose the password. 11/03/2014, Bing Li
   */
  public String getPassword()
  {
    return this.password;
  }
}
```

3.6 An Illustrative Large-Scale Distributed System

3.6.1 The Coordinator

- **CoorConfig**

```
package com.greatfree.testing.coordinator;

import com.greatfree.testing.data.ServerConfig;

/*
 * The class contains some configurations for the coordinator. 11/30/2014, Bing Li
 */

// Created: 11/24/2014, Bing Li
public class CoorConfig
{
   public final static String CSERVER_CONFIG = ServerConfig.CONFIG_HOME +
"CServerConfig.xml";

   public final static int CSERVER_CLIENT_POOL_SIZE = 500;
   public final static long CSERVER_CLIENT_IDLE_CHECK_DELAY = 3000;
   public final static long CSERVER_CLIENT_IDLE_CHECK_PERIOD = 3000;
   public final static long CSERVER_CLIENT_MAX_IDLE_TIME = 6000;

   public final static String SELECT_CRAWLSERVER_COUNT = "/CConfig/CrawlServerCount/text()";
   public final static String SELECT_MEMORYSERVER_COUNT =
"/CConfig/MemoryServerCount/text()";

   public final static long ANYCAST_REQUEST_WAIT_TIME = 5000;
   public final static long BROADCAST_REQUEST_WAIT_TIME = 5000;

   public final static int AUTHORITY_ANYCAST_READER_POOL_SIZE = 100;
   public final static int AUTHORITY_ANYCAST_READER_POOL_WAIT_TIME = 1000;

   public final static int BROADCAST_READER_POOL_SIZE = 100;
   public final static int BROADCAST_READER_POOL_WAIT_TIME = 1000;
}
```

- **StartCoordinator**

```
package com.greatfree.testing.coordinator;

import com.greatfree.testing.data.ServerConfig;
import com.greatfree.util.TerminateSignal;

/*
 * This is the entry and the exit of the coordinator process. 11/25/2014, Bing Li
 */

// Created: 11/10/2014, Bing Li
public class StartCoordinator
{
  public static void main(String[] args)
  {
    // Start the coordinator. 11/25/2014, Bing Li
    Coordinator.COORDINATOR().start(ServerConfig.COORDINATOR_PORT_FOR_CRAWLER,
ServerConfig.COORDINATOR_PORT_FOR_MEMORY,
ServerConfig.COORDINATOR_PORT_FOR_ADMIN,
ServerConfig.COORDINATOR_PORT_FOR_SEARCH);

    // After the coordinator is started, the loop check whether the flag of terminating is set. If the
terminating flag is true, the process is ended. Otherwise, the process keeps running. 11/25/2014, Bing Li
    while (!TerminateSignal.SIGNAL().isTerminated())
    {
      try
      {
        // If the terminating flag is false, it is required to sleep for some time. Otherwise, it might cause
the high CPU usage. 11/25/2014, Bing Li
        Thread.sleep(ServerConfig.TERMINATE_SLEEP);
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **Coordinator**

```
package com.greatfree.testing.coordinator;

import java.io.IOException;
import java.net.ServerSocket;
import java.util.ArrayList;
import java.util.List;

import com.greatfree.concurrency.Runner;
import com.greatfree.testing.coordinator.admin.AdminIORegistry;
import com.greatfree.testing.coordinator.admin.AdminMulticastor;
import com.greatfree.testing.coordinator.admin.AdminListener;
import com.greatfree.testing.coordinator.admin.AdminListenerDisposer;
import com.greatfree.testing.coordinator.crawling.CrawlIORegistry;
import com.greatfree.testing.coordinator.crawling.CrawlListener;
import com.greatfree.testing.coordinator.crawling.CrawlListenerDisposer;
import com.greatfree.testing.coordinator.crawling.CrawlServerClientPool;
import com.greatfree.testing.coordinator.memory.MemoryIORegistry;
import com.greatfree.testing.coordinator.memory.MemoryListener;
import com.greatfree.testing.coordinator.memory.MemoryListenerDisposer;
import com.greatfree.testing.coordinator.memory.MemoryServerClientPool;
import com.greatfree.testing.coordinator.search.CoordinatorMulticastReader;
import com.greatfree.testing.coordinator.search.SearchClientPool;
import com.greatfree.testing.coordinator.search.SearchIORegistry;
import com.greatfree.testing.coordinator.search.SearchListener;
import com.greatfree.testing.coordinator.search.SearchListenerDisposer;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.util.TerminateSignal;

/*
 * This is a sample of the coordinator which is responsible for managing all of the nodes in the form of a cluster. 11/25/2014, Bing Li
 *
 * The class is responsible for starting up and shutting down the coordinator. Since it is the unique entry and exit of the coordinator, it is implemented in the form of a singleton. 11/25/2014, Bing Li
 */

// Created: 11/24/2014, Bing Li
public class Coordinator
{
   // The ServerSocket waits for crawlers' connecting. The socket serves the coordinator in the sense that it not only responds to crawlers' requests but also notifies crawlers to manage the crawling process. 11/25/2014, Bing Li
   private ServerSocket crawlServerSocket;
   // The port number for the crawling socket. 11/25/2014, Bing Li
   private int crawlPort;

   // The ServerSocket waits for memory servers' connecting. The socket serves the coordinator in the sense that it not only responds to memory servers' requests but also notifies memory servers to manage the storing process. 11/28/2014, Bing Li
   private ServerSocket memServerSocket;
   // The port number for the memory socket. 11/29/2014, Bing Li
   private int memPort;

   // The ServerSocket waits for the administrator's connecting. 11/29/2014, Bing Li
   private ServerSocket adminServerSocket;
   // The port number for the administration socket. 11/29/2014, Bing Li
   private int adminPort;

   // The ServerSocket waits for the searchers' connecting. 11/29/2014, Bing Li
   private ServerSocket searchServerSocket;
   // The port number for the search socket. 11/29/2014, Bing Li
   private int searchPort;

   // The list keeps all of the threads that listen to connecting from crawlers of the cluster. When the
```

```
coordinator is shutdown, those threads can be killed to avoid possible missing. 11/25/2014, Bing Li
  private List<Runner<CrawlListener, CrawlListenerDisposer>> crawlListenerRunnerList;
  // The list keeps all of the threads that listen to connecting from memory servers of the cluster. When
the coordinator is shutdown, those threads can be killed to avoid possible missing. 11/28/2014, Bing Li
  private List<Runner<MemoryListener, MemoryListenerDisposer>> memListenerRunnerList;

  // Declare one runner for the administration. Since the load is lower, it is not necessary to initialize
multiple threads to listen to potential connections. 11/29/2014, Bing Li
  private Runner<AdminListener, AdminListenerDisposer> adminListenerRunner;

  // The list keeps all of the threads that listen to connecting from search clients. When the coordinator is
shutdown, those threads can be killed to avoid possible missing. 11/29/2014, Bing Li
  private List<Runner<SearchListener, SearchListenerDisposer>> searchListenerRunnerList;

  /*
   * A singleton is designed for the coordinator's startup and shutdown interface. 11/25/2014, Bing Li
   */
  private Coordinator()
  {
  }

  private static Coordinator instance = new Coordinator();

  public static Coordinator COORDINATOR()
  {
    if (instance == null)
    {
      instance = new Coordinator();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Start the coordinator and relevant listeners with concurrent threads for potential busy connecting.
11/25/2014, Bing Li
   */
  public void start(int crawlPort, int memPort, int adminPort, int searchPort)
  {
    // Initialize and start the crawling socket. 11/25/2014, Bing Li
    this.crawlPort = crawlPort;
    try
    {
      this.crawlServerSocket = new ServerSocket(this.crawlPort);
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }

    // Initialize and start the memory socket. 11/28/2014, Bing Li
    this.memPort = memPort;
    try
    {
      this.memServerSocket = new ServerSocket(this.memPort);
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }

    // Initialize and start administration socket. 11/29/2014, Bing Li
    this.adminPort = adminPort;
    try
    {
```

```
            this.adminServerSocket = new ServerSocket(this.adminPort);
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }

        // Initialize and start search socket. 11/29/2014, Bing Li
        this.searchPort = searchPort;
        try
        {
            this.searchServerSocket = new ServerSocket(this.searchPort);
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }

        // Initialize a disposer which collects the crawler listener. 11/25/2014, Bing Li
        CrawlListenerDisposer crawlDisposer = new CrawlListenerDisposer();
        // Initialize the runner list. 11/25/2014, Bing Li
        this.crawlListenerRunnerList = new ArrayList<Runner<CrawlListener, CrawlListenerDisposer>>();
        // Start up the threads to listen to connecting from crawlers which send and receive messages from
the coordinator. 11/25/2014, Bing Li
        Runner<CrawlListener, CrawlListenerDisposer> crawlRunner;
        for (int i = 0; i < ServerConfig.LISTENING_THREAD_COUNT; i++)
        {
            crawlRunner = new Runner<CrawlListener, CrawlListenerDisposer>(new
CrawlListener(this.crawlServerSocket, ServerConfig.LISTENER_THREAD_POOL_SIZE,
ServerConfig.LISTENER_THREAD_ALIVE_TIME), crawlDisposer, true);
            this.crawlListenerRunnerList.add(crawlRunner);
            crawlRunner.start();
        }

        // Initialize a disposer which collects the memory listener. 11/28/2014, Bing Li
        MemoryListenerDisposer memDisposer = new MemoryListenerDisposer();
        // Initialize the runner list. 11/28/2014, Bing Li
        this.memListenerRunnerList = new ArrayList<Runner<MemoryListener,
MemoryListenerDisposer>>();
        // Start up the threads to listen to connecting from memory servers which send and receive
messages from the coordinator. 11/28/2014, Bing Li
        Runner<MemoryListener, MemoryListenerDisposer> memRunner;
        for (int i = 0; i < ServerConfig.LISTENING_THREAD_COUNT; i++)
        {
            memRunner = new Runner<MemoryListener, MemoryListenerDisposer>(new
MemoryListener(this.memServerSocket, ServerConfig.LISTENER_THREAD_POOL_SIZE,
ServerConfig.LISTENER_THREAD_ALIVE_TIME), memDisposer, true);
            this.memListenerRunnerList.add(memRunner);
            memRunner.start();
        }

        // Initialize a disposer which collects the administration listener. 11/29/2014, Bing Li
        AdminListenerDisposer adminDisposer = new AdminListenerDisposer();
        // Initialize the runner to listen to connecting from the administrator which sends and notifications to
the coordinator. 11/29/2014, Bing Li
        this.adminListenerRunner = new Runner<AdminListener, AdminListenerDisposer>(new
AdminListener(this.memServerSocket, ServerConfig.LISTENER_THREAD_POOL_SIZE,
ServerConfig.LISTENER_THREAD_ALIVE_TIME), adminDisposer, true);
        // Start up the runner. 11/29/2014, Bing Li
        this.adminListenerRunner.start();


        // Initialize a disposer which collects the search listener. 11/29/2014, Bing Li
        SearchListenerDisposer searchDisposer = new SearchListenerDisposer();
        // Initialize the runner list. 11/29/2014, Bing Li
        this.searchListenerRunnerList = new ArrayList<Runner<SearchListener,
SearchListenerDisposer>>();
        // Start up the threads to listen to connecting from search clients which send messages from the
```

```
coordinator. 11/29/2014, Bing Li
    Runner<SearchListener, SearchListenerDisposer> searchRunner;
    for (int i = 0; i < ServerConfig.LISTENING_THREAD_COUNT; i++)
    {
      searchRunner = new Runner<SearchListener, SearchListenerDisposer>(new
SearchListener(this.memServerSocket, ServerConfig.LISTENER_THREAD_POOL_SIZE,
ServerConfig.LISTENER_THREAD_ALIVE_TIME), searchDisposer, true);
      this.searchListenerRunnerList.add(searchRunner);
      searchRunner.start();
    }

    // Initialize the profile. 11/25/2014, Bing Li
    Profile.CONFIG().init(CoorConfig.CSERVER_CONFIG);

    // Initialize the crawling IO registry. 11/25/2014, Bing Li
    CrawlIORegistry.REGISTRY().init();
    // Initialize a crawling client pool, which is used by the coordinator to connect to the remote crawler.
11/25/2014, Bing Li
    CrawlServerClientPool.COORDINATE().init();

    // Initialize the memory IO registry. 11/28/2014, Bing Li
    MemoryIORegistry.REGISTRY().init();
    // Initialize a memory client pool, which is used by the coordinator to connect to the remote memory
server. 11/25/2014, Bing Li
    MemoryServerClientPool.COORDINATE().init();

    // Initialize the administration IO registry. 11/29/2014, Bing Li
    AdminIORegistry.REGISTRY().init();

    // Initialize the search IO registry. 11/29/2014, Bing Li
    SearchIORegistry.REGISTRY().init();
    // Initialize a search client pool, which is used by the coordinator to connect to the remote search
client. 11/29/2014, Bing Li
    SearchClientPool.COORDINATE().init();

    // Initialize the multicastor for the clusters, crawlers and memory nodes. 11/27/2014, Bing Li
    CoordinatorMulticastor.COORDINATE().init();
    // Initialize the administration multicastor. 11/27/2014, Bing Li
    AdminMulticastor.ADMIN().init();

    // Initialize the memory multicast reader. 11/29/2014, Bing Li
    CoordinatorMulticastReader.COORDINATE().init();
  }

  /*
   * Shutdown the coordinator. 11/25/2014, Bing Li
   */
  public void stop() throws IOException, InterruptedException
  {
    // Set the terminating signal. 11/25/2014, Bing Li
    TerminateSignal.SIGNAL().setTerminated();
    // Close the sockets for the coordinator. 11/25/2014, Bing Li
    this.crawlServerSocket.close();
    this.memServerSocket.close();
    this.adminServerSocket.close();
    this.searchServerSocket.close();

    // Stop all of the threads that listen to crawlers' connecting to the coordinator. 11/25/2014, Bing Li
    for (Runner<CrawlListener, CrawlListenerDisposer> runner : this.crawlListenerRunnerList)
    {
      runner.stop();
    }

    // Stop all of the threads that listen to memory servers' connecting to the coordinator. 11/28/2014,
Bing Li
    for (Runner<MemoryListener, MemoryListenerDisposer> runner : this.memListenerRunnerList)
    {
      runner.stop();
```

```
    }

    // Stop the administration runner. 11/29/2014, Bing Li
    this.adminListenerRunner.stop();

    // Stop all of the threads that listen to search clients' connecting to the coordinator. 11/29/2014, Bing
Li
    for (Runner<SearchListener, SearchListenerDisposer> runner : this.searchListenerRunnerList)
    {
      runner.stop();
    }

    // Dispose the profile. 11/25/2014, Bing Li
    Profile.CONFIG().dispose();

    // Shutdown the IO registry. 11/25/2014, Bing Li
    CrawlIORegistry.REGISTRY().dispose();

    // Shutdown the client pool. 11/25/2014, Bing Li
    CrawlServerClientPool.COORDINATE().dispose();

    // Shutdown the IO registry. 11/28/2014, Bing Li
    MemoryIORegistry.REGISTRY().dispose();

    // Shutdown the client pool. 11/28/2014, Bing Li
    MemoryServerClientPool.COORDINATE().dispose();

    // Shutdown the IO registry. 11/29/2014, Bing Li
    AdminIORegistry.REGISTRY().dispose();

    // Shutdown the client pool. 11/29/2014, Bing Li
    SearchClientPool.COORDINATE().dispose();
    // Shutdown the IO registry. 11/29/2014, Bing Li
    SearchIORegistry.REGISTRY().dispose();

    // Dispose the multicastor for crawlers and memory nodes. 11/27/2014, Bing Li
    CoordinatorMulticastor.COORDINATE().dispose();
    // Dispose the administration multicastor. 11/27/2014, Bing Li
    AdminMulticastor.ADMIN().dispose();

    // Dispose the memory multicast reader. 11/29/2014, Bing Li
    CoordinatorMulticastReader.COORDINATE().dispose();
  }
}
```

- **Profile**

```
package com.greatfree.testing.coordinator;

import com.greatfree.util.XMLReader;

/*
 * This is a configuration file that contains the predefined information about the distributed system.
11/25/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class Profile
{
   // The crawler count to participate the system. 11/25/2014, Bing Li
   private int crawlServerCount;
   // The memory server count to participate the system. 11/28/2014, Bing Li
   private int memoryServerCount;

   private Profile()
   {
   }

   /*
    * A singleton definition. 11/25/2014, Bing Li
    */
   private static Profile instance = new Profile();

   public static Profile CONFIG()
   {
      if (instance == null)
      {
         instance = new Profile();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Dispose the profile. 11/25/2014, Bing Li
    */
   public void dispose()
   {
   }

   /*
    * Initialize the profile. 11/25/2014, Bing Li
    */
   public void init(String path)
   {
      // An XML reader is defined to retrieved predefined information from the XML file, which is managed
by the administrator manually. 11/25/2014, Bing Li
      XMLReader reader = new XMLReader(path, true);
      this.crawlServerCount = new
Integer(reader.read(CoorConfig.SELECT_CRAWLSERVER_COUNT));
      this.memoryServerCount = new
Integer(reader.read(CoorConfig.SELECT_MEMORYSERVER_COUNT));
      reader.close();
   }

   /*
    * Get the count of the crawlers that participate the crawling cluster. 11/25/2014, Bing Li
    */
```

```
    public int getCrawlServerCount()
    {
      return this.crawlServerCount;
    }

    /*
     * Get the count of the memory servers that participate the crawling cluster. 11/28/2014, Bing Li
     */
    public int getMemoryServerCount()
    {
      return this.memoryServerCount;
    }
}
```

- **CoordinatorMessageProducer**

```
package com.greatfree.testing.coordinator;

import com.greatfree.concurrency.MessageProducer;
import com.greatfree.concurrency.Threader;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.testing.coordinator.admin.AdminServerDispatcher;
import com.greatfree.testing.coordinator.admin.AdminServerProducerDisposer;
import com.greatfree.testing.coordinator.crawling.CrawlServerDispatcher;
import com.greatfree.testing.coordinator.crawling.CrawlServerProducerDisposer;
import com.greatfree.testing.coordinator.memory.MemoryServerDispatcher;
import com.greatfree.testing.coordinator.memory.MemoryServerProducerDisposer;
import com.greatfree.testing.coordinator.search.SearchServerDispatcher;
import com.greatfree.testing.coordinator.search.SearchServerProducerDisposer;
import com.greatfree.testing.data.ServerConfig;

/*
 * The class is a singleton to enclose the instances of MessageProducer. Each of the enclosed message producers serves for one particular client, such as the crawler server, that connects to a respective port on the coordinator. Usually, each port aims to provide one particular service. 11/24/2014, Bing Li
 *
 * The class is a wrapper that encloses all of the asynchronous message producers. It is responsible for assigning received messages to the corresponding producer in an asynchronous way. 08/22/2014, Bing Li
 */

// Created: 11/24/2014, Bing Li
public class CoordinatorMessageProducer
{
  // The Threader aims to associate with the crawler message producer to guarantee the producer can work concurrently. 11/24/2014, Bing Li
  private Threader<MessageProducer<CrawlServerDispatcher>, CrawlServerProducerDisposer> crawlProducerThreader;
  // The Threader aims to associate with the memory server message producer to guarantee the producer can work concurrently. 11/28/2014, Bing Li
  private Threader<MessageProducer<MemoryServerDispatcher>, MemoryServerProducerDisposer> memoryProducerThreader;
  // The Threader aims to associate with the administration message producer to guarantee the producer can work concurrently. 11/27/2014, Bing Li
  private Threader<MessageProducer<AdminServerDispatcher>, AdminServerProducerDisposer> adminProducerThreader;
  // The Threader aims to associate with the search message producer to guarantee the producer can work concurrently. 11/29/2014, Bing Li
  private Threader<MessageProducer<SearchServerDispatcher>, SearchServerProducerDisposer> searchProducerThreader;

  private CoordinatorMessageProducer()
  {
  }

  /*
   * The class is required to be a singleton since it is nonsense to initiate it for the producers are unique. 11/24/2014, Bing Li
   */
  private static CoordinatorMessageProducer instance = new CoordinatorMessageProducer();

  public static CoordinatorMessageProducer SERVER()
  {
    if (instance == null)
    {
      instance = new CoordinatorMessageProducer();
      return instance;
    }
    else
```

```
    {
      return instance;
    }
  }

  /*
   * Dispose the producers when the process of the coordinator is shutdown. 11/24/2014, Bing Li
   */
  public void dispose() throws InterruptedException
  {
    this.crawlProducerThreader.stop();
    this.memoryProducerThreader.stop();
    this.adminProducerThreader.stop();
    this.searchProducerThreader.stop();
  }

  /*
   * Initialize the message producers. It is invoked when the connection modules of the coordinator is
started since clients can send requests or notifications only after it is started. 11/24/2014, Bing Li
   */
  public void init()
  {
    // Initialize the crawling message producer. A threader is associated with the crawling message
producer such that the producer is able to work in a concurrent way. 11/24/2014, Bing Li
    this.crawlProducerThreader = new Threader<MessageProducer<CrawlServerDispatcher>,
CrawlServerProducerDisposer>(new MessageProducer<CrawlServerDispatcher>(new
CrawlServerDispatcher(ServerConfig.DISPATCHER_POOL_SIZE,
ServerConfig.DISPATCHER_POOL_THREAD_POOL_ALIVE_TIME)), new
CrawlServerProducerDisposer());
    // Start the associated thread for the crawling message producer. 11/24/2014, Bing Li
    this.crawlProducerThreader.start();

    // Initialize the memory server message producer. A threader is associated with the memory server
message producer such that the producer is able to work in a concurrent way. 11/28/2014, Bing Li
    this.memoryProducerThreader = new Threader<MessageProducer<MemoryServerDispatcher>,
MemoryServerProducerDisposer>(new MessageProducer<MemoryServerDispatcher>(new
MemoryServerDispatcher(ServerConfig.DISPATCHER_POOL_SIZE,
ServerConfig.DISPATCHER_POOL_THREAD_POOL_ALIVE_TIME)), new
MemoryServerProducerDisposer());
    // Start the associated thread for the memory server message producer. 11/28/2014, Bing Li
    this.memoryProducerThreader.start();

    // Initialize the administration message producer. A threader is associated with the administration
message producer such that the producer is able to work in a concurrent way. 11/27/2014, Bing Li
    this.adminProducerThreader = new Threader<MessageProducer<AdminServerDispatcher>,
AdminServerProducerDisposer>(new MessageProducer<AdminServerDispatcher>(new
AdminServerDispatcher(ServerConfig.DISPATCHER_POOL_SIZE,
ServerConfig.DISPATCHER_POOL_THREAD_POOL_ALIVE_TIME)), new
AdminServerProducerDisposer());
    // Start the associated thread for the administration message producer. 11/27/2014, Bing Li
    this.adminProducerThreader.start();

    // Initialize the search message producer. A threader is associated with the search message
producer such that the producer is able to work in a concurrent way. 11/29/2014, Bing Li
    this.searchProducerThreader = new Threader<MessageProducer<SearchServerDispatcher>,
SearchServerProducerDisposer>(new MessageProducer<SearchServerDispatcher>(new
SearchServerDispatcher(ServerConfig.DISPATCHER_POOL_SIZE,
ServerConfig.DISPATCHER_POOL_THREAD_POOL_ALIVE_TIME)), new
SearchServerProducerDisposer());
    // Start the associated thread for the search message producer. 11/29/2014, Bing Li
    this.searchProducerThreader.start();
  }

  /*
   * Assign crawling messages, requests or notifications, to the bound crawling message dispatcher
such that they can be responded or dealt with. 11/24/2014, Bing Li
   */
  public void produceCrawlingMessage(OutMessageStream<ServerMessage> message)
```

```
  {
    this.crawlProducerThreader.getFunction().produce(message);
  }

  /*
   * Assign memory server messages, requests or notifications, to the bound memory server message
dispatcher such that they can be responded or dealt with. 11/28/2014, Bing Li
   */
  public void produceMemoryMessage(OutMessageStream<ServerMessage> message)
  {
    this.memoryProducerThreader.getFunction().produce(message);
  }

  /*
   * Assign administration messages, requests or notifications, to the bound administration message
dispatcher such that they can be responded or dealt with. 11/27/2014, Bing Li
   */
  public void produceAdminMessage(OutMessageStream<ServerMessage> message)
  {
    this.adminProducerThreader.getFunction().produce(message);
  }

  /*
   * Assign search messages, requests or notifications, to the bound search message dispatcher such
that they can be responded or dealt with. 11/29/2014, Bing Li
   */
  public void produceSearchMessage(OutMessageStream<ServerMessage> message)
  {
    this.searchProducerThreader.getFunction().produce(message);
  }
}
```

- **CoordinatorMulticastor**

```
package com.greatfree.testing.coordinator;

import java.io.IOException;

import com.greatfree.reuse.ResourcePool;
import com.greatfree.testing.coordinator.crawling.CrawlServerClientPool;
import com.greatfree.testing.coordinator.crawling.StartCrawlMulticastor;
import com.greatfree.testing.coordinator.crawling.StartCrawlMulticastorCreator;
import com.greatfree.testing.coordinator.crawling.StartCrawlMulticastorDisposer;
import com.greatfree.testing.coordinator.crawling.StartCrawlMulticastorSource;
import com.greatfree.testing.coordinator.crawling.StartCrawlNotificationCreator;
import com.greatfree.testing.data.ServerConfig;

/*
 * This is a singleton that contains all of the multicastor pools. Those multicastors are critical to compose
a cluster for all of the crawlers. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public class CoordinatorMulticastor
{
   // The pool for the multicastor which multicasts the notification of StartCrawlMultiNotification.
11/26/2014, Bing Li
   private ResourcePool<StartCrawlMulticastorSource, StartCrawlMulticastor,
StartCrawlMulticastorCreator, StartCrawlMulticastorDisposer> startCrawlMulticastorPool;

   private CoordinatorMulticastor()
   {
   }

   /*
    * A singleton implementation. 11/26/2014, Bing Li
    */
   private static CoordinatorMulticastor instance = new CoordinatorMulticastor();

   public static CoordinatorMulticastor COORDINATE()
   {
      if (instance == null)
      {
         instance = new CoordinatorMulticastor();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Dispose all of the pools. 11/26/2014, Bing Li
    */
   public void dispose()
   {
      this.startCrawlMulticastorPool.shutdown();
   }

   /*
    * Initialize the pools. 11/26/2014, Bing Li
    */
   public void init()
   {
      this.startCrawlMulticastorPool = new ResourcePool<StartCrawlMulticastorSource,
StartCrawlMulticastor, StartCrawlMulticastorCreator,
StartCrawlMulticastorDisposer>(ServerConfig.MULTICASTOR_POOL_SIZE, new
```

```
StartCrawlMulticastorCreator(), new StartCrawlMulticastorDisposer(),
ServerConfig.MULTICASTOR_POOL_WAIT_TIME);
  }

  /*
   * Disseminate the notification of StartCrawlMultiNotification to all of the crawlers. 11/26/2014, Bing Li
   */
  public void disseminateStartCrawl() throws InstantiationException, IllegalAccessException,
IOException, InterruptedException
  {
    // Get an instance of StartCrawlMulticastor from the pool. 11/26/2014, Bing Li
    StartCrawlMulticastor multicastor = this.startCrawlMulticastorPool.get(new
StartCrawlMulticastorSource(CrawlServerClientPool.COORDINATE().getPool(),
ServerConfig.ROOT_MULTICAST_BRANCH_COUNT, ServerConfig.MULTICAST_BRANCH_COUNT,
new StartCrawlNotificationCreator()));
    // Check whether the multicastor is valid. 11/26/2014, Bing Li
    if (multicastor != null)
    {
      // Disseminate the notification. The notification contains no data. Thus, it is not necessary to put
any arguments here. Just place a null. 11/26/2014, Bing Li
      multicastor.disseminate(null);
      // Collect the instance of StartCrawlMulticastor. 11/26/2014, Bing Li
      this.startCrawlMulticastorPool.collect(multicastor);
    }
  }
}
```

3.6.1.1 The Administering

- **AdminListener**

```
package com.greatfree.testing.coordinator.admin;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

import com.greatfree.remote.ServerListener;
import com.greatfree.testing.data.ServerConfig;

/*
 * This is a coordinator listener that not only responds to the administrator. Since the administrator is controlled by a human, it is not necessary to consider so complicated as follows. The reason to do that as below is to keep the style consistent. And, the implementation does not affect the coordinator in terms of much additional resources. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class AdminListener extends ServerListener implements Runnable
{
  /*
   * Initialize the listener. 11/27/2014, Bing Li
   */
  public AdminListener(ServerSocket serverSocket, int threadPoolSize, long keepAliveTime)
  {
    super(serverSocket, threadPoolSize, keepAliveTime);
  }

  /*
   * Shutdown the listener. 11/27/2014, Bing Li
   */
  public void shutdown()
  {
    // Shutdown the listener. 11/27/2014, Bing Li
    super.shutdown();
  }

  /*
   * The task that must be executed concurrently. 11/27/2014, Bing Li
   */
  @Override
  public void run()
  {
    Socket clientSocket;
    AdminIO serverIO;

    // Detect whether the listener is shutdown. If not, it must be running all the time to wait for potential connections from crawlers. 11/27/2014, Bing Li
    while (!super.isShutdown())
    {
      try
      {
        // Wait and accept a connecting from a possible crawler. 11/27/2014, Bing Li
        clientSocket = super.accept();
        // Check whether the connected server IOs exceed the upper limit. 11/27/2014, Bing Li
        if (AdminIORegistry.REGISTRY().getIOCount() >= ServerConfig.MAX_SERVER_IO_COUNT)
        {
          try
          {
            // If the upper limit is reached, the listener has to wait until an existing server IO is disposed. 11/25/2014, Bing Li
            super.holdOn();
          }
          catch (InterruptedException e)
          {
```

```
                e.printStackTrace();
              }
            }

            // If the upper limit of IOs is not reached, a crawling server IO is initialized. A common
Collaborator and the socket are the initial parameters. The shared common collaborator guarantees all
of the crawling server IOs from a certain crawler could notify with each other with the same lock. Then,
the upper limit of crawling server IOs is under the control. 11/27/2014, Bing Li
            serverIO = new AdminIO(clientSocket, super.getCollaborator());
            // Add the new created server IO into the registry for further management. 11/27/2014, Bing Li
            AdminIORegistry.REGISTRY().addIO(serverIO);
            // Execute the new created administration IO concurrently to respond the crawlers requests and
notifications in an asynchronous manner. 11/27/2014, Bing Li
            super.execute(serverIO);
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
        }
      }
    }
```

- **AdminListenerDisposer**

```
package com.greatfree.testing.coordinator.admin;

import com.greatfree.reuse.RunDisposable;

/*
 * The class is responsible for disposing the instance of AdminServerListener by invoking its method of
shutdown(). 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class AdminListenerDisposer implements RunDisposable<AdminListener>
{
   /*
    * Dispose the instance of AdminServerListener. 11/29/2014, Bing Li
    */
   @Override
   public void dispose(AdminListener r)
   {
      r.shutdown();
   }

   /*
    * Dispose the instance of AdminServerListener. The method does not make sense to
AdminServerListener. Just leave it here for the requirement of the interface,
RunDisposable<AdminServerListener>. 11/29/2014, Bing Li
    */
   @Override
   public void dispose(AdminListener r, long time)
   {
      r.shutdown();
   }
}
```

- **AdminServerDispatcher**

```
package com.greatfree.testing.coordinator.admin;

import com.greatfree.concurrency.NotificationDispatcher;
import com.greatfree.concurrency.ServerMessageDispatcher;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.MessageType;
import com.greatfree.testing.message.ShutdownCrawlServerNotification;

/*
 * This is an implementation of ServerMessageDispatcher. It contains the concurrency mechanism to
respond the administrator's notifications for the coordinator. As the administrator is a human operator,
the design is more complicated than required. Just an exercise. :-) 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class AdminServerDispatcher extends ServerMessageDispatcher<ServerMessage>
{
   // Declare a notification dispatcher to process the crawler shutting-down concurrently. 11/27/2014,
Bing Li
   private NotificationDispatcher<ShutdownCrawlServerNotification, ShutdownCrawlServerThread,
ShutdownCrawlServerThreadCreator> shutdownCrawlServerNotificationDispatcher;

   /*
    * Initialize. 11/27/2014, Bing Li
    */
   public AdminServerDispatcher(int corePoolSize, long keepAliveTime)
   {
      // Set the pool size and threads' alive time. 11/27/2014, Bing Li
      super(corePoolSize, keepAliveTime);

      // Initialize the crawler registration dispatcher. 11/27/2014, Bing Li
      this.shutdownCrawlServerNotificationDispatcher = new
NotificationDispatcher<ShutdownCrawlServerNotification, ShutdownCrawlServerThread,
ShutdownCrawlServerThreadCreator>(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new
ShutdownCrawlServerThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
      // Set the parameters to check idle states of threads. 11/27/2014, Bing Li

   this.shutdownCrawlServerNotificationDispatcher.setIdleChecker(ServerConfig.NOTIFICATION_DISP
ATCHER_IDLE_CHECK_DELAY,
ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
      // Start the dispatcher. 11/27/2014, Bing Li
      super.execute(this.shutdownCrawlServerNotificationDispatcher);
   }

   /*
    * Shut down the administration message dispatcher. 11/27/2014, Bing Li
    */
   public void shutdown()
   {
      // Dispose the crawler shutting-down dispatcher. 11/27/2014, Bing Li
      this.shutdownCrawlServerNotificationDispatcher.dispose();
      // Shutdown the server message dispatcher. 11/27/2014, Bing Li
      super.shutdown();
   }

   /*
    * Process the available messages in a concurrent way. 11/27/2014, Bing Li
    */
   public void comsume(OutMessageStream<ServerMessage> message)
```

```
  {
    // Check the types of received messages. 11/27/2014, Bing Li
    switch (message.getMessage().getType())
    {
      // If the message is the notification to register the crawler server. 11/27/2014, Bing Li
      case MessageType.SHUTDOWN_CRAWL_SERVER_NOTIFICATION:
        // Enqueue the notification into the dispatcher for concurrent feedback. 11/27/2014, Bing Li

  this.shutdownCrawlServerNotificationDispatcher.enqueue((ShutdownCrawlServerNotification)messa
ge.getMessage());
        break;
    }
  }
}
```

- **AdminServerProducerDisposer**

```
package com.greatfree.testing.coordinator.admin;

import com.greatfree.concurrency.MessageProducer;
import com.greatfree.reuse.ThreadDisposable;

/*
 * The class is responsible for disposing the administration message producer of the coordinator.
11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class AdminServerProducerDisposer implements
ThreadDisposable<MessageProducer<AdminServerDispatcher>>
{
   /*
    * Dispose the message producer. 11/27/2014, Bing Li
    */
   @Override
   public void dispose(MessageProducer<AdminServerDispatcher> r)
   {
      r.dispose();
   }

   /*
    * The method does not make sense to the class of MessageProducer. Just leave it here. 11/27/2014,
Bing Li
    */
   @Override
   public void dispose(MessageProducer<AdminServerDispatcher> r, long time)
   {
      r.dispose();
   }
}
```

- **AdminMulticastor**

```
package com.greatfree.testing.coordinator.admin;

import java.io.IOException;

import com.greatfree.reuse.ResourcePool;
import com.greatfree.testing.coordinator.crawling.CrawlServerClientPool;
import com.greatfree.testing.data.ServerConfig;

/*
 * This is a singleton that contains all of the multicastor pools to administer the system. 11/27/2014, Bing
Li
 */

// Created: 11/27/2014, Bing Li
public class AdminMulticastor
{
   // The pool for the multicastor which multicasts the notification of StopCrawlMultiNotification.
11/27/2014, Bing Li
   private ResourcePool<StopCrawlMulticastorSource, StopCrawlMulticastor,
StopCrawlMulticastorCreator, StopCrawlMulticastorDisposer> stopCrawlMulticastorPool;

   private AdminMulticastor()
   {
   }

   /*
    * A singleton implementation. 11/27/2014, Bing Li
    */
   private static AdminMulticastor instance = new AdminMulticastor();

   public static AdminMulticastor ADMIN()
   {
      if (instance == null)
      {
         instance = new AdminMulticastor();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Dispose all of the pools. 11/27/2014, Bing Li
    */
   public void dispose()
   {
      this.stopCrawlMulticastorPool.shutdown();
   }

   /*
    * Initialize the pools. 11/27/2014, Bing Li
    */
   public void init()
   {
      this.stopCrawlMulticastorPool = new ResourcePool<StopCrawlMulticastorSource,
StopCrawlMulticastor, StopCrawlMulticastorCreator,
StopCrawlMulticastorDisposer>(ServerConfig.MULTICASTOR_POOL_SIZE, new
StopCrawlMulticastorCreator(), new StopCrawlMulticastorDisposer(),
ServerConfig.MULTICASTOR_POOL_WAIT_TIME);
   }

   /*
```

```
    * Disseminate the notification of StartCrawlMultiNotification to all of the crawlers. 11/27/2014, Bing Li
    */
   public void disseminateStopCrawl() throws InstantiationException, IllegalAccessException,
IOException, InterruptedException
   {
      // Get an instance of StartCrawlMulticastor from the pool. 11/27/2014, Bing Li
      StopCrawlMulticastor multicastor = this.stopCrawlMulticastorPool.get(new
StopCrawlMulticastorSource(CrawlServerClientPool.COORDINATE().getPool(),
ServerConfig.ROOT_MULTICAST_BRANCH_COUNT, ServerConfig.MULTICAST_BRANCH_COUNT,
new StopCrawlNotificationCreator()));
      // Check whether the multicastor is valid. 11/27/2014, Bing Li
      if (multicastor != null)
      {
         // Disseminate the notification. The notification contains no data. Thus, it is not necessary to put
any arguments here. Just place a null. 11/27/2014, Bing Li
         multicastor.disseminate(null);
         // Collect the instance of StartCrawlMulticastor. 11/27/2014, Bing Li
         this.stopCrawlMulticastorPool.collect(multicastor);
      }
   }
}
```

- **AdminIO**

```
package com.greatfree.testing.coordinator.admin;

import java.io.IOException;
import java.net.Socket;
import java.net.SocketException;

import com.greatfree.concurrency.Collaborator;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.remote.ServerIO;
import com.greatfree.testing.coordinator.CoordinatorMessageProducer;

/*
 * The class is actually an implementation of ServerIO, which serves for the administrator which interacts with the coordinator. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class AdminIO extends ServerIO
{
  /*
   * Initialize the server IO. The socket is the connection between the crawler and the coordinator. The collaborator is shared with other IOs to control the count of ServerIOs instances. 11/27/2014, Bing Li
   */
  public AdminIO(Socket clientSocket, Collaborator collaborator)
  {
    super(clientSocket, collaborator);
  }

  /*
   * A concurrent method to respond the received messages asynchronously. 11/27/2014, Bing Li
   */
  public void run()
  {
    ServerMessage message;
    while (!super.isShutdown())
    {
      try
      {
        // Wait and read messages from the administrator. 11/27/2014, Bing Li
        message = (ServerMessage)super.read();
        // Convert the received message to OutMessageStream and put it into the relevant dispatcher for concurrent processing. 11/27/2014, Bing Li
        CoordinatorMessageProducer.SERVER().produceAdminMessage(new OutMessageStream<ServerMessage>(super.getOutStream(), super.getLock(), message));
      }
      catch (SocketException e)
      {
        e.printStackTrace();
      }
      catch (IOException e)
      {
        e.printStackTrace();
      }
      catch (ClassNotFoundException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **AdminIORegistry**

```
package com.greatfree.testing.coordinator.admin;

import java.io.IOException;
import java.util.Set;

import com.greatfree.remote.ServerIORegistry;

/*
 * The class keeps all of AdminIOs. This is a singleton wrapper of ServerIORegistry. 11/24/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class AdminIORegistry
{
   // Declare an instance of ServerIORegistry for AdminIOs. 11/24/2014, Bing Li
   private ServerIORegistry<AdminIO> registry;

   /*
    * Initializing ... 11/24/2014, Bing Li
    */
   private AdminIORegistry()
   {
   }

   private static AdminIORegistry instance = new AdminIORegistry();

   public static AdminIORegistry REGISTRY()
   {
      if (instance == null)
      {
         instance = new AdminIORegistry();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Dispose the registry. 11/24/2014, Bing Li
    */
   public void dispose() throws IOException
   {
      this.registry.removeAllIOs();
   }

   /*
    * Initialize the registry. 11/24/2014, Bing Li
    */
   public void init()
   {
      this.registry = new ServerIORegistry<AdminIO>();
   }

   /*
    * Add a new instance of AdminIO to the registry. 11/24/2014, Bing Li
    */
   public void addIO(AdminIO io)
   {
      this.registry.addIO(io);
   }

   /*
```

```
    * Get all of the IPs of the connected clients from the corresponding AdminIOs. 11/24/2014, Bing Li
    */
   public Set<String> getIPs()
   {
      return this.registry.getIPs();
   }

   /*
    * Get the count of the registered AdminIOs. 11/24/2014, Bing Li
    */
   public int getIOCount()
   {
      return this.registry.getIOCount();
   }

   /*
    * Remove or unregister an AdminIO. It is executed when a client is down or the connection gets
something wrong. 11/24/2014, Bing Li
    */
   public void removeIO(AdminIO io) throws IOException
   {
      this.registry.removeIO(io);
   }

   /*
    * Remove or unregister all of the registered AdminIOs. It is executed when the server process is
shutdown. 11/24/2014, Bing Li
    */
   public void removeAllIOs() throws IOException
   {
      this.registry.removeAllIOs();
   }
}
```

- **ShutdownCrawlServerThread**

```
package com.greatfree.testing.coordinator.admin;

import java.io.IOException;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.testing.message.ShutdownCrawlServerNotification;

/*
 * The thread disseminates the multicasting notification to all of the clusters to stop the crawling
procedure. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class ShutdownCrawlServerThread extends
NotificationQueue<ShutdownCrawlServerNotification>
{
  /*
   * Initialize the thread. 11/27/2014, Bing Li
   */
  public ShutdownCrawlServerThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * Process the notification concurrently. 11/27/2014, Bing Li
   */
  public void run()
  {
    // The instance of ShutdownCrawlServerNotification. 11/27/2014, Bing Li
    ShutdownCrawlServerNotification notification;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/27/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/27/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/27/2014, Bing Li
          notification = this.getNotification();
          try
          {
            // Disseminate the notification to demand all of the crawlers to stop crawling. 11/27/2014,
Bing Li
            AdminMulticastor.ADMIN().disseminateStopCrawl();
          }
          catch (InstantiationException e)
          {
            e.printStackTrace();
          }
          catch (IllegalAccessException e)
          {
            e.printStackTrace();
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
          this.disposeMessage(notification);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
```

```
        }
      }
    }
  }
}
```

- **ShutdownCrawlServerThreadCreator**

```
package com.greatfree.testing.coordinator.admin;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.ShutdownCrawlServerNotification;

/*
 * The creator here attempts to create instances of ShutdownCrawlServerThread. It works with the
notification dispatcher to schedule the tasks concurrently. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class ShutdownCrawlServerThreadCreator implements
NotificationThreadCreatable<ShutdownCrawlServerNotification, ShutdownCrawlServerThread>
{
  @Override
  public ShutdownCrawlServerThread createNotificationThreadInstance(int taskSize)
  {
    return new ShutdownCrawlServerThread(taskSize);
  }
}
```

- **StopCrawlMulticastor**

```
package com.greatfree.testing.coordinator.admin;

import com.greatfree.multicast.RootObjectMulticastor;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.message.StopCrawlMultiNotification;
import com.greatfree.util.NullObject;

/*
 * This is an extending of RootObjectMulticastor to transfer the notification of
StopCrawlServerMultiNotification to all of the crawlers. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StopCrawlMulticastor extends RootObjectMulticastor<NullObject,
StopCrawlMultiNotification, StopCrawlNotificationCreator>
{
   public StopCrawlMulticastor(FreeClientPool clientPool, int rootBranchCount, int treeBranchCount,
StopCrawlNotificationCreator messageCreator)
   {
      super(clientPool, rootBranchCount, treeBranchCount, messageCreator);
   }
}
```

- **StopCrawlMulticastorCreator**

```
package com.greatfree.testing.coordinator.admin;

import com.greatfree.reuse.HashCreatable;

/*
 * The class intends to create the interface of StopCrawlMulticastor. It is used by the resource pool to
manage the multicastors efficiently. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StopCrawlMulticastorCreator implements HashCreatable<StopCrawlMulticastorSource,
StopCrawlMulticastor>
{
   /*
    * Define the method to create the instances of StopCrawlMulticastor upon the source,
StopCrawlMulticastorSource. 11/27/2014, Bing Li
    */
   @Override
   public StopCrawlMulticastor createResourceInstance(StopCrawlMulticastorSource source)
   {
      return new StopCrawlMulticastor(source.getClientPool(), source.getRootBranchCount(),
source.getTreeBranchCount(), source.getCreator());
   }
}
```

- **StopCrawlMulticastorDisposer**

```
package com.greatfree.testing.coordinator.admin;

import com.greatfree.reuse.HashDisposable;

/*
 * The disposer collects the instance of StopCrawlMulticastor. 11/26/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StopCrawlMulticastorDisposer implements HashDisposable<StopCrawlMulticastor>
{
   @Override
   public void dispose(StopCrawlMulticastor t)
   {
      t.dispose();
   }
}
```

- **StopCrawlMulticastorSource**

```
package com.greatfree.testing.coordinator.admin;

import com.greatfree.multicast.RootMessageCreatorGettable;
import com.greatfree.multicast.RootMulticastorSource;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.message.StopCrawlMultiNotification;
import com.greatfree.util.NullObject;

/*
 * The class provides the pool with the initial values to create a StopCrawlMulticastor. The sources that are needed to create an instance of RootMulticastor are enclosed in the class. That is required by the pool to create multicastors. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StopCrawlMulticastorSource extends RootMulticastorSource<NullObject, StopCrawlMultiNotification, StopCrawlNotificationCreator> implements RootMessageCreatorGettable<StopCrawlMultiNotification, NullObject>
{
  /*
   * Initialize the source. 11/27/2014, Bing Li
   */
  public StopCrawlMulticastorSource(FreeClientPool clientPool, int rootBranchCount, int treeBranchCount, StopCrawlNotificationCreator creator)
  {
    super(clientPool, rootBranchCount, treeBranchCount, creator);
  }

  /*
   * Expose the message creator. 11/26/2014, Bing Li
   */
  @Override
  public StopCrawlNotificationCreator getMessageCreator()
  {
    return super.getCreator();
  }
}
```

- **StopCrawlNotificationCreator**

```
package com.greatfree.testing.coordinator.admin;

import java.util.HashMap;

import com.greatfree.multicast.ObjectMulticastCreatable;
import com.greatfree.testing.message.StopCrawlMultiNotification;
import com.greatfree.util.NullObject;
import com.greatfree.util.Tools;

/*
 * The creator is used to create the instance of StopCrawlServerMultiNotification. It works with a
multicastor to do that. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StopCrawlNotificationCreator implements
ObjectMulticastCreatable<StopCrawlMultiNotification, NullObject>
{
  /*
   * Create an instance of StopCrawlServerMultiNotification. 11/27/2014, Bing Li
   *
   * For the specific notification, StopCrawlServerMultiNotification, no arguments are needed to send.
Therefore, the NullObject is put here. For other notification, an object that contains all of the arguments
must be enclosed in the object.
   *
   * The constructor needs to input the children nodes for further multicast.
   */
  @Override
  public StopCrawlMultiNotification createInstanceWithChildren(NullObject message,
HashMap<String, String> childrenMap)
  {
    return new StopCrawlMultiNotification(Tools.generateUniqueKey(), childrenMap);
  }

  /*
   * Create an instance of StartCrawlMultiNotification. 11/27/2014, Bing Li
   *
   * For the specific notification, StartCrawlMultiNotification, no arguments are needed to send.
Therefore, the NullObject is put here. For other notification, an object that contains all of the arguments
must be enclosed in the object.
   */
  @Override
  public StopCrawlMultiNotification createInstanceWithoutChildren(NullObject message)
  {
    return new StopCrawlMultiNotification(Tools.generateUniqueKey());
  }
}
```

3.6.1.2 The Crawling

- **CrawlListener**

```
package com.greatfree.testing.coordinator.crawling;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

import com.greatfree.remote.IPPort;
import com.greatfree.remote.ServerListener;
import com.greatfree.testing.data.ServerConfig;

/*
 * This is a coordinator listener that not only responds to crawlers but also intends to connect and communicate with them. It assists the coordinator to interact with crawlers in an eventing manner other than waiting for requests and then responding only. 11/24/2014, Bing Li
 */

// Created: 11/24/2014, Bing Li
public class CrawlListener extends ServerListener implements Runnable
{
  // A thread to connect the remote crawler concurrently. 11/24/2014, Bing Li
  private ConnectCrawlServerThread connectThread;

  /*
   * Initialize the listener. 11/24/2014, Bing Li
   */
  public CrawlListener(ServerSocket serverSocket, int threadPoolSize, long keepAliveTime)
  {
    super(serverSocket, threadPoolSize, keepAliveTime);
  }

  /*
   * Shutdown the listener. 11/24/2014, Bing Li
   */
  public void shutdown()
  {
    // Dispose the connecting thread. 11/24/2014, Bing Li
    this.connectThread.dispose();
    // Shutdown the listener. 11/24/2014, Bing Li
    super.shutdown();
  }

  /*
   * The task that must be executed concurrently. 11/24/2014, Bing Li
   */
  @Override
  public void run()
  {
    Socket clientSocket;
    CrawlIO serverIO;

    // Detect whether the listener is shutdown. If not, it must be running all the time to wait for potential connections from crawlers. 11/24/2014, Bing Li
    while (!super.isShutdown())
    {
      try
      {
        // Wait and accept a connecting from a possible crawler. 11/24/2014, Bing Li
        clientSocket = super.accept();
        // Check whether the connected server IOs exceed the upper limit. 11/24/2014, Bing Li
        if (CrawlIORegistry.REGISTRY().getIOCount() >= ServerConfig.MAX_SERVER_IO_COUNT)
        {
          try
          {
            // If the upper limit is reached, the listener has to wait until an existing server IO is disposed.
```

```
11/24/2014, Bing Li
                super.holdOn();
              }
              catch (InterruptedException e)
              {
                e.printStackTrace();
              }
            }

            // If the upper limit of IOs is not reached, a crawling server IO is initialized. A common
Collaborator and the socket are the initial parameters. The shared common collaborator guarantees all
of the crawling server IOs from a certain crawler could notify with each other with the same lock. Then,
the upper limit of crawling server IOs is under the control. 11/24/2014, Bing Li
            serverIO = new CrawlIO(clientSocket, super.getCollaborator());

            /*
             * Since the listener servers need to form a peer-to-peer architecture, it is required to connect to
the remote crawling server. Thus, the local server can send messages without waiting for any requests
from the remote end. 11/24/2014, Bing Li
             */

            // Check whether a client to the IP and the port number of the remote crawling server is existed
in the client pool. If such a client does not exist, it is required to connect the remote crawler by the thread
concurrently. Doing that concurrently is to speed up the rate of responding to the crawling server.
11/24/2014, Bing Li
            if (!CrawlServerClientPool.COORDINATE().getPool().isClientExisted(serverIO.getIP(),
ServerConfig.CRAWL_SERVER_PORT))
            {
              // Since the client does not exist in the pool, input the IP address and the port number to the
thread. 11/25/2014, Bing Li
              this.connectThread.enqueue(new IPPort(serverIO.getIP(),
ServerConfig.CRAWL_SERVER_PORT));
              // Execute the thread to connect to the remote crawler. 11/25/2014, Bing Li
              super.execute(this.connectThread);
            }

            // Add the new created server IO into the registry for further management. 11/25/2014, Bing Li
            CrawlIORegistry.REGISTRY().addIO(serverIO);
            // Execute the new created crawling IO concurrently to respond the crawlers' requests and
notifications in an asynchronous manner. 11/25/2014, Bing Li
            super.execute(serverIO);
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
        }
      }
    }
```

- **CrawlListenerDisposer**

```
package com.greatfree.testing.coordinator.crawling;

import com.greatfree.reuse.RunDisposable;

/*
 * The class is responsible for disposing the instance of CrawlListener by invoking its method of
shutdown(). 11/25/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class CrawlListenerDisposer implements RunDisposable<CrawlListener>
{
   /*
    * Dispose the instance of CrawlListener. 11/25/2014, Bing Li
    */
   @Override
   public void dispose(CrawlListener r)
   {
      r.shutdown();
   }

   /*
    * Dispose the instance of CrawlListener. The method does not make sense to CrawlListener. Just
leave it here for the requirement of the interface, RunDisposable<CrawlListener>. 11/25/2014, Bing Li
    */
   @Override
   public void dispose(CrawlListener r, long time)
   {
      r.shutdown();
   }
}
```

- **CrawlIO**

```
package com.greatfree.testing.coordinator.crawling;

import java.io.IOException;
import java.net.Socket;
import java.net.SocketException;

import com.greatfree.concurrency.Collaborator;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.remote.ServerIO;
import com.greatfree.testing.coordinator.CoordinatorMessageProducer;

/*
 * The class is actually an implementation of ServerIO, which serves for the crawlers which access the coordinator. 11/24/2014, Bing Li
 */

// Created: 11/24/2014, Bing Li
public class CrawlIO extends ServerIO
{
  /*
   * Initialize the server IO. The socket is the connection between the crawler and the coordinator. The collaborator is shared with other IOs to control the count of ServerIOs instances. 11/24/2014, Bing Li
   */
  public CrawlIO(Socket clientSocket, Collaborator collaborator)
  {
    super(clientSocket, collaborator);
  }

  /*
   * A concurrent method to respond the received messages asynchronously. 11/24/2014, Bing Li
   */
  public void run()
  {
    ServerMessage message;
    while (!super.isShutdown())
    {
      try
      {
        // Wait and read messages from a crawling server. 11/24/2014, Bing Li
        message = (ServerMessage)super.read();
        // Convert the received message to OutMessageStream and put it into the relevant dispatcher for concurrent processing. 11/24/2014, Bing Li
        CoordinatorMessageProducer.SERVER().produceCrawlingMessage(new OutMessageStream<ServerMessage>(super.getOutStream(), super.getLock(), message));
      }
      catch (SocketException e)
      {
        e.printStackTrace();
      }
      catch (IOException e)
      {
        e.printStackTrace();
      }
      catch (ClassNotFoundException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **CrawlIORegistry**

```
package com.greatfree.testing.coordinator.crawling;

import java.io.IOException;
import java.util.Set;

import com.greatfree.remote.ServerIORegistry;

/*
 * The class keeps all of CrawlIOs. This is a singleton wrapper of ServerIORegistry. 11/24/2014, Bing Li
 */

// Created: 11/24/2014, Bing Li
public class CrawlIORegistry
{
  // Declare an instance of ServerIORegistry for CrawlIOs. 11/24/2014, Bing Li
  private ServerIORegistry<CrawlIO> registry;

  /*
   * Initializing ... 11/24/2014, Bing Li
   */
  private CrawlIORegistry()
  {
  }

  private static CrawlIORegistry instance = new CrawlIORegistry();

  public static CrawlIORegistry REGISTRY()
  {
    if (instance == null)
    {
      instance = new CrawlIORegistry();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the registry. 11/24/2014, Bing Li
   */
  public void dispose() throws IOException
  {
    this.registry.removeAllIOs();
  }

  /*
   * Initialize the registry. 11/24/2014, Bing Li
   */
  public void init()
  {
    this.registry = new ServerIORegistry<CrawlIO>();
  }

  /*
   * Add a new instance of CrawlIO to the registry. 11/24/2014, Bing Li
   */
  public void addIO(CrawlIO io)
  {
    this.registry.addIO(io);
  }

  /*
```

```
    * Get all of the IPs of the connected clients from the corresponding CrawlIOs. 11/24/2014, Bing Li
    */
   public Set<String> getIPs()
   {
      return this.registry.getIPs();
   }

   /*
    * Get the count of the registered CrawlIOs. 11/24/2014, Bing Li
    */
   public int getIOCount()
   {
      return this.registry.getIOCount();
   }

   /*
    * Remove or unregister an CrawlIO. It is executed when a client is down or the connection gets
something wrong. 11/24/2014, Bing Li
    */
   public void removeIO(CrawlIO io) throws IOException
   {
      this.registry.removeIO(io);
   }

   /*
    * Remove or unregister all of the registered CrawlIOs. It is executed when the server process is
shutdown. 11/24/2014, Bing Li
    */
   public void removeAllIOs() throws IOException
   {
      this.registry.removeAllIOs();
   }
}
```

- **CrawlServerClientPool**

```
package com.greatfree.testing.coordinator.crawling;

import java.io.IOException;

import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.coordinator.CoorConfig;

/*
 * This is a pool that creates and manages instances of FreeClient to achieve the goal of using minimum instances of clients to reach a high performance. 11/24/2014, Bing Li
 */

// Created: 11/24/2014, Bing Li
public class CrawlServerClientPool
{
  // An instance of FreeClientPool is defined to interact with the crawler. 11/23/2014, Bing Li
  private FreeClientPool clientPool;

  private CrawlServerClientPool()
  {
  }

  /*
   * A singleton definition. 11/24/2014, Bing Li
   */
  private static CrawlServerClientPool instance = new CrawlServerClientPool();

  public static CrawlServerClientPool COORDINATE()
  {
    if (instance == null)
    {
      instance = new CrawlServerClientPool();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the free client pool. 11/24/2014, Bing Li
   */
  public void dispose()
  {
    try
    {
      this.clientPool.dispose();
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
  }

  /*
   * Initialize the client pool. The method is called when the coordinator process is started. 11/24/2014, Bing Li
   */
  public void init()
  {
    // Initialize the client pool. 11/24/2014, Bing Li
    this.clientPool = new FreeClientPool(CoorConfig.CSERVER_CLIENT_POOL_SIZE);
    // Set idle checking for the client pool. 11/24/2014, Bing Li
```

```
    this.clientPool.setIdleChecker(CoorConfig.CSERVER_CLIENT_IDLE_CHECK_DELAY,
CoorConfig.CSERVER_CLIENT_IDLE_CHECK_PERIOD,
CoorConfig.CSERVER_CLIENT_MAX_IDLE_TIME);
  }

  /*
   * Expose the client pool. 11/24/2014, Bing Li
   */
  public FreeClientPool getPool()
  {
    return this.clientPool;
  }
}
```

- **CrawlServerDispatcher**

```
package com.greatfree.testing.coordinator.crawling;

import com.greatfree.concurrency.NotificationDispatcher;
import com.greatfree.concurrency.ServerMessageDispatcher;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.CrawledLinksNotification;
import com.greatfree.testing.message.MessageType;
import com.greatfree.testing.message.RegisterCrawlServerNotification;
import com.greatfree.testing.message.UnregisterCrawlServerNotification;

/*
 * This is an implementation of ServerMessageDispatcher. It contains the concurrency mechanism to
respond crawlers' requests and receive crawlers' notifications for the coordinator. 11/24/2014, Bing Li
 */

// Created: 11/24/2014, Bing Li
public class CrawlServerDispatcher extends ServerMessageDispatcher<ServerMessage>
{
   // Declare a notification dispatcher to process the crawler registration concurrently. 11/27/2014, Bing Li
   private NotificationDispatcher<RegisterCrawlServerNotification, RegisterCrawlServerThread,
RegisterCrawlServerThreadCreator> registerCrawlServerNotificationDispatcher;
   // Declare a notification dispatcher to process the crawler unregistering concurrently. 11/28/2014, Bing
Li
   private NotificationDispatcher<UnregisterCrawlServerNotification, UnregisterCrawlServerThread,
UnregisterCrawlServerThreadCreator> unregisterCrawlServerNotificationDispatcher;
   // Declare a notification dispatcher to distribute crawled links concurrently. 11/28/2014, Bing Li
   private NotificationDispatcher<CrawledLinksNotification, DistributeLinksThread,
DistributeLinksThreadCreator> distributeCrawledLinksNotificationDispatcher;

   /*
    * Initialize. 11/24/2014, Bing Li
    */
   public CrawlServerDispatcher(int corePoolSize, long keepAliveTime)
   {
      // Set the pool size and threads' alive time. 11/27/2014, Bing Li
      super(corePoolSize, keepAliveTime);

      // Initialize the crawler registration dispatcher. 11/27/2014, Bing Li
      this.registerCrawlServerNotificationDispatcher = new
NotificationDispatcher<RegisterCrawlServerNotification, RegisterCrawlServerThread,
RegisterCrawlServerThreadCreator>(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new
RegisterCrawlServerThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
      // Set the parameters to check idle states of threads. 11/27/2014, Bing Li

   this.registerCrawlServerNotificationDispatcher.setIdleChecker(ServerConfig.NOTIFICATION_DISPA
TCHER_IDLE_CHECK_DELAY,
ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
      // Start the dispatcher. 11/27/2014, Bing Li
      super.execute(this.registerCrawlServerNotificationDispatcher);

      // Initialize the crawler unregistering dispatcher. 11/27/2014, Bing Li
      this.unregisterCrawlServerNotificationDispatcher = new
NotificationDispatcher<UnregisterCrawlServerNotification, UnregisterCrawlServerThread,
UnregisterCrawlServerThreadCreator>(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new
UnregisterCrawlServerThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
      // Set the parameters to check idle states of threads. 11/27/2014, Bing Li
```

```
    this.unregisterCrawlServerNotificationDispatcher.setIdleChecker(ServerConfig.NOTIFICATION_DISP
ATCHER_IDLE_CHECK_DELAY,
ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
        // Start the dispatcher. 11/27/2014, Bing Li
        super.execute(this.unregisterCrawlServerNotificationDispatcher);

        // Initialize the distributing crawled links dispatcher. 11/28/2014, Bing Li
        this.distributeCrawledLinksNotificationDispatcher = new
NotificationDispatcher<CrawledLinksNotification, DistributeLinksThread,
DistributeLinksThreadCreator>(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new
DistributeLinksThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
        // Set the parameters to check idle states of threads. 11/28/2014, Bing Li

    this.distributeCrawledLinksNotificationDispatcher.setIdleChecker(ServerConfig.NOTIFICATION_DIS
PATCHER_IDLE_CHECK_DELAY,
ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
        // Start the dispatcher. 11/28/2014, Bing Li
        super.execute(this.distributeCrawledLinksNotificationDispatcher);
    }

    /*
     * Shut down the crawling message dispatcher. 11/24/2014, Bing Li
     */
    public void shutdown()
    {
        // Dispose the crawler registration dispatcher. 11/27/2014, Bing Li
        this.registerCrawlServerNotificationDispatcher.dispose();
        // Dispose the crawler unregistering dispatcher. 11/28/2014, Bing Li
        this.unregisterCrawlServerNotificationDispatcher.dispose();
        // Dispose the distributing crawled links dispatcher. 11/28/2014, Bing Li
        this.distributeCrawledLinksNotificationDispatcher.dispose();
        // Shutdown the server message dispatcher. 11/27/2014, Bing Li
        super.shutdown();
    }

    /*
     * Process the available messages in a concurrent way. 11/24/2014, Bing Li
     */
    public void consume(OutMessageStream<ServerMessage> message)
    {
        // Check the types of received messages. 11/27/2014, Bing Li
        switch (message.getMessage().getType())
        {
            // If the message is the notification to register the crawler server. 11/27/2014, Bing Li
            case MessageType.REGISTER_CRAWL_SERVER_NOTIFICATION:
                // Enqueue the notification into the dispatcher for concurrent feedback. 11/27/2014, Bing Li

    this.registerCrawlServerNotificationDispatcher.enqueue((RegisterCrawlServerNotification)message.g
etMessage());
                break;

            // If the message is the notification to unregister the crawler server. 11/28/2014, Bing Li
            case MessageType.UNREGISTER_CRAWL_SERVER_NOTIFICATION:
                // Enqueue the notification into the dispatcher for concurrent feedback. 11/28/2014, Bing Li

    this.unregisterCrawlServerNotificationDispatcher.enqueue((UnregisterCrawlServerNotification)messa
ge.getMessage());
                break;

            // If the message is the notification which contains the crawled link. 11/28/2014, Bing Li
            case MessageType.CRAWLED_LINKS_NOTIFICATION:
                // Enqueue the notification into the dispatcher for concurrent feedback. 11/28/2014, Bing Li

    this.distributeCrawledLinksNotificationDispatcher.enqueue((CrawledLinksNotification)message.getM
```

```
essage());
            break;
        }
    }
}
```

- **CrawlServerProducerDisposer**

```
package com.greatfree.testing.coordinator.crawling;

import com.greatfree.concurrency.MessageProducer;
import com.greatfree.reuse.ThreadDisposable;

/*
 * The class is responsible for disposing the crawling message producer of the coordinator. 11/24/2014,
Bing Li
 */

// Created: 11/24/2014, Bing Li
public class CrawlServerProducerDisposer implements
ThreadDisposable<MessageProducer<CrawlServerDispatcher>>
{
   /*
    * Dispose the message producer. 11/24/2014, Bing Li
    */
   @Override
   public void dispose(MessageProducer<CrawlServerDispatcher> r)
   {
      r.dispose();
   }

   /*
    * The method does not make sense to the class of MessageProducer. Just leave it here. 11/24/2014,
Bing Li
    */
   @Override
   public void dispose(MessageProducer<CrawlServerDispatcher> r, long time)
   {
      r.dispose();
   }
}
```

- **ConnectCrawlServerThread**

```
package com.greatfree.testing.coordinator.crawling;

import java.io.IOException;
import java.util.Queue;
import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.remote.IPPort;
import com.greatfree.testing.message.NodeKeyNotification;

/*
 * The class is derived from Thread. It is responsible for connecting the remote crawling server such that it is feasible for the server to notify the client even without the crawling server's request. 08/10/2014, Bing Li
 */

// Created: 11/24/2014, Bing Li
public class ConnectCrawlServerThread extends Thread
{
  // The queue keeps the clients' IPs which need to be connected to. 11/24/2014, Bing Li
  private Queue<IPPort> ipQueue;

  /*
   * Initialize. 11/24/2014, Bing Li
   */
  public ConnectCrawlServerThread()
  {
    this.ipQueue = new LinkedBlockingQueue<IPPort>();
  }

  /*
   * Dispose the resource of the class. 11/24/2014, Bing Li
   */
  public synchronized void dispose()
  {
    if (this.ipQueue != null)
    {
      this.ipQueue.clear();
      this.ipQueue = null;
    }
  }

  /*
   * Input the IP address and the port number into the queue. They are connected in the order of first-in-first-out. The CrawlServerClientPool is responsible for the connections. 11/24/2014, Bing Li
   */
  public void enqueue(IPPort ipPort)
  {
    this.ipQueue.add(ipPort);
  }

  /*
   * Connect the remote IP addresses and the associated port numbers concurrently. 11/24/2014, Bing Li
   */
  @Override
  public void run()
  {
    IPPort ipPort;
    // The thread keeps working until all of the IP addresses are connected. 11/24/2014, Bing Li
    while (this.ipQueue.size() > 0)
    {
      // Dequeue the IP addresses. 11/24/2014, Bing Li
      ipPort = this.ipQueue.poll();
      try
```

```
        {
            // Notify the remote node its unique ID. This is usually used to set up a multicasting cluster which
contains a bunch of nodes. Each of them is identified by the ID. 11/24/2014, Bing Li
            CrawlServerClientPool.COORDINATE().getPool().send(ipPort, new
NodeKeyNotification(ipPort.getObjectKey()));
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
  }
}
```

- **CrawlRegistry**

```
package com.greatfree.testing.coordinator.crawling;

import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

/*
 * This is a registry to assist the management of all of the crawlers. 11/25/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class CrawlRegistry
{
  // A list that contains all of the keys of the crawlers. 11/25/2014, Bing Li
  private List<String> crawlDCKeys;
  // It is the current registered total count of URLs. 11/25/2014, Bing Li
  private long totalURLCount;
  // A flag represents why a crawler has any tasks. If it is false, it denotes that it is necessary to distribute crawling load again, i.e., resetting. 11/25/2014, Bing Li
  private boolean shouldReset;

  private CrawlRegistry()
  {
    this.crawlDCKeys = new CopyOnWriteArrayList<String>();
    this.totalURLCount = 0;
    this.shouldReset = false;
  }

  /*
   * A singleton definition. 11/25/2014, Bing Li
   */
  private static CrawlRegistry instance = new CrawlRegistry();

  public static CrawlRegistry COORDINATE()
  {
    if (instance == null)
    {
      instance = new CrawlRegistry();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the registry. 11/25/2014, Bing Li
   */
  public void dispose()
  {
    this.crawlDCKeys.clear();
  }

  /*
   * Register a new online crawler and its URL load. 11/25/2014, Bing Li
   */
  public synchronized void register(String dcKey, long localURLCount)
  {
    // Check whether the list contains the key of the crawler. 11/25/2014, Bing Li
    if (!this.crawlDCKeys.contains(dcKey))
    {
      // Put the crawler key into the list. 11/25/2014, Bing Li
      this.crawlDCKeys.add(dcKey);
    }
```

```
    // Accumulate the URL load. 11/25/2014, Bing Li
    this.totalURLCount += localURLCount;
    // Detect whether the URL load of the crawler is zero, it represents that the crawler has no workload.
Thus, it is required to reset the crawling load to the crawlers. 11/25/2014, Bing Li
    if (localURLCount == 0)
    {
      // Set the reset flag. 11/25/2014, Bing Li
      this.shouldReset = true;
    }
  }

  /*
   * Unregister a crawler. 11/25/2014, Bing Li
   */
  public synchronized void unregister(String dcKey)
  {
    // Check whether the list contains the crawler key. 11/25/2014, Bing Li
    if (this.crawlDCKeys.contains(dcKey))
    {
      // Remove the crawler key. 11/25/2014, Bing Li
      this.crawlDCKeys.remove(dcKey);
    }
  }

  /*
   * Expose the total registered URL count. 11/25/2014, Bing Li
   */
  public synchronized long getTotalURLCount()
  {
    return this.totalURLCount;
  }

  /*
   * Expose the total registered crawler count. 11/25/2014, Bing Li
   */
  public int getCrawlDCCount()
  {
    return this.crawlDCKeys.size();
  }

  /*
   * Expose the total registered crawler keys. 11/25/2014, Bing Li
   */
  public List<String> getCrawlDCKeys()
  {
    return this.crawlDCKeys;
  }

  /*
   * Clear the total registered URL count. 11/25/2014, Bing Li
   */
  public synchronized void clearURLCount()
  {
    this.totalURLCount = 0;
  }

  /*
   * Check whether the crawling load should be redistributed. 11/25/2014, Bing Li
   */
  public synchronized boolean shouldReset()
  {
    return this.shouldReset;
  }
}
```

- **CrawlCoordinator**

```
package com.greatfree.testing.coordinator.crawling;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import com.greatfree.concurrency.Threader;
import com.greatfree.testing.data.URLValue;
import com.greatfree.testing.db.DBConfig;
import com.greatfree.testing.db.URLDB;
import com.greatfree.testing.db.URLDBPool;

/*
 * This class is responsible for distributing the crawling load, URLs, to each online crawlers. In the
version, it is assumed that each crawler has the identical capacity. 11/25/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class CrawlCoordinator
{
   // The threader takes the instance of CrawlLoadDistributer to assign crawling URLs to each crawler.
11/25/2014, Bing Li
   private Threader<CrawlLoadDistributer, CrawlLoadDistributerDisposer> loadDistributer;

   private CrawlCoordinator()
   {
   }

   /*
    * A singleton implementation. 11/25/2014, Bing Li
    */
   private static CrawlCoordinator instance = new CrawlCoordinator();

   public static CrawlCoordinator COORDINATOR()
   {
      if (instance == null)
      {
         instance = new CrawlCoordinator();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Dispose the coordinator. 11/25/2014, Bing Li
    */
   public void dispose() throws InterruptedException
   {
      this.loadDistributer.stop();
   }

   /*
    * Initialize the threader to distribute crawling loads. 11/25/2014, Bing Li
    */
   public void init()
   {
      // Initialize the threader. 11/25/2014, Bing Li
      this.loadDistributer = new Threader<CrawlLoadDistributer, CrawlLoadDistributerDisposer>(new
CrawlLoadDistributer(new CrawlLoadSender()), new CrawlLoadDistributerDisposer());
      // Start the threader. 11/25/2014, Bing Li
```

```
    this.loadDistributer.start();
  }

  /*
   * Get the crawling load count. 11/25/2014, Bing Li
   */
  public long getURLCount()
  {
    // Initialize an instance of URLDB. 11/25/2014, Bing Li
    URLDB db = URLDBPool.PERSISTENT().getDB(DBConfig.URL_DB_PATH);
    // Get the count of all of URLs. 11/25/2014, Bing Li
    long count = db.loadAllURLCount();
    // Collect the instance of URLDB. 11/25/2014, Bing Li
    URLDBPool.PERSISTENT().collectDB(db);
    // Return the count of all of the URLs. 11/25/2014, Bing Li
    return count;
  }

  /*
   * Distribute the crawling load to each crawler. 11/25/2014, Bing Li
   */
  public void distributeCrawlLoads()
  {
    // Define an instance of ArryList that contains all of the crawler keys. 11/25/2014, Bing Li
    List<String> dcKeys = new ArrayList<String>(CrawlRegistry.COORDINATE().getCrawlDCKeys());
    // Check whether the count of crawlers is greater than zero. 11/25/2014, Bing Li
    if (dcKeys.size() > 0)
    {
      // Get an instance of URLDB. 11/25/2014, Bing Li
      URLDB db = URLDBPool.PERSISTENT().getDB(DBConfig.URL_DB_PATH);
      // Load all of the URLs to be crawled. 11/25/2014, Bing Li
      Map<String, URLValue> allURLs = db.loadAllURLs();
      // Collect the instance of URLDB. 11/25/2014, Bing Li
      URLDBPool.PERSISTENT().collectDB(db);
      // Initialize a collection to take a certain number of URLs, which must be assigned to one of the
crawler. 11/26/2014, Bing Li
      HashMap<String, URLValue> urls = new HashMap<String, URLValue>();
      // Check whether the count of crawlers is less than that of the URLs to be crawled. 11/26/2014,
Bing Li
      if (dcKeys.size() <= allURLs.size())
      {
        // Usually, the count of crawlers is much less than that of the URLs to be crawled. If so, estimate
how many URLs each crawler needs to take, the value of task load. 11/26/2014, Bing Li
        int taskLoad = allURLs.size() / dcKeys.size() + 1;
        // Initialize an index that is a number to select a crawler from the list containing crawlers.
11/26/2014, Bing Li
        int index = 0;
        // Scan all of the URLs to assign them to each crawler. 11/26/2014, Bing Li
        for (URLValue url : allURLs.values())
        {
          // Put one URL into a list. 11/26/2014, Bing Li
          urls.put(url.getKey(), url);
          // Check whether the size of the list is greater than the task load each crawler needs to take.
11/26/2014, Bing Li
          if (urls.size() >= taskLoad)
          {
            // Put the crawling load into the load distributer, which is responsible for sending the URLs to
the crawler indicated by the index. 11/26/2014, Bing Li
            this.loadDistributer.getFunction().produce(new CrawlLoad(dcKeys.get(index), urls));
            // Increment the index. 11/26/2014, Bing Li
            index++;
            // Initialize a collection to take the rest collection. 11/26/2014, Bing Li
            urls = new HashMap<String, URLValue>();
          }
        }

        // When the above quits, it is necessary to check whether the collection contains some URLs
and whether some crawlers has not got any URLs to crawl. 11/26/2014, Bing Li
```

```
            if (urls.size() > 0 && index < dcKeys.size())
            {
               // Put the crawling load into the load distributer, which is responsible for sending the URLs to
the crawler indicated by the index. 11/26/2014, Bing Li
               this.loadDistributer.getFunction().produce(new CrawlLoad(dcKeys.get(index), urls));
            }
         }
         else
         {
            // Usually, the count of crawlers is much less than that of the URLs to be crawled. If not, just
send all of the URLs to the first crawler indicated by the index. 11/26/2014, Bing Li
            this.loadDistributer.getFunction().produce(new CrawlLoad(dcKeys.get(0), urls));
         }
         // Set the flag that all of the URLs are assigned. Then, the load distributer must end after all of the
loads are sent to the crawlers respectively. 11/26/2014, Bing Li
         this.loadDistributer.getFunction().setIsFoodQueued(true);
      }
   }
}
```

- **CrawlLoadDistributer**

```
package com.greatfree.testing.coordinator.crawling;

import com.greatfree.concurrency.Consumable;
import com.greatfree.concurrency.ConsumerThread;
import com.greatfree.testing.data.ServerConfig;

/*
 * This is the class that works in the way of producer/consumer to distribute crawling loads, URLs, to all
registered crawlers. 11/25/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class CrawlLoadDistributer extends ConsumerThread<CrawlLoad, CrawlLoadSender>
{
   /*
    * Initialize the distributer, which extends the class, ConsumerThread. 11/25/2014, Bing Li
    */
   public CrawlLoadDistributer(Consumable<CrawlLoad> consumer)
   {
      super(consumer, ServerConfig.DISTRIBUTE_DATA_WAIT_TIME);
   }
}
```

- **CrawlLoadDistributerDisposer**

```
package com.greatfree.testing.coordinator.crawling;

import com.greatfree.reuse.ThreadDisposable;

/*
 * This class aims to dispose the instances of CrawlLoadDistributer. It works with the producer/consumer
pattern. 11/25/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class CrawlLoadDistributerDisposer implements ThreadDisposable<CrawlLoadDistributer>
{
   /*
    * Dispose the instance of CrawlLoadDistributer. 11/25/2014, Bing Li
    */
   @Override
   public void dispose(CrawlLoadDistributer r)
   {
      r.dispose();
   }

   /*
    * The method does not make sense in the version. Just leave it here. 11/25/2014, Bing Li
    */
   @Override
   public void dispose(CrawlLoadDistributer r, long time)
   {
      r.dispose();
   }
}
```

- **CrawlLoad**

```
package com.greatfree.testing.coordinator.crawling;

import java.util.HashMap;

import com.greatfree.testing.data.URLValue;

/*
 * This is a workload to crawl, assigned to a particular crawler represented as the DC key. DC stands for the term, Distributed Component. 11/25/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class CrawlLoad
{
	// The key of the crawler. 11/25/2014, Bing Li
	private String dcKey;
	// The URLs that the crawler to be crawled. 11/25/2014, Bing Li
	private HashMap<String, URLValue> urls;

	/*
	 * Initialize. 11/25/2014, Bing Li
	 */
	public CrawlLoad(String dcKey, HashMap<String, URLValue> urls)
	{
		this.dcKey = dcKey;
		this.urls = urls;
	}

	/*
	 * Expose the crawler key. 11/25/2014, Bing Li
	 */
	public String getDCKey()
	{
		return this.dcKey;
	}

	/*
	 * Expose the URLs to be crawled by the crawler. 11/25/2014, Bing Li
	 */
	public HashMap<String, URLValue> getURLs()
	{
		return this.urls;
	}
}
```

- **CrawlLoadSender**

```
package com.greatfree.testing.coordinator.crawling;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import com.greatfree.concurrency.Consumable;
import com.greatfree.testing.data.Constants;
import com.greatfree.testing.data.URLValue;
import com.greatfree.testing.message.CrawlLoadNotification;
import com.greatfree.util.Tools;

/*
 * This is a procedure to send crawling load to a particular distributed crawler. 11/25/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class CrawlLoadSender implements Consumable<CrawlLoad>
{
  @Override
  public void consume(CrawlLoad load)
  {
    try
    {
      // Initialize a notification to take the load. 11/25/2014, Bing Li
      CrawlLoadNotification notification = new CrawlLoadNotification(load.getDCKey(), true);
      // Get all of the load to be sent to the crawler. 11/25/2014, Bing Li
      HashMap<String, URLValue> urls = load.getURLs();
      // Initialize a collection to take the URLs. 11/25/2014, Bing Li
      Map<String, URLValue> assignedURLs = new HashMap<String, URLValue>();
      // Initialize the batchSize, the approximate size of the notification. It is the amount of the batch load dividing the size of a collection when it takes one URL. With the value, it is possible to estimate how many URLs should be added into a notification. It is the way to measure the size of a notification to send the load each time in an approximately balanced burden to the network. 11/25/2014, Bing Li
      double batchSize = 0;
      // Initialize the count which is to be added into a notification. 11/25/2014, Bing Li
      long loadedCount = 0;
      // Initialize whether it is time to estimate the batchSize. It becomes false when it is time to do that. 11/25/2014, Bing Li
      boolean isReady = false;
      // Add the URL load one by one to notifications. 11/25/2014, Bing Li
      for (URLValue url : urls.values())
      {
        // Check whether it is time to estimate the batchSize. 11/25/2014, Bing Li
        if (!isReady)
        {
          // Add one URL to the notification. 11/25/2014, Bing Li
          assignedURLs.put(url.getKey(), url);
          // Increment the load count. 11/25/2014, Bing Li
          loadedCount++;
          // Estimate the batchSize. 11/25/2014, Bing Li
          batchSize = Constants.MEMORY_BATCH_LOAD / Tools.sizeOf(assignedURLs);
          // Set the flag to true. It denotes that the it needs to add URLs to the load notification with the current batchSize. 11/25/2014, Bing Li
          isReady = true;
          // Since the current URL is added to the notification, it is time to add the next one. 11/25/2014, Bing Li
          continue;
        }

        // Add the URL load to the collection. 11/25/2014, Bing Li
        assignedURLs.put(url.getKey(), url);
        // Increment the load count. 11/25/2014, Bing Li
        loadedCount++;
```

```
            // Check whether the count of URLs loaded in the collection, loadedCount, exceeds the
approximate size of the notification. 11/25/2014, Bing Li
            if (loadedCount >= batchSize)
            {
              // If the load is large enough, put the URLs into the notification. 11/25/2014, Bing Li
              notification.setURLs(assignedURLs);
              // Send the notification to the crawler. 11/25/2014, Bing Li
              CrawlServerClientPool.COORDINATE().getPool().send(load.getDCKey(), notification);
              // Initialize another notification to send additional URLs. 11/25/2014, Bing Li
              notification = new CrawlLoadNotification(load.getDCKey(), false);
              // Initialize a new collection to take the additional URLs. 11/25/2014, Bing Li
              assignedURLs = new HashMap<String, URLValue>();
              // Reset the loadedCount. 11/25/2014, Bing Li
              loadedCount = 0;
              // It time to estimate the batchSize again. 11/25/2014, Bing Li
              isReady = false;
            }
          }
          // When the loop quits, put the rest URLs into the notification. 11/25/2014, Bing Li
          notification.setURLs(assignedURLs);
          // Set the flag that this is the last notification. 11/25/2014, Bing Li
          notification.setDone();
          // Send the last notification. 11/25/2014, Bing Li
          CrawlServerClientPool.COORDINATE().getPool().send(load.getDCKey(), notification);
        }
        catch (IOException e)
        {
          e.printStackTrace();
        }
      }
    }
```

- **DistributeLinksThread**

```
package com.greatfree.testing.coordinator.crawling;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.testing.coordinator.memory.MemoryCoordinator;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.CrawledLinksNotification;

/*
 * The thread gets the crawled links and distribute them to the distributed memory servers. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class DistributeLinksThread extends NotificationQueue<CrawledLinksNotification>
{
  /*
   * Initialize the thread. The argument, taskSize, is used to limit the count of tasks to be queued. 11/28/2014, Bing Li
   */
  public DistributeLinksThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * Once if a crawled links notification is received, it is processed concurrently as follows. 11/28/2014, Bing Li
   */
  public void run()
  {
    // Declare an instance of CrawledLinksNotification. 11/28/2014, Bing Li
    CrawledLinksNotification notification;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/28/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/28/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/28/2014, Bing Li
          notification = this.getNotification();
          // Distribute the crawled links to distributed memory servers. 11/28/2014, Bing Li
          MemoryCoordinator.COORDINATOR().distributeCrawledLink(notification.getLinks());
          // Dispose the notification. 11/28/2014, Bing Li
          this.disposeMessage(notification);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      try
      {
        // Wait for a moment after all of the existing notifications are processed. 11/28/2014, Bing Li
        this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **DistributeLinksThreadCreator**

```
package com.greatfree.testing.coordinator.crawling;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.CrawledLinksNotification;

/*
 * The creator here attempts to create instances of DistributeLinksThread. It works with the notification
dispatcher to schedule the tasks concurrently. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class DistributeLinksThreadCreator implements
NotificationThreadCreatable<CrawledLinksNotification, DistributeLinksThread>
{
  @Override
  public DistributeLinksThread createNotificationThreadInstance(int taskSize)
  {
    return new DistributeLinksThread(taskSize);
  }
}
```

- **RegisterCrawlServerThread**

```
package com.greatfree.testing.coordinator.crawling;

import java.io.IOException;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.testing.coordinator.CoordinatorMulticastor;
import com.greatfree.testing.coordinator.Profile;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.RegisterCrawlServerNotification;

/*
 * The thread is responsible for determining whether to distribute the crawling workload again or to start
the crawling after receiving registration notifications. 11/26/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class RegisterCrawlServerThread extends
NotificationQueue<RegisterCrawlServerNotification>
{
  /*
   * Initialize the thread. 11/26/2014, Bing Li
   */
  public RegisterCrawlServerThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * Process the notification concurrently. 11/26/2014, Bing Li
   */
  public void run()
  {
    // The instance of RegisterCrawlServerNotification. 11/26/2014, Bing Li
    RegisterCrawlServerNotification notification;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/26/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/26/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/26/2014, Bing Li
          notification = this.getNotification();
          // Register the crawler. 11/26/2014, Bing Li
          CrawlRegistry.COORDINATE().register(notification.getDCKey(), notification.getURLCount());
          // Check whether the count of registered crawlers is equal to that of predefined ones.
11/26/2014, Bing Li
          if (CrawlRegistry.COORDINATE().getCrawlDCCount() ==
Profile.CONFIG().getCrawlServerCount())
          {
            // If all of the crawlers are registered, it needs to check whether the total URLs those
crawlers take is less than that of the ones to be crawled. In addition, it also needs to check if one crawler
has no any URLs. 11/26/2014, Bing Li
            if (CrawlRegistry.COORDINATE().getTotalURLCount() <
CrawlCoordinator.COORDINATOR().getURLCount() || CrawlRegistry.COORDINATE().shouldReset())
            {
              // Clear the count of registered URLs. 11/26/2014, Bing Li
              CrawlRegistry.COORDINATE().clearURLCount();
              // Distribute the URLs, crawling loads, to each registered clusters. 11/26/2014, Bing Li
              CrawlCoordinator.COORDINATOR().distributeCrawlLoads();
            }
            else
            {
```

```
                try
                {
                  // Disseminate the notification to demand all of the registered crawlers to start crawling.
11/26/2014, Bing Li
                  CoordinatorMulticastor.COORDINATE().disseminateStartCrawl();
                }
                catch (InstantiationException e)
                {
                  e.printStackTrace();
                }
                catch (IllegalAccessException e)
                {
                  e.printStackTrace();
                }
                catch (IOException e)
                {
                  e.printStackTrace();
                }
              }
            }
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
        }

        try
        {
          // Wait for a moment after all of the existing notifications are processed. 11/28/2014, Bing Li
          this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
    }
  }
}
```

- **RegisterCrawlServerThreadCreator**

```
package com.greatfree.testing.coordinator.crawling;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.RegisterCrawlServerNotification;

/*
 * The creator here attempts to create instances of RegisterCrawlServerThread. It works with the
notification dispatcher to schedule the tasks concurrently. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class RegisterCrawlServerThreadCreator implements
NotificationThreadCreatable<RegisterCrawlServerNotification, RegisterCrawlServerThread>
{
  @Override
  public RegisterCrawlServerThread createNotificationThreadInstance(int taskSize)
  {
    return new RegisterCrawlServerThread(taskSize);
  }
}
```

- **UnregisterCrawlServerThread**

```
package com.greatfree.testing.coordinator.crawling;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.UnregisterCrawlServerNotification;

/*
 * The thread processes the unregister notification from a crawling server concurrently. 11/27/2014, Bing
Li
 */

// Created: 11/27/2014, Bing Li
public class UnregisterCrawlServerThread extends
NotificationQueue<UnregisterCrawlServerNotification>
{
  /*
   * Initialize the thread. 11/27/2014, Bing Li
   */
  public UnregisterCrawlServerThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * Process the notification concurrently. 11/27/2014, Bing Li
   */
  public void run()
  {
    // Declare an instance of UnregisterCrawlServerNotification. 11/28/2014, Bing Li
    UnregisterCrawlServerNotification notification;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/28/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/28/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/28/2014, Bing Li
          notification = this.getNotification();
          // Unregister the crawler from the registry. 11/28/2014, Bing Li
          CrawlRegistry.COORDINATE().unregister(notification.getDCKey());
          // Dispose the notification. 11/28/2014, Bing Li
          this.disposeMessage(notification);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      try
      {
        // Wait for a moment after all of the existing notifications are processed. 11/28/2014, Bing Li
        this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **UnregisterCrawlServerThreadCreator**

```
package com.greatfree.testing.coordinator.crawling;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.UnregisterCrawlServerNotification;

/*
 * The creator here attempts to create instances of UnregisterCrawlServerThread. It works with the
notification dispatcher to schedule the tasks concurrently. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class UnregisterCrawlServerThreadCreator implements
NotificationThreadCreatable<UnregisterCrawlServerNotification, UnregisterCrawlServerThread>
{
  @Override
  public UnregisterCrawlServerThread createNotificationThreadInstance(int taskSize)
  {
    return new UnregisterCrawlServerThread(taskSize);
  }
}
```

- **StartCrawlMulticastor**

```
package com.greatfree.testing.coordinator.crawling;

import com.greatfree.multicast.RootObjectMulticastor;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.message.StartCrawlMultiNotification;
import com.greatfree.util.NullObject;

/*
 * This is an extending of RootObjectMulticastor to transfer the notification of StartCrawlMultiNotification
to all of the crawlers. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public class StartCrawlMulticastor extends RootObjectMulticastor<NullObject,
StartCrawlMultiNotification, StartCrawlNotificationCreator>
{
   /*
    * Initialize the multicastor. 11/26/2014, Bing Li
    */
   public StartCrawlMulticastor(FreeClientPool clientPool, int rootBranchCount, int treeBranchCount,
StartCrawlNotificationCreator messageCreator)
   {
      super(clientPool, rootBranchCount, treeBranchCount, messageCreator);
   }
}
```

- **StartCrawlMulticastorCreator**

```
package com.greatfree.testing.coordinator.crawling;

import com.greatfree.reuse.HashCreatable;

/*
 * The class intends to create the interface of StartCrawlMulticastor. It is used by the resource pool to
manage the multicastors efficiently. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public class StartCrawlMulticastorCreator implements HashCreatable<StartCrawlMulticastorSource,
StartCrawlMulticastor>
{
   /*
    * Define the method to create the instances of StartCrawlMulticastor upon the source,
StartCrawlMulticastorSource. 11/26/2014, Bing Li
    */
   @Override
   public StartCrawlMulticastor createResourceInstance(StartCrawlMulticastorSource source)
   {
      return new StartCrawlMulticastor(source.getClientPool(), source.getRootBranchCount(),
source.getTreeBranchCount(), source.getCreator());
   }
}
```

- **StartCrawlMulticastorDisposer**

```
package com.greatfree.testing.coordinator.crawling;

import com.greatfree.reuse.HashDisposable;

/*
 * The disposer collects the instance of StartCrawlMulticastor. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public class StartCrawlMulticastorDisposer implements HashDisposable<StartCrawlMulticastor>
{
   @Override
   public void dispose(StartCrawlMulticastor t)
   {
      t.dispose();
   }
}
```

- **StartCrawlMulticastorSource**

```
package com.greatfree.testing.coordinator.crawling;

import com.greatfree.multicast.RootMessageCreatorGettable;
import com.greatfree.multicast.RootMulticastorSource;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.message.StartCrawlMultiNotification;
import com.greatfree.util.NullObject;

/*
 * The class provides the pool with the initial values to create a StartCrawlMulticastor. The sources that are needed to create an instance of RootMulticastor are enclosed in the class. That is required by the pool to create multicastors. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public class StartCrawlMulticastorSource extends RootMulticastorSource<NullObject, StartCrawlMultiNotification, StartCrawlNotificationCreator> implements RootMessageCreatorGettable<StartCrawlMultiNotification, NullObject>
{
  /*
   * Initialize the source. 11/26/2014, Bing Li
   */
  public StartCrawlMulticastorSource(FreeClientPool clientPool, int rootBranchCount, int treeBranchCount, StartCrawlNotificationCreator creator)
  {
    super(clientPool, rootBranchCount, treeBranchCount, creator);
  }

  /*
   * Expose the message creator. 11/26/2014, Bing Li
   */
  @Override
  public StartCrawlNotificationCreator getMessageCreator()
  {
    return super.getCreator();
  }
}
```

- **StartCrawlNotificationCreator**

```
package com.greatfree.testing.coordinator.crawling;

import java.util.HashMap;

import com.greatfree.multicast.ObjectMulticastCreatable;
import com.greatfree.testing.message.StartCrawlMultiNotification;
import com.greatfree.util.NullObject;
import com.greatfree.util.Tools;

/*
 * The creator is used to create the instance of StartCrawlMultiNotification. It works with a multicastor to do that. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public class StartCrawlNotificationCreator implements ObjectMulticastCreatable<StartCrawlMultiNotification, NullObject>
{
  /*
   * Create an instance of StartCrawlMultiNotification. 11/26/2014, Bing Li
   *
   * For the specific notification, StartCrawlMultiNotification, no arguments are needed to send. Therefore, the NullObject is put here. For other notification, an object that contains all of the arguments must be enclosed in the object.
   *
   * The constructor needs to input the children nodes for further multicast.
   */
  @Override
  public StartCrawlMultiNotification createInstanceWithChildren(NullObject message, HashMap<String, String> childrenMap)
  {
    return new StartCrawlMultiNotification(Tools.generateUniqueKey(), childrenMap);
  }

  /*
   * Create an instance of StartCrawlMultiNotification. 11/26/2014, Bing Li
   *
   * For the specific notification, StartCrawlMultiNotification, no arguments are needed to send. Therefore, the NullObject is put here. For other notification, an object that contains all of the arguments must be enclosed in the object.
   */
  @Override
  public StartCrawlMultiNotification createInstanceWithoutChildren(NullObject message)
  {
    return new StartCrawlMultiNotification(Tools.generateUniqueKey());
  }
}
```

3.6.1.3 The Memorizing

- **MemoryListener**

```
package com.greatfree.testing.coordinator.memorizing;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

import com.greatfree.remote.IPPort;
import com.greatfree.remote.ServerListener;
import com.greatfree.testing.data.ServerConfig;

/*
 * This is a coordinator listener that not only responds to memory servers but also intends to connect and communicate with them. It assists the coordinator to interact with memory servers in an eventing manner other than waiting for requests and then responding only. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class MemoryListener extends ServerListener implements Runnable
{
  // A thread to connect the remote memory server concurrently. 11/28/2014, Bing Li
  private ConnectMemServerThread connectThread;

  /*
   * Initialize the listener. 11/28/2014, Bing Li
   */
  public MemoryListener(ServerSocket serverSocket, int threadPoolSize, long keepAliveTime)
  {
    super(serverSocket, threadPoolSize, keepAliveTime);
  }

  /*
   * Shutdown the listener. 11/28/2014, Bing Li
   */
  public void shutdown()
  {
    // Dispose the connecting thread. 11/28/2014, Bing Li
    this.connectThread.dispose();
    // Shutdown the listener. 11/28/2014, Bing Li
    super.shutdown();
  }

  /*
   * The task that must be executed concurrently. 11/28/2014, Bing Li
   */
  @Override
  public void run()
  {
    Socket clientSocket;
    MemoryIO serverIO;

    // Detect whether the listener is shutdown. If not, it must be running all the time to wait for potential connections from crawlers. 11/28/2014, Bing Li
    while (!super.isShutdown())
    {
      try
      {
        // Wait and accept a connecting from a possible memory server. 11/28/2014, Bing Li
        clientSocket = super.accept();
        // Check whether the connected server IOs exceed the upper limit. 11/28/2014, Bing Li
        if (MemoryIORegistry.REGISTRY().getIOCount() >= ServerConfig.MAX_SERVER_IO_COUNT)
        {
          try
          {
            // If the upper limit is reached, the listener has to wait until an existing server IO is disposed.
```

```
11/28/2014, Bing Li
                super.holdOn();
              }
              catch (InterruptedException e)
              {
                e.printStackTrace();
              }
            }

            // If the upper limit of IOs is not reached, a memory server IO is initialized. A common
Collaborator and the socket are the initial parameters. The shared common collaborator guarantees all
of the memory server IOs from a certain memory server could notify with each other with the same lock.
Then, the upper limit of memory server IOs is under the control. 11/28/2014, Bing Li
            serverIO = new MemoryIO(clientSocket, super.getCollaborator());

            /*
             * Since the listener servers need to form a peer-to-peer architecture, it is required to connect to
the remote memory server. Thus, the local server can send messages without waiting for any requests
from the remote end. 11/28/2014, Bing Li
             */

            // Check whether a client to the IP and the port number of the remote memory server is existed
in the client pool. If such a client does not exist, it is required to connect the remote memory server by
the thread concurrently. Doing that concurrently is to speed up the rate of responding to the memory
server. 11/28/2014, Bing Li
            if (!MemoryServerClientPool.COORDINATE().getPool().isClientExisted(serverIO.getIP(),
ServerConfig.MEMORY_SERVER_PORT))
            {
              // Since the client does not exist in the pool, input the IP address and the port number to the
thread. 11/28/2014, Bing Li
              this.connectThread.enqueue(new IPPort(serverIO.getIP(),
ServerConfig.MEMORY_SERVER_PORT));
              // Execute the thread to connect to the remote memory server. 11/28/2014, Bing Li
              super.execute(this.connectThread);
            }

            // Add the new created server IO into the registry for further management. 11/28/2014, Bing Li
            MemoryIORegistry.REGISTRY().addIO(serverIO);
            // Execute the new created memory IO concurrently to respond the memory servers' requests
and notifications in an asynchronous manner. 11/28/2014, Bing Li
            super.execute(serverIO);
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
        }
      }
    }
```

- **MemoryListenerDisposer**

```
package com.greatfree.testing.coordinator.memorizing;

import com.greatfree.reuse.RunDisposable;

/*
 * The class is responsible for disposing the instance of MemoryListener by invoking its method of shutdown(). 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class MemoryListenerDisposer implements RunDisposable<MemoryListener>
{
	/*
	 * Dispose the instance of MemoryListener. 11/28/2014, Bing Li
	 */
	@Override
	public void dispose(MemoryListener r)
	{
		r.shutdown();
	}

	/*
	 * Dispose the instance of MemoryListener. The method does not make sense to MemoryListener. Just leave it here for the requirement of the interface, RunDisposable<MemoryListener>. 11/28/2014, Bing Li
	 */
	@Override
	public void dispose(MemoryListener r, long time)
	{
		r.shutdown();
	}
}
```

- **MemoryIO**

```
package com.greatfree.testing.coordinator.memorizing;

import java.io.IOException;
import java.net.Socket;
import java.net.SocketException;

import com.greatfree.concurrency.Collaborator;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.remote.ServerIO;
import com.greatfree.testing.coordinator.CoordinatorMessageProducer;

/*
 * The class is actually an implementation of ServerIO, which serves for the memory nodes which access the coordinator. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class MemoryIO extends ServerIO
{
  /*
   * Initialize the server IO. The socket is the connection between the memory server and the coordinator. The collaborator is shared with other IOs to control the count of ServerIOs instances. 11/28/2014, Bing Li
   */
  public MemoryIO(Socket clientSocket, Collaborator collaborator)
  {
    super(clientSocket, collaborator);
  }

  /*
   * A concurrent method to respond the received messages asynchronously. 11/28/2014, Bing Li
   */
  public void run()
  {
    ServerMessage message;
    while (!super.isShutdown())
    {
      try
      {
        // Wait and read messages from a memory server. 11/28/2014, Bing Li
        message = (ServerMessage)super.read();
        // Convert the received message to OutMessageStream and put it into the relevant dispatcher for concurrent processing. 11/28/2014, Bing Li
        CoordinatorMessageProducer.SERVER().produceMemoryMessage(new OutMessageStream<ServerMessage>(super.getOutStream(), super.getLock(), message));
      }
      catch (SocketException e)
      {
        e.printStackTrace();
      }
      catch (IOException e)
      {
        e.printStackTrace();
      }
      catch (ClassNotFoundException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **MemoryIORegistry**

```
package com.greatfree.testing.coordinator.memorizing;

import java.io.IOException;
import java.util.Set;

import com.greatfree.remote.ServerIORegistry;

/*
 * The class keeps all of MemoryIOs. This is a singleton wrapper of ServerIORegistry. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class MemoryIORegistry
{
  // Declare an instance of ServerIORegistry for MemoryIOs. 11/28/2014, Bing Li
  private ServerIORegistry<MemoryIO> registry;

  /*
   * Initializing ... 11/28/2014, Bing Li
   */
  private MemoryIORegistry()
  {
  }

  private static MemoryIORegistry instance = new MemoryIORegistry();

  public static MemoryIORegistry REGISTRY()
  {
    if (instance == null)
    {
      instance = new MemoryIORegistry();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the registry. 11/28/2014, Bing Li
   */
  public void dispose() throws IOException
  {
    this.registry.removeAllIOs();
  }

  /*
   * Initialize the registry. 11/28/2014, Bing Li
   */
  public void init()
  {
    this.registry = new ServerIORegistry<MemoryIO>();
  }

  /*
   * Add a new instance of MemoryIO to the registry. 11/28/2014, Bing Li
   */
  public void addIO(MemoryIO io)
  {
    this.registry.addIO(io);
  }
```

```
    /*
     * Get all of the IPs of the connected clients from the corresponding MemoryIOs. 11/28/2014, Bing Li
     */
    public Set<String> getIPs()
    {
      return this.registry.getIPs();
    }

    /*
     * Get the count of the registered MemoryIOs. 11/28/2014, Bing Li
     */
    public int getIOCount()
    {
      return this.registry.getIOCount();
    }

    /*
     * Remove or unregister an MemoryIO. It is executed when a client is down or the connection gets
something wrong. 11/28/2014, Bing Li
     */
    public void removeIO(MemoryIO io) throws IOException
    {
      this.registry.removeIO(io);
    }

    /*
     * Remove or unregister all of the registered MemoryIOs. It is executed when the server process is
shutdown. 11/28/2014, Bing Li
     */
    public void removeAllIOs() throws IOException
    {
      this.registry.removeAllIOs();
    }
}
```

- **MemoryRegistry**

```
package com.greatfree.testing.coordinator.memorizing;

import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

/*
 * Since a bunch of nodes are composed together to form a distributed memory system, a centralized registry is required to manage all of them. This is what the registry should do. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class MemoryRegistry
{
  // A list that contains all of the keys of the memory servers. 11/28/2014, Bing Li
  private List<String> memDCKeys;

  private MemoryRegistry()
  {
    this.memDCKeys = new CopyOnWriteArrayList<String>();
  }

  /*
   * A singleton definition. 11/28/2014, Bing Li
   */
  private static MemoryRegistry instance = new MemoryRegistry();

  public static MemoryRegistry COORDINATE()
  {
    if (instance == null)
    {
      instance = new MemoryRegistry();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the registry. 11/28/2014, Bing Li
   */
  public void dispose()
  {
    this.memDCKeys.clear();
  }

  /*
   * Register a new online memory server and its URL load. 11/28/2014, Bing Li
   */
  public synchronized void register(String dcKey)
  {
    // Check whether the list contains the key of the memory server. 11/28/2014, Bing Li
    if (!this.memDCKeys.contains(dcKey))
    {
      // Put the memory server key into the list. 11/28/2014, Bing Li
      this.memDCKeys.add(dcKey);
    }
  }

  /*
   * Unregister a memory server. 11/28/2014, Bing Li
   */
  public synchronized void unregister(String dcKey)
```

```
    {
        // Check whether the list contains the memory server key. 11/28/2014, Bing Li
        if (this.memDCKeys.contains(dcKey))
        {
            // Remove the memory server key. 11/28/2014, Bing Li
            this.memDCKeys.remove(dcKey);
        }
    }

    /*
     * Expose the total registered memory server count. 11/28/2014, Bing Li
     */
    public int getCrawlDCCount()
    {
        return this.memDCKeys.size();
    }

    /*
     * Expose the total registered memory server keys. 11/28/2014, Bing Li
     */
    public List<String> getCrawlDCKeys()
    {
        return this.memDCKeys;
    }
}
```

- **ConnectMemServerThread**

```
package com.greatfree.testing.coordinator.memorizing;

import java.io.IOException;
import java.util.Queue;
import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.remote.IPPort;
import com.greatfree.testing.message.NodeKeyNotification;

/*
 * The class is derived from Thread. It is responsible for connecting the remote memory server such that it is feasible for the server to notify the client even without the memory server's request. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class ConnectMemServerThread extends Thread
{
  // The queue keeps the clients' IPs which need to be connected to. 11/28/2014, Bing Li
  private Queue<IPPort> ipQueue;

  /*
   * Initialize. 11/28/2014, Bing Li
   */
  public ConnectMemServerThread()
  {
    this.ipQueue = new LinkedBlockingQueue<IPPort>();
  }

  /*
   * Dispose the resource of the class. 11/28/2014, Bing Li
   */
  public synchronized void dispose()
  {
    if (this.ipQueue != null)
    {
      this.ipQueue.clear();
      this.ipQueue = null;
    }
  }

  /*
   * Input the IP address and the port number into the queue. They are connected in the order of first-in-first-out. The CrawlServerClientPool is responsible for the connections. 11/28/2014, Bing Li
   */
  public void enqueue(IPPort ipPort)
  {
    this.ipQueue.add(ipPort);
  }

  /*
   * Connect the remote IP addresses and the associated port numbers concurrently. 11/28/2014, Bing Li
   */
  @Override
  public void run()
  {
    IPPort ipPort;
    // The thread keeps working until all of the IP addresses are connected. 11/28/2014, Bing Li
    while (this.ipQueue.size() > 0)
    {
      // Dequeue the IP addresses. 11/28/2014, Bing Li
      ipPort = this.ipQueue.poll();
      try
```

```
        {
            // Notify the remote node its unique ID. This is usually used to set up a multicasting cluster which
contains a bunch of nodes. Each of them is identified by the ID. 11/28/2014, Bing Li
            MemoryServerClientPool.COORDINATE().getPool().send(ipPort, new
NodeKeyNotification(ipPort.getObjectKey()));
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
  }
}
```

- **MemoryCoordinator**

```
package com.greatfree.testing.coordinator.memorizing;

import java.util.Set;

import com.greatfree.testing.data.CrawledLink;
import com.greatfree.util.Tools;

/*
 * The coordinator is responsible to send the crawled links to one of the distributed memory servers in a
hash-based load-balancing approach. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class MemoryCoordinator
{
  private MemoryCoordinator()
  {
  }

  /*
   * A singleton implementation. 11/25/2014, Bing Li
   */
  private static MemoryCoordinator instance = new MemoryCoordinator();

  public static MemoryCoordinator COORDINATOR()
  {
    if (instance == null)
    {
      instance = new MemoryCoordinator();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the coordinator. 11/25/2014, Bing Li
   */
  public void dispose() throws InterruptedException
  {
  }

  /*
   * Send the crawled links to the distributed memory servers. 11/28/2014, Bing Li
   */
  public void distributeCrawledLink(Set<CrawledLink> links)
  {
    for (CrawledLink link : links)
    {
      // To achieve the goal of load-balance, a hash-based algorithm is used. 11/28/2014, Bing Li
      MemoryEventer.NOTIFY().addCrawledLink(Tools.getClosestKey(link.getKey(),
MemoryRegistry.COORDINATE().getCrawlDCKeys()), link);
    }
  }
}
```

- **MemoryEventer**

```
package com.greatfree.testing.coordinator.memorizing;

import com.greatfree.concurrency.ThreadPool;
import com.greatfree.remote.AsyncRemoteEventer;
import com.greatfree.testing.data.ClientConfig;
import com.greatfree.testing.data.CrawledLink;
import com.greatfree.testing.message.AddCrawledLinkNotification;

/*
 * This is an eventer that sends notifications to one of the distributed memory node in a synchronous or
asynchronous manner. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class MemoryEventer
{
   // The thread pool that starts up the asynchronous eventer. 11/28/2014, Bing Li
   private ThreadPool pool;
   // The asynchronous eventer sends crawled links to one of the distributed memory servers.
11/28/2014, Bing Li
   private AsyncRemoteEventer<AddCrawledLinkNotification> addCrawledLinkEventer;

   private MemoryEventer()
   {
   }

   /*
    * Initialize a singleton. 11/28/2014, Bing Li
    */
   private static MemoryEventer instance = new MemoryEventer();

   public static MemoryEventer NOTIFY()
   {
      if (instance == null)
      {
         instance = new MemoryEventer();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Dispose the eventer. 11/28/2014, Bing Li
    */
   public void dispose()
   {
      this.addCrawledLinkEventer.dispose();
      this.pool.shutdown();
   }

   /*
    * Initialize. 11/28/2014, Bing Li
    */
   public void init()
   {
      this.pool = new ThreadPool(ClientConfig.EVENTER_THREAD_POOL_SIZE,
ClientConfig.EVENTER_THREAD_POOL_ALIVE_TIME);

      // Initialize the adding crawled link eventer. 11/28/2014, Bing Li
      this.addCrawledLinkEventer = new
AsyncRemoteEventer<AddCrawledLinkNotification>(MemoryServerClientPool.COORDINATE().getPool
```

```
(), this.pool, ClientConfig.EVENT_QUEUE_SIZE, ClientConfig.EVENTER_SIZE,
ClientConfig.EVENTING_WAIT_TIME, ClientConfig.EVENTER_WAIT_TIME);
      // Set the idle checking for the adding crawled link eventer. 11/28/2014, Bing Li
      this.addCrawledLinkEventer.setIdleChecker(ClientConfig.EVENT_IDLE_CHECK_DELAY,
ClientConfig.EVENT_IDLE_CHECK_PERIOD);
      // Start up the adding crawled link eventer. 11/28/2014, Bing Li
      this.pool.execute(this.addCrawledLinkEventer);
    }

    /*
     * Add the crawled links to the remote memory server identified by the key, dcKey. 11/28/2014, Bing Li
     */
    public void addCrawledLink(String dcKey, CrawledLink link)
    {
      this.addCrawledLinkEventer.notify(dcKey, new AddCrawledLinkNotification(dcKey, link));
    }
}
```

- **MemoryServerClientPool**

```
package com.greatfree.testing.coordinator.memorizing;

import java.io.IOException;

import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.coordinator.CoorConfig;

/*
 * This is a pool that creates and manages instances of FreeClient to achieve the goal of using minimum instances of clients to reach a high performance. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class MemoryServerClientPool
{
  // An instance of FreeClientPool is defined to interact with the crawler. 11/28/2014, Bing Li
  private FreeClientPool clientPool;

  private MemoryServerClientPool()
  {
  }

  /*
   * A singleton definition. 11/28/2014, Bing Li
   */
  private static MemoryServerClientPool instance = new MemoryServerClientPool();

  public static MemoryServerClientPool COORDINATE()
  {
    if (instance == null)
    {
      instance = new MemoryServerClientPool();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the free client pool. 11/28/2014, Bing Li
   */
  public void dispose()
  {
    try
    {
      this.clientPool.dispose();
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
  }

  /*
   * Initialize the client pool. The method is called when the coordinator process is started. 11/28/2014, Bing Li
   */
  public void init()
  {
    // Initialize the client pool. 11/28/2014, Bing Li
    this.clientPool = new FreeClientPool(CoorConfig.CSERVER_CLIENT_POOL_SIZE);
    // Set idle checking for the client pool. 11/28/2014, Bing Li
```

```
    this.clientPool.setIdleChecker(CoorConfig.CSERVER_CLIENT_IDLE_CHECK_DELAY,
CoorConfig.CSERVER_CLIENT_IDLE_CHECK_PERIOD,
CoorConfig.CSERVER_CLIENT_MAX_IDLE_TIME);
  }

  /*
   * Expose the client pool. 11/28/2014, Bing Li
   */
  public FreeClientPool getPool()
  {
    return this.clientPool;
  }
}
```

- **MemoryServerDispatcher**

```
package com.greatfree.testing.coordinator.memorizing;

import com.greatfree.concurrency.NotificationDispatcher;
import com.greatfree.concurrency.ServerMessageDispatcher;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.IsPublisherExistedAnycastResponse;
import com.greatfree.testing.message.MessageType;
import com.greatfree.testing.message.RegisterMemoryServerNotification;
import com.greatfree.testing.message.SearchKeywordBroadcastResponse;
import com.greatfree.testing.message.UnregisterMemoryServerNotification;

/*
 * This is an implementation of ServerMessageDispatcher. It contains the concurrency mechanism to
respond memory servers' requests and receive their notifications for the coordinator. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class MemoryServerDispatcher extends ServerMessageDispatcher<ServerMessage>
{
  // Declare a notification dispatcher to process the memory server registration concurrently.
11/28/2014, Bing Li
  private NotificationDispatcher<RegisterMemoryServerNotification, RegisterMemoryServerThread,
RegisterMemoryServerThreadCreator> registerMemoryServerNotificationDispatcher;
  // Declare a notification dispatcher to process the memory server unregistering concurrently.
11/28/2014, Bing Li
  private NotificationDispatcher<UnregisterMemoryServerNotification,
UnregisterMemoryServerThread, UnregisterMemoryServerThreadCreator>
unregisterMemoryServerNotificationDispatcher;

  private NotificationDispatcher<IsPublisherExistedAnycastResponse,
NotifyIsPublisherExistedThread, NotifyIsPublisherExistedThreadCreator>
notifyIsPublishedReceivedDispatcher;
  private NotificationDispatcher<SearchKeywordBroadcastResponse, NotifySearchKeywordThread,
NotifySearchKeywordThreadCreator> notifySearchKeywordReceivedDispatcher;

  /*
   * Initialize. 11/28/2014, Bing Li
   */
  public MemoryServerDispatcher(int corePoolSize, long keepAliveTime)
  {
    super(corePoolSize, keepAliveTime);

    // Initialize the memory server registration dispatcher. 11/28/2014, Bing Li
    this.registerMemoryServerNotificationDispatcher = new
NotificationDispatcher<RegisterMemoryServerNotification, RegisterMemoryServerThread,
RegisterMemoryServerThreadCreator>(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new
RegisterMemoryServerThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
    // Set the parameters to check idle states of threads. 11/28/2014, Bing Li

  this.registerMemoryServerNotificationDispatcher.setIdleChecker(ServerConfig.NOTIFICATION_DISP
ATCHER_IDLE_CHECK_DELAY,
ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
    // Start the dispatcher. 11/28/2014, Bing Li
    super.execute(this.registerMemoryServerNotificationDispatcher);

    // Initialize the memory server unregistering dispatcher. 11/27/2014, Bing Li
    this.unregisterMemoryServerNotificationDispatcher = new
NotificationDispatcher<UnregisterMemoryServerNotification, UnregisterMemoryServerThread,
UnregisterMemoryServerThreadCreator>(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
```

```
ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new
UnregisterMemoryServerThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
      // Set the parameters to check idle states of threads. 11/27/2014, Bing Li

    this.unregisterMemoryServerNotificationDispatcher.setIdleChecker(ServerConfig.NOTIFICATION_DI
SPATCHER_IDLE_CHECK_DELAY,
ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
      // Start the dispatcher. 11/27/2014, Bing Li
      super.execute(this.unregisterMemoryServerNotificationDispatcher);

      // Initialize the anycast notifying dispatcher. 11/27/2014, Bing Li
      this.notifyIsPublishedReceivedDispatcher = new
NotificationDispatcher<IsPublisherExistedAnycastResponse, NotifyIsPublisherExistedThread,
NotifyIsPublisherExistedThreadCreator>(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new
NotifyIsPublisherExistedThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
      // Set the parameters to check idle states of threads. 11/27/2014, Bing Li

    this.notifyIsPublishedReceivedDispatcher.setIdleChecker(ServerConfig.NOTIFICATION_DISPATCH
ER_IDLE_CHECK_DELAY, ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
      // Start the dispatcher. 11/27/2014, Bing Li
      super.execute(this.notifyIsPublishedReceivedDispatcher);

      // Initialize the broadcast notifying dispatcher. 11/27/2014, Bing Li
      this.notifySearchKeywordReceivedDispatcher = new
NotificationDispatcher<SearchKeywordBroadcastResponse, NotifySearchKeywordThread,
NotifySearchKeywordThreadCreator>(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new
NotifySearchKeywordThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
      // Set the parameters to check idle states of threads. 11/27/2014, Bing Li

    this.notifySearchKeywordReceivedDispatcher.setIdleChecker(ServerConfig.NOTIFICATION_DISPAT
CHER_IDLE_CHECK_DELAY, ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
      // Start the dispatcher. 11/27/2014, Bing Li
      super.execute(this.notifySearchKeywordReceivedDispatcher);
    }

    /*
     * Shut down the storing message dispatcher. 11/28/2014, Bing Li
     */
    public void shutdown()
    {
      // Dispose the memory server registration dispatcher. 11/28/2014, Bing Li
      this.registerMemoryServerNotificationDispatcher.dispose();
      // Dispose the memory server unregistering dispatcher. 11/28/2014, Bing Li
      this.unregisterMemoryServerNotificationDispatcher.dispose();
      // Dispose the anycast notifying dispatcher. 11/29/2014, Bing Li
      this.notifyIsPublishedReceivedDispatcher.dispose();
      // Dispose the broadcast notifying dispatcher. 11/29/2014, Bing Li
      this.notifySearchKeywordReceivedDispatcher.dispose();
      // Shutdown the server message dispatcher. 11/28/2014, Bing Li
      super.shutdown();
    }

    /*
     * Process the available messages in a concurrent way. 11/28/2014, Bing Li
     */
    public void consume(OutMessageStream<ServerMessage> message)
    {
      // Check the types of received messages. 11/28/2014, Bing Li
      switch (message.getMessage().getType())
      {
```

```
        // If the message is the notification to register the memory server. 11/28/2014, Bing Li
        case MessageType.REGISTER_MEMORY_SERVER_NOTIFICATION:
          // Enqueue the notification into the dispatcher for concurrent feedback. 11/28/2014, Bing Li

  this.registerMemoryServerNotificationDispatcher.enqueue((RegisterMemoryServerNotification)messa
ge.getMessage());
          break;

        // If the message is the notification to unregister the memory server. 11/28/2014, Bing Li
        case MessageType.UNREGISTER_MEMORY_SERVER_NOTIFICATION:
          // Enqueue the notification into the dispatcher for concurrent feedback. 11/28/2014, Bing Li

  this.unregisterMemoryServerNotificationDispatcher.enqueue((UnregisterMemoryServerNotification)m
essage.getMessage());
          break;

        // If the message is an anycast response, IsPublisherExistedAnycastResponse. 11/29/2014, Bing
Li
        case MessageType.IS_PUBLISHER_EXISTED_ANYCAST_RESPONSE:
          // Notify the received response. 11/29/2014, Bing Li

  this.notifyIsPublishedReceivedDispatcher.enqueue((IsPublisherExistedAnycastResponse)message.g
etMessage());
          break;

        // If the message is an broadcast response, SearchKeywordBroadcastResponse. 11/29/2014, Bing
Li
        case MessageType.SEARCH_KEYWORD_BROADCAST_RESPONSE:
          // Notify the received response. 11/29/2014, Bing Li

  this.notifySearchKeywordReceivedDispatcher.enqueue((SearchKeywordBroadcastResponse)messa
ge.getMessage());
          break;
      }
    }
  }
```

- **MemoryServerProducerDisposer**

```
package com.greatfree.testing.coordinator.memorizing;

import com.greatfree.concurrency.MessageProducer;
import com.greatfree.reuse.ThreadDisposable;

/*
 * The class is responsible for disposing the memory server message producer of the coordinator.
11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class MemoryServerProducerDisposer implements
ThreadDisposable<MessageProducer<MemoryServerDispatcher>>
{
   /*
    * Dispose the message producer. 11/28/2014, Bing Li
    */
   @Override
   public void dispose(MessageProducer<MemoryServerDispatcher> r)
   {
      r.dispose();
   }

   /*
    * The method does not make sense to the class of MessageProducer. Just leave it here. 11/28/2014,
Bing Li
    */
   @Override
   public void dispose(MessageProducer<MemoryServerDispatcher> r, long time)
   {
      r.dispose();
   }
}
```

- **RegisterMemoryServerThread**

```
package com.greatfree.testing.coordinator.memorizing;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.RegisterMemoryServerNotification;

/*
 * The thread is responsible for registering the distributed memory nodes for data storing. 11/28/2014,
Bing Li
 */

// Created: 11/28/2014, Bing Li
public class RegisterMemoryServerThread extends
NotificationQueue<RegisterMemoryServerNotification>
{
  /*
   * Initialize the thread. 11/28/2014, Bing Li
   */
  public RegisterMemoryServerThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * Process the notification concurrently. 11/28/2014, Bing Li
   */
  public void run()
  {
    // The instance of RegisterMemoryServerNotification. 11/28/2014, Bing Li
    RegisterMemoryServerNotification notification;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/28/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/28/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/29/2014, Bing Li
          notification = this.getNotification();
          // Register the memory server. 11/29/2014, Bing Li
          MemoryRegistry.COORDINATE().register(notification.getDCKey());
          // Dispose the notification. 11/29/2014, Bing Li
          this.disposeMessage(notification);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      try
      {
        // Wait for a moment after all of the existing requests are processed. 11/29/2014, Bing Li
        this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **RegisterMemoryServerThreadCreator**

```
package com.greatfree.testing.coordinator.memorizing;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.RegisterMemoryServerNotification;

/*
 * The creator here attempts to create instances of RegisterMemoryServerThread. It works with the
notification dispatcher to schedule the tasks concurrently. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class RegisterMemoryServerThreadCreator implements
NotificationThreadCreatable<RegisterMemoryServerNotification, RegisterMemoryServerThread>
{
  @Override
  public RegisterMemoryServerThread createNotificationThreadInstance(int taskSize)
  {
    return new RegisterMemoryServerThread(taskSize);
  }
}
```

- **UnregisterMemoryServerThread**

```
package com.greatfree.testing.coordinator.memorizing;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.UnregisterMemoryServerNotification;

/*
 * The thread processes the unregister notification from a memory server concurrently. 11/28/2014, Bing
Li
 */

// Created: 11/28/2014, Bing Li
public class UnregisterMemoryServerThread extends
NotificationQueue<UnregisterMemoryServerNotification>
{
  /*
   * Initialize the thread. 11/28/2014, Bing Li
   */
  public UnregisterMemoryServerThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * Process the notification concurrently. 11/28/2014, Bing Li
   */
  public void run()
  {
    // Declare an instance of UnregisterMemoryServerNotification. 11/28/2014, Bing Li
    UnregisterMemoryServerNotification notification;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/28/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/28/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/28/2014, Bing Li
          notification = this.getNotification();
          // Unregister the memory server from the registry. 11/28/2014, Bing Li
          MemoryRegistry.COORDINATE().unregister(notification.getDCKey());
          // Dispose the notification. 11/28/2014, Bing Li
          this.disposeMessage(notification);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      try
      {
        // Wait for a moment after all of the existing notifications are processed. 11/28/2014, Bing Li
        this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **UnregisterMemoryServerThreadCreator**

```
package com.greatfree.testing.coordinator.memorizing;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.UnregisterMemoryServerNotification;

/*
 * The creator here attempts to create instances of UnregisterMemoryServerThread. It works with the
notification dispatcher to schedule the tasks concurrently. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class UnregisterMemoryServerThreadCreator implements
NotificationThreadCreatable<UnregisterMemoryServerNotification, UnregisterMemoryServerThread>
{
  @Override
  public UnregisterMemoryServerThread createNotificationThreadInstance(int taskSize)
  {
    return new UnregisterMemoryServerThread(taskSize);
  }
}
```

- **NotifyIsPublisherExistedThread**

```
package com.greatfree.testing.coordinator.memorizing;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.testing.coordinator.searching.CoordinatorMulticastReader;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.IsPublisherExistedAnycastResponse;

/*
 * The thread notifies the coordinator multicast reader that an anycast response is received. Then, it
determines if the anycast requesting process is terminated or not. 11/29/2014, Bing Li
 *
 * Usually, the notifying takes very short time. So it is not necessary to use the structure. 11/29/2014,
Bing Li
 */

// Created: 11/29/2014, Bing Li
public class NotifyIsPublisherExistedThread extends
NotificationQueue<IsPublisherExistedAnycastResponse>
{
  /*
   * Initialize the thread. 11/28/2014, Bing Li
   */
  public NotifyIsPublisherExistedThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * Process the notification concurrently. 11/28/2014, Bing Li
   */
  public void run()
  {
    // The instance of response. 11/28/2014, Bing Li
    IsPublisherExistedAnycastResponse response;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/28/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/28/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the response. 11/29/2014, Bing Li
          response = this.getNotification();
          // Notify the coordinator multicast reader. 11/29/2014, Bing Li
          CoordinatorMulticastReader.COORDINATE().notifyResponseReceived(response);
          // Dispose the response. 11/29/2014, Bing Li
          this.disposeMessage(response);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      try
      {
        // Wait for a moment after all of the existing requests are processed. 11/29/2014, Bing Li
        this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
```

```
  }
}
```

- **NotifyIsPublisherExistedThreadCreator**

```
package com.greatfree.testing.coordinator.memorizing;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.IsPublisherExistedAnycastResponse;

/*
 * The creator here attempts to create instances of NotifyIsPublisherExistedThread. It works with the
notification dispatcher to schedule the tasks concurrently. 11/28/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class NotifyIsPublisherExistedThreadCreator implements
NotificationThreadCreatable<IsPublisherExistedAnycastResponse, NotifyIsPublisherExistedThread>
{
  @Override
  public NotifyIsPublisherExistedThread createNotificationThreadInstance(int taskSize)
  {
    return new NotifyIsPublisherExistedThread(taskSize);
  }
}
```

- **NotifySearchKeywordThread**

```
package com.greatfree.testing.coordinator.memorizing;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.testing.coordinator.searching.CoordinatorMulticastReader;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.SearchKeywordBroadcastResponse;

/*
 * The thread notifies the coordinator multicast reader that an broadcast response is received. Then, it
determines if the broadcast requesting process is terminated or not. 11/29/2014, Bing Li
 *
 * Usually, the notifying takes very short time. So it is not necessary to use the structure. 11/29/2014,
Bing Li
 */

// Created: 11/29/2014, Bing Li
public class NotifySearchKeywordThread extends
NotificationQueue<SearchKeywordBroadcastResponse>
{
  /*
   * Initialize the thread. 11/28/2014, Bing Li
   */
  public NotifySearchKeywordThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * Process the notification concurrently. 11/28/2014, Bing Li
   */
  public void run()
  {
    // The instance of response. 11/28/2014, Bing Li
    SearchKeywordBroadcastResponse response;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/28/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/28/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the response. 11/29/2014, Bing Li
          response = this.getNotification();
          // Notify the coordinator multicast reader. 11/29/2014, Bing Li
          CoordinatorMulticastReader.COORDINATE().notifyResponseReceived(response);
          // Dispose the response. 11/29/2014, Bing Li
          this.disposeMessage(response);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      try
      {
        // Wait for a moment after all of the existing requests are processed. 11/29/2014, Bing Li
        this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
```

```
    }
}
```

- **NotifySearchKeywordThreadCreator**

```
package com.greatfree.testing.coordinator.memorizing;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.SearchKeywordBroadcastResponse;

/*
 * The creator here attempts to create instances of NotifySearchKeywordThread. It works with the
notification dispatcher to schedule the tasks concurrently. 11/28/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class NotifySearchKeywordThreadCreator implements
NotificationThreadCreatable<SearchKeywordBroadcastResponse, NotifySearchKeywordThread>
{
  @Override
  public NotifySearchKeywordThread createNotificationThreadInstance(int taskSize)
  {
    return new NotifySearchKeywordThread(taskSize);
  }
}
```

3.6.1.4 The Searching

- **SearchListener**

```
package com.greatfree.testing.coordinator.searching;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

import com.greatfree.remote.IPPort;
import com.greatfree.remote.ServerListener;
import com.greatfree.testing.coordinator.memorizing.MemoryServerClientPool;
import com.greatfree.testing.data.ServerConfig;

/*
 * This is a coordinator listener that not only responds to search clients. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchListener extends ServerListener implements Runnable
{
  // A thread to connect the search client concurrently. 11/29/2014, Bing Li
  private ConnectSearcherThread connectThread;

  /*
   * Initialize the listener. 11/29/2014, Bing Li
   */
  public SearchListener(ServerSocket serverSocket, int threadPoolSize, long keepAliveTime)
  {
    super(serverSocket, threadPoolSize, keepAliveTime);
  }

  /*
   * Shutdown the listener. 11/29/2014, Bing Li
   */
  public void shutdown()
  {
    // Dispose the connecting thread. 11/29/2014, Bing Li
    this.connectThread.dispose();
    // Shutdown the listener. 11/29/2014, Bing Li
    super.shutdown();
  }

  /*
   * The task that must be executed concurrently. 11/29/2014, Bing Li
   */
  @Override
  public void run()
  {
    Socket clientSocket;
    SearchIO serverIO;

    // Detect whether the listener is shutdown. If not, it must be running all the time to wait for potential
connections from crawlers. 11/29/2014, Bing Li
    while (!super.isShutdown())
    {
      try
      {
        // Wait and accept a connecting from a possible memory server. 11/29/2014, Bing Li
        clientSocket = super.accept();
        // Check whether the connected server IOs exceed the upper limit. 11/29/2014, Bing Li
        if (SearchIORegistry.REGISTRY().getIOCount() >= ServerConfig.MAX_SERVER_IO_COUNT)
        {
          try
          {
            // If the upper limit is reached, the listener has to wait until an existing server IO is disposed.
11/29/2014, Bing Li
```

```
                super.holdOn();
              }
              catch (InterruptedException e)
              {
                e.printStackTrace();
              }
            }

            // If the upper limit of IOs is not reached, a search server IO is initialized. A common Collaborator
and the socket are the initial parameters. The shared common collaborator guarantees all of the search
server IOs from a certain search client could notify with each other with the same lock. Then, the upper
limit of search server IOs is under the control. 11/29/2014, Bing Li
            serverIO = new SearchIO(clientSocket, super.getCollaborator());

            /*
             * Since the listener servers need to form a peer-to-peer architecture, it is required to connect to
the remote search client. Thus, the local server can send messages without waiting for any requests
from the searcher. 11/29/2014, Bing Li
             */

            // Check whether a client to the IP and the port number of the remote search clients is existed in
the client pool. If such a client does not exist, it is required to connect the remote search client by the
thread concurrently. Doing that concurrently is to speed up the rate of responding to the search client.
11/29/2014, Bing Li
            if (!MemoryServerClientPool.COORDINATE().getPool().isClientExisted(serverIO.getIP(),
ServerConfig.SEARCH_CLIENT_PORT))
            {
              // Since the client does not exist in the pool, input the IP address and the port number to the
thread. 11/29/2014, Bing Li
              this.connectThread.enqueue(new IPPort(serverIO.getIP(),
ServerConfig.SEARCH_CLIENT_PORT));
              // Execute the thread to connect to the remote search client. 11/29/2014, Bing Li
              super.execute(this.connectThread);
            }

            // Add the new created server IO into the registry for further management. 11/29/2014, Bing Li
            SearchIORegistry.REGISTRY().addIO(serverIO);
            // Execute the new created search IO concurrently to respond the search clients' requests and
notifications in an asynchronous manner. 11/29/2014, Bing Li
            super.execute(serverIO);
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
        }
      }
    }
```

- **SearchListenerDisposer**

```
package com.greatfree.testing.coordinator.searching;

import com.greatfree.reuse.RunDisposable;

/*
 * The class is responsible for disposing the instance of SearchListener by invoking its method of
shutdown(). It is invoked by the runner to execute the server IOs concurrently. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchListenerDisposer implements RunDisposable<SearchListener>
{
   /*
    * Dispose the instance of SearchListener. 11/29/2014, Bing Li
    */
   @Override
   public void dispose(SearchListener r)
   {
      r.shutdown();
   }

   /*
    * Dispose the instance of SearchListener. The method does not make sense to SearchListener. Just
leave it here for the requirement of the interface, RunDisposable<SearchListener>. 11/29/2014, Bing Li
    */
   @Override
   public void dispose(SearchListener r, long time)
   {
      r.shutdown();
   }
}
```

- **SearchIO**

```
package com.greatfree.testing.coordinator.searching;

import java.io.IOException;
import java.net.Socket;
import java.net.SocketException;

import com.greatfree.concurrency.Collaborator;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.remote.ServerIO;
import com.greatfree.testing.coordinator.CoordinatorMessageProducer;

/*
 * The class is actually an implementation of ServerIO, which serves for the searchers which access the coordinator. 11/28/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchIO extends ServerIO
{
  /*
   * Initialize the server IO. The socket is the connection between the searchers and the coordinator. The collaborator is shared with other IOs to control the count of ServerIOs instances. 11/29/2014, Bing Li
   */
  public SearchIO(Socket clientSocket, Collaborator collaborator)
  {
    super(clientSocket, collaborator);
  }

  public void run()
  {
    ServerMessage message;
    while (!super.isShutdown())
    {
      try
      {
        // Wait and read messages from a search client. 11/29/2014, Bing Li
        message = (ServerMessage)super.read();
        // Convert the received message to OutMessageStream and put it into the relevant dispatcher for concurrent processing. 11/29/2014, Bing Li
        CoordinatorMessageProducer.SERVER().produceSearchMessage(new OutMessageStream<ServerMessage>(super.getOutStream(), super.getLock(), message));
      }
      catch (SocketException e)
      {
        e.printStackTrace();
      }
      catch (IOException e)
      {
        e.printStackTrace();
      }
      catch (ClassNotFoundException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **SearchIORegistry**

```
package com.greatfree.testing.coordinator.searching;

import java.io.IOException;
import java.util.Set;

import com.greatfree.remote.ServerIORegistry;

/*
 * The class keeps all of SearchIOs. This is a singleton wrapper of ServerIORegistry. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchIORegistry
{
  // Declare an instance of ServerIORegistry for SearchIOs. 11/29/2014, Bing Li
  private ServerIORegistry<SearchIO> registry;

  /*
   * Initializing ... 11/29/2014, Bing Li
   */
  private SearchIORegistry()
  {
  }

  private static SearchIORegistry instance = new SearchIORegistry();

  public static SearchIORegistry REGISTRY()
  {
    if (instance == null)
    {
      instance = new SearchIORegistry();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the registry. 11/29/2014, Bing Li
   */
  public void dispose() throws IOException
  {
    this.registry.removeAllIOs();
  }

  /*
   * Initialize the registry. 11/29/2014, Bing Li
   */
  public void init()
  {
    this.registry = new ServerIORegistry<SearchIO>();
  }

  /*
   * Add a new instance of SearchIO to the registry. 11/29/2014, Bing Li
   */
  public void addIO(SearchIO io)
  {
    this.registry.addIO(io);
  }

  /*
```

```
    * Get all of the IPs of the connected clients from the corresponding SearchIOs. 11/29/2014, Bing Li
    */
   public Set<String> getIPs()
   {
      return this.registry.getIPs();
   }

   /*
    * Get the count of the registered SearchIOs. 11/29/2014, Bing Li
    */
   public int getIOCount()
   {
      return this.registry.getIOCount();
   }

   /*
    * Remove or unregister an SearchIO. It is executed when a client is down or the connection gets
something wrong. 11/29/2014, Bing Li
    */
   public void removeIO(SearchIO io) throws IOException
   {
      this.registry.removeIO(io);
   }

   /*
    * Remove or unregister all of the registered SearchIOs. It is executed when the server process is
shutdown. 11/29/2014, Bing Li
    */
   public void removeAllIOs() throws IOException
   {
      this.registry.removeAllIOs();
   }
}
```

- **ConnectSearcherThread**

```
package com.greatfree.testing.coordinator.searching;

import java.io.IOException;
import java.util.Queue;
import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.remote.IPPort;
import com.greatfree.testing.message.NodeKeyNotification;

/*
 * The class is derived from Thread. It is responsible for connecting the search client such that it is feasible for the server to notify the client even without the memory server's request. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class ConnectSearcherThread extends Thread
{
  // The queue keeps the clients' IPs which need to be connected to. 11/29/2014, Bing Li
  private Queue<IPPort> ipQueue;

  /*
   * Initialize. 11/29/2014, Bing Li
   */
  public ConnectSearcherThread()
  {
    this.ipQueue = new LinkedBlockingQueue<IPPort>();
  }

  /*
   * Dispose the resource of the class. 11/29/2014, Bing Li
   */
  public synchronized void dispose()
  {
    if (this.ipQueue != null)
    {
      this.ipQueue.clear();
      this.ipQueue = null;
    }
  }

  /*
   * Input the IP address and the port number into the queue. They are connected in the order of first-in-first-out. The CrawlServerClientPool is responsible for the connections. 11/29/2014, Bing Li
   */
  public void enqueue(IPPort ipPort)
  {
    this.ipQueue.add(ipPort);
  }

  /*
   * Connect the remote IP addresses and the associated port numbers concurrently. 11/29/2014, Bing Li
   */
  @Override
  public void run()
  {
    IPPort ipPort;
    // The thread keeps working until all of the IP addresses are connected. 11/29/2014, Bing Li
    while (this.ipQueue.size() > 0)
    {
      // Dequeue the IP addresses. 11/29/2014, Bing Li
      ipPort = this.ipQueue.poll();
      try
      {
```

```
            // Notify the remote node its unique ID. This is usually used to set up a multicasting cluster which
contains a bunch of nodes. Each of them is identified by the ID. 11/29/2014, Bing Li
            SearchClientPool.COORDINATE().getPool().send(ipPort, new
NodeKeyNotification(ipPort.getObjectKey()));
        }
        catch (IOException e)
        {
          e.printStackTrace();
        }
      }
    }
}
```

- **SearchClientPool**

```
package com.greatfree.testing.coordinator.searching;

import java.io.IOException;

import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.coordinator.CoorConfig;

/*
 * This is a pool that creates and manages instances of FreeClient to connect the search clients in a high performance. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchClientPool
{
  // An instance of FreeClientPool is defined to interact with the crawler. 11/29/2014, Bing Li
  private FreeClientPool clientPool;

  private SearchClientPool()
  {
  }

  /*
   * A singleton definition. 11/29/2014, Bing Li
   */
  private static SearchClientPool instance = new SearchClientPool();

  public static SearchClientPool COORDINATE()
  {
    if (instance == null)
    {
      instance = new SearchClientPool();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the free client pool. 11/29/2014, Bing Li
   */
  public void dispose()
  {
    try
    {
      this.clientPool.dispose();
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
  }

  /*
   * Initialize the client pool. The method is called when the coordinator process is started. 11/29/2014, Bing Li
   */
  public void init()
  {
    // Initialize the client pool. 11/29/2014, Bing Li
    this.clientPool = new FreeClientPool(CoorConfig.CSERVER_CLIENT_POOL_SIZE);
    // Set idle checking for the client pool. 11/29/2014, Bing Li
```

```
    this.clientPool.setIdleChecker(CoorConfig.CSERVER_CLIENT_IDLE_CHECK_DELAY,
CoorConfig.CSERVER_CLIENT_IDLE_CHECK_PERIOD,
CoorConfig.CSERVER_CLIENT_MAX_IDLE_TIME);
  }

  /*
   * Expose the client pool. 11/29/2014, Bing Li
   */
  public FreeClientPool getPool()
  {
    return this.clientPool;
  }
}
```

- **SearchServerDispatcher**

```
package com.greatfree.testing.coordinator.searching;

import com.greatfree.concurrency.NotificationDispatcher;
import com.greatfree.concurrency.RequestDispatcher;
import com.greatfree.concurrency.ServerMessageDispatcher;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.InitReadNotification;
import com.greatfree.testing.message.IsPublisherExistedRequest;
import com.greatfree.testing.message.IsPublisherExistedResponse;
import com.greatfree.testing.message.IsPublisherExistedStream;
import com.greatfree.testing.message.MessageType;
import com.greatfree.testing.message.SearchKeywordRequest;
import com.greatfree.testing.message.SearchKeywordResponse;
import com.greatfree.testing.message.SearchKeywordStream;

/*
 * This is an implementation of ServerMessageDispatcher. It contains the concurrency mechanism to
respond searchers' requests and receive their notifications for the coordinator. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchServerDispatcher extends ServerMessageDispatcher<ServerMessage>
{
   // Declare a notification dispatcher to deal with instances of InitReadNotification from a client
concurrently such that the client can initialize its ObjectInputStream. 11/29/2014, Bing Li
   private NotificationDispatcher<InitReadNotification, InitReadFeedbackThread,
InitReadFeedbackThreadCreator> initReadFeedbackNotificationDispatcher;
   // Declare a request dispatcher to respond users requests concurrently. 11/29/2014, Bing Li
   private RequestDispatcher<IsPublisherExistedRequest, IsPublisherExistedStream,
IsPublisherExistedResponse, IsPublisherExistedThread, IsPublisherExistedThreadCreator>
isPublisherExistedRequestDispatcher;
   // Declare a request dispatcher to respond users requests concurrently. 11/29/2014, Bing Li
   private RequestDispatcher<SearchKeywordRequest, SearchKeywordStream,
SearchKeywordResponse, SearchKeywordThread, SearchKeywordThreadCreator>
searchKeywordRequestDispatcher;

   /*
    * Initialize. 11/29/2014, Bing Li
    */
   public SearchServerDispatcher(int corePoolSize, long keepAliveTime)
   {
      super(corePoolSize, keepAliveTime);

      this.initReadFeedbackNotificationDispatcher = new NotificationDispatcher<InitReadNotification,
InitReadFeedbackThread,
InitReadFeedbackThreadCreator>(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new
InitReadFeedbackThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);

   this.initReadFeedbackNotificationDispatcher.setIdleChecker(ServerConfig.NOTIFICATION_DISPAT
CHER_IDLE_CHECK_DELAY, ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
      super.execute(this.initReadFeedbackNotificationDispatcher);

      // Initialize the request dispatcher. 11/29/2014, Bing Li
      this.isPublisherExistedRequestDispatcher = new RequestDispatcher<IsPublisherExistedRequest,
IsPublisherExistedStream, IsPublisherExistedResponse, IsPublisherExistedThread,
IsPublisherExistedThreadCreator>(ServerConfig.REQUEST_DISPATCHER_POOL_SIZE,
ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME, new
IsPublisherExistedThreadCreator(), ServerConfig.MAX_REQUEST_TASK_SIZE,
ServerConfig.MAX_REQUEST_THREAD_SIZE,
```

```
ServerConfig.REQUEST_DISPATCHER_WAIT_TIME);
      // Set the parameters to check idle states of threads. 11/29/2014, Bing Li

    this.isPublisherExistedRequestDispatcher.setIdleChecker(ServerConfig.REQUEST_DISPATCHER_I
DLE_CHECK_DELAY, ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD);
      // Start the dispatcher. 11/29/2014, Bing Li
      super.execute(this.isPublisherExistedRequestDispatcher);

      // Initialize the request dispatcher. 11/29/2014, Bing Li
      this.searchKeywordRequestDispatcher = new RequestDispatcher<SearchKeywordRequest,
SearchKeywordStream, SearchKeywordResponse, SearchKeywordThread,
SearchKeywordThreadCreator>(ServerConfig.REQUEST_DISPATCHER_POOL_SIZE,
ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME, new
SearchKeywordThreadCreator(), ServerConfig.MAX_REQUEST_TASK_SIZE,
ServerConfig.MAX_REQUEST_THREAD_SIZE,
ServerConfig.REQUEST_DISPATCHER_WAIT_TIME);
      // Set the parameters to check idle states of threads. 11/29/2014, Bing Li

    this.searchKeywordRequestDispatcher.setIdleChecker(ServerConfig.REQUEST_DISPATCHER_IDL
E_CHECK_DELAY, ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD);
      // Start the dispatcher. 11/29/2014, Bing Li
      super.execute(this.searchKeywordRequestDispatcher);
    }

    /*
     * Shut down the server message dispatcher. 11/29/2014, Bing Li
     */
    public void shutdown()
    {
      this.initReadFeedbackNotificationDispatcher.dispose();
      this.isPublisherExistedRequestDispatcher.dispose();
      this.searchKeywordRequestDispatcher.dispose();
      super.shutdown();
    }

    /*
     * Process the available messages in a concurrent way. 11/29/2014, Bing Li
     */
    public void consume(OutMessageStream<ServerMessage> message)
    {
      // Check the types of received messages. 11/29/2014, Bing Li
      switch (message.getMessage().getType())
      {
        // If the message is the one of initializing notification. 11/29/2014, Bing Li
        case MessageType.INIT_READ_NOTIFICATION:
          // Enqueue the notification into the dispatcher for concurrent feedback. 11/29/2014, Bing Li

    this.initReadFeedbackNotificationDispatcher.enqueue((InitReadNotification)message.getMessage());
          break;

        // Process the search request. 11/29/2014, Bing Li
        case MessageType.IS_PUBLISHER_EXISTED_REQUEST:
          // Enqueue the request into the dispatcher for concurrent responding. 11/29/2014, Bing Li
          this.isPublisherExistedRequestDispatcher.enqueue(new
IsPublisherExistedStream(message.getOutStream(), message.getLock(),
(IsPublisherExistedRequest)message.getMessage()));
          break;

        // Process the search request. 11/29/2014, Bing Li
        case MessageType.SEARCH_KEYWORD_REQUEST:
          // Enqueue the request into the dispatcher for concurrent responding. 11/29/2014, Bing Li
          this.searchKeywordRequestDispatcher.enqueue(new
SearchKeywordStream(message.getOutStream(), message.getLock(),
(SearchKeywordRequest)message.getMessage()));
          break;
      }
    }
}
```

- **SearchServerProducerDisposer**

```
package com.greatfree.testing.coordinator.searching;

import com.greatfree.concurrency.MessageProducer;
import com.greatfree.reuse.ThreadDisposable;

/*
 * The class is responsible for disposing the searchers' message producer of the coordinator.
11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchServerProducerDisposer implements
ThreadDisposable<MessageProducer<SearchServerDispatcher>>
{
   /*
    * Dispose the message producer. 11/29/2014, Bing Li
    */
   @Override
   public void dispose(MessageProducer<SearchServerDispatcher> r)
   {
      r.dispose();
   }

   /*
    * The method does not make sense to the class of MessageProducer. Just leave it here. 11/29/2014,
Bing Li
    */
   @Override
   public void dispose(MessageProducer<SearchServerDispatcher> r, long time)
   {
      r.dispose();
   }
}
```

- **InitReadFeedbackThread**

```
package com.greatfree.testing.coordinator.searching;

import java.io.IOException;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.InitReadFeedbackNotification;
import com.greatfree.testing.message.InitReadNotification;

/*
 * This is an important thread since it ensure the local ObjectOutputStream is initialized and it notifies to
the relevant remote ObjectInputStream can be initialized. 11/09/2014, Bing Li
 */

// Created: 11/09/2014, Bing Li
public class InitReadFeedbackThread extends NotificationQueue<InitReadNotification>
{
  /*
   * Initialize the thread. The argument, taskSize, is used to limit the count of tasks to be queued.
11/09/2014, Bing Li
   */
  public InitReadFeedbackThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * Once if a node key notification is received, it is processed concurrently as follows. 11/09/2014, Bing
Li
   */
  public void run()
  {
    // Declare an instance of InitReadNotification. 11/09/2014, Bing Li
    InitReadNotification notification;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/09/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/09/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/09/2014, Bing Li
          notification = this.getNotification();
          try
          {
            // Send the instance of InitReadFeedbackNotification to the client which needs to initialize
the ObjectInputStream of an instance of FreeClient. 11/09/2014, Bing Li
            SearchClientPool.COORDINATE().getPool().send(notification.getClientKey(), new
InitReadFeedbackNotification());
            this.disposeMessage(notification);
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      try
      {
```

```
            // Wait for a moment after all of the existing notifications are processed. 11/09/2014, Bing Li
            this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
        }
      }
    }
```

- **InitReadFeedbackThreadCreator**

```
package com.greatfree.testing.coordinator.searching;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.InitReadNotification;

/*
 * The code here attempts to create instances of SetInputStreamThread. 11/07/2014, Bing Li
 */

// Created: 11/09/2014, Bing Li
public class InitReadFeedbackThreadCreator implements
NotificationThreadCreatable<InitReadNotification, InitReadFeedbackThread>
{
  @Override
  public InitReadFeedbackThread createNotificationThreadInstance(int taskSize)
  {
    return new InitReadFeedbackThread(taskSize);
  }
}
```

- **IsPublisherExistedThread**

```
package com.greatfree.testing.coordinator.searching;

import java.io.IOException;

import com.greatfree.concurrency.RequestQueue;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.IsPublisherExistedRequest;
import com.greatfree.testing.message.IsPublisherExistedResponse;
import com.greatfree.testing.message.IsPublisherExistedStream;

/*
 * This is an example to use RequestQueue, which receives users' requests and responds concurrently.
11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedThread extends RequestQueue<IsPublisherExistedRequest,
IsPublisherExistedStream, IsPublisherExistedResponse>
{
  /*
   * Initialize the thread. The value of maxTaskSize is the length of the queue to take the count of
requests. 11/29/2014, Bing Li
   */
  public IsPublisherExistedThread(int maxTaskSize)
  {
    super(maxTaskSize);
  }

  /*
   * Respond users' requests concurrently. 11/29/2014, Bing Li
   */
  public void run()
  {
    // Declare the request stream. 11/29/2014, Bing Li
    IsPublisherExistedStream request;
    // Declare the response. 11/29/2014, Bing Li
    IsPublisherExistedResponse response;
    // The thread is shutdown when it is idle long enough. Before that, the thread keeps alive. It is
necessary to detect whether it is time to end the task. 11/29/2014, Bing Li
    while (!this.isShutdown())
    {
      // The loop detects whether the queue is empty or not. 11/29/2014, Bing Li
      while (!this.isEmpty())
      {
        // Dequeue a request. 11/29/2014, Bing Li
        request = this.getRequest();
        // Invoke the multicastor reader to retrieve the data in the cluster of the memory nodes in an
anycast manner. 11/29/2014, Bing Li
        response = new
IsPublisherExistedResponse(CoordinatorMulticastReader.COORDINATE().isPublisherExisted(request.
getMessage().getURL()));
        try
        {
          // Respond to the client. 11/29/2014, Bing Li
          this.respond(request.getOutStream(), request.getLock(), response);
        }
        catch (IOException e)
        {
          e.printStackTrace();
        }
        // Dispose the request and the response. 11/29/2014, Bing Li
        this.disposeMessage(request, response);
      }
      try
```

```
        {
          // Wait for some time when the queue is empty. During the period and before the thread is killed,
some new requests might be received. If so, the thread can keep working. 11/29/2014, Bing Li
          this.holdOn(ServerConfig.RETRIEVE_THREAD_WAIT_TIME);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
    }
  }
```

- **IsPublisherExistedThreadCreator**

```
package com.greatfree.testing.coordinator.searching;

import com.greatfree.concurrency.RequestThreadCreatable;
import com.greatfree.testing.message.IsPublisherExistedRequest;
import com.greatfree.testing.message.IsPublisherExistedResponse;
import com.greatfree.testing.message.IsPublisherExistedStream;

/*
 * A creator to initialize instances of IsPublisherExistedThread. It is used in the instance of
RequestDispatcher. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedThreadCreator implements
RequestThreadCreatable<IsPublisherExistedRequest, IsPublisherExistedStream,
IsPublisherExistedResponse, IsPublisherExistedThread>
{
  @Override
  public IsPublisherExistedThread createRequestThreadInstance(int taskSize)
  {
    return new IsPublisherExistedThread(taskSize);
  }
}
```

- **SearchKeywordThread**

```
package com.greatfree.testing.coordinator.searching;

import java.io.IOException;

import com.greatfree.concurrency.RequestQueue;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.SearchKeywordRequest;
import com.greatfree.testing.message.SearchKeywordResponse;
import com.greatfree.testing.message.SearchKeywordStream;

/*
 * This is an example to use RequestQueue, which receives users' requests and responds concurrently.
11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordThread extends RequestQueue<SearchKeywordRequest,
SearchKeywordStream, SearchKeywordResponse>
{
  /*
   * Initialize the thread. The value of maxTaskSize is the length of the queue to take the count of
requests. 11/29/2014, Bing Li
   */
  public SearchKeywordThread(int maxTaskSize)
  {
    super(maxTaskSize);
  }

  /*
   * Respond users' requests concurrently. 11/29/2014, Bing Li
   */
  public void run()
  {
    // Declare the request stream. 11/29/2014, Bing Li
    SearchKeywordStream request;
    // Declare the response. 11/29/2014, Bing Li
    SearchKeywordResponse response;
    // The thread is shutdown when it is idle long enough. Before that, the thread keeps alive. It is
necessary to detect whether it is time to end the task. 11/29/2014, Bing Li
    while (!this.isShutdown())
    {
      // The loop detects whether the queue is empty or not. 11/29/2014, Bing Li
      while (!this.isEmpty())
      {
        // Dequeue a request. 11/29/2014, Bing Li
        request = this.getRequest();
        // Invoke the multicastor reader to retrieve the data in the cluster of the memory nodes in an
anycast manner. 11/29/2014, Bing Li
        response = new SearchKeywordResponse(request.getMessage().getKeyword(),
CoordinatorMulticastReader.COORDINATE().searchKeyword(request.getMessage().getKeyword()));
        try
        {
          // Respond to the client. 11/29/2014, Bing Li
          this.respond(request.getOutStream(), request.getLock(), response);
        }
        catch (IOException e)
        {
          e.printStackTrace();
        }
        // Dispose the request and the response. 11/29/2014, Bing Li
        this.disposeMessage(request, response);
      }
      try
      {
```

```
            // Wait for some time when the queue is empty. During the period and before the thread is killed,
some new requests might be received. If so, the thread can keep working. 11/29/2014, Bing Li
            this.holdOn(ServerConfig.RETRIEVE_THREAD_WAIT_TIME);
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
        }
      }
    }
```

- **SearchKeywordThreadCreator**

```
package com.greatfree.testing.coordinator.searching;

import com.greatfree.concurrency.RequestThreadCreatable;
import com.greatfree.testing.message.SearchKeywordRequest;
import com.greatfree.testing.message.SearchKeywordResponse;
import com.greatfree.testing.message.SearchKeywordStream;

/*
 * A creator to initialize instances of SearchKeywordThread. It is used in the instance of
RequestDispatcher. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordThreadCreator implements
RequestThreadCreatable<SearchKeywordRequest, SearchKeywordStream, SearchKeywordResponse,
SearchKeywordThread>
{
  @Override
  public SearchKeywordThread createRequestThreadInstance(int taskSize)
  {
    return new SearchKeywordThread(taskSize);
  }
}
```

- **CoordinatorMulticastReader**

```
package com.greatfree.testing.coordinator.searching;

import java.io.IOException;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;

import com.google.common.collect.Sets;
import com.greatfree.reuse.ResourcePool;
import com.greatfree.testing.coordinator.CoorConfig;
import com.greatfree.testing.coordinator.memorizing.MemoryServerClientPool;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.IsPublisherExistedAnycastResponse;
import com.greatfree.testing.message.SearchKeywordBroadcastResponse;

/*
 * This is a wrapper that encloses all of anycast and broadcast readers to accomplish the goal to retrieve data from the cluster of memory nodes. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class CoordinatorMulticastReader
{
  // The collection to save the current working anycast readers. 11/29/2014, Bing Li
  private Map<String, IsPublisherExistedAnycastReader> isPublisherExistedAnycastReaders;
  // The resource pool to manage the instances of anycast readers. 11/29/2014, Bing Li
  private ResourcePool<IsPublisherExistedAnycastReaderSource, IsPublisherExistedAnycastReader, IsPublisherExistedAnycastReaderCreator, IsPublisherExistedAnycastReaderDisposer> isPublisherExistedAnycastReaderPool;

  // The collection to save the current working broadcast readers. 11/29/2014, Bing Li
  private Map<String, SearchKeywordBroadcastReader> searchKeywordBroadcastReaders;
  // The resource pool to manage the instances of broadcast readers. 11/29/2014, Bing Li
  private ResourcePool<SearchKeywordBroadcastReaderSource, SearchKeywordBroadcastReader, SearchKeywordBroadcastReaderCreator, SearchKeywordBroadcastReaderDisposer> searchKeywordBroadcastReaderPool;

  private CoordinatorMulticastReader()
  {
  }

  /*
   * A singleton implementation. 11/26/2014, Bing Li
   */
  private static CoordinatorMulticastReader instance = new CoordinatorMulticastReader();

  public static CoordinatorMulticastReader COORDINATE()
  {
    if (instance == null)
    {
      instance = new CoordinatorMulticastReader();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose all of the pools. 11/26/2014, Bing Li
   */
  public void dispose()
  {
```

```
    // Dispose each anycast reader. 11/29/2014, Bing Li
    for (IsPublisherExistedAnycastReader reader: this.isPublisherExistedAnycastReaders.values())
    {
      reader.dispose();
    }
    this.isPublisherExistedAnycastReaders.clear();
    // Shutdown the anycast reader pool. 11/29/2014, Bing Li
    this.isPublisherExistedAnycastReaderPool.shutdown();

    // Dispose each broadcast reader. 11/29/2014, Bing Li
    for (SearchKeywordBroadcastReader reader: this.searchKeywordBroadcastReaders.values())
    {
      reader.dispose();
    }
    this.searchKeywordBroadcastReaders.clear();
    // Shutdown the broadcast reader pool. 11/29/2014, Bing Li
    this.searchKeywordBroadcastReaderPool.shutdown();
  }

  /*
   * Initialize. 11/29/2014, Bing Li
   */
  public void init()
  {
    // Initialize the anycast reader collection. 11/29/2014, Bing Li
    this.isPublisherExistedAnycastReaders = new ConcurrentHashMap<String,
IsPublisherExistedAnycastReader>();
    // Initialize the pool to manage the readers. 11/29/2014, Bing Li
    this.isPublisherExistedAnycastReaderPool = new
ResourcePool<IsPublisherExistedAnycastReaderSource, IsPublisherExistedAnycastReader,
IsPublisherExistedAnycastReaderCreator,
IsPublisherExistedAnycastReaderDisposer>(CoorConfig.AUTHORITY_ANYCAST_READER_POOL_SI
ZE, new IsPublisherExistedAnycastReaderCreator(), new
IsPublisherExistedAnycastReaderDisposer(),
CoorConfig.AUTHORITY_ANYCAST_READER_POOL_WAIT_TIME);

    // Initialize the broadcast reader collection. 11/29/2014, Bing Li
    this.searchKeywordBroadcastReaders = new ConcurrentHashMap<String,
SearchKeywordBroadcastReader>();
    this.searchKeywordBroadcastReaderPool = new
ResourcePool<SearchKeywordBroadcastReaderSource, SearchKeywordBroadcastReader,
SearchKeywordBroadcastReaderCreator,
SearchKeywordBroadcastReaderDisposer>(CoorConfig.BROADCAST_READER_POOL_SIZE, new
SearchKeywordBroadcastReaderCreator(), new SearchKeywordBroadcastReaderDisposer(),
CoorConfig.BROADCAST_READER_POOL_WAIT_TIME);
  }

  /*
   * Retrieve whether the URL is existed by anycast. 11/29/2014, Bing Li
   */
  public boolean isPublisherExisted(String url)
  {
    try
    {
      // Get one instance of the anycast reader. 11/29/2014, Bing Li
      IsPublisherExistedAnycastReader reader = this.isPublisherExistedAnycastReaderPool.get(new
IsPublisherExistedAnycastReaderSource(MemoryServerClientPool.COORDINATE().getPool(),
ServerConfig.ROOT_MULTICAST_BRANCH_COUNT, ServerConfig.MULTICAST_BRANCH_COUNT,
new IsPublisherExistedAnycastRequestCreator()));
      // Get the collaborator key of the reader. The key is unique to the reader. 11/29/2014, Bing Li
      String collaboratorKey = reader.resetAnycast();
      // Save the instance of the anycast reader. 11/29/2014, Bing Li
      this.isPublisherExistedAnycastReaders.put(collaboratorKey, reader);
      try
      {
        // Anycast the request and wait for the response. 11/29/2014, Bing Li
        IsPublisherExistedAnycastResponse response = reader.disseminate(url);
        // After the response is received, the reader must be collected. 11/29/2014, Bing Li
```

```
            this.isPublisherExistedAnycastReaderPool.collect(reader);
            // Remove the reader from the collection. 11/29/2014, Bing Li
            this.isPublisherExistedAnycastReaders.remove(collaboratorKey);
            // Check whether the response is valid. 11/29/2014, Bing Li
            if (response != null)
            {
              // Return the value of the response. 11/29/2014, Bing Li
              return response.isExisted();
            }
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
        }
        catch (InstantiationException e)
        {
          e.printStackTrace();
        }
        catch (IllegalAccessException e)
        {
          e.printStackTrace();
        }
        // Return false in other cases. 11/29/2014, Bing Li
        return false;
      }

      /*
       * Once if a response to an anycast response is received, it is required to check whether a
corresponding collaborator is waiting. If so, just signal the collaborator and the response is what is
needed. 11/29/2014, Bing Li
       */
      public void notifyResponseReceived(IsPublisherExistedAnycastResponse response)
      {
        // Check whether the response has a matched unique corresponding collaborator. 11/29/2014, Bing
Li
        if (this.isPublisherExistedAnycastReaders.containsKey(response.getCollaboratorKey()))
        {
          // If it is, notify the collaborator. And the anycast requesting is accomplished. 11/29/2014, Bing Li
          this.isPublisherExistedAnycastReaders.get(response.getCollaboratorKey()).notify(response);
        }
      }

      /*
       * Retrieve keyword for the matched links by broadcast. 11/29/2014, Bing Li
       */
      public Set<String> searchKeyword(String keyword)
      {
        try
        {
          // Get one instance of the broadcast reader. 11/29/2014, Bing Li
          SearchKeywordBroadcastReader reader = this.searchKeywordBroadcastReaderPool.get(new
SearchKeywordBroadcastReaderSource(MemoryServerClientPool.COORDINATE().getPool(),
ServerConfig.ROOT_MULTICAST_BRANCH_COUNT, ServerConfig.MULTICAST_BRANCH_COUNT,
new SearchKeywordBroadcastRequestCreator()));
          // Get the collaborator key of the reader. The key is unique to the reader. 11/29/2014, Bing Li
          String collaboratorKey = reader.resetBroadcast();
          // Save the instance of the broadcast reader. 11/29/2014, Bing Li
          this.searchKeywordBroadcastReaders.put(collaboratorKey, reader);
          try
          {
            // Broadcast the request and wait for the responses. 11/29/2014, Bing Li
            Map<String, SearchKeywordBroadcastResponse> responses = reader.disseminate(keyword);
            // After the responses are received, the reader must be collected. 11/29/2014, Bing Li
```

```
            this.searchKeywordBroadcastReaderPool.collect(reader);
            // Remove the reader from the collection. 11/29/2014, Bing Li
            this.searchKeywordBroadcastReaders.remove(collaboratorKey);
            // Check whether some responses are received. 11/29/2014, Bing Li
            if (responses.size() > 0)
            {
              // Initialize an empty set to take all of the matched links. 11/29/2014, Bing Li
              Set<String> links = Sets.newHashSet();
              // Scan each response to accumulate their results. 11/29/2014, Bing Li
              for (SearchKeywordBroadcastResponse response : responses.values())
              {
                // Check whether the links are valid. 11/29/2014, Bing Li
                if (response.getLinks() != null)
                {
                  // Add them together. 11/29/2014, Bing Li
                  links.addAll(response.getLinks());
                }
              }
              // Return the retrieved results. 11/29/2014, Bing Li
              return links;
            }
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
        }
        catch (InstantiationException e)
        {
          e.printStackTrace();
        }
        catch (IllegalAccessException e)
        {
          e.printStackTrace();
        }
        // Return null if any problems. 11/29/2014, Bing Li
        return null;
      }

      /*
       * Once if a response to a broadcast request is received, it is required to save it in the reader until all of
the node respond or the waiting time is over. 11/29/2014, Bing Li
       */
      public void notifyResponseReceived(SearchKeywordBroadcastResponse response)
      {
        // Check whether the response is the result to one of the reader by comparing the collaborator keys.
11/29/2014, Bing Li
        if (this.searchKeywordBroadcastReaders.containsKey(response.getCollaboratorKey()))
        {
          // If the reader is waiting, save the response. The requesting process is waiting until all of the node
respond or the waiting time is over. 11/29/2014, Bing Li

      this.searchKeywordBroadcastReaders.get(response.getCollaboratorKey()).saveResponse(response)
;
        }
      }
    }
```

- **IsPublisherExistedAnycastReader**

```
package com.greatfree.testing.coordinator.searching;

import com.greatfree.multicast.RootRequestAnycastor;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.coordinator.CoorConfig;
import com.greatfree.testing.message.IsPublisherExistedAnycastRequest;
import com.greatfree.testing.message.IsPublisherExistedAnycastResponse;

/*
 * The reader is derived from the RootRequestAnycastor. It attempts to retrieve data in the way of
anycast among the cluster of memory nodes. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedAnycastReader extends RootRequestAnycastor<String,
IsPublisherExistedAnycastRequest, IsPublisherExistedAnycastResponse,
IsPublisherExistedAnycastRequestCreator>
{
   /*
    * Initialize the anycastor. 11/29/2014, Bing Li
    */
   public IsPublisherExistedAnycastReader(FreeClientPool clientPool, int rootBranchCount, int
treeBranchCount, IsPublisherExistedAnycastRequestCreator requestCreator)
   {
      super(clientPool, rootBranchCount, treeBranchCount, requestCreator,
CoorConfig.ANYCAST_REQUEST_WAIT_TIME);
   }
}
```

- **IsPublisherExistedAnycastReaderCreator**

```
package com.greatfree.testing.coordinator.searching;

import com.greatfree.reuse.HashCreatable;

/*
 * The creator initializes the instances of IsPublisherExistedAnycastReader for the resource pool.
11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedAnycastReaderCreator implements
HashCreatable<IsPublisherExistedAnycastReaderSource, IsPublisherExistedAnycastReader>
{
   @Override
   public IsPublisherExistedAnycastReader
createResourceInstance(IsPublisherExistedAnycastReaderSource source)
   {
      return new IsPublisherExistedAnycastReader(source.getClientPool(),
source.getRootBranchCount(), source.getTreeBranchCount(), source.getCreator());
   }
}
```

- **IsPublisherExistedAnycastReaderDisposer**

```
package com.greatfree.testing.coordinator.searching;

import com.greatfree.reuse.HashDisposable;

/*
 * This is a disposer that collects the resources of the anycast reader, IsPublisherExistedAnycastReader.
It is used by the resource pool for IsPublisherExistedAnycastReader. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedAnycastReaderDisposer implements
HashDisposable<IsPublisherExistedAnycastReader>
{
   @Override
   public void dispose(IsPublisherExistedAnycastReader t)
   {
      t.dispose();
   }
}
```

- **IsPublisherExistedAnycastReaderSource**

```
package com.greatfree.testing.coordinator.searching;

import com.greatfree.multicast.RootAnycastReaderSource;
import com.greatfree.multicast.RootAnycastRequestCreatorGettable;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.message.IsPublisherExistedAnycastRequest;

/*
 * The source contains all of the arguments to create the instance of anycast reader,
IsPublisherExistedAnycastReader. It is used by the resource pool that manages the instances of
IsPublisherExistedAnycastReader. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedAnycastReaderSource extends RootAnycastReaderSource<String,
IsPublisherExistedAnycastRequest, IsPublisherExistedAnycastRequestCreator> implements
RootAnycastRequestCreatorGettable<IsPublisherExistedAnycastRequest, String>
{
  /*
   * Initialize the source. 11/29/2014, Bing Li
   */
  public IsPublisherExistedAnycastReaderSource(FreeClientPool clientPool, int rootBranchCount, int
treeBranchCount, IsPublisherExistedAnycastRequestCreator creator)
  {
    super(clientPool, rootBranchCount, treeBranchCount, creator);
  }

  /*
   * Expose the anycast request creator. 11/29/2014, Bing Li
   */
  @Override
  public IsPublisherExistedAnycastRequestCreator getRequestCreator()
  {
    return super.getCreator();
  }
}
```

- **IsPublisherExistedAnycastRequestCreator**

```
package com.greatfree.testing.coordinator.searching;

import java.util.HashMap;

import com.greatfree.multicast.RootAnycastRequestCreatable;
import com.greatfree.testing.message.IsPublisherExistedAnycastRequest;
import com.greatfree.util.Tools;

/*
 * This creator aims to create requests of IsPublisherExistedAnycastRequest, in the root anycast
requester. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedAnycastRequestCreator implements
RootAnycastRequestCreatable<IsPublisherExistedAnycastRequest, String>
{
   /*
    * Create the anycast request for the node which has children. 11/29/2014, Bing Li
    */
   @Override
   public IsPublisherExistedAnycastRequest createInstanceWithChildren(String t, String
collaboratorKey, HashMap<String, String> childrenMap)
   {
      return new IsPublisherExistedAnycastRequest(Tools.generateUniqueKey(), t, collaboratorKey,
childrenMap);
   }

   /*
    * Create the anycast request for the node which has no children. 11/29/2014, Bing Li
    */
   @Override
   public IsPublisherExistedAnycastRequest createInstanceWithoutChildren(String t, String
collaboratorKey)
   {
      return new IsPublisherExistedAnycastRequest(Tools.generateUniqueKey(), t, collaboratorKey);
   }
}
```

- **SearchKeywordBroadcastReader**

```
package com.greatfree.testing.coordinator.searching;

import com.greatfree.multicast.RootRequestBroadcastor;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.coordinator.CoorConfig;
import com.greatfree.testing.message.SearchKeywordBroadcastRequest;
import com.greatfree.testing.message.SearchKeywordBroadcastResponse;

/*
 * The reader is derived from the RootRequestBroadcastor. It attempts to retrieve data in the way of broadcast among the cluster of memory nodes. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordBroadcastReader extends RootRequestBroadcastor<String, SearchKeywordBroadcastRequest, SearchKeywordBroadcastResponse, SearchKeywordBroadcastRequestCreator>
{
	/*
	 * Initialize the broadcastor. 11/29/2014, Bing Li
	 */
	public SearchKeywordBroadcastReader(FreeClientPool clientPool, int rootBranchCount, int treeBranchCount, SearchKeywordBroadcastRequestCreator requestCreator)
	{
		super(clientPool, rootBranchCount, treeBranchCount, requestCreator, CoorConfig.BROADCAST_REQUEST_WAIT_TIME);
	}
}
```

- **SearchKeywordBroadcastReaderCreator**

```
package com.greatfree.testing.coordinator.searching;

import com.greatfree.reuse.HashCreatable;

/*
 * The creator initializes the instances of SearchKeywordBroadcastReader for the resource pool.
11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordBroadcastReaderCreator implements
HashCreatable<SearchKeywordBroadcastReaderSource, SearchKeywordBroadcastReader>
{
  @Override
  public SearchKeywordBroadcastReader
createResourceInstance(SearchKeywordBroadcastReaderSource source)
  {
    return new SearchKeywordBroadcastReader(source.getClientPool(),
source.getRootBranchCount(), source.getTreeBranchCount(), source.getCreator());
  }
}
```

- **SearchKeywordBroadcastReaderDisposer**

```
package com.greatfree.testing.coordinator.searching;

import com.greatfree.reuse.HashDisposable;

/*
 * This is a disposer that collects the resources of the broadcast reader,
SearchKeywordBroadcastReader. It is used by the resource pool for SearchKeywordBroadcastReader.
11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordBroadcastReaderDisposer implements
HashDisposable<SearchKeywordBroadcastReader>
{
   @Override
   public void dispose(SearchKeywordBroadcastReader t)
   {
      t.dispose();
   }
}
```

- **SearchKeywordBroadcastReaderSource**

```
package com.greatfree.testing.coordinator.searching;

import com.greatfree.multicast.RootBroadcastReaderSource;
import com.greatfree.multicast.RootBroadcastRequestCreatorGettable;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.message.SearchKeywordBroadcastRequest;

/*
 * The source contains all of the arguments to create the instance of broadcast reader,
SearchKeywordBroadcastReader. It is used by the resource pool that manages the instances of
SearchKeywordBroadcastReader. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordBroadcastReaderSource extends RootBroadcastReaderSource<String,
SearchKeywordBroadcastRequest, SearchKeywordBroadcastRequestCreator> implements
RootBroadcastRequestCreatorGettable<SearchKeywordBroadcastRequest, String>
{
  /*
   * Initialize the source. 11/29/2014, Bing Li
   */
  public SearchKeywordBroadcastReaderSource(FreeClientPool clientPool, int rootBranchCount, int
treeBranchCount, SearchKeywordBroadcastRequestCreator creator)
  {
    super(clientPool, rootBranchCount, treeBranchCount, creator);
  }

  /*
   * Expose the broadcast request creator. 11/29/2014, Bing Li
   */
  @Override
  public SearchKeywordBroadcastRequestCreator getRequestCreator()
  {
    return super.getCreator();
  }
}
```

- **SearchKeywordBroadcastRequestCreator**

```
package com.greatfree.testing.coordinator.searching;

import java.util.HashMap;

import com.greatfree.multicast.RootBroadcastRequestCreatable;
import com.greatfree.testing.message.SearchKeywordBroadcastRequest;
import com.greatfree.util.Tools;

/*
 * This creator aims to create requests of SearchKeywordBroadcastRequest, in the root broadcast
requester. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordBroadcastRequestCreator implements
RootBroadcastRequestCreatable<SearchKeywordBroadcastRequest, String>
{
   /*
    * Create the broadcast request for the node which has children. 11/29/2014, Bing Li
    */
   @Override
   public SearchKeywordBroadcastRequest createInstanceWithChildren(String keyword, String
collaboratorKey, HashMap<String, String> childrenMap)
   {
      return new SearchKeywordBroadcastRequest(keyword, Tools.generateUniqueKey(),
collaboratorKey, childrenMap);
   }

   /*
    * Create the broadcast request for the node which has no children. 11/29/2014, Bing Li
    */
   @Override
   public SearchKeywordBroadcastRequest createInstanceWithoutChildren(String keyword, String
collaboratorKey)
   {
      return new SearchKeywordBroadcastRequest(keyword, Tools.generateUniqueKey(),
collaboratorKey);
   }
}
```

3.6.2 The Administrator

- **Administrator**

```
package com.greatfree.testing.admin;

import java.util.Scanner;

/*
 * The program is controlled by the administrator to manage the distributed system manually.
11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class Administrator
{
  public static void main(String[] args)
  {
    // Initialize the option, which represents the commands. 11/27/2014, Bing Li
    int option = AdminConfig.NO_OPTION;
    // Initialize a scanner to wait for the administrator's commands. 11/27/2014, Bing Li
    Scanner in = new Scanner(System.in);
    String optionStr;

    // Initialize the client pool. 11/27/2014, Bing Li
    ClientPool.ADMIN().init();
    // Initialize the eventer. 11/27/2014, Bing Li
    AdminEventer.CONSOLE().init();

    // Check whether the administrator selects the end command. 11/27/2014, Bing Li
    while (option != AdminConfig.END)
    {
      // Print the console menu. 11/27/2014, Bing Li
      printMenu();
      // Wait for the administrator's input from the console. 11/27/2014, Bing Li
      optionStr = in.nextLine();
      try
      {
        // Convert the string to the integer. 11/27/2014, Bing Li
        option = Integer.parseInt(optionStr);
        // Print the option. 11/27/2014, Bing Li
        System.out.println("Your choice is: " + option);
        // Notify the coordinator. 11/27/2014, Bing Li
        notifyServer(option);
      }
      catch (NumberFormatException e)
      {
        option = AdminConfig.NO_OPTION;
        System.out.println(Menu.WRONG_OPTION);
      }
    }

    // Dispose the client pool. 11/27/2014, Bing Li
    ClientPool.ADMIN().dispose();
    // Dispose the eventer. 11/27/2014, Bing Li
    AdminEventer.CONSOLE().dispose();
  }

  /*
   * Print the console menu for the administrator. 11/27/2014, Bing Li
   */
  private static void printMenu()
  {
    System.out.println(Menu.MENU_HEAD);
    System.out.println(Menu.STOP_CRAWLSERVER);
    System.out.println(Menu.STOP_MSERVER);
    System.out.println(Menu.STOP_CSERVER);
    System.out.println(Menu.END);
```

```
        System.out.println(Menu.MENU_TAIL);
        System.out.println(Menu.INPUT_PROMPT);
    }

    /*
     * Notify the coordinator to manage the distributed system. 11/27/2014, Bing Li
     */
    private static void notifyServer(int option)
    {
        // Check the option of the administrator. 11/27/2014, Bing Li
        switch (option)
        {
            // Shutdown the coordinator. 11/27/2014, Bing Li
            case AdminConfig.STOP_CSERVER:
                System.out.println(Menu.NOTIFYING_SHUTDOWN_COORDINATOR);
                // Notify the coordinator to shutdown the coordinator. 11/27/2014, Bing Li
                AdminEventer.CONSOLE().notifyShutdownCoordinator();
                break;

            // Shutdown the memory servers. 11/27/2014, Bing Li
            case AdminConfig.STOP_MSERVER:
                System.out.println(Menu.NOTIFYING_SHUTDOWN_MEMSERVER);
                // Notify the coordinator to shutdown the cluster of memory servers. 11/27/2014, Bing Li
                AdminEventer.CONSOLE().notifyShutdownMemoryServer();
                break;

            // Shutdown the crawling servers. 11/27/2014, Bing Li
            case AdminConfig.STOP_CRAWLSERVER:
                System.out.println(Menu.NOTIFYING_SHUTDOWN_CRAWLSERVER);
                // Notify the coordinator to shutdown the cluster of crawling servers. 11/27/2014, Bing Li
                AdminEventer.CONSOLE().notifyShutdownCrawlServer();
                break;
        }
    }
}
```

- **ClientPool**

```
package com.greatfree.testing.admin;

import java.io.IOException;

import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.data.ClientConfig;

/*
 * The pool is responsible for creating and managing instance of FreeClient to achieve the goal of using as small number of instances of FreeClient to send messages to the coordinator in a high performance. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class ClientPool
{
  // An instance of FreeClientPool is defined to interact with the coordinator. 11/27/2014, Bing Li
  private FreeClientPool pool;

  private ClientPool()
  {
    this.pool = new FreeClientPool(ClientConfig.CLIENT_POOL_SIZE);
  }

  /*
   * A singleton definition. 11/27/2014, Bing Li
   */
  private static ClientPool instance = new ClientPool();

  public static ClientPool ADMIN()
  {
    if (instance == null)
    {
      instance = new ClientPool();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the free client pool. 11/27/2014, Bing Li
   */
  public void dispose()
  {
    try
    {
      this.pool.dispose();
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
  }

  /*
   * Initialize the client pool. The method is called when the crawler process is started. 11/27/2014, Bing Li
   */
  public void init()
  {
    // Initialize the client pool. 11/27/2014, Bing Li
```

```
        this.pool = new FreeClientPool(ClientConfig.CLIENT_POOL_SIZE);
        // Set idle checking for the client pool. 11/27/2014, Bing Li
        this.pool.setIdleChecker(ClientConfig.CLIENT_IDLE_CHECK_DELAY,
ClientConfig.CLIENT_IDLE_CHECK_PERIOD, ClientConfig.CLIENT_MAX_IDLE_TIME);
    }

    /*
     * Expose the client pool. 11/27/2014, Bing Li
     */
    public FreeClientPool getPool()
    {
        return this.pool;
    }
}
```

- **AdminEventer**

```
package com.greatfree.testing.admin;

import com.greatfree.concurrency.ThreadPool;
import com.greatfree.remote.AsyncRemoteEventer;
import com.greatfree.testing.data.ClientConfig;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.ShutdownCoordinatorServerNotification;
import com.greatfree.testing.message.ShutdownCrawlServerNotification;
import com.greatfree.testing.message.ShutdownMemoryServerNotification;

/*
 * This is an eventer that sends notifications to the coordinator in a synchronous or asynchronous
manner. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class AdminEventer
{
   // The notification of ShutdownCrawlServerNotification is sent to the coordinator in an asynchronous
manner. 11/27/2014, Bing Li
   private AsyncRemoteEventer<ShutdownCrawlServerNotification>
shutdownCrawlServerNotificationEventer;
   // The notification of ShutdownMemoryServerNotification is sent to the coordinator in an asynchronous
manner. 11/27/2014, Bing Li
   private AsyncRemoteEventer<ShutdownMemoryServerNotification>
shutdownMemServerNotificationEventer;
   // The notification of ShutdownCoordinatorServerNotification is sent to the coordinator in an
asynchronous manner. 11/27/2014, Bing Li
   private AsyncRemoteEventer<ShutdownCoordinatorServerNotification>
shutdownCoordinatorNotificationEventer;
   // The thread pool that starts up the asynchronous eventer. 11/27/2014, Bing Li
   private ThreadPool pool;

   private AdminEventer()
   {
   }

   /*
    * Initialize a singleton. 11/27/2014, Bing Li
    */
   private static AdminEventer instance = new AdminEventer();

   public static AdminEventer CONSOLE()
   {
      if (instance == null)
      {
         instance = new AdminEventer();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Dispose the eventer. 11/27/2014, Bing Li
    */
   public void dispose()
   {
      // Shutdown the eventer to send the notification of ShutdownCrawlServerNotification in an
asynchronous manner. 11/27/2014, Bing Li
      this.shutdownCrawlServerNotificationEventer.dispose();
      // Shutdown the eventer to send the notification of ShutdownMemoryServerNotification in an
```

```
asynchronous manner. 11/27/2014, Bing Li
    this.shutdownMemServerNotificationEventer.dispose();
    // Shutdown the eventer to send the notification of ShutdownCoordinatorServerNotification in an
asynchronous manner. 11/27/2014, Bing Li
    this.shutdownCoordinatorNotificationEventer.dispose();
    // Shutdown the thread pool. 11/27/2014, Bing Li
    this.pool.shutdown();
  }

  /*
   * Initialize the eventer. 11/27/2014, Bing Li
   */
  public void init()
  {
    this.pool = new ThreadPool(ClientConfig.EVENTER_THREAD_POOL_SIZE,
ClientConfig.EVENTER_THREAD_POOL_ALIVE_TIME);

    // Initialize the shutting down crawling server notification eventer. 11/27/2014, Bing Li
    this.shutdownCrawlServerNotificationEventer = new
AsyncRemoteEventer<ShutdownCrawlServerNotification>(ClientPool.ADMIN().getPool(), this.pool,
ClientConfig.EVENT_QUEUE_SIZE, ClientConfig.EVENTER_SIZE,
ClientConfig.EVENTING_WAIT_TIME, ClientConfig.EVENTER_WAIT_TIME);
    // Set the idle checking for the shutdown notification eventer. 11/27/2014, Bing Li

  this.shutdownCrawlServerNotificationEventer.setIdleChecker(ClientConfig.EVENT_IDLE_CHECK_D
ELAY, ClientConfig.EVENT_IDLE_CHECK_PERIOD);
    // Start up the shutdown notification eventer. 11/27/2014, Bing Li
    this.pool.execute(this.shutdownCrawlServerNotificationEventer);

    // Initialize the shutting down memory server notification eventer. 11/27/2014, Bing Li
    this.shutdownMemServerNotificationEventer = new
AsyncRemoteEventer<ShutdownMemoryServerNotification>(ClientPool.ADMIN().getPool(), this.pool,
ClientConfig.EVENT_QUEUE_SIZE, ClientConfig.EVENTER_SIZE,
ClientConfig.EVENTING_WAIT_TIME, ClientConfig.EVENTER_WAIT_TIME);
    // Set the idle checking for the shutdown notification eventer. 11/27/2014, Bing Li

  this.shutdownMemServerNotificationEventer.setIdleChecker(ClientConfig.EVENT_IDLE_CHECK_DE
LAY, ClientConfig.EVENT_IDLE_CHECK_PERIOD);
    // Start up the shutdown notification eventer. 11/27/2014, Bing Li
    this.pool.execute(this.shutdownMemServerNotificationEventer);

    // Initialize the shutting down coordinator server notification eventer. 11/27/2014, Bing Li
    this.shutdownCoordinatorNotificationEventer = new
AsyncRemoteEventer<ShutdownCoordinatorServerNotification>(ClientPool.ADMIN().getPool(),
this.pool, ClientConfig.EVENT_QUEUE_SIZE, ClientConfig.EVENTER_SIZE,
ClientConfig.EVENTING_WAIT_TIME, ClientConfig.EVENTER_WAIT_TIME);
    // Set the idle checking for the shutdown notification eventer. 11/27/2014, Bing Li

  this.shutdownCoordinatorNotificationEventer.setIdleChecker(ClientConfig.EVENT_IDLE_CHECK_DE
LAY, ClientConfig.EVENT_IDLE_CHECK_PERIOD);
    // Start up the shutdown notification eventer. 11/27/2014, Bing Li
    this.pool.execute(this.shutdownCoordinatorNotificationEventer);
  }

  /*
   * Notify the coordinator to shut down the cluster of crawler servers. 11/27/2014, Bing Li
   */
  public void notifyShutdownCrawlServer()
  {
    this.shutdownCrawlServerNotificationEventer.notify(ServerConfig.COORDINATOR_ADDRESS,
ServerConfig.COORDINATOR_PORT_FOR_ADMIN, new ShutdownCrawlServerNotification());
  }

  /*
   * Notify the coordinator to shut down the cluster of memory servers. 11/27/2014, Bing Li
   */
  public void notifyShutdownMemoryServer()
  {
```

```
    this.shutdownMemServerNotificationEventer.notify(ServerConfig.COORDINATOR_ADDRESS,
ServerConfig.COORDINATOR_PORT_FOR_ADMIN, new ShutdownMemoryServerNotification());
  }

  /*
   * Notify the coordinator to shut down the coordinator. 11/27/2014, Bing Li
   */
  public void notifyShutdownCoordinator()
  {
    this.shutdownCoordinatorNotificationEventer.notify(ServerConfig.COORDINATOR_ADDRESS,
ServerConfig.COORDINATOR_PORT_FOR_ADMIN, new ShutdownCoordinatorServerNotification());
  }
}
```

- **AdminConfig**

```
package com.greatfree.testing.admin;

/*
 * The configuration contains the constants of the administration program. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class AdminConfig
{
   public final static int NO_OPTION = -1;
   public final static int STOP_CRAWLSERVER = 1;
   public final static int STOP_MSERVER = 2;
   public final static int STOP_CSERVER = 3;
   public final static int END = 0;
   public final static int MANSERVER_CLIENT_POOL_SIZE = 50;
}
```

- **Menu**

```
package com.greatfree.testing.admin;

/*
 * This is a menu for the administrator to display a control interface. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class Menu
{
   public final static String TAB = "  ";
   public final static String MENU_HEAD = "\n========== Menu Head ==========";
   public final static String STOP_CRAWLSERVER = Menu.TAB + "1) Stop CrawlServer";
   public final static String STOP_MSERVER = Menu.TAB + "2) Stop MServer";
   public final static String STOP_CSERVER = Menu.TAB + "3) Stop CServer";
   public final static String END = Menu.TAB + "0) End";
   public final static String MENU_TAIL = "========== Menu Tail ==========\n";
   public final static String INPUT_PROMPT = "Input an option:";

   public final static String WRONG_OPTION = "Wrong option!";
   public final static String NOTIFYING_SHUTDOWN_COORDINATOR = "Notifying shutdown
CServer";
   public final static String NOTIFYING_SHUTDOWN_MEMSERVER = "Notifying shutdown
MServer";
   public final static String NOTIFYING_SHUTDOWN_CRAWLSERVER = "Notifying shutdown
CrawlServer";
}
```

3.6.3 The Cluster of Web Crawlers

- **StartCrawler**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.testing.data.ServerConfig;
import com.greatfree.util.TerminateSignal;

/*
 * This is the unique entry and exit for the crawling server. 11/28/2014, Bing Li
 */

// Created: 11/11/2014, Bing Li
public class StartCrawler
{
  public static void main(String[] args)
  {
    // Start the crawling server. 11/28/2014, Bing Li
    CrawlServer.CRAWL().start(ServerConfig.CRAWL_SERVER_PORT);
    // Detect whether the process is shutdown. 11/28/2014, Bing Li
    while (!TerminateSignal.SIGNAL().isTerminated())
    {
      try
      {
        // Sleep for some time if the process is not shutdown. 11/28/2014, Bing Li
        Thread.sleep(ServerConfig.TERMINATE_SLEEP);
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **CrawlServer**

```
package com.greatfree.testing.crawlserver;

import java.io.IOException;
import java.net.ServerSocket;
import java.util.ArrayList;
import java.util.List;
import java.util.Timer;

import com.greatfree.concurrency.Runner;
import com.greatfree.concurrency.Threader;
import com.greatfree.testing.data.ClientConfig;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.util.NullObject;
import com.greatfree.util.TerminateSignal;
import com.greatfree.util.UtilConfig;

/*
 * This is an example to demonstrate the distributed patterns introduced in the tutorial. 11/11/2014, Bing Li
 *
 *  1) The code is used to crawl a large number of URLs. In this case, the count of URLs exceeds 10,000.
 *
 *    2) Because of the heavy burden to crawl, it is required to distribute the crawling tasks to multiple crawlers to balance the load.
 *
 *    3) The crawling results must be submitted to the coordinator for distributed caching and summarizing.
 *
 *    4) This server is one of the components to receive and forward crawling tasks, crawl URLs and submit the results.
 *
 */

// Created: 11/11/2014, Bing Li
public class CrawlServer
{
   // The socket that waits for connections from other crawlers and the coordinator. In this case, the crawlers and the coordinator form a cluster such that it is possible to crawl the Web in a large scale. 11/24/2014, Bing Li
   private ServerSocket serverSocket;
   // The port to wait for connections. 11/24/2014, Bing Li
   private int serverPort;

   // The listener list contains all of the listeners to wait for connections. 11/24/2014, Bing Li
   private List<Runner<CrawlingListener, CrawlingListenerDisposer>> listeners;

   // The threader contains the instance of CrawlConsumer, which is a producer/consumer pattern. The consumer schedules and crawls the URLs until the threader is stopped. 11/24/2014, Bing Li
   private Threader<CrawlConsumer, CrawlConsumerDisposer> crawlConsumerThreader;

   // Declare a timer to control the crawling state checking periodically. 11/27/2014, Bing Li
   private Timer crawlCheckingTimer;
   // Declare The crawling state checker. 11/27/2014, Bing Li
   private CrawlingStateChecker stateChecker;

   private CrawlServer()
   {
   }

   /*
    * A singleton implementation since this is the unique entry and exit of the crawler. 11/24/2014, Bing Li
    */
   private static CrawlServer instance = new CrawlServer();
```

```
  public static CrawlServer CRAWL()
  {
    if (instance == null)
    {
      instance = new CrawlServer();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Start up the crawler. 11/24/2014, Bing Li
   */
  public void start(int serverPort)
  {
    // On JD7, the sorting algorithm is replaced with TimSort rather than MargeSort. To run correctly, it is
necessary to use the old one. the following line sets that up. 11/23/2014, Bing Li
    System.setProperty(UtilConfig.MERGE_SORT, UtilConfig.TRUE);

    // Assign the crawler port. 11/24/2014, Bing Li
    this.serverPort = serverPort;
    try
    {
      // Initialize the socket of the crawler. 11/24/2014, Bing Li
      this.serverSocket = new ServerSocket(this.serverPort);
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }

    // Initialize a list to contain the listeners. 11/24/2014, Bing Li
    this.listeners = new ArrayList<Runner<CrawlingListener, CrawlingListenerDisposer>>();
    // Initialize a disposer that collects the listeners. 11/24/2014, Bing Li
    CrawlingListenerDisposer disposer = new CrawlingListenerDisposer();
    // The runner contains the listener and listener disposer to start the listeners concurrently.
11/24/2014, Bing Li
    Runner<CrawlingListener, CrawlingListenerDisposer> runner;
    // Initialize and start a certain number of listeners concurrently. 11/24/2014, Bing Li
    for (int i = 0; i < ServerConfig.MAX_CLIENT_LISTEN_THREAD_COUNT; i++)
    {
      // Initialize the runner that contains the listener and its disposer. 11/24/2014, Bing Li
      runner = new Runner<CrawlingListener, CrawlingListenerDisposer>(new
CrawlingListener(this.serverSocket), disposer, true);
      // Put the runner into a list for management. 11/24/2014, Bing Li
      this.listeners.add(runner);
      // Start up the runner. 11/24/2014, Bing Li
      runner.start();
    }

    // Initialize the crawling server IO registry. 11/24/2014, Bing Li
    CrawlIORegistry.REGISTRY().init();
    // Initialize the client pool that is used to connect the coordinator. 11/24/2014, Bing Li
    ClientPool.CRAWL().init();
    // Initialize the sub client pool that is used to connect the children of the crawler. 11/27/2014, Bing Li
    SubClientPool.CRAWL().init();
    // Initialize the crawling eventer that sends notifications to the coordinator. For example, the crawled
links are sent to the coordinator asynchronously by the eventer. 11/24/2014, Bing Li
    CrawlEventer.NOTIFY().init(ServerConfig.COORDINATOR_ADDRESS,
ServerConfig.COORDINATOR_PORT_FOR_CRAWLER);

    // Initialize the message producer which dispatches received requests and notifications from the
coordinator. 11/24/2014, Bing Li
    CrawlMessageProducer.CRAWL().Init();
```

```
    // Initialize the crawling scheduler. 11/24/2014, Bing Li
    CrawlScheduler.CRAWL().init();

    // Initialize the threader that contains the crawling consumer and its disposer. 11/24/2014, Bing Li
    this.crawlConsumerThreader = new Threader<CrawlConsumer, CrawlConsumerDisposer>(new
CrawlConsumer(new CrawlEater(), CrawlConfig.CRAWL_SCHEDULER_WAIT_TIME), new
CrawlConsumerDisposer());

    // Initialize th crawling multicastor. 12/01/2014, Bing Li
    CrawlerMulticastor.CRAWLER().init();

    try
    {
      // Notify the coordinator that the crawler is online. 11/24/2014, Bing Li
      CrawlEventer.NOTIFY().notifyOnline();
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
  }

  /*
   * Start the crawling. It is actually invoked remotely by the coordinator when a
StartCrawlMultiNotification is received. 11/24/2014, Bing Li
   */
  public void startCrawl()
  {
    // Start the crawling consumer. 11/24/2014, Bing Li
    this.crawlConsumerThreader.start();
    // Produce a null object, which is not processed by the consumer. However, an infinite loop to
schedule URLs is started for that. 11/24/2014, Bing Li
    this.crawlConsumerThreader.getFunction().produce(new NullObject());

    // Initialize the crawling state checker. 11/27/2014, Bing Li
    this.stateChecker = new CrawlingStateChecker();
    // Initialize the timer. 11/27/2014, Bing Li
    this.crawlCheckingTimer = new Timer();
    // Schedule the crawling state checker. 11/27/2014, Bing Li
    this.crawlCheckingTimer.schedule(this.stateChecker, 0,
CrawlConfig.CRAWLING_STATE_CHECK_PERIOD);
  }

  /*
   * Stop the crawler. 11/24/2014, Bing Li
   */
  public void stop() throws InterruptedException, IOException
  {
    // Set the terminating signal. The long time running task, scheduling the crawling URLs, needs to be
interrupted when the signal is set. 11/24/2014, Bing Li
    TerminateSignal.SIGNAL().setTerminated();

    // Stop each listener one by one. 11/24/2014, Bing Li
    for (Runner<CrawlingListener, CrawlingListenerDisposer> runner : this.listeners)
    {
      runner.stop(ClientConfig.TIME_TO_WAIT_FOR_THREAD_TO_DIE);
    }

    // Close the socket of the crawling server. 11/24/2014, Bing Li
    this.serverSocket.close();

    try
    {
      // Stop the crawling consumer. 11/24/2014, Bing Li
```

```
        this.crawlConsumerThreader.stop(ClientConfig.TIME_TO_WAIT_FOR_THREAD_TO_DIE);
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }

      // Check whether the crawling state checker is valid. It might not be initialized if the crawling is not
started. 11/27/2014, Bing Li
      if (this.stateChecker != null)
      {
        // End the checking. 11/27/2014, Bing Li
        this.stateChecker.cancel();
        // End the timer. 11/27/2014, Bing Li
        this.crawlCheckingTimer.cancel();
      }

      // Unregister the crawling eventer. 11/24/2014, Bing Li
      CrawlEventer.NOTIFY().unregister();
      // Dispose the crawling eventer. 11/24/2014, Bing Li
      CrawlEventer.NOTIFY().dispose();

      // Dispose the message producer. 11/24/2014, Bing Li
      CrawlMessageProducer.CRAWL().dispose();
      // Shutdown the crawling server IOs. 11/24/2014, Bing Li
      CrawlIORegistry.REGISTRY().dispose();
      // Dispose the client pool. 11/24/2014, Bing Li
      ClientPool.CRAWL().dispose();
      // Dispose the sub client pool/ 11/27/2014, Bing Li
      SubClientPool.CRAWL().dispose();
      // Dispose the crawling multicastor. 12/01/2014, Bing L
      CrawlerMulticastor.CRAWLER().dispose();
      // Dispose the crawling scheduler. 11/24/2014, Bing Li
      CrawlScheduler.CRAWL().dispose();
    }
}
```

- **CrawlingListener**

```
package com.greatfree.testing.crawlserver;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

import com.greatfree.remote.ServerListener;
import com.greatfree.testing.data.ServerConfig;

/*
 * The class is a server listener that waits for the coordinator to distribute crawling tasks. The design is the same as the general server listener. 07/30/2014, Bing Li
 */

// Created: 11/11/2014, Bing Li
public class CrawlingListener extends ServerListener implements Runnable
{
  /*
   * Initialize the listener. 11/23/2014, Bing Li
   */
  public CrawlingListener(ServerSocket serverSocket)
  {
    super(serverSocket, CrawlConfig.CRAWLING_TASK_LISTENER_THREAD_POOL_SIZE, CrawlConfig.CRAWLING_TASK_LISTENER_THREAD_ALIVE_TIME);
  }

  /*
   * Waiting for connection concurrently. The connection request is invoked by a coordinator, which sends crawling tasks and other management notifications. 11/23/2014, Bing Li
   */
  @Override
  public void run()
  {
    Socket clientSocket;
    CrawlServerIO serverIO;

    // Detect whether the listener is shutdown. If not, it must be running all the time to wait for potential connections from clients. 11/23/2014, Bing Li
    while (!super.isShutdown())
    {
      try
      {
        // Wait and accept a connecting from a possible client residing on the coordinator. 11/23/2014, Bing Li
        clientSocket = super.accept();
        // Check whether the connected server IOs exceed the upper limit. 11/23/2014, Bing Li
        if (CrawlIORegistry.REGISTRY().getIOCount() >= ServerConfig.MAX_SERVER_IO_COUNT)
        {
          try
          {
            // If the upper limit is reached, the listener has to wait until an existing server IO is disposed. 11/23/2014, Bing Li
            super.holdOn();
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
        }
        // If the upper limit of IOs is not reached, a server IO is initialized. A common Collaborator and the socket are the initial parameters. The shared common collaborator guarantees all of the server IOs from a certain client could notify with each other with the same lock. Then, the upper limit of server IOs is under the control. 11/23/2014, Bing Li
        serverIO = new CrawlServerIO(clientSocket, super.getCollaborator());
```

```
            // Add the new created server IO into the registry for further management. 11/23/2014, Bing Li
            CrawlIORegistry.REGISTRY().addIO(serverIO);
            // Execute the new created server IO concurrently to respond the client requests in an
asynchronous manner. 11/23/2014, Bing Li
            super.execute(serverIO);
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
        }
      }
    }
```

- **CrawlingListenerDisposer**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.reuse.RunDisposable;

/*
 * The class is responsible for disposing the instance of CrawlingListener by invoking its method of shutdown(). It works with the thread container, Threader or Runner. 11/23/2014, Bing Li
 */

// Created: 11/11/2014, Bing Li
public class CrawlingListenerDisposer implements RunDisposable<CrawlingListener>
{
	/*
	 * Dispose the instance of CrawlingListener. 11/23/2014, Bing Li
	 */
	@Override
	public void dispose(CrawlingListener r)
	{
		r.shutdown();
	}

	/*
	 * Dispose the instance of CrawlingListener. The method does not make sense to CrawlingListener. Just leave it here for the requirement of the interface, RunDisposable<CrawlingListener>. 11/23/2014, Bing Li
	 */
	@Override
	public void dispose(CrawlingListener r, long time)
	{
		r.shutdown();
	}
}
```

- **CrawlServerIO**

```
package com.greatfree.testing.crawlserver;

import java.io.IOException;
import java.net.Socket;
import java.net.SocketException;

import com.greatfree.concurrency.Collaborator;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.remote.ServerIO;

/*
 * The class receives requests/notifications from the coordinator to accomplish the task of crawling. 11/23/2014, Bing Li
 */

// Created: 11/11/2014, Bing Li
public class CrawlServerIO extends ServerIO
{
  /*
   * Initialize the server IO. The socket is the connection between the coordinator and the crawling server. The collaborator is shared with other IOs to control the count of ServerIOs instances. 11/23/2014, Bing Li
   */
  public CrawlServerIO(Socket clientSocket, Collaborator collaborator)
  {
    super(clientSocket, collaborator);
  }

  /*
   * Concurrently respond the coordinator's requests/notification. 11/23/2014, Bing Li
   */
  public void run()
  {
    ServerMessage message;
    while (!super.isShutdown())
    {
      try
      {
        // Wait and read messages from the coordinator. 11/23/2014, Bing Li
        message = (ServerMessage)super.read();
        // Convert the received message to OutMessageStream and put it into the relevant dispatcher for concurrent processing. 11/23/2014, Bing Li
        CrawlMessageProducer.CRAWL().produceMessage(new OutMessageStream<ServerMessage>(super.getOutStream(), super.getLock(), message));
      }
      catch (SocketException e)
      {
        e.printStackTrace();
      }
      catch (IOException e)
      {
        e.printStackTrace();
      }
      catch (ClassNotFoundException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **CrawlIORegistry**

```
package com.greatfree.testing.crawlserver;

import java.io.IOException;
import java.util.Set;

import com.greatfree.remote.ServerIORegistry;

/*
 * The class keeps all of CrawlServerIOs. This is a singleton wrapper of ServerIORegistry. 11/11/2014,
Bing Li
 */

// Created: 11/11/2014, Bing Li
public class CrawlIORegistry
{
  // Declare an instance of ServerIORegistry for CrawlServerIOs. 11/11/2014, Bing Li
  private ServerIORegistry<CrawlServerIO> registry;

  /*
   * Initializing ... 11/11/2014, Bing Li
   */
  private CrawlIORegistry()
  {
  }

  private static CrawlIORegistry instance = new CrawlIORegistry();

  public static CrawlIORegistry REGISTRY()
  {
    if (instance == null)
    {
      instance = new CrawlIORegistry();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the registry. 11/11/2014, Bing Li
   */
  public void dispose() throws IOException
  {
    this.registry.removeAllIOs();
  }

  /*
   * Initialize the registry. 11/11/2014, Bing Li
   */
  public void init()
  {
    this.registry = new ServerIORegistry<CrawlServerIO>();
  }

  /*
   * Add a new instance of CrawlServerIO to the registry. 11/11/2014, Bing Li
   */
  public void addIO(CrawlServerIO io)
  {
    this.registry.addIO(io);
  }
```

```
  /*
   * Get all of the IPs of the connected clients from the corresponding CrawlServerIOs. 11/11/2014, Bing
Li
   */
  public Set<String> getIPs()
  {
    return this.registry.getIPs();
  }

  /*
   * Get the count of the registered CrawlServerIOs. 11/11/2014, Bing Li
   */
  public int getIOCount()
  {
    return this.registry.getIOCount();
  }

  /*
   * Remove or unregister a CrawlServerIO. It is executed when a client is down or the connection gets
something wrong. 11/11/2014, Bing Li
   */
  public void removeIO(CrawlServerIO io) throws IOException
  {
    this.registry.removeIO(io);
  }

  /*
   * Remove or unregister all of the registered CrawlServerIOs. It is executed when the server process is
shutdown. 11/11/2014, Bing Li
   */
  public void removeAllIOs() throws IOException
  {
    this.registry.removeAllIOs();
  }
}
```

- **CrawlMessageProducer**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.concurrency.MessageProducer;
import com.greatfree.concurrency.Threader;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.testing.data.ServerConfig;

/*
 * The class is a singleton to enclose the instances of MessageProducer. Each of the enclosed message producers serves for one particular client that connects to a respective port on the crawling server. Usually, each port aims to provide one particular service. 11/23/2014, Bing Li
 *
 * The class is a wrapper that encloses all of the asynchronous message producers. It is responsible for assigning received messages to the corresponding producer in an asynchronous way. 11/23/2014, Bing Li
 */

// Created: 11/23/2014, Bing Li
public class CrawlMessageProducer
{
  private Threader<MessageProducer<CrawlDispatcher>, CrawlMessageProducerDisposer> producerThreader;

  private CrawlMessageProducer()
  {
  }

  /*
   * The class is required to be a singleton since it is nonsense to initiate it for the producers are unique. 11/23/2014, Bing Li
   */
  private static CrawlMessageProducer instance = new CrawlMessageProducer();

  public static CrawlMessageProducer CRAWL()
  {
    if (instance == null)
    {
      instance = new CrawlMessageProducer();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the producers when the process of the server is shutdown. 11/23/2014, Bing Li
   */
  public void dispose() throws InterruptedException
  {
    this.producerThreader.stop();
  }

  /*
   * Initialize the message producers. It is invoked when the connection modules of the server is started since clients can send requests or notifications only after it is started. 08/22/2014, Bing Li
   */
  public void Init()
  {
    // Initialize the crawling message producer. A threader is associated with the message producer such that the producer is able to work in a concurrent way. 11/23/2014, Bing Li
    this.producerThreader = new Threader<MessageProducer<CrawlDispatcher>,
```

```
CrawlMessageProducerDisposer>(new MessageProducer<CrawlDispatcher>(new
CrawlDispatcher(ServerConfig.DISPATCHER_POOL_SIZE,
ServerConfig.DISPATCHER_POOL_THREAD_POOL_ALIVE_TIME)), new
CrawlMessageProducerDisposer());
      // Start the associated thread for the crawling message producer. 11/23/2014, Bing Li
      this.producerThreader.start();
    }

    /*
     * Assign messages, requests or notifications, to the bound crawling message dispatcher such that
they can be responded or dealt with. 11/23/2014, Bing Li
     */
    public void produceMessage(OutMessageStream<ServerMessage> message)
    {
      this.producerThreader.getFunction().produce(message);
    }
}
```

- **CrawlMessageProducerDisposer**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.concurrency.MessageProducer;
import com.greatfree.reuse.ThreadDisposable;

/*
 * The class is responsible for disposing the message producer of the server. 11/23/2014, Bing Li
 */

// Created: 11/23/2014, Bing Li
public class CrawlMessageProducerDisposer implements
ThreadDisposable<MessageProducer<CrawlDispatcher>>
{
   /*
    * Dispose the message producer. 11/23/2014, Bing Li
    */
   @Override
   public void dispose(MessageProducer<CrawlDispatcher> r)
   {
      r.dispose();
   }

   /*
    * The method does not make sense to the class of MessageProducer. Just leave it here. 11/23/2014,
Bing Li
    */
   @Override
   public void dispose(MessageProducer<CrawlDispatcher> r, long time)
   {
      r.dispose();
   }
}
```

- **CrawlDispatcher**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.concurrency.BoundNotificationDispatcher;
import com.greatfree.concurrency.NotificationDispatcher;
import com.greatfree.concurrency.ServerMessageDispatcher;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.reuse.MulticastMessageDisposer;
import com.greatfree.testing.data.ClientConfig;
import com.greatfree.testing.message.CrawlLoadNotification;
import com.greatfree.testing.message.MessageType;
import com.greatfree.testing.message.NodeKeyNotification;
import com.greatfree.testing.message.StartCrawlMultiNotification;
import com.greatfree.testing.message.StopCrawlMultiNotification;

/*
 * This is an implementation of ServerMessageDispatcher. It contains the concurrency mechanism to respond the coordinator's requests and notifications for the crawling server. 11/23/2014, Bing Li
 */

// Created: 11/23/2014, Bing Li
public class CrawlDispatcher extends ServerMessageDispatcher<ServerMessage>
{
   // Declare a instance of notification dispatcher to deal with received the notification that contains the node key. 11/25/2014, Bing Li
   private NotificationDispatcher<NodeKeyNotification, RegisterThread, RegisterThreadCreator> nodeKeyNotificationDispatcher;

   // The disposer is the binder that synchronizes the two bound notification dispatchers, startCrawlNotificationDispatcher and multicastStartCrawlNotificationDispatcher. After both of them finish their respective tasks concurrently, it disposes the notification of StartCrawlMultiNotification finally. 11/27/2014, Bing Li
   private MulticastMessageDisposer<StartCrawlMultiNotification> startCrawlNotificationDisposer;
   // The dispatcher to start the crawling after getting the notification of StartCrawlMultiNotification. It must be synchronized by the binder, startCrawlNotificationDisposer. So it is implemented as a bound notification dispatcher. 11/27/2014, Bing Li
   private BoundNotificationDispatcher<StartCrawlMultiNotification, MulticastMessageDisposer<StartCrawlMultiNotification>, StartCrawlThread, StartCrawlThreadCreator> startCrawlNotificationDispatcher;
   // The dispatcher to disseminate the notification of StartCrawlMultiNotification to children nodes. It must be synchronized by the binder, startCrawlNotificationDisposer. So it is implemented as a bound notification dispatcher. 11/27/2014, Bing Li
   private BoundNotificationDispatcher<StartCrawlMultiNotification, MulticastMessageDisposer<StartCrawlMultiNotification>, MulticastStartCrawlNotificationThread, MulticastStartCrawlNotificationThreadCreator> multicastStartCrawlNotificationDispatcher;

   // Declare a instance of notification dispatcher to deal with received the notification that contains the crawling workload. 11/28/2014, Bing Li
   private NotificationDispatcher<CrawlLoadNotification, AssignURLLoadThread, AssignURLLoadThreadCreator> crawlLoadNotificationDispatcher;

   // A instance of notification dispatcher to deal with received the stop crawling notification. 11/27/2014, Bing Li
   private NotificationDispatcher<StopCrawlMultiNotification, StopCrawlThread, StopCrawlThreadCreator> stopCrawlNotificationDispatcher;

   /*
    * Initialize the dispatcher. 11/25/2014, Bing Li
    */
   public CrawlDispatcher(int corePoolSize, long keepAliveTime)
   {
      super(corePoolSize, keepAliveTime);

      // Initialize the notification dispatcher for the notification, NodeKeyNotification. 11/25/2014, Bing Li
```

```
    this.nodeKeyNotificationDispatcher = new NotificationDispatcher<NodeKeyNotification,
RegisterThread, RegisterThreadCreator>(ClientConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
ClientConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new RegisterThreadCreator(),
ClientConfig.MAX_NOTIFICATION_TASK_SIZE, ClientConfig.MAX_NOTIFICATION_THREAD_SIZE,
ClientConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
    // Set the idle checking of the nodeKeyNotificationDispatcher. 11/25/2014, Bing Li

  this.nodeKeyNotificationDispatcher.setIdleChecker(ClientConfig.NOTIFICATION_DISPATCHER_IDL
E_CHECK_DELAY, ClientConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
    // Start the nodeKeyNotificationDispatcher. 11/25/2014, Bing Li
    super.execute(this.nodeKeyNotificationDispatcher);

    // Initialize the disposer for the notification of StartCrawlMultiNotification, which works as a binder.
11/27/2014, Bing Li
    this.startCrawlNotificationDisposer = new
MulticastMessageDisposer<StartCrawlMultiNotification>();
    // Initialize the bound notification dispatcher for the notification, StartCrawlMultiNotification, to start
crawling. 11/27/2014, Bing Li
    this.startCrawlNotificationDispatcher = new
BoundNotificationDispatcher<StartCrawlMultiNotification,
MulticastMessageDisposer<StartCrawlMultiNotification>, StartCrawlThread,
StartCrawlThreadCreator>(ClientConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
ClientConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME,
this.startCrawlNotificationDisposer, new StartCrawlThreadCreator(),
ClientConfig.MAX_NOTIFICATION_TASK_SIZE, ClientConfig.MAX_NOTIFICATION_THREAD_SIZE,
ClientConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
    // Set the idle checking of the startCrawlNotificationDispatcher. 11/27/2014, Bing Li

  this.startCrawlNotificationDispatcher.setIdleChecker(ClientConfig.NOTIFICATION_DISPATCHER_ID
LE_CHECK_DELAY, ClientConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
    // Start the startCrawlNotificationDispatcher. 11/27/2014, Bing Li
    super.execute(this.startCrawlNotificationDispatcher);

    // Initialize the bound notification dispatcher for the notification, StartCrawlMultiNotification, to
disseminate the notification to children crawlers. 11/27/2014, Bing Li
    this.multicastStartCrawlNotificationDispatcher = new
BoundNotificationDispatcher<StartCrawlMultiNotification,
MulticastMessageDisposer<StartCrawlMultiNotification>, MulticastStartCrawlNotificationThread,
MulticastStartCrawlNotificationThreadCreator>(ClientConfig.NOTIFICATION_DISPATCHER_POOL_SI
ZE, ClientConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME,
this.startCrawlNotificationDisposer, new MulticastStartCrawlNotificationThreadCreator(),
ClientConfig.MAX_NOTIFICATION_TASK_SIZE, ClientConfig.MAX_NOTIFICATION_THREAD_SIZE,
ClientConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
    // Set the idle checking of the multicastStartCrawlNotificationDispatcher. 11/27/2014, Bing Li

  this.multicastStartCrawlNotificationDispatcher.setIdleChecker(ClientConfig.NOTIFICATION_DISPAT
CHER_IDLE_CHECK_DELAY, ClientConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
    // Start the multicastStartCrawlNotificationDispatcher. 11/27/2014, Bing Li
    super.execute(this.multicastStartCrawlNotificationDispatcher);

    // Initialize the notification dispatcher for the notification, CrawlLoadNotification. 11/28/2014, Bing Li
    this.crawlLoadNotificationDispatcher = new NotificationDispatcher<CrawlLoadNotification,
AssignURLLoadThread,
AssignURLLoadThreadCreator>(ClientConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
ClientConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new
AssignURLLoadThreadCreator(), ClientConfig.MAX_NOTIFICATION_TASK_SIZE,
ClientConfig.MAX_NOTIFICATION_THREAD_SIZE,
ClientConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
    // Set the idle checking of the nodeKeyNotificationDispatcher. 11/25/2014, Bing Li

  this.crawlLoadNotificationDispatcher.setIdleChecker(ClientConfig.NOTIFICATION_DISPATCHER_ID
LE_CHECK_DELAY, ClientConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
    // Start the nodeKeyNotificationDispatcher. 11/25/2014, Bing Li
    super.execute(this.crawlLoadNotificationDispatcher);

    // Initialize the notification dispatcher for the notification, StopCrawlMultiNotification. 11/27/2014,
Bing Li
    this.stopCrawlNotificationDispatcher = new NotificationDispatcher<StopCrawlMultiNotification,
```

```
StopCrawlThread,
StopCrawlThreadCreator>(ClientConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
ClientConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new StopCrawlThreadCreator(),
ClientConfig.MAX_NOTIFICATION_TASK_SIZE, ClientConfig.MAX_NOTIFICATION_THREAD_SIZE,
ClientConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
    // Set the idle checking of the stopCrawlNotificationDispatcher. 11/27/2014, Bing Li

  this.stopCrawlNotificationDispatcher.setIdleChecker(ClientConfig.NOTIFICATION_DISPATCHER_ID
LE_CHECK_DELAY, ClientConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
    // Start the stopCrawlNotificationDispatcher. 11/27/2014, Bing Li
    super.execute(this.stopCrawlNotificationDispatcher);
  }

  /*
   * Shutdown the dispatcher. 11/25/2014, Bing Li
   */
  public void shutdown()
  {
    // Shutdown the notification dispatcher for setting the node key. 11/25/2014, Bing Li
    this.nodeKeyNotificationDispatcher.dispose();
    // Shutdown the bound notification dispatcher for starting crawling. 11/27/2014, Bing Li
    this.startCrawlNotificationDispatcher.dispose();
    // Shutdown the bound notification dispatcher for disseminating the crawling notification. 11/27/2014,
Bing Li
    this.multicastStartCrawlNotificationDispatcher.dispose();
    // Shutdown the notification dispatcher for stopping crawling. 11/27/2014, Bing Li
    this.stopCrawlNotificationDispatcher.dispose();
    // Shutdown the parent dispatcher. 11/25/2014, Bing Li
    super.shutdown();
  }

  /*
   * Dispatch received messages to corresponding threads respectively for concurrent processing.
11/25/2014, Bing Li
   */
  public void consume(OutMessageStream<ServerMessage> message)
  {
    // The notification is shared by multiple threads. 11/27/2014, Bing Li
    StartCrawlMultiNotification startCrawlMultiNotification;
    switch (message.getMessage().getType())
    {
      // Process the notification of NodeKeyNotification. 11/25/2014, Bing Li
      case MessageType.NODE_KEY_NOTIFICATION:
        // Enqueue the notification into the notification dispatcher. The notifications are queued and
processed asynchronously. 11/25/2014, Bing Li
        this.nodeKeyNotificationDispatcher.enqueue((NodeKeyNotification)message.getMessage());
        break;

      // Process the notification of StartCrawlMultiNotification. 11/27/2014, Bing Li
      case MessageType.START_CRAWL_MULTI_NOTIFICATION:
        // Cast the message. 11/27/2014, Bing Li
        startCrawlMultiNotification = (StartCrawlMultiNotification)message.getMessage();
        // Enqueue the notification into those bound notification dispatchers. The notifications are
queued and processed asynchronously with a proper synchronization. 11/27/2014, Bing Li
        this.startCrawlNotificationDispatcher.enqueue(startCrawlMultiNotification);
        this.multicastStartCrawlNotificationDispatcher.enqueue(startCrawlMultiNotification);
        break;

      // Process the notification of CrawlLoadNotification. 11/27/2014, Bing Li
      case MessageType.CRAWL_LOAD_NOTIFICATION:
        // Enqueue the notification into the notification dispatcher. The notifications are queued and
processed asynchronously. 11/27/2014, Bing Li
        this.crawlLoadNotificationDispatcher.enqueue((CrawlLoadNotification)message.getMessage());
        break;

      // Process the notification of StopCrawlMultiNotification. 11/27/2014, Bing Li
      case MessageType.STOP_CRAWL_MULTI_NOTIFICATION:
        // Enqueue the notification into the notification dispatcher. The notifications are queued and
```

processed asynchronously. 11/27/2014, Bing Li

```
      this.stopCrawlNotificationDispatcher.enqueue((StopCrawlMultiNotification)message.getMessage());
            break;
        }
      }
}
```

- **RegisterThread**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.NodeKeyNotification;
import com.greatfree.util.NodeID;

/*
 * After getting a notification from the coordinator, it denotes that the coordinator is ready such that the crawler can register itself to the coordinator and prepare for the coming crawling. 11/25/2014, Bing Li
 *
 * The notification contains the key for the crawler. The key is the identification of the crawler. The coordinator uses the ID to manage the cluster of crawlers. 11/25/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class RegisterThread extends NotificationQueue<NodeKeyNotification>
{
  /*
   * Initialize the thread. The argument, taskSize, is used to limit the count of tasks to be queued. 11/25/2014, Bing Li
   */
  public RegisterThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * Once if a node key notification is received, it is processed concurrently as follows. 11/25/2014, Bing Li
   */
  public void run()
  {
    // Declare an instance of NodeKeyNotification. 11/25/2014, Bing Li
    NodeKeyNotification notification;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/25/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/25/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/25/2014, Bing Li
          notification = this.getNotification();
          // Set the crawler key. 11/25/2014, Bing Li
          NodeID.DISTRIBUTED().setKey(notification.getKey());
          // Register the crawler after getting the key. 11/25/2014, Bing Li
          CrawlEventer.NOTIFY().register();
          // Dispose the notification. 11/25/2014, Bing Li
          this.disposeMessage(notification);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      try
      {
        // Wait for a moment after all of the existing notifications are processed. 11/25/2014, Bing Li
        this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
      }
      catch (InterruptedException e)
      {
```

```
            e.printStackTrace();
        }
    }
  }
}
```

- **RegisterThreadCreator**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.NodeKeyNotification;

/*
 * The code here attempts to create instances of RegisterThread. It is used by the notification dispatcher.
11/25/2014, Bing Li
 */

// Created: 11/25/2014, Bing Li
public class RegisterThreadCreator implements NotificationThreadCreatable<NodeKeyNotification,
RegisterThread>
{
  // Create the instance of RegisterThread. 11/25/2014, Bing Li
  @Override
  public RegisterThread createNotificationThreadInstance(int taskSize)
  {
    return new RegisterThread(taskSize);
  }
}
```

- **StartCrawlThread**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.concurrency.BoundNotificationQueue;
import com.greatfree.reuse.MulticastMessageDisposer;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.StartCrawlMultiNotification;

/*
 * The thread processes the instances of StartCrawlMultiNotification from the coordinator. Since the notification is shared by the one that needs to forward it, both of them are derived from BoundNotificationQueue for synchronization. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public class StartCrawlThread extends BoundNotificationQueue<StartCrawlMultiNotification, MulticastMessageDisposer<StartCrawlMultiNotification>>
{
  /*
   * Initialize the thread. 11/26/2014, Bing Li
   *
   * In the constructor, the parameters are explained as follows.
   *
   *      int taskSize: the size of the notification queue.
   *
   *      String dispatcherKey: the key of the dispatcher which manages the thread. It is the thread key that shares the notification in the binder below.
   *
   *      MulticastMessageDisposer<StartCrawlMultiNotification> binder: the binder that synchronizes the relevant threads that share the notification and disposes the notification after the sharing is ended.
   *
   */
  public StartCrawlThread(int taskSize, String dispatcherKey, MulticastMessageDisposer<StartCrawlMultiNotification> binder)
  {
    super(taskSize, dispatcherKey, binder);
  }

  /*
   * The thread to process notifications asynchronously. 11/26/2014, Bing Li
   */
  public void run()
  {
    // Declare a notification instance of StartCrawlMultiNotification. 11/26/2014, Bing Li
    StartCrawlMultiNotification notification;
    // The thread always runs until it is shutdown by the BoundNotificationDispatcher. 11/26/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/26/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/26/2014, Bing Li
          notification = this.getNotification();
          // Start the crawling. 11/26/2014, Bing Li
          CrawlServer.CRAWL().startCrawl();
          // Notify the binder that the notification is consumed by the thread. 11/26/2014, Bing Li
          this.bind(super.getDispatcherKey(), notification);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
```

```
        }
        try
        {
          // Wait for a moment after all of the existing notifications are processed. 11/26/2014, Bing Li
          this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
    }
  }
```

- **StartCrawlThreadCreator**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.concurrency.BoundNotificationThreadCreatable;
import com.greatfree.reuse.MulticastMessageDisposer;
import com.greatfree.testing.message.StartCrawlMultiNotification;

/*
 * This is an implementation of the interface BoundNotificationThreadCreatable to create the instance of
StartCrawlThread inside the pool, BoundNotificationDispatcher. 11/26/2014, Bing Li
 */

// Created: 11/26/2014, Bing Li
public class StartCrawlThreadCreator implements
BoundNotificationThreadCreatable<StartCrawlMultiNotification,
MulticastMessageDisposer<StartCrawlMultiNotification>, StartCrawlThread>
{
  @Override
  public StartCrawlThread createNotificationThreadInstance(int taskSize, String dispatcherKey,
MulticastMessageDisposer<StartCrawlMultiNotification> binder)
  {
    return new StartCrawlThread(taskSize, dispatcherKey, binder);
  }
}
```

- **MulticastStartCrawlNotificationThread**

```
package com.greatfree.testing.crawlserver;

import java.io.IOException;

import com.greatfree.concurrency.BoundNotificationQueue;
import com.greatfree.reuse.MulticastMessageDisposer;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.StartCrawlMultiNotification;

/*
 * The thread processes the notification of StartCrawlMultiNotification concurrently. Since it shares the notifications with other threads, i.e., StartCrawlThread, it must have the mechanism of synchronization. Therefore, it derives the BoundNotificationQueue rather than NotificationQueue. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class MulticastStartCrawlNotificationThread extends
BoundNotificationQueue<StartCrawlMultiNotification,
MulticastMessageDisposer<StartCrawlMultiNotification>>
{
  /*
   * Initialize the thread. 11/27/2014, Bing Li
   *
   * In the constructor, the parameters are explained as follows.
   *
   *      int taskSize: the size of the notification queue.
   *
   *      String dispatcherKey: the key of the dispatcher which manages the thread. It is the thread key that shares the notification in the binder below.
   *
   *      MulticastMessageDisposer<StartCrawlMultiNotification> disposer: the disposer that synchronizes the relevant threads that share the notification and disposes the notification after the sharing is ended.
   *
   */
  public MulticastStartCrawlNotificationThread(int taskSize, String dispatcherKey,
MulticastMessageDisposer<StartCrawlMultiNotification> disposer)
  {
    super(taskSize, dispatcherKey, disposer);
  }

  /*
   * The thread to process notifications asynchronously. 11/27/2014, Bing Li
   */
  public void run()
  {
    // Declare a notification instance of StartCrawlMultiNotification. 11/27/2014, Bing Li
    StartCrawlMultiNotification notification;
    // The thread always runs until it is shutdown by the BoundNotificationDispatcher. 11/27/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/27/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/27/2014, Bing Li
          notification = this.getNotification();
          try
          {
            // Disseminate the notification of StartCrawlNotification among children crawlers. 11/27/2014, Bing Li
```

```
   CrawlerMulticastor.CRAWLER().disseminateStartCrawlNotificationAmongSubCrawlServers(notificatio
n);
                }
                catch (InstantiationException e)
                {
                  e.printStackTrace();
                }
                catch (IllegalAccessException e)
                {
                  e.printStackTrace();
                }
                catch (IOException e)
                {
                  e.printStackTrace();
                }
                // Notify the binder that the notification is consumed by the thread. 11/27/2014, Bing Li
                this.bind(super.getDispatcherKey(), notification);
              }
              catch (InterruptedException e)
              {
                e.printStackTrace();
              }
            }
            try
            {
              // Wait for a moment after all of the existing notifications are processed. 11/26/2014, Bing Li
              this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
            }
            catch (InterruptedException e)
            {
              e.printStackTrace();
            }
          }
        }
      }
```

- **MulticastStartCrawlNotificationThreadCreator**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.concurrency.BoundNotificationThreadCreatable;
import com.greatfree.reuse.MulticastMessageDisposer;
import com.greatfree.testing.message.StartCrawlMultiNotification;

/*
 * The creator aims to generate the instance of MulticastStartCrawlNotificationThread for the
BoundNotificationDispatcher so as to schedule the notification as tasks concurrently. 11/27/2014, Bing
Li
 */

// Created: 11/27/2014, Bing Li
public class MulticastStartCrawlNotificationThreadCreator implements
BoundNotificationThreadCreatable<StartCrawlMultiNotification,
MulticastMessageDisposer<StartCrawlMultiNotification>, MulticastStartCrawlNotificationThread>
{
  @Override
  public MulticastStartCrawlNotificationThread createNotificationThreadInstance(int taskSize, String
dispatcherKey, MulticastMessageDisposer<StartCrawlMultiNotification> binder)
  {
    return new MulticastStartCrawlNotificationThread(taskSize, dispatcherKey, binder);
  }
}
```

- **AssignURLLoadThread**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.data.URLValue;
import com.greatfree.testing.message.CrawlLoadNotification;

/*
 * The thread gets the crawling workload from the coordinator and injects them into the crawling
scheduler. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class AssignURLLoadThread extends NotificationQueue<CrawlLoadNotification>
{
  /*
   * Initialize the thread. The argument, taskSize, is used to limit the count of tasks to be queued.
11/28/2014, Bing Li
   */
  public AssignURLLoadThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * Once if a workload notification is received, it is processed concurrently as follows. 11/28/2014, Bing
Li
   */
  public void run()
  {
    // Declare an instance of CrawlLoadNotification. 11/28/2014, Bing Li
    CrawlLoadNotification notification;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/28/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/28/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/28/2014, Bing Li
          notification = this.getNotification();
          // Convert the instances of URLValue to HubURLs and place them into the crawling scheduler.
11/28/2014, Bing Li
          for (URLValue url : notification.getURLs().values())
          {
            CrawlScheduler.CRAWL().enqueue(new HubURL(url.getKey(), url.getURL(),
url.getUpdatingPeriod()));
          }
          // Dispose the notification. 11/28/2014, Bing Li
          this.disposeMessage(notification);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      try
      {
        // Wait for a moment after all of the existing notifications are processed. 11/28/2014, Bing Li
        this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
      }
      catch (InterruptedException e)
      {
```

```
            e.printStackTrace();
        }
    }
  }
}
```

- **AssignURLLoadThreadCreator**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.CrawlLoadNotification;

/*
 * The creator here attempts to create instances of AssignURLLoadThread. It works with the notification dispatcher to schedule the tasks concurrently. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class AssignURLLoadThreadCreator implements
NotificationThreadCreatable<CrawlLoadNotification, AssignURLLoadThread>
{
  @Override
  public AssignURLLoadThread createNotificationThreadInstance(int taskSize)
  {
    return new AssignURLLoadThread(taskSize);
  }
}
```

- **StopCrawlThread**

```
package com.greatfree.testing.crawlserver;

import java.io.IOException;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.testing.message.StopCrawlMultiNotification;

/*
 * The thread receives stopping crawling notification from the coordinator. It transfers the notification to
all of its children and then stop itself. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StopCrawlThread extends NotificationQueue<StopCrawlMultiNotification>
{
  /*
   * Initialize the thread. The argument, taskSize, is used to limit the count of tasks to be queued.
11/27/2014, Bing Li
   */
  public StopCrawlThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * Once if the stop crawling notification is received, it is processed concurrently as follows. 11/27/2014,
Bing Li
   */
  public void run()
  {
    // Declare an instance of StopCrawlMultiNotification. 11/27/2014, Bing Li
    StopCrawlMultiNotification notification;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/27/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/27/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/27/2014, Bing Li
          notification = this.getNotification();
          try
          {
            // Disseminate the stop notification to all of the current crawler's children nodes. 11/27/2014,
Bing Li

  CrawlerMulticastor.CRAWLER().disseminateStopCrawlNotificationAmongSubCrawlServers(notificatio
n);
          }
          catch (InstantiationException e)
          {
            e.printStackTrace();
          }
          catch (IllegalAccessException e)
          {
            e.printStackTrace();
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
          try
          {
```

```
                // Stop the current crawler. 11/27/2014, Bing Li
                CrawlServer.CRAWL().stop();
              }
              catch (IOException e)
              {
                e.printStackTrace();
              }
              // Dispose the notification. 11/27/2014, Bing Li
              this.disposeMessage(notification);
              return;
            }
            catch (InterruptedException e)
            {
              e.printStackTrace();
            }
          }
        }
      }
    }
```

- **StopCrawlThreadCreator**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.StopCrawlMultiNotification;

/*
 * The code here attempts to create instances of StopCrawlThread. It works with the notification
dispatcher. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StopCrawlThreadCreator implements
NotificationThreadCreatable<StopCrawlMultiNotification, StopCrawlThread>
{
  // Create the instance of StopCrawlThread. 11/27/2014, Bing Li
  @Override
  public StopCrawlThread createNotificationThreadInstance(int taskSize)
  {
    return new StopCrawlThread(taskSize);
  }
}
```

- **CrawlConsumer**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.concurrency.Consumable;
import com.greatfree.concurrency.ConsumerThread;
import com.greatfree.util.NullObject;

/*
 * This class is derived from ConsumerThread. It keeps running all the time immediately after the crawler is started to perform crawling following the manner of producer/consumer. 11/20/2014, Bing Li
 */

// Created: 11/20/2014, Bing Li
public class CrawlConsumer extends ConsumerThread<NullObject, CrawlEater>
{
   /*
    * Initialize. 11/20/2014, Bing Li
    */
   public CrawlConsumer(Consumable<NullObject> consumer, long waitTime)
   {
      super(consumer, waitTime);
   }
}
```

- **CrawlConsumerDisposer**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.reuse.ThreadDisposable;

/*
 * The class is responsible for disposing the instance of HubCrawlConsumer in a threader. 11/20/2014,
Bing Li
 */

// Created: 11/20/2014, Bing Li
public class CrawlConsumerDisposer implements ThreadDisposable<CrawlConsumer>
{
   /*
    * Dispose one instance of HubCrawlConsumer. 11/20/2014, Bing Li
    */
   @Override
   public void dispose(CrawlConsumer r)
   {
      r.dispose();
   }

   /*
    * Dispose one instance of HubCrawlConsumer with time to wait being considered. It is not
implemented yet. Just leave the interface. 11/20/2014, Bing Li
    */
   @Override
   public void dispose(CrawlConsumer r, long time)
   {
      r.dispose();
   }
}
```

- **CrawlEater**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.concurrency.Consumable;
import com.greatfree.util.NullObject;
import com.greatfree.util.TerminateSignal;

/*
 * The process runs in an infinite loop to crawl all of the URLs until the process is terminated.
12/01/2014, Bing Li
 */

// Created: 11/20/2014, Bing Li
public class CrawlEater implements Consumable<NullObject>
{
  @Override
  public void consume(NullObject food)
  {
    // The crawler must work for ever until the crawling server is shutdown. Thus, the crawler needs to
check whether the crawler is terminated. 11/23/2014, ing Li
    while (!TerminateSignal.SIGNAL().isTerminated())
    {
      try
      {
        // Submit one hub URL to crawl. 11/23/2014, Bing Li
        CrawlScheduler.CRAWL().submit();
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
      // Check whether the crawling procedure is demanded to be paused. 11/23/2014, Bing Li
      if (CrawlScheduler.CRAWL().isPaused())
      {
        try
        {
          // If so, it is required to hold on. 11/23/2014, Bing Li
          CrawlScheduler.CRAWL().holdOn();
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
    }
  }
}
```

- **CrawlScheduler**

```
package com.greatfree.testing.crawlserver;

import java.util.Date;
import java.util.Queue;
import java.util.Set;
import java.util.concurrent.LinkedBlockingQueue;

import com.greatfree.reuse.InteractiveDispatcher;
import com.greatfree.util.FileManager;
import com.greatfree.util.Time;
import com.greatfree.util.UtilConfig;

/*
 * This is the crawling scheduler. It works on the way defined in InteractiveDispatcher. 11/23/2014, Bing Li
 */

// Created: 11/21/2014, Bing Li
public class CrawlScheduler
{
   // The interactive dispatcher is responsible for creating instances of crawling threads and interact with them to adjust the crawling procedure to reach the goal of the high performance crawling. 11/23/2014, Bing Li
   private InteractiveDispatcher<HubURL, CrawlNotifier, CrawlThread, CrawlThreadCreator, CrawlThreadDisposer> dispatcher;
   // The queue to take all of the URLs to be crawled. Since each URL needs to be crawled periodically, the queue is defined to schedule them in a FIFO way infinitely. 11/23/2014, Bing Li
   private Queue<HubURL> urlQueue;

   private CrawlScheduler()
   {
   }

   /*
    * A singleton implementation. 11/23/2014, Bing Li
    */
   private static CrawlScheduler instance = new CrawlScheduler();

   public static CrawlScheduler CRAWL()
   {
      if (instance == null)
      {
         instance = new CrawlScheduler();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Dispose the scheduler. 11/23/2014, Bing Li
    */
   public void dispose()
   {
      try
      {
         // Clear the queue. 11/23/2014, Bing Li
         this.urlQueue.clear();
         // Dispose the interactive dispatcher. 11/23/2014, Bing Li
         this.dispatcher.dispose();
      }
      catch (InterruptedException e)
```

```
    {
      e.printStackTrace();
    }
  }

  /*
   * Initialize the interactive dispatcher. 11/23/2014, Bing Li
   */
  public void init()
  {
    /*
     * A directory on the hard disk is required to save files for crawling. 11/23/2014, Bing Li
     */
    // Check whether the required directory exists. 11/23/2014, Bing Li
    if (!FileManager.isDirExisted(CrawlConfig.CRAWLED_FILE_PATH))
    {
      // Create the directory if it does not exist. 11/23/2014, Bing Li
      FileManager.makeDir(CrawlConfig.CRAWLED_FILE_PATH);
    }

    // Initialize the queue. 11/23/2014, Bing Li
    this.urlQueue = new LinkedBlockingQueue<HubURL>();

    // Initialize the interactive dispatcher. 11/23/2014, Bing Li
    this.dispatcher = new InteractiveDispatcher<HubURL, CrawlNotifier, CrawlThread,
CrawlThreadCreator, CrawlThreadDisposer>(CrawlConfig.CRAWLER_FAST_POOL_SIZE,
CrawlConfig.CRAWLER_SLOW_POOL_SIZE, CrawlConfig.CRAWLER_THREAD_TASK_SIZE,
CrawlConfig.CRAWLER_THREAD_IDLE_TIME, new CrawlNotifier(), new CrawlThreadCreator(), new
CrawlThreadDisposer());
    // Set the idle checking for the interactive dispatcher. 11/23/2014, Bing Li
    this.dispatcher.setIdleChecker(CrawlConfig.CRAWLER_IDLE_CHECK_DELAY,
CrawlConfig.CRAWLER_IDLE_CHECK_PERIOD);
  }

  /*
   * Check whether the interactive dispatcher is shutdown. 11/23/2014, Bing Li
   */
  public boolean isShutdown()
  {
    return this.dispatcher.isShutdown();
  }

  /*
   * Check whether the interactive dispatcher is holding on. 11/23/2014, Bing Li
   */
  public boolean isPaused()
  {
    return this.dispatcher.isPaused();
  }

  /*
   * Demand the interactive dispatcher to hold on. 11/23/2014, Bing Li
   */
  public void holdOn() throws InterruptedException
  {
    this.dispatcher.holdOn();
  }

  /*
   * Demand the interactive dispatcher to keep on working. 11/23/2014, Bing Li
   */
  public void keepOn()
  {
    this.dispatcher.keepOn();
  }

  /*
   * Get the keys of fast crawling threads. 11/23/2014, Bing Li
```

```
 */
public Set<String> getFastCrawlThreadKeys()
{
  return this.dispatcher.getFastThreadKeys();
}

/*
 * Get the starting time of a specific fast crawling thread by its key. 11/23/2014, Bing Li
 */
public Date getFastStartTime(String key)
{
  // Get the specific fast thread. 11/23/2014, Bing Li
  CrawlThread thread = this.dispatcher.getFastFunction(key);
  // Check whether the thread is valid. 11/23/2014, Bing Li
  if (thread != CrawlConfig.NO_CRAWL_THREAD)
  {
    // Return the starting time if the thread is valid. 11/23/2014, Bing Li
    return thread.getStartTime();
  }
  // Return null if the thread is invalid. 11/23/2014, Bing Li
  return UtilConfig.NO_TIME;
}

/*
 * Get the ending time of a specific fast crawling by its key. 11/23/2014, Bing Li
 */
public Date getFastEndTime(String key)
{
  // Get the specific fast thread. 11/23/2014, Bing Li
  CrawlThread thread = this.dispatcher.getFastFunction(key);
  // Check whether the thread is valid. 11/23/2014, Bing Li
  if (thread != CrawlConfig.NO_CRAWL_THREAD)
  {
    // Return the ending time if the thread is valid. 11/23/2014, Bing Li
    return thread.getEndTime();
  }
  // Return null if the thread is invalid. 11/23/2014, Bing Li
  return UtilConfig.NO_TIME;
}

/*
 * Get the URL which a specific fast thread is working on. 11/23/2014, Bing Li
 */
public String getFastURL(String key)
{
  // Get the specific fast thread. 11/23/2014, Bing Li
  CrawlThread thread = this.dispatcher.getFastFunction(key);
  // Check whether the thread is valid. 11/23/2014, Bing Li
  if (thread != CrawlConfig.NO_CRAWL_THREAD)
  {
    // Return the URL if the thread is valid. 11/23/2014, Bing Li
    return thread.getURL();
  }
  // Return null if the thread is invalid. 11/23/2014, Bing Li
  return CrawlConfig.NO_URL_LINK;
}

/*
 * Get the starting time of a specific slow crawling thread by its key. 11/23/2014, Bing Li
 */
public Date getSlowStartTime(String key)
{
  // Get the specific slow thread. 11/23/2014, Bing Li
  CrawlThread thread = this.dispatcher.getSlowFunction(key);
  // Check whether the thread is valid. 11/23/2014, Bing Li
  if (thread != CrawlConfig.NO_CRAWL_THREAD)
  {
    // Return the starting time if the thread is valid. 11/23/2014, Bing Li
```

```
      return thread.getStartTime();
    }
    // Return null if the thread is invalid. 11/23/2014, Bing Li
    return UtilConfig.NO_TIME;
  }

  /*
   * Get the ending time of a specific slow crawling by its key. 11/23/2014, Bing Li
   */
  public Date getSlowEndTime(String key)
  {
    // Get the specific slow thread. 11/23/2014, Bing Li
    CrawlThread thread = this.dispatcher.getSlowFunction(key);
    // Check whether the thread is valid. 11/23/2014, Bing Li
    if (thread != CrawlConfig.NO_CRAWL_THREAD)
    {
      // Return the ending time if the thread is valid. 11/23/2014, Bing Li
      return thread.getEndTime();
    }
    // Return null if the thread is invalid. 11/23/2014, Bing Li
    return UtilConfig.NO_TIME;
  }

  /*
   * Get the URL which a specific slow thread is working on. 11/23/2014, Bing Li
   */
  public String getSlowURL(String key)
  {
    // Get the specific slow thread. 11/23/2014, Bing Li
    CrawlThread thread = this.dispatcher.getSlowFunction(key);
    // Check whether the thread is valid. 11/23/2014, Bing Li
    if (thread != CrawlConfig.NO_CRAWL_THREAD)
    {
      // Return the URL if the thread is valid. 11/23/2014, Bing Li
      return thread.getURL();
    }
    // Return null if the thread is invalid. 11/23/2014, Bing Li
    return CrawlConfig.NO_URL_LINK;
  }

  /*
   * Check whether a fast thread's task queue is empty. 11/27/2014, Bing Li
   */
  public int IsFastEmpty(String key)
  {
    // Get the specific fast thread. 11/27/2014, Bing Li
    CrawlThread thread = this.dispatcher.getFastFunction(key);
    // Check whether the thread is valid. 11/27/2014, Bing Li
    if (thread != CrawlConfig.NO_CRAWL_THREAD)
    {
      // Check whether the thread is empty. 11/27/2014, Bing Li
      if (thread.isEmpty())
      {
        // Return empty. 11/27/2014, Bing Li
        return CrawlConfig.EMPTY;
      }
      else
      {
        // Return not empty. 11/27/2014, Bing Li
        return CrawlConfig.NOT_EMPTY;
      }
    }
    // Return dead. 11/27/2014, Bing Li
    return CrawlConfig.DEAD;
  }

  /*
   * Check whether a fast crawling thread is idle or not. 11/23/2014, Bing Li
```

```
 */
public int isIdle(String key)
{
  // Get the specific fast thread. 11/23/2014, Bing Li
  CrawlThread thread = this.dispatcher.getFastFunction(key);
  // Check whether the thread is valid. 11/23/2014, Bing Li
  if (thread != CrawlConfig.NO_CRAWL_THREAD)
  {
    // Check whether the thread is idle. 11/23/2014, Bing Li
    if (thread.getIdleTime() != Time.INIT_TIME)
    {
      // Return idle if it is. 11/23/2014, Bing Li
      return CrawlConfig.IDLE;
    }
    else
    {
      // Return busy if it is not. 11/23/2014, Bing Li
      return CrawlConfig.BUSY;
    }
  }
  // Return dead it does not exist. 11/23/2014, Bing Li
  return CrawlConfig.DEAD;
}

/*
 * Get the idle time of a fast crawling thread. 11/23/2014, Bing Li
 */
public Date getIdleTime(String key)
{
  // Get the specific fast thread. 11/23/2014, Bing Li
  CrawlThread thread = this.dispatcher.getFastFunction(key);
  // Check whether the thread is valid. 11/23/2014, Bing Li
  if (thread != CrawlConfig.NO_CRAWL_THREAD)
  {
    // Check whether the thread is idle. 11/23/2014, Bing Li
    if (thread.getIdleTime() != Time.INIT_TIME)
    {
      // Return the idle time if it is. 11/23/2014, Bing Li
      return thread.getIdleTime();
    }
    else
    {
      // Return a meaningless time if it is not. 11/23/2014, Bing Li
      return Time.INIT_TIME;
    }
  }
  // Return a meaningless time if it does not exist. 11/23/2014, Bing Li
  return Time.INIT_TIME;
}

/*
 * Get the keys of slow crawling threads. 11/23/2014, Bing Li
 */
public Set<String> getSlowCrawlThreadKeys()
{
  return this.dispatcher.getSlowThreadKeys();
}

/*
 * Solve the problem if a thread is detected slow. 11/23/2014, Bing Li
 */
public synchronized void alleviateSlow(String key) throws InterruptedException
{
  this.dispatcher.alleviateSlow(key);
}

/*
 * Restore the slow thread to fast if applicable. 11/23/2014, Bing Li
```

```
   */
  public synchronized void restoreFast(String key) throws InterruptedException
  {
    this.dispatcher.restoreFast(key);
  }

  /*
   * Expose the URL count to be crawled. 11/23/2014, Bing Li
   */
  public long getURLCount()
  {
    return this.urlQueue.size();
  }

  /*
   * Enqueue a hub URL for scheduling. 11/23/2014, Bing Li
   */
  public void enqueue(HubURL hub)
  {
    this.urlQueue.add(hub);
  }

  /*
   * Submit a hub URL for crawling. 11/23/2014, Bing Li
   */
  public void submit() throws InterruptedException
  {
    // Dequeue a hub URL from the queue. 11/23/2014, Bing Li
    HubURL hub = this.urlQueue.poll();
    // Check whether hub URL is valid. 11/23/2014, Bing Li
    if (hub != CrawlConfig.NO_URL)
    {
      // Check whether the passed time of the hub URL is longer than its updating period. 11/23/2014,
Bing Li
      if (hub.getPassedTime() >= hub.getUpdatingPeriod())
      {
        // If so, it denotes that it is time to crawl it. Thus, the hub URL is enqueued into the interactive
dispatcher. 11/23/2014, Bing Li
        this.dispatcher.enqueue(hub);
      }
      else
      {
        // If not, it denotes that it is not the proper moment to crawl the hub because it might not be
updated. 11/23/2014, Bing Li
        hub.incrementTime();
        // Enqueue the hub URL to the end of the queue for future scheduling. 11/23/2014, Bing Li
        this.urlQueue.add(hub);
      }
    }
  }
}
```

- **HubURL**

```
package com.greatfree.testing.crawlserver;

import java.util.Date;

import com.greatfree.testing.data.URLValue;
import com.greatfree.testing.db.URLEntity;
import com.greatfree.util.Time;

/*
 * This is the URL that must be crawled by the sample crawler. The hub indicates that the URL is super on the Web. For the detailed explanation of the concept of hub URL, refer to the knowledge of WWW. 11/21/2014, Bing Li
 */

// Created: 11/21/2014, Bing Li
public class HubURL
{
   // The key to represent the URL. The key is unique for each URL. It is the hash string generated upon the URL. 11/23/2014, Bing Li
   private String key;
   // The URL to be crawled. 11/23/2014, Bing Li
   private String url;
   // The time passed after a request is sent to the URL. It is an indicator whether the server on which the URL resides is fast or slow. 11/23/2014, Bing Li
   private long passedTime;
   // The updating period of the URL. According to it, the crawler can determine the moment to crawl it. 11/23/2014, Bing Li
   private long updatingPeriod;
   // The time that the content of the URL was modified. 11/23/2014, Bing Li
   private Date lastModified;

   /*
    * Initialize the URL. 11/23/2014, Bing Li
    */
   public HubURL(String key, String url, long updatingPeriod)
   {
      this.key = key;
      this.url = url;
      this.passedTime = 0;
      this.updatingPeriod = updatingPeriod;
      this.lastModified = Time.NO_TIME;
   }

   /*
    * Dispose the URL. 11/23/2014, Bing Li
    */
   public void dispose()
   {
      this.lastModified = null;
   }

   /*
    * Update the instance of the URL. The properties, passedTime, updatingPeriod and lastModified, must change during the crawling procedure. Thus, it is required to update them when new values are detected. 11/23/2014, Bing Li
    */
   public void update(long passedTime, long updatingPeriod, Date lastModified)
   {
      this.passedTime = passedTime;
      this.updatingPeriod = updatingPeriod;
      this.lastModified = lastModified;
   }

   /*
```

```
   * Get the URLValueEntity to be persisted. 11/23/2014, Bing Li
   */
  public URLEntity getEntity()
  {
    return new URLEntity(this.key, this.url, this.updatingPeriod);
  }

  /*
   * Get the value of the URL. The instance of URLValue must be transferred over the network. Thus, it
is serialized and the logic to process is filtered. 11/23/2014, Bing Li
   */
  public URLValue getValue()
  {
    return new URLValue(this.key, this.url, this.updatingPeriod);
  }

  /*
   * Expose the key of the URL. 11/23/2014, Bing Li
   */
  public String getKey()
  {
    return this.key;
  }

  /*
   * Expose the URL. 11/23/2014, Bing Li
   */
  public String getURL()
  {
    return this.url;
  }

  /*
   * Expose the passed time. 11/23/2014, Bing Li
   */
  public long getPassedTime()
  {
    return this.passedTime;
  }

  /*
   * Clear the passed time. 11/23/2014, Bing Li
   */
  public void clearPassedTime()
  {
    this.passedTime = 0;
  }

  /*
   * Increment the passed time. The larger the passed time, the slower the server the URL resides.
11/23/2014, Bing Li
   */
  public void incrementTime()
  {
    this.passedTime += CrawlConfig.CRAWLING_TIMER_PERIOD;
  }

  /*
   * Expose the updating period. 11/23/2014, Bing Li
   */
  public long getUpdatingPeriod()
  {
    return this.updatingPeriod;
  }

  /*
   * Increment the updating period. Since the crawling is performed periodically, the period is determined
in a heuristic way. It is not the actual value. 11/23/2014, Bing Li
```

```
   */
  public void incrementPeriod()
  {
    this.updatingPeriod += CrawlConfig.UPDATING_VALUE;
  }

  /*
   * Decrement the updating period. 11/23/2014, Bing Li
   */
  public void decrementPeriod()
  {
    // Check whether the updating period is larger than the updating value. 11/23/2014, Bing Li
    if (this.updatingPeriod > CrawlConfig.UPDATING_VALUE)
    {
      // The current updating period is reduced by the updating value if the updating period is larger than
it. 11/23/2014, Bing Li
      this.updatingPeriod -= CrawlConfig.UPDATING_VALUE;
    }
    // Check whether the updating period is lower than the minimum updating period. 11/23/2014, Bing
Li
    if (this.updatingPeriod < CrawlConfig.MIN_UPDATING_PERIOD)
    {
      // If the current updating period is too low, it is not necessary to cut it. Just set it to the minimum
one. 11/23/2014, Bing Li
      this.updatingPeriod = CrawlConfig.MIN_UPDATING_PERIOD;
    }
  }

  /*
   * Expose the last modified time. 11/23/2014, Bing Li
   */
  public Date getLastModified()
  {
    return this.lastModified;
  }

  /*
   * Update the last modified time. 11/23/2014, Bing Li
   */
  public void updateLastModified(Date lastModified)
  {
    this.lastModified = lastModified;
  }
}
```

- **CrawlNotifier**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.concurrency.Interactable;

/*
 * This is the notifier that calls back the methods provided by the InteractiveDispatcher to notify the dynamic state of crawling. The InteractiveDispatcher then can perform relevant reactions on those notifications. 11/23/2014, Bing Li
 */

// Created: 11/23/2014, Bing Li
public class CrawlNotifier implements Interactable<HubURL>
{
  /*
   * Demand the scheduler to pause. 11/23/2014, Bing Li
   */
  @Override
  public void pause()
  {
    try
    {
      CrawlScheduler.CRAWL().holdOn();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
  }

  /*
   * Demand the scheduler to keep on crawling. 11/23/2014, Bing Li
   */
  @Override
  public void keepOn()
  {
    CrawlScheduler.CRAWL().keepOn();
  }

  /*
   * Restore a slow thread to fast. 11/23/2014, Bing Li
   */
  @Override
  public void restoreFast(String key)
  {
    try
    {
      CrawlScheduler.CRAWL().restoreFast(key);
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
  }

  /*
   * One hub URL is crawled, thus it is time to enqueue it for later scheduling. 11/23/2014, Bing Li
   */
  @Override
  public void done(HubURL task)
  {
    CrawlScheduler.CRAWL().enqueue(task);
  }
}
```

- **CrawlThread**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.concurrency.InteractiveQueue;
import com.greatfree.testing.data.ServerConfig;

/*
 * This is the thread that takes the task of crawling concurrently. It is derived from InteractiveQueue.
11/23/2014, Bing Li
 */

// Created: 11/23/2014, Bing Li
public class CrawlThread extends InteractiveQueue<HubURL, CrawlNotifier>
{
  /*
   * Initialize the crawling thread. 11/23/2014, Bing Li
   */
  public CrawlThread(int taskSize, CrawlNotifier notifier)
  {
    super(taskSize, notifier);
  }

  /*
   * Dispose the thread. 11/23/2014, Bing Li
   */
  public void dispose()
  {
    super.dispose();
  }

  /*
   * Expose the URL being crawled. 11/23/2014, Bing Li
   */
  public String getURL()
  {
    // Get the current URL being crawled. 11/23/2014, Bing Li
    HubURL url = this.getCurrentTask();
    // Check whether the URL is valid. 11/23/2014, Bing Li
    if (url != CrawlConfig.NO_URL)
    {
      // Return the URL. 11/23/2014, Bing Li
      return url.getURL();
    }
    // Return null if no crawling URLs are available. 11/23/2014, Bing Li
    return CrawlConfig.NO_URL_LINK;
  }

  /*
   * Crawl each URL scheduled to the thread one by one concurrently. 11/23/2014, Bing Li
   */
  public void run()
  {
    // An instance of HubURL. 11/23/2014, Bing Li
    HubURL url = CrawlConfig.NO_URL;
    // Check whether the crawling thread is terminated. 11/23/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the task queue is empty. 11/23/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // If the queue is not empty, dequeue the instance of URL. 11/23/2014, Bing Li
          url = this.getTask();
          // Set the starting time stamp. 11/23/2014, Bing Li
```

```
            this.setStartTime();
            // Crawl the dequeued URL. 11/23/2014, Bing Li
            Crawler.crawl(url);
            // Notify the interactive dispatcher that the URL is crawled. 11/23/2014, Bing Li
            this.done(url);
            // Set the ending time stamp. 11/23/2014, Bing Li
            this.setEndTime();
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
        }
        try
        {
          // Wait for some time if the task queue is empty. 11/23/2014, Bing Li
          this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
    }
  }
```

- **CrawlThreadCreator**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.concurrency.InteractiveThreadCreatable;

/*
 * The creator generates instances of CrawlThread in the interactive dispatcher. 11/23/2014, Bing Li
 */

// Created: 11/23/2014, Bing Li
public class CrawlThreadCreator implements InteractiveThreadCreatable<HubURL, CrawlNotifier,
CrawlThread>
{
  // Create an instance of crawling thread. 11/23/2014, Bing Li
  @Override
  public CrawlThread createInteractiveThreadInstance(int taskSize, CrawlNotifier notifier)
  {
    return new CrawlThread(taskSize, notifier);
  }
}
```

- **CrawlThreadDisposer**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.reuse.ThreadDisposable;

/*
 * The disposer collects the instance of crawling thread in the interactive dispatcher. 11/23/2014, Bing Li
 */

// Created: 11/23/2014, Bing Li
public class CrawlThreadDisposer implements ThreadDisposable<CrawlThread>
{
   // Dispose the instance of crawling thread. 11/23/2014, Bing Li
   @Override
   public void dispose(CrawlThread r)
   {
      r.dispose();
   }

   // Dispose the instance of crawling thread. 11/23/2014, Bing Li
   @Override
   public void dispose(CrawlThread r, long time)
   {
      r.dispose();
   }
}
```

- **Crawler**

```
package com.greatfree.testing.crawlserver;

import java.io.File;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Set;

import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import org.jsoup.select.Elements;

import com.google.common.collect.Sets;
import com.greatfree.testing.data.CrawledLink;
import com.greatfree.util.FileManager;
import com.greatfree.util.Tools;
import com.greatfree.util.UtilConfig;

/*
 * This is the class that crawls a URL. For the sample, it just gets the links from the URL, no further accessing on those links. 11/23/2014, Bing Li
 */

// Created: 11/23/2014, Bing Li
public class Crawler
{
  /*
   * Crawl the URL contained in the instance of HubURL. 11/23/2014, Bing Li
   */
  public static Set<CrawledLink> crawl(HubURL hubURL)
  {
    // Create the file name, which is formed mainly by the URL key. The file saves the HTML file crawled from the URL. 11/23/2014, Bing Li
    String fileName = CrawlConfig.CRAWLED_FILE_PATH + hubURL.getKey();
    // The instance of URL, which is defined by JDK. 11/23/2014, Bing Li
    URL url;
    try
    {
      // Initialize the instance of URL. 11/23/2014, Bing Li
      url = new URL(hubURL.getURL());
    }
    catch (MalformedURLException e)
    {
      e.printStackTrace();
      return CrawlConfig.NO_LINK_KEYS;
    }

    // The instance of Document, which is used to take the parsed the HTML information. 11/23/2014, Bing Li
    Document doc;
    try
    {
      // Assign the value of the instance of Document. The value is parsed by JSoup. 11/23/2014, Bing Li
      doc = Jsoup.parse(url, CrawlConfig.CRAW_TIMEOUT);
    }
    catch (IOException e)
    {
      e.printStackTrace();
      return CrawlConfig.NO_LINK_KEYS;
    }
    catch (IllegalArgumentException e)
    {
```

```
            e.printStackTrace();
            return CrawlConfig.NO_LINK_KEYS;
        }
        try
        {
            // Save the HTML text into the file system. 11/23/2014, Bing Li
            FileManager.createTextFile(fileName, doc.html());
        }
        catch (IOException e)
        {
            e.printStackTrace();
            return CrawlConfig.NO_LINK_KEYS;
        }
        // Create an instance of File upon the HTML text file. 11/23/2014, Bing Li
        File html = new File(fileName);
        // The instance of Document, which is used to take the parsed the HTML information. 11/23/2014,
Bing Li
        Document loadedDoc;
        try
        {
            // Assign the value of the instance of Document. The value is parsed by JSoup. 11/23/2014, Bing
Li
            loadedDoc = Jsoup.parse(html, UtilConfig.UTF_8, hubURL.getURL());
        }
        catch (IOException e)
        {
            e.printStackTrace();
            return CrawlConfig.NO_LINK_KEYS;
        }
        // Get the links by parsing the HTML text on the tag, "a". 11/23/2014, Bing Li
        Elements links = loadedDoc.getElementsByTag(CrawlConfig.TAG_A);
        // The link string. 11/23/2014, Bing Li
        String linkHref;
        // Initialize a set to contain all the link strings. 11/23/2014, Bing Li
        Set<CrawledLink> crawledLinks = Sets.newHashSet();
        // Parse the link to get the link string and save them into the set. 11/23/2014, Bing Li
        for (Element link : links)
        {
            // Parse the link to get the link string. 11/23/2014, Bing Li
            linkHref = link.attr(CrawlConfig.HREF);
            // Save the link string into the set. 11/23/2014, Bing Li
            crawledLinks.add(new CrawledLink(Tools.getHash(linkHref), linkHref, link.text(),
hubURL.getKey()));
        }
        // Send the crawled links from the URL to the coordinator. 11/23/2014, Bing Li
        CrawlEventer.NOTIFY().sendCrawledLinks(crawledLinks);
        // Return the crawled links.
        return crawledLinks;
    }
}
```

- **CrawlingStateChecker**

```
package com.greatfree.testing.crawlserver;

import java.util.Calendar;
import java.util.Date;
import java.util.Set;
import java.util.TimerTask;

import com.greatfree.util.Time;
import com.greatfree.util.UtilConfig;

/*
 * The class is responsible for checking the states of all of the currently working crawling threads. Each
thread has some workloads to crawl the URLs. However, for the complicated circumstance of the Web,
some URLs might be slow. Some might be fast. Some might be dead. It must affect the scheduling the
tasks if some threads work on slow and dead URLs. Those threads must identified by the checker. If
necessary, their workloads must be rescheduled in order to save the resources. Some interactions
between those threads and the interactive dispatchers are required to happen in the checker.
11/27/2014, Bing Li
 *
 * The checker must work periodically. So it is derived from TimerTask. 11/27/2014, Bing Li
 *
 * In addition, the checker also displays the states on the screen for possible monitors of administrators.
11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class CrawlingStateChecker extends TimerTask
{
  /*
   * The checking is performed concurrently by a timer. 11/27/2014, Bing Li
   */
  @Override
  public void run()
  {
    // Get the fast crawling thread keys. 11/27/2014, Bing Li
    Set<String> fastCrawlThreadKeys = CrawlScheduler.CRAWL().getFastCrawlThreadKeys();

    // Display the necessary information on the screen. 11/27/2014, Bing Li
    System.out.println("\n============= Fast Crawler =============");
    // Display the current time. 11/27/2014, Bing Li
    System.out.println("Current time: " + Calendar.getInstance().getTime());
    // Display the size of the fast threads. 11/27/2014, Bing Li
    System.out.println("------> Fast crawler size: " + fastCrawlThreadKeys.size());
    // Declare the starting time of crawling for threads. 11/27/2014, Bing Li
    Date startTime;
    // Scan each thread to check their states. 11/27/2014, Bing Li
    for (String key : fastCrawlThreadKeys)
    {
      // Get the starting time of one particular fast thread by its key. 11/27/2014, Bing Li
      startTime = CrawlScheduler.CRAWL().getFastStartTime(key);
      // Check whether the starting time is valid or not. 11/27/2014, Bing Li
      if (startTime != UtilConfig.NO_TIME)
      {
        // Check whether the starting time is a meaningless value. 11/27/2014, Bing Li
        if (startTime != Time.INIT_TIME)
        {
          // Check whether the fast thread has been starting crawling for long enough to exceed the
upper limit. 11/27/2014, Bing Li
          if (Time.getTimeSpanInSecond(Calendar.getInstance().getTime(), startTime) >=
CrawlConfig.MAX_BUSY_TIME_LENGTH)
          {
            System.out.println("Killed: " + CrawlScheduler.CRAWL().getFastURL(key) + " takes " +
Time.getTimeSpanInSecond(Calendar.getInstance().getTime(), startTime) + " seconds");
            try
```

```
                {
                  // If so, the fast thread must be turned to a slow one. 11/27/2014, Bing Li
                  CrawlScheduler.CRAWL().alleviateSlow(key);
                }
                catch (InterruptedException e)
                {
                  e.printStackTrace();
                }
              }
              else
              {
                System.out.println("Working: " + CrawlScheduler.CRAWL().getFastURL(key) + " takes " +
Time.getTimeSpanInSecond(Calendar.getInstance().getTime(), startTime) + " seconds");
              }
            }
            else
            {
              System.out.println("-----------------------------------");
              // If the starting time is meaningless, it denotes that the thread has no job to do. 11/27/2014,
Bing Li
              System.out.println("Done: " + CrawlScheduler.CRAWL().getFastURL(key) + " takes " +
Time.getTimeSpanInSecond(Calendar.getInstance().getTime(),
CrawlScheduler.CRAWL().getFastEndTime(key)) + " seconds");
            }
          }
        }
        System.out.println("=========================================\n");

        System.out.println("\n============ Slow Crawler ==============");
        // Get the slow thread keys. 11/27/2014, Bing Li
        Set<String> slowCrawlThreadKeys = CrawlScheduler.CRAWL().getSlowCrawlThreadKeys();
        System.out.println("Current time: " + Calendar.getInstance().getTime());
        // Display the size of the slow threads. 11/27/2014, Bing Li
        System.out.println("------> Slow crawler size: " + slowCrawlThreadKeys.size());
        // Scan each slow thread to check their states. 11/27/2014, Bing Li
        for (String key : slowCrawlThreadKeys)
        {
          // Get the starting time of one particular slow thread. 11/27/2014, Bing Li
          startTime = CrawlScheduler.CRAWL().getSlowStartTime(key);
          // Check whether the starting time is valid. 11/27/2014, Bing Li
          if (startTime != UtilConfig.NO_TIME)
          {
            // Check whether the starting time make sense. 11/27/2014, Bing Li
            if (startTime != Time.INIT_TIME)
            {
              // If the starting time is effective, display the time the slow thread work on the specific URL.
11/27/2014, Bing Li
              System.out.println("Working: " + CrawlScheduler.CRAWL().getSlowURL(key) + " takes " +
Time.getTimeSpanInSecond(Calendar.getInstance().getTime(), startTime) + " seconds");
            }
            else
            {
              // If the starting time does not make sense, it denotes that the thread has finished its task.
Display the consumed time. 11/27/2014, Bing Li
              System.out.println("Accomplished: " + CrawlScheduler.CRAWL().getSlowURL(key) + " takes "
+ Time.getTimeSpanInSecond(Calendar.getInstance().getTime(),
CrawlScheduler.CRAWL().getSlowEndTime(key)) + " seconds");
            }
          }
        }
        System.out.println("=========================================\n");
      }
    }
```

- **ClientPool**

```
package com.greatfree.testing.crawlserver;

import java.io.IOException;

import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.data.ClientConfig;

/*
 * The pool is responsible for creating and managing instance of FreeClient to achieve the goal of using as small number of instances of FreeClient to send messages to the coordinator in a high performance. 11/23/2014, Bing Li
 */

// Created: 11/23/2014, Bing Li
public class ClientPool
{
  // An instance of FreeClientPool is defined to interact with the coordinator. 11/23/2014, Bing Li
  private FreeClientPool pool;

  private ClientPool()
  {
  }

  /*
   * A singleton definition. 11/23/2014, Bing Li
   */
  private static ClientPool instance = new ClientPool();

  public static ClientPool CRAWL()
  {
    if (instance == null)
    {
      instance = new ClientPool();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the free client pool. 11/23/2014, Bing Li
   */
  public void dispose() throws IOException
  {
    this.pool.dispose();
  }

  /*
   * Initialize the client pool. The method is called when the crawler process is started. 11/23/2014, Bing Li
   */
  public void init()
  {
    // Initialize the client pool. 11/23/2014, Bing Li
    this.pool = new FreeClientPool(ClientConfig.CLIENT_POOL_SIZE);
    // Set idle checking for the client pool. 11/23/2014, Bing Li
    this.pool.setIdleChecker(ClientConfig.CLIENT_IDLE_CHECK_DELAY, ClientConfig.CLIENT_IDLE_CHECK_PERIOD, ClientConfig.CLIENT_MAX_IDLE_TIME);
  }

  /*
   * Expose the client pool. 11/23/2014, Bing Li
```

```
     */
    public FreeClientPool getPool()
    {
      return this.pool;
    }
}
```

- **SubClientPool**

```
package com.greatfree.testing.crawlserver;

import java.io.IOException;

import com.greatfree.remote.FreeClientPool;

/*
 * The pool is responsible for creating and managing instance of FreeClient to achieve the goal of using as small number of instances of FreeClient to send messages to the children crawlers in a high performance. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class SubClientPool
{
  // An instance of FreeClientPool is defined to interact with the children crawlers. 11/27/2014, Bing Li
  private FreeClientPool clientPool;

  private SubClientPool()
  {
  }

  /*
   * A singleton definition. 11/27/2014, Bing Li
   */
  private static SubClientPool instance = new SubClientPool();

  public static SubClientPool CRAWL()
  {
    if (instance == null)
    {
      instance = new SubClientPool();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the free client pool. 11/27/2014, Bing Li
   */
  public void dispose()
  {
    try
    {
      this.clientPool.dispose();
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
  }

  /*
   * Initialize the client pool. The method is called when the crawler process is started. 11/27/2014, Bing Li
   */
  public void init()
  {
    // Initialize the client pool. 11/23/2014, Bing Li
    this.clientPool = new FreeClientPool(CrawlConfig.SUB_CLIENT_POOL_SIZE);
    this.clientPool.setIdleChecker(CrawlConfig.SUB_CLIENT_IDLE_CHECK_DELAY,
```

```
CrawlConfig.SUB_CLIENT_IDLE_CHECK_PERIOD, CrawlConfig.SUB_CLIENT_MAX_IDLE_TIME);
  }

  /*
   * Expose the client pool. 11/27/2014, Bing Li
   */
  public FreeClientPool getPool()
  {
    return this.clientPool;
  }
}
```

- **CrawlEventer**

```
package com.greatfree.testing.crawlserver;

import java.io.IOException;
import java.util.Set;

import com.greatfree.concurrency.ThreadPool;
import com.greatfree.remote.AsyncRemoteEventer;
import com.greatfree.remote.SyncRemoteEventer;
import com.greatfree.testing.data.ClientConfig;
import com.greatfree.testing.data.CrawledLink;
import com.greatfree.testing.message.CrawledLinksNotification;
import com.greatfree.testing.message.OnlineNotification;
import com.greatfree.testing.message.RegisterCrawlServerNotification;
import com.greatfree.testing.message.UnregisterCrawlServerNotification;
import com.greatfree.util.NodeID;

/*
 * This is an eventer that sends notifications to the coordinator in a synchronous or asynchronous manner. 11/27/2014, Bing Li
 */

// Created: 11/23/2014, Bing Li
public class CrawlEventer
{
  // The IP of the coordinator the eventer needs to notify. 11/23/2014, Bing Li
  private String ip;
  // The port of the coordinator the eventer needs to notify. 11/23/2014, Bing Li
  private int port;
  // The thread pool that starts up the asynchronous eventer. 11/23/2014, Bing Li
  private ThreadPool pool;
  // The synchronous eventer notifies the coordinator that a crawler is online. After receiving the notification, the coordinator can assign tasks and interact with the crawler for crawling. 11/23/2014, Bing Li
  private SyncRemoteEventer<OnlineNotification> onlineNotificationEventer;
  // The synchronous eventer notifies the coordinator that a crawler needs to register. 11/23/2014, Bing Li
  private SyncRemoteEventer<RegisterCrawlServerNotification> registerCrawlServerEventer;
  // The synchronous eventer notifies the coordinator that a crawler needs to unregister. 11/23/2014, Bing Li
  private SyncRemoteEventer<UnregisterCrawlServerNotification> unregisterCrawlServerEventer;
  // The asynchronous eventer sends crawled links to the coordinator. 11/23/2014, Bing Li
  private AsyncRemoteEventer<CrawledLinksNotification> crawledLinkEventer;

  private CrawlEventer()
  {
  }

  /*
   * Initialize a singleton. 11/23/2014, Bing Li
   */
  private static CrawlEventer instance = new CrawlEventer();

  public static CrawlEventer NOTIFY()
  {
    if (instance == null)
    {
      instance = new CrawlEventer();
      return instance;
    }
    else
    {
      return instance;
    }
  }
```

```
  /*
   * Dispose the eventer. 11/23/2014, Bing Li
   */
  public void dispose()
  {
    // Dispose the online eventer. 11/23/2014, Bing Li
    this.onlineNotificationEventer.dispose();
    // Dispose the registering eventer. 11/23/2014, Bing Li
    this.registerCrawlServerEventer.dispose();
    // Dispose the unregistering eventer. 11/23/2014, Bing Li
    this.unregisterCrawlServerEventer.dispose();
    // Dispose the crawled link eventer. 11/23/2014, Bing Li
    this.crawledLinkEventer.dispose();
    // Shutdown the thread pool. 11/23/2014, Bing Li
    this.pool.shutdown();
  }

  /*
   * Initialize the eventer. The IP/port is the coordinator to be notified. 11/23/2014, Bing Li
   */
  public void init(String ipAddress, int port)
  {
    // Assign the IP. 11/23/2014, Bing Li
    this.ip = ipAddress;
    // Assign the port. 11/23/2014, Bing Li
    this.port = port;
    // Initialize a thread pool. 11/23/2014, Bing Li
    this.pool = new ThreadPool(ClientConfig.EVENTER_THREAD_POOL_SIZE,
ClientConfig.EVENTER_THREAD_POOL_ALIVE_TIME);

    this.onlineNotificationEventer = new
SyncRemoteEventer<OnlineNotification>(ClientPool.CRAWL().getPool());
    this.registerCrawlServerEventer = new
SyncRemoteEventer<RegisterCrawlServerNotification>(ClientPool.CRAWL().getPool());
    this.unregisterCrawlServerEventer = new
SyncRemoteEventer<UnregisterCrawlServerNotification>(ClientPool.CRAWL().getPool());

    // Initialize the crawled links eventer. 11/23/2014, Bing Li
    this.crawledLinkEventer = new
AsyncRemoteEventer<CrawledLinksNotification>(ClientPool.CRAWL().getPool(), this.pool,
ClientConfig.EVENT_QUEUE_SIZE, ClientConfig.EVENTER_SIZE,
ClientConfig.EVENTING_WAIT_TIME, ClientConfig.EVENTER_WAIT_TIME);
    // Set the idle checking for the crawled links eventer. 11/23/2014, Bing Li
    this.crawledLinkEventer.setIdleChecker(ClientConfig.EVENT_IDLE_CHECK_DELAY,
ClientConfig.EVENT_IDLE_CHECK_PERIOD);
    // Start up the crawled links eventer. 11/23/2014, Bing Li
    this.pool.execute(this.crawledLinkEventer);
  }

  /*
   * Notify the coordinator that the crawler is online. 11/23/2014, Bing Li
   */
  public void notifyOnline() throws IOException, InterruptedException
  {
    this.onlineNotificationEventer.notify(this.ip, this.port, new OnlineNotification());
  }

  /*
   * Register the crawler. 11/23/2014, Bing Li
   */
  public void register()
  {
    try
    {
      this.registerCrawlServerEventer.notify(this.ip, this.port, new
RegisterCrawlServerNotification(NodeID.DISTRIBUTED().getKey(),
CrawlScheduler.CRAWL().getURLCount()));
```

```
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
  }

  /*
   * Unregister the crawler. 11/23/2014, Bing Li
   */
  public void unregister()
  {
    try
    {
      this.unregisterCrawlServerEventer.notify(this.ip, this.port, new
UnregisterCrawlServerNotification(NodeID.DISTRIBUTED().getKey()));
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
  }

  /*
   * Send the crawled links to the coordinator concurrently. 11/23/2014, Bing Li
   */
  public void sendCrawledLinks(Set<CrawledLink> links)
  {
    this.crawledLinkEventer.notify(this.ip, this.port, new CrawledLinksNotification(links));
  }
}
```

- **CrawlerMulticastor**

```
package com.greatfree.testing.crawlserver;

import java.io.IOException;

import com.greatfree.reuse.ResourcePool;
import com.greatfree.testing.data.ClientConfig;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.StartCrawlMultiNotification;
import com.greatfree.testing.message.StopCrawlMultiNotification;

/*
 * This is the multicasting mechanism to manage all of the multicastors to transfer notifications to its children crawlers. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class CrawlerMulticastor
{
   // The pool for the children multicastor which multicasts the notification of StartCrawlMultiNotification. 11/27/2014, Bing Li
   private ResourcePool<StartCrawlNotificationMulticastorSource, StartCrawlNotificationMulticastor, StartCrawlNotificationMulticastorCreator, StartCrawlNotificationMulticastorDisposer> startCrawlMulticastorPool;
   // The pool for the children multicastor which multicasts the notification of StopCrawlMultiNotification. 11/27/2014, Bing Li
   private ResourcePool<StopCrawlNotificationMulticastorSource, StopCrawlNotificationMulticastor, StopCrawlNotificationMulticastorCreator, StopCrawlNotificationMulticastorDisposer> stopCrawlMulticastorPool;

   private CrawlerMulticastor()
   {
   }

   /*
    * A singleton implementation. 11/27/2014, Bing Li
    */
   private static CrawlerMulticastor instance = new CrawlerMulticastor();

   public static CrawlerMulticastor CRAWLER()
   {
      if (instance == null)
      {
         instance = new CrawlerMulticastor();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Dispose all of the pools. 11/27/2014, Bing Li
    */
   public void dispose()
   {
      this.startCrawlMulticastorPool.shutdown();
      this.stopCrawlMulticastorPool.shutdown();
   }

   /*
    * Initialize the pools. 11/27/2014, Bing Li
    */
   public void init()
```

```
  {
    this.startCrawlMulticastorPool = new ResourcePool<StartCrawlNotificationMulticastorSource,
StartCrawlNotificationMulticastor, StartCrawlNotificationMulticastorCreator,
StartCrawlNotificationMulticastorDisposer>(ClientConfig.MULTICASTOR_POOL_SIZE, new
StartCrawlNotificationMulticastorCreator(), new StartCrawlNotificationMulticastorDisposer(),
ClientConfig.MULTICASTOR_WAIT_TIME);
    this.stopCrawlMulticastorPool = new ResourcePool<StopCrawlNotificationMulticastorSource,
StopCrawlNotificationMulticastor, StopCrawlNotificationMulticastorCreator,
StopCrawlNotificationMulticastorDisposer>(ClientConfig.MULTICASTOR_POOL_SIZE, new
StopCrawlNotificationMulticastorCreator(), new StopCrawlNotificationMulticastorDisposer(),
ClientConfig.MULTICASTOR_WAIT_TIME);
  }

  /*
   * Disseminate the notification of StartCrawlMultiNotification to the children crawlers of the local one.
11/27/2014, Bing Li
   */
  public void disseminateStartCrawlNotificationAmongSubCrawlServers(StartCrawlMultiNotification
notification) throws InstantiationException, IllegalAccessException, IOException, InterruptedException
  {
    // Get an instance of StartCrawlNotificationMulticastor from the pool. 11/27/2014, Bing Li
    StartCrawlNotificationMulticastor notifier = this.startCrawlMulticastorPool.get(new
StartCrawlNotificationMulticastorSource(SubClientPool.CRAWL().getPool(),
ClientConfig.MULTICAST_BRANCH_COUNT, ServerConfig.CRAWL_SERVER_PORT, new
StartCrawlMultiNotificationCreator()));
    // Check whether the notifier is valid. 11/26/2014, Bing Li
    if (notifier != null)
    {
      // Disseminate the notification. 11/27/2014, Bing Li
      notifier.disseminate(notification);
      // Collect the instance of StartCrawlNotificationMulticastor. 11/27/2014, Bing Li
      this.startCrawlMulticastorPool.collect(notifier);
    }
  }

  /*
   * Disseminate the notification of StartCrawlMultiNotification to the children crawlers of the local one.
11/27/2014, Bing Li
   */
  public void disseminateStopCrawlNotificationAmongSubCrawlServers(StopCrawlMultiNotification
notification) throws InstantiationException, IllegalAccessException, IOException, InterruptedException
  {
    // Get an instance of StartCrawlNotificationMulticastor from the pool. 11/27/2014, Bing Li
    StopCrawlNotificationMulticastor notifier = this.stopCrawlMulticastorPool.get(new
StopCrawlNotificationMulticastorSource(SubClientPool.CRAWL().getPool(),
ClientConfig.MULTICAST_BRANCH_COUNT, ServerConfig.CRAWL_SERVER_PORT, new
StopCrawlMultiNotificationCreator()));
    // Check whether the notifier is valid. 11/26/2014, Bing Li
    if (notifier != null)
    {
      // Disseminate the notification. 11/27/2014, Bing Li
      notifier.disseminate(notification);
      // Collect the instance of Stop`CrawlNotificationMulticastor. 11/27/2014, Bing Li
      this.stopCrawlMulticastorPool.collect(notifier);
    }
  }
}
```

- **StartCrawlMultiNotificationCreator**

```
package com.greatfree.testing.crawlserver;

import java.util.HashMap;

import com.greatfree.multicast.ChildMulticastMessageCreatable;
import com.greatfree.testing.message.StartCrawlMultiNotification;
import com.greatfree.util.Tools;

/*
 * The creator generates the notifications of StartCrawlMultiNotification that should be multicast to the
local crawler's children. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StartCrawlMultiNotificationCreator implements
ChildMulticastMessageCreatable<StartCrawlMultiNotification>
{
  @Override
  public StartCrawlMultiNotification createInstanceWithChildren(StartCrawlMultiNotification msg,
HashMap<String, String> children)
  {
    return new StartCrawlMultiNotification(Tools.generateUniqueKey(), children);
  }
}
```

- **StartCrawlNotificationMulticastorSource**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.multicast.ChildMessageCreatorGettable;
import com.greatfree.multicast.ChildMulticastorSource;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.message.StartCrawlMultiNotification;

/*
 * The sources that are needed to create an instance of ChildMulticastor are enclosed in the class. That is required by the pool to create children multicastors. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StartCrawlNotificationMulticastorSource extends ChildMulticastorSource<StartCrawlMultiNotification, StartCrawlMultiNotificationCreator> implements ChildMessageCreatorGettable<StartCrawlMultiNotification>
{
  /*
   * Initialize the source. 11/27/2014, Bing Li
   */
  public StartCrawlNotificationMulticastorSource(FreeClientPool clientPool, int treeBranchCount, int serverPort, StartCrawlMultiNotificationCreator creator)
  {
    super(clientPool, treeBranchCount, serverPort, creator);
  }

  /*
   * Expose the message creator. 11/27/2014, Bing Li
   */
  @Override
  public StartCrawlMultiNotificationCreator getMessageCreator()
  {
    return super.getMessageCreator();
  }
}
```

- **StartCrawlNotificationMulticastor**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.multicast.ChildMulticastor;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.message.StartCrawlMultiNotification;

/*
 * The multicastor is derived from ChildMulticastor to transfer the notification, StartCrawlMultiNotification,
to the children nodes. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StartCrawlNotificationMulticastor extends ChildMulticastor<StartCrawlMultiNotification,
StartCrawlMultiNotificationCreator>
{
   public StartCrawlNotificationMulticastor(FreeClientPool clientPool, int treeBranchCount, int
clusterServerPort, StartCrawlMultiNotificationCreator messageCreator)
   {
      super(clientPool, treeBranchCount, clusterServerPort, messageCreator);
   }
}
```

- **StartCrawlNotificationMulticastorCreator**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.reuse.HashCreatable;

/*
 * The class intends to create the interface of StartCrawlNotificationMulticastor. It is used by the resource pool to manage the children multicastors efficiently. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StartCrawlNotificationMulticastorCreator implements HashCreatable<StartCrawlNotificationMulticastorSource, StartCrawlNotificationMulticastor>
{
	/*
	 * Define the method to create the instances of StartCrawlNotificationMulticastor upon the source, StartCrawlNotificationMulticastorSource. 11/27/2014, Bing Li
	 */
	@Override
	public StartCrawlNotificationMulticastor createResourceInstance(StartCrawlNotificationMulticastorSource source)
	{
		return new StartCrawlNotificationMulticastor(source.getClientPool(), source.getTreeBranchCount(), source.getServerPort(), source.getMessageCreator());
	}
}
```

- **StartCrawlNotificationMulticastorDisposer**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.reuse.HashDisposable;

/*
 * The disposer collects the instance of StartCrawlNotificationMulticastor. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StartCrawlNotificationMulticastorDisposer implements
HashDisposable<StartCrawlNotificationMulticastor>
{
   @Override
   public void dispose(StartCrawlNotificationMulticastor t)
   {
      t.dispose();
   }
}
```

- **StopCrawlMultiNotificationCreator**

```
package com.greatfree.testing.crawlserver;

import java.util.HashMap;

import com.greatfree.multicast.ChildMulticastMessageCreatable;
import com.greatfree.testing.message.StopCrawlMultiNotification;
import com.greatfree.util.Tools;

/*
 * The creator generates the notifications of StopCrawlMultiNotification that should be multicast to the
local crawler's children. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StopCrawlMultiNotificationCreator implements
ChildMulticastMessageCreatable<StopCrawlMultiNotification>
{
  @Override
  public StopCrawlMultiNotification createInstanceWithChildren(StopCrawlMultiNotification msg,
HashMap<String, String> children)
  {
    return new StopCrawlMultiNotification(Tools.generateUniqueKey(), children);
  }
}
```

- **StopCrawlNotificationMulticastorSource**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.multicast.ChildMessageCreatorGettable;
import com.greatfree.multicast.ChildMulticastorSource;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.message.StopCrawlMultiNotification;

/*
 * The sources that are needed to create an instance of ChildMulticastor are enclosed in the class. That
is required by the pool to create children multicastors. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StopCrawlNotificationMulticastorSource extends
ChildMulticastorSource<StopCrawlMultiNotification, StopCrawlMultiNotificationCreator> implements
ChildMessageCreatorGettable<StopCrawlMultiNotification>
{
  /*
   * Initialize the source. 11/27/2014, Bing Li
   */
  public StopCrawlNotificationMulticastorSource(FreeClientPool clientPool, int treeBranchCount, int
serverPort, StopCrawlMultiNotificationCreator creator)
  {
    super(clientPool, treeBranchCount, serverPort, creator);
  }

  /*
   * Expose the message creator. 11/27/2014, Bing Li
   */
  @Override
  public StopCrawlMultiNotificationCreator getMessageCreator()
  {
    return super.getMessageCreator();
  }
}
```

- **StopCrawlNotificationMulticastor**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.multicast.ChildMulticastor;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.message.StopCrawlMultiNotification;

/*
 * The multicastor is derived from ChildMulticastor to transfer the notification, StopCrawlMultiNotification,
to the children nodes. 11/27/2014, Bing Li
 */

// Created; 11/27/2014, Bing Li
public class StopCrawlNotificationMulticastor extends ChildMulticastor<StopCrawlMultiNotification,
StopCrawlMultiNotificationCreator>
{
   public StopCrawlNotificationMulticastor(FreeClientPool clientPool, int treeBranchCount, int
clusterServerPort, StopCrawlMultiNotificationCreator messageCreator)
   {
      super(clientPool, treeBranchCount, clusterServerPort, messageCreator);
   }
}
```

- **StopCrawlNotificationMulticastorCreator**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.reuse.HashCreatable;

/*
 * The class intends to create the interface of StopCrawlNotificationMulticastor. It is used by the resource pool to manage the children multicastors efficiently. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StopCrawlNotificationMulticastorCreator implements HashCreatable<StopCrawlNotificationMulticastorSource, StopCrawlNotificationMulticastor>
{
	/*
	 * Define the method to create the instances of StartCrawlNotificationMulticastor upon the source, StopCrawlNotificationMulticastorCreator. 11/27/2014, Bing Li
	 */
	@Override
	public StopCrawlNotificationMulticastor createResourceInstance(StopCrawlNotificationMulticastorSource source)
	{
		return new StopCrawlNotificationMulticastor(source.getClientPool(), source.getTreeBranchCount(), source.getServerPort(), source.getMessageCreator());
	}
}
```

- **StopCrawlNotificationMulticastorDisposer**

```
package com.greatfree.testing.crawlserver;

import com.greatfree.reuse.HashDisposable;

/*
 * The disposer collects the instance of StopCrawlNotificationMulticastor. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StopCrawlNotificationMulticastorDisposer implements
HashDisposable<StopCrawlNotificationMulticastor>
{
   @Override
   public void dispose(StopCrawlNotificationMulticastor t)
   {
      t.dispose();
   }
}
```

- **CrawlConfig**

```
package com.greatfree.testing.crawlserver;

import java.util.Set;

import com.greatfree.testing.data.CrawledLink;
import com.greatfree.testing.db.DBConfig;

/*
 * The class contains the constants and configurations for the crawler. 11/28/2014, Bing Li
 */

// Created: 11/11/2014, Bing Li
public class CrawlConfig
{
  public final static int CRAWLING_TASK_LISTENER_THREAD_POOL_SIZE = 100;
  public final static long CRAWLING_TASK_LISTENER_THREAD_ALIVE_TIME = 10000;

  public final static HubURL NO_URL = null;
  public final static String NO_URL_LINK = "";

  public final static long LOCK_TIME_OUT = 0;

  public final static int CRAWLING_TIMER_PERIOD = 5;
  public final static int UPDATING_VALUE = 5;
  public final static int MIN_UPDATING_PERIOD = 1000;

  public final static String CRAWLING_DB_ROOT = DBConfig.DB_HOME + "CrawlDB/";
  public final static String CRAWLED_FILE_PATH = CrawlConfig.CRAWLING_DB_ROOT +
"HubCrawledFiles/";

  public final static Set<CrawledLink> NO_LINK_KEYS = null;

  public final static int CRAW_TIMEOUT = 50000;

  public final static String TAG_A = "a";
  public final static String HREF = "href";

  public final static int CRAWLER_FAST_POOL_SIZE = 320;
  public final static int CRAWLER_SLOW_POOL_SIZE = 30;
  public final static int CRAWLER_THREAD_TASK_SIZE = 5;
  public final static long CRAWLER_THREAD_IDLE_TIME = 30000;
  public final static long CRAWLER_IDLE_CHECK_DELAY = 5000;
  public final static long CRAWLER_IDLE_CHECK_PERIOD = 10000;

  public final static CrawlThread NO_CRAWL_THREAD = null;

  public final static int IDLE = 1;
  public final static int BUSY = 0;
  public final static int DEAD = -1;

  public final static long CRAWL_SCHEDULER_WAIT_TIME = 2000;

  public final static int SUB_CLIENT_POOL_SIZE = 500;
  public final static long SUB_CLIENT_IDLE_CHECK_DELAY = 3000;
  public final static long SUB_CLIENT_IDLE_CHECK_PERIOD = 3000;
  public final static long SUB_CLIENT_MAX_IDLE_TIME = 3000;

  public final static int MAX_BUSY_TIME_LENGTH = 200;

  public final static int EMPTY = 1;
  public final static int NOT_EMPTY = 0;

  public final static long CRAWLING_STATE_CHECK_PERIOD = 2000;
}
```

3.6.4 The Cluster of Memory Servers

- **StartMemoryServer**

```
package com.greatfree.testing.memory;

import com.greatfree.testing.data.ServerConfig;
import com.greatfree.util.TerminateSignal;

/*
 * This is the entry and exit for the memory server. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class StartMemoryServer
{
  public static void main(String[] args)
  {
    // Start the memory server. 11/28/2014, Bing Li
    MemoryServer.STORE().start(ServerConfig.MEMORY_SERVER_PORT);
    // Detect whether the process is shutdown. 11/28/2014, Bing Li
    while (!TerminateSignal.SIGNAL().isTerminated())
    {
      try
      {
        // Sleep for some time if the process is not shutdown. 11/28/2014, Bing Li
        Thread.sleep(ServerConfig.TERMINATE_SLEEP);
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **MemoryServer**

```
package com.greatfree.testing.memory;

import java.io.IOException;
import java.net.ServerSocket;
import java.util.ArrayList;
import java.util.List;

import com.greatfree.concurrency.Runner;
import com.greatfree.testing.data.ClientConfig;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.util.TerminateSignal;
import com.greatfree.util.UtilConfig;

/*
 * This is the server to save a large amount of data into its storage system. It is one node to take one portion of the data. In the sample, the data is saved in the memory for fast accessing. Developers can follow it to keep a large amount of data either in a persistent or in a volatile way. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class MemoryServer
{
  private ServerSocket serverSocket;
  private int serverPort;

  private List<Runner<MemServerListener, MemServerListenerDisposer>> listeners;

  private MemoryServer()
  {
  }

  /*
   * A singleton implementation since this is the unique entry and exit of the crawler. 11/24/2014, Bing Li
   */
  private static MemoryServer instance = new MemoryServer();

  public static MemoryServer STORE()
  {
    if (instance == null)
    {
      instance = new MemoryServer();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  public void start(int port)
  {
    // On JD7, the sorting algorithm is replaced with TimSort rather than MargeSort. To run correctly, it is necessary to use the old one. the following line sets that up. 11/28/2014, Bing Li
    System.setProperty(UtilConfig.MERGE_SORT, UtilConfig.TRUE);

    // Assign the memory server port. 11/28/2014, Bing Li
    this.serverPort = port;
    try
    {
      // Initialize the socket of the crawler. 11/28/2014, Bing Li
      this.serverSocket = new ServerSocket(this.serverPort);
    }
    catch (IOException e)
    {
```

```
            e.printStackTrace();
        }

        // Initialize a list to contain the listeners. 11/28/2014, Bing Li
        this.listeners = new ArrayList<Runner<MemServerListener, MemServerListenerDisposer>>();
        // Initialize a disposer that collects the listeners. 11/28/2014, Bing Li
        MemServerListenerDisposer disposer = new MemServerListenerDisposer();
        // The runner contains the listener and listener disposer to start the listeners concurrently.
11/28/2014, Bing Li
        Runner<MemServerListener, MemServerListenerDisposer> runner;
        // Initialize and start a certain number of listeners concurrently. 11/28/2014, Bing Li
        for (int i = 0; i < ServerConfig.MAX_CLIENT_LISTEN_THREAD_COUNT; i++)
        {
            // Initialize the runner that contains the listener and its disposer. 11/24/2014, Bing Li
            runner = new Runner<MemServerListener, MemServerListenerDisposer>(new
MemServerListener(this.serverSocket), disposer, true);
            // Put the runner into a list for management. 11/24/2014, Bing Li
            this.listeners.add(runner);
            // Start up the runner. 11/24/2014, Bing Li
            runner.start();
        }

        // Initialize the memory server IO registry. 11/28/2014, Bing Li
        MemoryIORegistry.REGISTRY().init();
        // Initialize the client pool that is used to connect the coordinator. 11/28/2014, Bing Li
        ClientPool.STORE().init();
        // Initialize the sub client pool that is used to connect the children of the memory server. 11/28/2014,
Bing Li
        SubClientPool.STORE().init();
        // Initialize the memory eventer that sends notifications to the coordinator. For example, the crawled
links are sent to the coordinator asynchronously by the eventer. 11/28/2014, Bing Li
        MemoryEventer.NOTIFY().init(ServerConfig.COORDINATOR_ADDRESS,
ServerConfig.COORDINATOR_PORT_FOR_MEMORY);
        // Initialize the message producer which dispatches received requests and notifications from the
coordinator. 11/28/2014, Bing Li
        MemoryMessageProducer.STORE().Init();

        // Initialize the memory. 11/28/2014, Bing Li
        LinkPond.STORE().init();

        // Start the multicastor. 11/29/2014, Bing Li
        MemoryMulticastor.STORE().init();

        try
        {
            // Notify the coordinator that the memory server is online. 11/28/2014, Bing Li
            MemoryEventer.NOTIFY().notifyOnline();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }

    /*
     * Stop the crawler. 11/28/2014, Bing Li
     */
    public void stop() throws InterruptedException, IOException
    {
        // Set the terminating signal. The long time running task needs to be interrupted when the signal is
set. 11/28/2014, Bing Li
        TerminateSignal.SIGNAL().setTerminated();

        // Stop each listener one by one. 11/28/2014, Bing Li
```

```
        for (Runner<MemServerListener, MemServerListenerDisposer> runner : this.listeners)
        {
            runner.stop(ClientConfig.TIME_TO_WAIT_FOR_THREAD_TO_DIE);
        }

        // Close the socket of the memory server. 11/28/2014, Bing Li
        this.serverSocket.close();

        // Dispose the memory. 11/28/2014, Bing Li
        LinkPond.STORE().dispose();

        // Unregister the memory eventer. 11/28/2014, Bing Li
        MemoryEventer.NOTIFY().unregister();
        // Dispose the memory eventer. 11/28/2014, Bing Li
        MemoryEventer.NOTIFY().dispose();

        // Dispose the message producer. 11/28/2014, Bing Li
        MemoryMessageProducer.STORE().dispose();
        // Shutdown the memory server IOs. 11/28/2014, Bing Li
        MemoryIORegistry.REGISTRY().dispose();
        // Dispose the client pool. 11/28/2014, Bing Li
        ClientPool.STORE().dispose();
        // Dispose the sub client pool/ 11/28/2014, Bing Li
        SubClientPool.STORE().dispose();
        // Shutdown the multicastor. 11/29/2014, Bing Li
        MemoryMulticastor.STORE().dispose();
    }
}
```

- **MemServerListener**

```
package com.greatfree.testing.memory;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

import com.greatfree.remote.ServerListener;
import com.greatfree.testing.data.ServerConfig;

/*
 * The class is a server listener that waits for the coordinator to distribute data. The design is the same as the general server listener. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class MemServerListener extends ServerListener implements Runnable
{
  /*
   * Initialize the listener. 11/27/2014, Bing Li
   */
  public MemServerListener(ServerSocket serverSocket)
  {
    super(serverSocket, MemConfig.MEMORY_LISTENER_THREAD_POOL_SIZE, MemConfig.MEMORY_LISTENER_THREAD_ALIVE_TIME);
  }

  /*
   * Waiting for connection concurrently. The connection request is invoked by a coordinator, which sends data storing/retrieving tasks and other management notifications. 11/27/2014, Bing Li
   */
  @Override
  public void run()
  {
    Socket clientSocket;
    MemoryIO serverIO;
    // Detect whether the listener is shutdown. If not, it must be running all the time to wait for potential connections from clients. 11/27/2014, Bing Li
    while (!super.isShutdown())
    {
      try
      {
        // Wait and accept a connecting from a possible client residing on the coordinator. 11/27/2014, Bing Li
        clientSocket = super.accept();
        // Check whether the connected server IOs exceed the upper limit. 11/27/2014, Bing Li
        if (MemoryIORegistry.REGISTRY().getIOCount() >= ServerConfig.MAX_SERVER_IO_COUNT)
        {
          try
          {
            // If the upper limit is reached, the listener has to wait until an existing server IO is disposed. 11/27/2014, Bing Li
            super.holdOn();
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
        }
        // If the upper limit of IOs is not reached, a server IO is initialized. A common Collaborator and the socket are the initial parameters. The shared common collaborator guarantees all of the server IOs from a certain client could notify with each other with the same lock. Then, the upper limit of server IOs is under the control. 11/27/2014, Bing Li
        serverIO = new MemoryIO(clientSocket, super.getCollaborator());
        // Add the new created server IO into the registry for further management. 11/27/2014, Bing Li
```

```
            MemoryIORegistry.REGISTRY().addIO(serverIO);
            // Execute the new created server IO concurrently to respond the client requests in an
asynchronous manner. 11/27/2014, Bing Li
            super.execute(serverIO);
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
        }
      }
    }
```

- **MemServerListenerDisposer**

```
package com.greatfree.testing.memory;

import com.greatfree.reuse.RunDisposable;

/*
 * The class is responsible for disposing the instance of MemServerListener by invoking its method of shutdown(). It works with the thread container, Threader or Runner. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class MemServerListenerDisposer implements RunDisposable<MemServerListener>
{
  /*
   * Dispose the instance of CrawlingListener. 11/28/2014, Bing Li
   */
  @Override
  public void dispose(MemServerListener r)
  {
    r.shutdown();
  }

  /*
   * Dispose the instance of MemServerListener. The method does not make sense to MemServerListener. Just leave it here for the requirement of the interface, RunDisposable<MemServerListener>. 11/28/2014, Bing Li
   */
  @Override
  public void dispose(MemServerListener r, long time)
  {
    r.shutdown();
  }
}
```

- **MemoryIO**

```
package com.greatfree.testing.memory;

import java.io.IOException;
import java.net.Socket;
import java.net.SocketException;

import com.greatfree.concurrency.Collaborator;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.remote.ServerIO;

/*
 * The class receives requests/notifications from the coordinator to accomplish the task of data storing and even retrieving. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class MemoryIO extends ServerIO
{
  /*
   * Initialize the server IO. The socket is the connection between the coordinator and the memory server. The collaborator is shared with other IOs to control the count of ServerIOs instances. 11/27/2014, Bing Li
   */
  public MemoryIO(Socket clientSocket, Collaborator collaborator)
  {
    super(clientSocket, collaborator);
  }

  public void run()
  {
    ServerMessage message;
    while (!super.isShutdown())
    {
      try
      {
        // Wait and read messages from the coordinator. 11/27/2014, Bing Li
        message = (ServerMessage)super.read();
        // Convert the received message to OutMessageStream and put it into the relevant dispatcher for concurrent processing. 11/27/2014, Bing Li
        MemoryMessageProducer.STORE().produceMessage(new OutMessageStream<ServerMessage>(super.getOutStream(), super.getLock(), message));
      }
      catch (SocketException e)
      {
        e.printStackTrace();
      }
      catch (IOException e)
      {
        e.printStackTrace();
      }
      catch (ClassNotFoundException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **MemoryIORegistry**

```
package com.greatfree.testing.memory;

import java.io.IOException;
import java.util.Set;

import com.greatfree.remote.ServerIORegistry;

/*
 * The class keeps all of MemoryIOs. This is a singleton wrapper of ServerIORegistry. 11/27/2014, Bing
Li
 */

// Created: 11/27/2014, Bing Li
public class MemoryIORegistry
{
   // Declare an instance of ServerIORegistry for MemoryIOs. 11/27/2014, Bing Li
   private ServerIORegistry<MemoryIO> registry;

   /*
    * Initializing ... 11/27/2014, Bing Li
    */
   private MemoryIORegistry()
   {
   }

   private static MemoryIORegistry instance = new MemoryIORegistry();

   public static MemoryIORegistry REGISTRY()
   {
      if (instance == null)
      {
         instance = new MemoryIORegistry();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Dispose the registry. 11/27/2014, Bing Li
    */
   public void dispose() throws IOException
   {
      this.registry.removeAllIOs();
   }

   /*
    * Initialize the registry. 11/27/2014, Bing Li
    */
   public void init()
   {
      this.registry = new ServerIORegistry<MemoryIO>();
   }

   /*
    * Add a new instance of MemoryIO to the registry. 11/27/2014, Bing Li
    */
   public void addIO(MemoryIO io)
   {
      this.registry.addIO(io);
   }
```

```
  /*
   * Get all of the IPs of the connected clients from the corresponding MemoryIOs. 11/27/2014, Bing Li
   */
  public Set<String> getIPs()
  {
    return this.registry.getIPs();
  }

  /*
   * Get the count of the registered MemoryIOs. 11/27/2014, Bing Li
   */
  public int getIOCount()
  {
    return this.registry.getIOCount();
  }

  /*
   * Remove or unregister a MemoryIO. It is executed when a client is down or the connection gets
something wrong. 11/27/2014, Bing Li
   */
  public void removeIO(MemoryIO io) throws IOException
  {
    this.registry.removeIO(io);
  }

  /*
   * Remove or unregister all of the registered MemoryIOs. It is executed when the server process is
shutdown. 11/27/2014, Bing Li
   */
  public void removeAllIOs() throws IOException
  {
    this.registry.removeAllIOs();
  }
}
```

- **ClientPool**

```
package com.greatfree.testing.memory;

import java.io.IOException;

import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.data.ClientConfig;

/*
 * The pool is responsible for creating and managing instance of FreeClient to achieve the goal of using as small number of instances of FreeClient to send messages to the coordinator in a high performance. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class ClientPool
{
  // An instance of FreeClientPool is defined to interact with the coordinator. 11/28/2014, Bing Li
  private FreeClientPool pool;

  private ClientPool()
  {
  }

  /*
   * A singleton definition. 11/28/2014, Bing Li
   */
  private static ClientPool instance = new ClientPool();

  public static ClientPool STORE()
  {
    if (instance == null)
    {
      instance = new ClientPool();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the free client pool. 11/28/2014, Bing Li
   */
  public void dispose() throws IOException
  {
    this.pool.dispose();
  }

  /*
   * Initialize the client pool. The method is called when the crawler process is started. 11/28/2014, Bing Li
   */
  public void init()
  {
    // Initialize the client pool. 11/28/2014, Bing Li
    this.pool = new FreeClientPool(ClientConfig.CLIENT_POOL_SIZE);
    // Set idle checking for the client pool. 11/28/2014, Bing Li
    this.pool.setIdleChecker(ClientConfig.CLIENT_IDLE_CHECK_DELAY, ClientConfig.CLIENT_IDLE_CHECK_PERIOD, ClientConfig.CLIENT_MAX_IDLE_TIME);
  }

  /*
   * Expose the client pool. 11/28/2014, Bing Li
```

```
     */
    public FreeClientPool getPool()
    {
      return this.pool;
    }
}
```

- **SubClientPool**

```
package com.greatfree.testing.memory;

import java.io.IOException;

import com.greatfree.remote.FreeClientPool;

/*
 * The pool is responsible for creating and managing instance of FreeClient to achieve the goal of using as small number of instances of FreeClient to send messages to the children memory servers in a high performance. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class SubClientPool
{
   // An instance of FreeClientPool is defined to interact with the children memory servers. 11/28/2014, Bing Li
   private FreeClientPool clientPool;

   private SubClientPool()
   {
   }

   /*
    * A singleton definition. 11/28/2014, Bing Li
    */
   private static SubClientPool instance = new SubClientPool();

   public static SubClientPool STORE()
   {
      if (instance == null)
      {
         instance = new SubClientPool();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Dispose the free client pool. 11/28/2014, Bing Li
    */
   public void dispose()
   {
      try
      {
         this.clientPool.dispose();
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
   }

   /*
    * Initialize the client pool. The method is called when the memory server process is started. 11/28/2014, Bing Li
    */
   public void init()
   {
      // Initialize the client pool. 11/28/2014, Bing Li
      this.clientPool = new FreeClientPool(MemConfig.SUB_CLIENT_POOL_SIZE);
```

```
    this.clientPool.setIdleChecker(MemConfig.SUB_CLIENT_IDLE_CHECK_DELAY,
MemConfig.SUB_CLIENT_IDLE_CHECK_PERIOD, MemConfig.SUB_CLIENT_MAX_IDLE_TIME);
  }

  /*
   * Expose the client pool. 11/28/2014, Bing Li
   */
  public FreeClientPool getPool()
  {
    return this.clientPool;
  }
}
```

- **MemoryEventer**

```
package com.greatfree.testing.memory;

import java.io.IOException;

import com.greatfree.concurrency.ThreadPool;
import com.greatfree.remote.SyncRemoteEventer;
import com.greatfree.testing.data.ClientConfig;
import com.greatfree.testing.message.OnlineNotification;
import com.greatfree.testing.message.RegisterMemoryServerNotification;
import com.greatfree.testing.message.UnregisterMemoryServerNotification;
import com.greatfree.util.NodeID;

/*
 * This is an eventer that sends notifications to the coordinator in a synchronous or asynchronous
manner. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class MemoryEventer
{
   // The IP of the coordinator the eventer needs to notify. 11/28/2014, Bing Li
   private String ip;
   // The port of the coordinator the eventer needs to notify. 11/28/2014, Bing Li
   private int port;
   // The thread pool that starts up the asynchronous eventer. 11/28/2014, Bing Li
   private ThreadPool pool;
   // The synchronous eventer notifies the coordinator that a crawler is online. After receiving the
notification, the coordinator can assign tasks and interact with the crawler for crawling. 11/28/2014, Bing
Li
   private SyncRemoteEventer<OnlineNotification> onlineNotificationEventer;
   // The synchronous eventer notifies the coordinator that a memory server needs to register.
11/28/2014, Bing Li
   private SyncRemoteEventer<RegisterMemoryServerNotification> registerMemoryServerEventer;
   // The synchronous eventer notifies the coordinator that a memory server needs to unregister.
11/28/2014, Bing Li
   private SyncRemoteEventer<UnregisterMemoryServerNotification>
unregisterMemoryServerEventer;

   private MemoryEventer()
   {
   }

   /*
    * Initialize a singleton. 11/28/2014, Bing Li
    */
   private static MemoryEventer instance = new MemoryEventer();

   public static MemoryEventer NOTIFY()
   {
      if (instance == null)
      {
         instance = new MemoryEventer();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Dispose the eventer. 11/28/2014, Bing Li
    */
   public void dispose()
```

```
  {
    // Dispose the online eventer. 11/28/2014, Bing Li
    this.onlineNotificationEventer.dispose();
    // Dispose the registering eventer. 11/28/2014, Bing Li
    this.registerMemoryServerEventer.dispose();
    // Dispose the unregistering eventer. 11/28/2014, Bing Li
    this.unregisterMemoryServerEventer.dispose();
    // Shutdown the thread pool. 11/28/2014, Bing Li
    this.pool.shutdown();
  }

  /*
   * Initialize the eventer. The IP/port is the coordinator to be notified. 11/28/2014, Bing Li
   */
  public void init(String ip, int port)
  {
    // Assign the IP. 11/28/2014, Bing Li
    this.ip = ip;
    // Assign the port. 11/28/2014, Bing Li
    this.port = port;
    // Initialize a thread pool. 11/28/2014, Bing Li
    this.pool = new ThreadPool(ClientConfig.EVENTER_THREAD_POOL_SIZE,
ClientConfig.EVENTER_THREAD_POOL_ALIVE_TIME);

    this.onlineNotificationEventer = new
SyncRemoteEventer<OnlineNotification>(ClientPool.STORE().getPool());
    this.registerMemoryServerEventer = new
SyncRemoteEventer<RegisterMemoryServerNotification>(ClientPool.STORE().getPool());
    this.unregisterMemoryServerEventer = new
SyncRemoteEventer<UnregisterMemoryServerNotification>(ClientPool.STORE().getPool());
  }

  /*
   * Notify the coordinator that the memory server is online. 11/28/2014, Bing Li
   */
  public void notifyOnline() throws IOException, InterruptedException
  {
    this.onlineNotificationEventer.notify(this.ip, this.port, new OnlineNotification());
  }

  /*
   * Register the memory server. 11/28/2014, Bing Li
   */
  public void register()
  {
    try
    {
      this.registerMemoryServerEventer.notify(this.ip, this.port, new
RegisterMemoryServerNotification(NodeID.DISTRIBUTED().getKey()));
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
  }

  /*
   * Unregister the memory server. 11/23/2014, Bing Li
   */
  public void unregister()
  {
    try
    {
      this.unregisterMemoryServerEventer.notify(this.ip, this.port, new
```

```
UnregisterMemoryServerNotification(NodeID.DISTRIBUTED().getKey()));
      }
      catch (IOException e)
      {
        e.printStackTrace();
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
}
```

- **MemoryMessageProducer**

```
package com.greatfree.testing.memory;

import com.greatfree.concurrency.MessageProducer;
import com.greatfree.concurrency.Threader;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.testing.data.ServerConfig;

/*
 * The class is a singleton to enclose the instances of MessageProducer. Each of the enclosed message producers serves for one particular client that connects to a respective port on the memory server. Usually, each port aims to provide one particular service. 11/27/2014, Bing Li
 *
 * The class is a wrapper that encloses all of the asynchronous message producers. It is responsible for assigning received messages to the corresponding producer in an asynchronous way. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class MemoryMessageProducer
{
  private Threader<MessageProducer<MemoryDispatcher>, MemoryMessageProducerDisposer> producerThreader;

  private MemoryMessageProducer()
  {
  }

  /*
   * The class is required to be a singleton since it is nonsense to initiate it for the producers are unique. 11/27/2014, Bing Li
   */
  private static MemoryMessageProducer instance = new MemoryMessageProducer();

  public static MemoryMessageProducer STORE()
  {
    if (instance == null)
    {
      instance = new MemoryMessageProducer();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the producers when the process of the server is shutdown. 11/27/2014, Bing Li
   */
  public void dispose() throws InterruptedException
  {
    this.producerThreader.stop();
  }

  /*
   * Initialize the message producers. It is invoked when the connection modules of the server is started since clients can send requests or notifications only after it is started. 11/27/2014, Bing Li
   */
  public void Init()
  {
    // Initialize the memory message producer. A threader is associated with the message producer such that the producer is able to work in a concurrent way. 11/27/2014, Bing Li
    this.producerThreader = new Threader<MessageProducer<MemoryDispatcher>,
```

```
MemoryMessageProducerDisposer>(new MessageProducer<MemoryDispatcher>(new
MemoryDispatcher(ServerConfig.DISPATCHER_POOL_SIZE,
ServerConfig.DISPATCHER_POOL_THREAD_POOL_ALIVE_TIME)), new
MemoryMessageProducerDisposer());
      // Start the associated thread for the crawling message producer. 11/27/2014, Bing Li
      this.producerThreader.start();
    }

    /*
     * Assign messages, requests or notifications, to the bound crawling message dispatcher such that
they can be responded or dealt with. 11/27/2014, Bing Li
     */
    public void produceMessage(OutMessageStream<ServerMessage> message)
    {
      this.producerThreader.getFunction().produce(message);
    }
}
```

- **MemoryDispatcher**

```
package com.greatfree.testing.memory;

import com.greatfree.concurrency.AnycastRequestDispatcher;
import com.greatfree.concurrency.BoundBroadcastRequestDispatcher;
import com.greatfree.concurrency.BoundNotificationDispatcher;
import com.greatfree.concurrency.NotificationDispatcher;
import com.greatfree.concurrency.ServerMessageDispatcher;
import com.greatfree.multicast.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.reuse.MulticastMessageDisposer;
import com.greatfree.testing.data.ClientConfig;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.AddCrawledLinkNotification;
import com.greatfree.testing.message.IsPublisherExistedAnycastRequest;
import com.greatfree.testing.message.IsPublisherExistedAnycastResponse;
import com.greatfree.testing.message.MessageType;
import com.greatfree.testing.message.NodeKeyNotification;
import com.greatfree.testing.message.SearchKeywordBroadcastRequest;
import com.greatfree.testing.message.SearchKeywordBroadcastResponse;

/*
 * This is an implementation of ServerMessageDispatcher. It contains the concurrency mechanism to respond the coordinator's requests and notifications for the memory server. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class MemoryDispatcher extends ServerMessageDispatcher<ServerMessage>
{
  // Declare a instance of notification dispatcher to deal with received the notification that contains the node key. 11/28/2014, Bing Li
  private NotificationDispatcher<NodeKeyNotification, RegisterThread, RegisterThreadCreator> nodeKeyNotificationDispatcher;
  private NotificationDispatcher<AddCrawledLinkNotification, SaveCrawledLinkThread, SaveCrawledLinkThreadCreator> saveCrawledLinkNotificationDispatcher;

  private AnycastRequestDispatcher<IsPublisherExistedAnycastRequest, IsPublisherExistedAnycastResponse, IsPublisherExistedThread, IsPublisherExistedThreadCreator> isPublisherExistedyAnycastRequestDispatcher;

  private MulticastMessageDisposer<SearchKeywordBroadcastRequest> searchKeywordRequestDisposer;
  private BoundBroadcastRequestDispatcher<SearchKeywordBroadcastRequest, SearchKeywordBroadcastResponse, MulticastMessageDisposer<SearchKeywordBroadcastRequest>, SearchKeywordThread, SearchKeywordThreadCreator> searchKeywordRequestDispatcher;
  private BoundNotificationDispatcher<SearchKeywordBroadcastRequest, MulticastMessageDisposer<SearchKeywordBroadcastRequest>, BroadcastSearchKeywordRequestThread, BroadcastSearchKeywordRequestThreadCreator> broadcastSearchKeywordRequestDispatcher;

  /*
   * Initialize the dispatcher. 11/28/2014, Bing Li
   */
  public MemoryDispatcher(int corePoolSize, long keepAliveTime)
  {
    super(corePoolSize, keepAliveTime);

    // Initialize the notification dispatcher for the notification, NodeKeyNotification. 11/28/2014, Bing Li
    this.nodeKeyNotificationDispatcher = new NotificationDispatcher<NodeKeyNotification, RegisterThread, RegisterThreadCreator>(ClientConfig.NOTIFICATION_DISPATCHER_POOL_SIZE, ClientConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new RegisterThreadCreator(), ClientConfig.MAX_NOTIFICATION_TASK_SIZE, ClientConfig.MAX_NOTIFICATION_THREAD_SIZE, ClientConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
    // Set the idle checking of the nodeKeyNotificationDispatcher. 11/28/2014, Bing Li
```

```
  this.nodeKeyNotificationDispatcher.setIdleChecker(ClientConfig.NOTIFICATION_DISPATCHER_IDL
E_CHECK_DELAY, ClientConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
      // Start the nodeKeyNotificationDispatcher. 11/28/2014, Bing Li
      super.execute(this.nodeKeyNotificationDispatcher);

      // Initialize the notification dispatcher for the notification, AddCrawledLinkNotification. 11/28/2014,
Bing Li
      this.saveCrawledLinkNotificationDispatcher = new
NotificationDispatcher<AddCrawledLinkNotification, SaveCrawledLinkThread,
SaveCrawledLinkThreadCreator>(ClientConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
ClientConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new
SaveCrawledLinkThreadCreator(), ClientConfig.MAX_NOTIFICATION_TASK_SIZE,
ClientConfig.MAX_NOTIFICATION_THREAD_SIZE,
ClientConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);
      // Set the idle checking of the saveCrawledLinkNotificationDispatcher. 11/28/2014, Bing Li

  this.saveCrawledLinkNotificationDispatcher.setIdleChecker(ClientConfig.NOTIFICATION_DISPATC
HER_IDLE_CHECK_DELAY, ClientConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
      // Start the saveCrawledLinkNotificationDispatcher. 11/28/2014, Bing Li
      super.execute(this.saveCrawledLinkNotificationDispatcher);

      // Initialize the anycast dispatcher for the request, IsPublisherExistedAnycastRequest. 11/29/2014,
Bing Li
      this.isPublisherExistedyAnycastRequestDispatcher = new
AnycastRequestDispatcher<IsPublisherExistedAnycastRequest, IsPublisherExistedAnycastResponse,
IsPublisherExistedThread, IsPublisherExistedThreadCreator>(ClientPool.STORE().getPool(),
ServerConfig.COORDINATOR_ADDRESS, ServerConfig.COORDINATOR_PORT_FOR_MEMORY,
ServerConfig.REQUEST_DISPATCHER_POOL_SIZE,
ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME, new
IsPublisherExistedThreadCreator(), ServerConfig.MAX_REQUEST_TASK_SIZE,
ServerConfig.MAX_REQUEST_THREAD_SIZE,
ServerConfig.REQUEST_DISPATCHER_WAIT_TIME);
      // Set the idle checking. 11/29/2014, Bing Li

  this.isPublisherExistedyAnycastRequestDispatcher.setIdleChecker(ServerConfig.REQUEST_DISPA
TCHER_IDLE_CHECK_DELAY, ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD);
      // Start the dispatcher. 11/29/2014, Bing Li
      super.execute(this.isPublisherExistedyAnycastRequestDispatcher);

      this.searchKeywordRequestDisposer = new
MulticastMessageDisposer<SearchKeywordBroadcastRequest>();
      this.searchKeywordRequestDispatcher = new
BoundBroadcastRequestDispatcher<SearchKeywordBroadcastRequest,
SearchKeywordBroadcastResponse, MulticastMessageDisposer<SearchKeywordBroadcastRequest>,
SearchKeywordThread, SearchKeywordThreadCreator>(ClientPool.STORE().getPool(),
ServerConfig.COORDINATOR_ADDRESS, ServerConfig.COORDINATOR_PORT_FOR_MEMORY,
ServerConfig.REQUEST_DISPATCHER_POOL_SIZE,
ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME,
this.searchKeywordRequestDisposer, new SearchKeywordThreadCreator(),
ServerConfig.MAX_REQUEST_TASK_SIZE, ServerConfig.MAX_REQUEST_THREAD_SIZE,
ServerConfig.REQUEST_DISPATCHER_WAIT_TIME);

  this.searchKeywordRequestDispatcher.setIdleChecker(ServerConfig.REQUEST_DISPATCHER_IDL
E_CHECK_DELAY, ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD);
      super.execute(this.searchKeywordRequestDispatcher);

      this.broadcastSearchKeywordRequestDispatcher = new
BoundNotificationDispatcher<SearchKeywordBroadcastRequest,
MulticastMessageDisposer<SearchKeywordBroadcastRequest>,
BroadcastSearchKeywordRequestThread,
BroadcastSearchKeywordRequestThreadCreator>(ServerConfig.NOTIFICATION_DISPATCHER_POO
L_SIZE, ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME,
this.searchKeywordRequestDisposer, new BroadcastSearchKeywordRequestThreadCreator(),
ServerConfig.MAX_NOTIFICATION_TASK_SIZE, ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME);

  this.broadcastSearchKeywordRequestDispatcher.setIdleChecker(ServerConfig.NOTIFICATION_DIS
PATCHER_IDLE_CHECK_DELAY,
```

```
ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD);
      super.execute(this.broadcastSearchKeywordRequestDispatcher);
    }

    /*
     * Shutdown the dispatcher. 11/28/2014, Bing Li
     */
    public void shutdown()
    {
      this.nodeKeyNotificationDispatcher.dispose();
      this.saveCrawledLinkNotificationDispatcher.dispose();
      this.isPublisherExistedyAnycastRequestDispatcher.dispose();
      this.searchKeywordRequestDispatcher.dispose();
      this.broadcastSearchKeywordRequestDispatcher.dispose();
      super.shutdown();
    }

    /*
     * Dispatch received messages to corresponding threads respectively for concurrent processing.
11/28/2014, Bing Li
     */
    public void consume(OutMessageStream<ServerMessage> message)
    {
      SearchKeywordBroadcastRequest broadcastRequest;
      switch (message.getMessage().getType())
      {
        // Process the notification of NodeKeyNotification. 11/28/2014, Bing Li
        case MessageType.NODE_KEY_NOTIFICATION:
          // Enqueue the notification into the notification dispatcher. The notifications are queued and
processed asynchronously. 11/28/2014, Bing Li
          this.nodeKeyNotificationDispatcher.enqueue((NodeKeyNotification)message.getMessage());
          break;

        // Process the notification of AddCrawledLinkNotification. 11/28/2014, Bing Li
        case MessageType.ADD_CRAWLED_LINK_NOTIFICATION:
          // Enqueue the notification into the notification dispatcher. The notifications are queued and
processed asynchronously. 11/28/2014, Bing Li

    this.saveCrawledLinkNotificationDispatcher.enqueue((AddCrawledLinkNotification)message.getMess
age());
          break;

        // Process the anycast request of IsPublisherExistedAnycastRequest. 11/29/2014, Bing Li
        case MessageType.IS_PUBLISHER_EXISTED_ANYCAST_REQUEST:
          // Enqueue the anycast request into the anycast request dispatcher. The requests are queued
and processed asynchronously. 11/29/2014, Bing Li

    this.isPublisherExistedyAnycastRequestDispatcher.enqueue((IsPublisherExistedAnycastRequest)me
ssage.getMessage());
          break;

        // Process the broadcast request of SearchKeywordBroadcastRequest. 11/29/2014, Bing Li
        case MessageType.SEARCH_KEYWORD_BROADCAST_REQUEST:
          // Cast the request. 11/29/2014, Bing Li
          broadcastRequest = (SearchKeywordBroadcastRequest)message.getMessage();
          // Put the request into the search dispatcher to retrieve. 11/29/2014, Bing Li
          this.searchKeywordRequestDispatcher.enqueue(broadcastRequest);
          // Put the request into the broadcast dispatcher to forward it to the local node's children.
11/29/2014, Bing Li
          this.broadcastSearchKeywordRequestDispatcher.enqueue(broadcastRequest);
          break;
      }
    }
}
```

- **MemoryMessageProducerDisposer**

```
package com.greatfree.testing.memory;

import com.greatfree.concurrency.MessageProducer;
import com.greatfree.reuse.ThreadDisposable;

/*
 * The class is responsible for disposing the message producer of the server. 11/27/2014, Bing Li
 */

// Created: 11/27/2014, Bing Li
public class MemoryMessageProducerDisposer implements
ThreadDisposable<MessageProducer<MemoryDispatcher>>
{
   /*
    * Dispose the message producer. 11/27/2014, Bing Li
    */
   @Override
   public void dispose(MessageProducer<MemoryDispatcher> r)
   {
      r.dispose();
   }

   /*
    * The method does not make sense to the class of MessageProducer. Just leave it here. 11/27/2014,
Bing Li
    */
   @Override
   public void dispose(MessageProducer<MemoryDispatcher> r, long time)
   {
      r.dispose();
   }
}
```

- **RegisterThread**

```
package com.greatfree.testing.memory;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.NodeKeyNotification;
import com.greatfree.util.NodeID;

/*
 * After getting a notification from the coordinator, it denotes that the coordinator is ready such that the memory server can register itself to the coordinator and prepare for the coming storing. 11/28/2014, Bing Li
 *
 * The notification contains the key for the memory server. The key is the identification of the memory server. The coordinator uses the ID to manage the cluster of memory servers. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class RegisterThread extends NotificationQueue<NodeKeyNotification>
{
  /*
   * Initialize the thread. The argument, taskSize, is used to limit the count of tasks to be queued. 11/28/2014, Bing Li
   */
  public RegisterThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * Once if a node key notification is received, it is processed concurrently as follows. 11/28/2014, Bing Li
   */
  public void run()
  {
    // Declare an instance of NodeKeyNotification. 11/28/2014, Bing Li
    NodeKeyNotification notification;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/28/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/28/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/28/2014, Bing Li
          notification = this.getNotification();
          // Set the crawler key. 11/28/2014, Bing Li
          NodeID.DISTRIBUTED().setKey(notification.getKey());
          // Register the memory server after getting the key. 11/28/2014, Bing Li
          MemoryEventer.NOTIFY().register();
          // Dispose the notification. 11/28/2014, Bing Li
          this.disposeMessage(notification);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      try
      {
        // Wait for a moment after all of the existing notifications are processed. 11/28/2014, Bing Li
        this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
      }
      catch (InterruptedException e)
```

```
        {
          e.printStackTrace();
        }
      }
    }
  }
```

- **RegisterThreadCreator**

```
package com.greatfree.testing.memory;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.NodeKeyNotification;

/*
 * The code here attempts to create instances of RegisterThread. It is used by the notification dispatcher.
11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class RegisterThreadCreator implements NotificationThreadCreatable<NodeKeyNotification,
RegisterThread>
{
  @Override
  public RegisterThread createNotificationThreadInstance(int taskSize)
  {
    return new RegisterThread(taskSize);
  }
}
```

- **SaveCrawledLinkThread**

```
package com.greatfree.testing.memory;

import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.AddCrawledLinkNotification;

/*
 * The thread gets the crawled link and save it in the local server. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class SaveCrawledLinkThread extends NotificationQueue<AddCrawledLinkNotification>
{
  /*
   * Initialize the thread. The argument, taskSize, is used to limit the count of tasks to be queued. 11/28/2014, Bing Li
   */
  public SaveCrawledLinkThread(int taskSize)
  {
    super(taskSize);
  }

  /*
   * Once if a crawled link notification is received, it is processed concurrently as follows. 11/28/2014, Bing Li
   */
  public void run()
  {
    // Declare an instance of AddCrawledLinkNotification. 11/28/2014, Bing Li
    AddCrawledLinkNotification notification;
    // The thread always runs until it is shutdown by the NotificationDispatcher. 11/28/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/28/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the notification. 11/28/2014, Bing Li
          notification = this.getNotification();
          // Save the crawled link into the memory cache. 11/28/2014, Bing Li
          LinkPond.STORE().save(notification.getLink());
          // Dispose the notification. 11/28/2014, Bing Li
          this.disposeMessage(notification);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      try
      {
        // Wait for a moment after all of the existing notifications are processed. 11/28/2014, Bing Li
        this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
  }
}
```

- **SaveCrawledLinkThreadCreator**

```
package com.greatfree.testing.memory;

import com.greatfree.concurrency.NotificationThreadCreatable;
import com.greatfree.testing.message.AddCrawledLinkNotification;

/*
 * The code here attempts to create instances of SaveCrawledLinkThread. It is used by the notification
dispatcher. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class SaveCrawledLinkThreadCreator implements
NotificationThreadCreatable<AddCrawledLinkNotification, SaveCrawledLinkThread>
{
  @Override
  public SaveCrawledLinkThread createNotificationThreadInstance(int taskSize)
  {
    return new SaveCrawledLinkThread(taskSize);
  }
}
```

- **IsPublisherExistedThread**

```
package com.greatfree.testing.memory;

import java.io.IOException;

import com.greatfree.concurrency.AnycastRequestQueue;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.remote.IPPort;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.IsPublisherExistedAnycastRequest;
import com.greatfree.testing.message.IsPublisherExistedAnycastResponse;

/*
 * The thread is responsible for processing the anycast requests in a concurrent way. 11/29/2014, Bing
Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedThread extends
AnycastRequestQueue<IsPublisherExistedAnycastRequest, IsPublisherExistedAnycastResponse>
{
  /*
   * Initialize the thread. 11/29/2014, Bing Li
   */
  public IsPublisherExistedThread(IPPort ipPort, FreeClientPool pool, int taskSize)
  {
    super(ipPort, pool, taskSize);
  }

  /*
   * Process the anycast concurrently. 11/29/2014, Bing Li
   */
  public void run()
  {
    // The instance of the received anycast request. 11/29/2014, Bing Li
    IsPublisherExistedAnycastRequest request;
    // The retrieval result. 11/29/2014, Bing Li
    boolean isExisted;
    // The thread always runs until it is shutdown by the AnycastRequestDispatcher. 11/29/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/29/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the request. 11/29/2014, Bing Li
          request = this.getRequest();
          // Retrieve whether the publisher exists. 11/29/2014, Bing Li
          isExisted = LinkPond.STORE().isPublisherExisted(request.getURL());
          // Check whether the publisher exists. 11/29/2014, Bing Li
          if (isExisted)
          {
            try
            {
              // If the publisher exists, it is required to send the response back to the initiator
immediately. Then, the anycast requesting process can be terminated. 11/29/2014, Bing Li
              this.respond(new IsPublisherExistedAnycastResponse(isExisted,
request.getCollaboratorKey()));
            }
            catch (IOException e)
            {
              e.printStackTrace();
            }
          }
```

```
            else
            {
               // If the publisher does not exist, it is required to forward the request to its children for further
retrieval. 11/29/2014, Bing Li

   MemoryMulticastor.STORE().disseminateIsPublisherExistedRequestAmongSubMemServers(request)
;
            }
            // Dispose the request. 11/29/2014, Bing Li
            this.disposeMessage(request);
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
        }
        try
        {
          // Wait for a moment after all of the existing requests are processed. 11/29/2014, Bing Li
          this.holdOn(ServerConfig.RETRIEVE_THREAD_WAIT_TIME);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
    }
}
```

- **IsPublisherExistedThreadCreator**

```
package com.greatfree.testing.memory;

import com.greatfree.concurrency.AnycastRequestThreadCreatable;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.remote.IPPort;
import com.greatfree.testing.message.IsPublisherExistedAnycastRequest;
import com.greatfree.testing.message.IsPublisherExistedAnycastResponse;

/*
 * The creator initializes an instance of IsPublisherExistedThread. It is invoked by the
AnycastRequestDispatcher to create new threads to process the requests concurrently and efficiently.
11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedThreadCreator implements
AnycastRequestThreadCreatable<IsPublisherExistedAnycastRequest,
IsPublisherExistedAnycastResponse, IsPublisherExistedThread>
{
  @Override
  public IsPublisherExistedThread createRequestThreadInstance(IPPort ipPort, FreeClientPool pool,
int taskSize)
  {
    return new IsPublisherExistedThread(ipPort, pool, taskSize);
  }
}
```

- **SearchKeywordThread**

```
package com.greatfree.testing.memory;

import java.io.IOException;
import java.util.Set;

import com.greatfree.concurrency.BoundBroadcastRequestQueue;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.remote.IPPort;
import com.greatfree.reuse.MulticastMessageDisposer;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.SearchKeywordBroadcastRequest;
import com.greatfree.testing.message.SearchKeywordBroadcastResponse;
import com.greatfree.util.Tools;

/*
 * The thread retrieves links by the keyword. Since the request that contains the keyword must be broadcast to the local node's children, the disposal of the request must be controlled by the binder. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordThread extends BoundBroadcastRequestQueue<SearchKeywordBroadcastRequest, SearchKeywordBroadcastResponse, MulticastMessageDisposer<SearchKeywordBroadcastRequest>>
{
  /*
   * Initialize the thread. The binder is assigned at this point. 11/29/2014, Bing Li
   */
  public SearchKeywordThread(IPPort ipPort, FreeClientPool pool, int taskSize, String dispatcherKey, MulticastMessageDisposer<SearchKeywordBroadcastRequest> reqBinder)
  {
    super(ipPort, pool, taskSize, dispatcherKey, reqBinder);
  }

  /*
   * Retrieve links concurrently. After that, it must notify the binder that its task is accomplished. 11/29/2014, Bing Li
   */
  public void run()
  {
    // The instance of the received broadcast request. 11/29/2014, Bing Li
    SearchKeywordBroadcastRequest request;
    // The instance of the response to be responded to the broadcast request initiator. 11/29/2014, Bing Li
    SearchKeywordBroadcastResponse response;
    // The set to take the results. 11/29/2014, Bing Li
    Set<String> links;
    // The thread always runs until it is shutdown by the BoundBroadcastRequestDispatcher. 11/29/2014, Bing Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/29/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the request. 11/29/2014, Bing Li
          request = this.getRequest();
          // Retrieve the links. 11/29/2014, Bing Li
          links = LinkPond.STORE().getLinksByKeyword(request.getKeyword());
          // Form the response. The unique key is required so that the initiator is able to estimate the count of responses. That is different from the anycast response, which does not need to have the key. 11/29/2014, Bing Li
          response = new SearchKeywordBroadcastResponse(links, Tools.generateUniqueKey(),
```

```
request.getCollaboratorKey());
            try
            {
              // Respond the initiator of the broadcast requesting. 11/29/2014, Bing Li
              this.respond(response);
            }
            catch (IOException e)
            {
              e.printStackTrace();
            }
            // Notify the binder that the thread task on the request has done. 11/29/2014, Bing Li
            this.bind(super.getDispatcherKey(), request);
            // Dispose the response. 11/29/2014, Bing Li
            this.disposeResponse(response);
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
        }
        try
        {
          // Wait for a moment after all of the existing requests are processed. 11/29/2014, Bing Li
          this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
    }
}
```

- **SearchKeywordThreadCreator**

```
package com.greatfree.testing.memory;

import com.greatfree.concurrency.BoundBroadcastRequestThreadCreatable;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.remote.IPPort;
import com.greatfree.reuse.MulticastMessageDisposer;
import com.greatfree.testing.message.SearchKeywordBroadcastRequest;
import com.greatfree.testing.message.SearchKeywordBroadcastResponse;

/*
 * The creator is responsible for initializing an instance of SearchKeywordThread. It works with the
BoundBroadcastRequestDispatcher to manage the searching process efficiently and concurrently.
11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordThreadCreator implements
BoundBroadcastRequestThreadCreatable<SearchKeywordBroadcastRequest,
SearchKeywordBroadcastResponse, MulticastMessageDisposer<SearchKeywordBroadcastRequest>,
SearchKeywordThread>
{
  @Override
  public SearchKeywordThread createRequestThreadInstance(IPPort ipPort, FreeClientPool pool, int
taskSize, String dispatcherKey, MulticastMessageDisposer<SearchKeywordBroadcastRequest>
reqBinder)
  {
    return new SearchKeywordThread(ipPort, pool, taskSize, dispatcherKey, reqBinder);
  }
}
```

- **BroadcastSearchKeywordRequestThread**

```
package com.greatfree.testing.memory;

import com.greatfree.concurrency.BoundNotificationQueue;
import com.greatfree.reuse.MulticastMessageDisposer;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.SearchKeywordBroadcastRequest;

/*
 * The thread broadcasts the instance of SearchKeywordBroadcastRequest to the local node's children.
11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class BroadcastSearchKeywordRequestThread extends
BoundNotificationQueue<SearchKeywordBroadcastRequest,
MulticastMessageDisposer<SearchKeywordBroadcastRequest>>
{
  /*
   * Initialize the bound thread. 11/29/2014, Bing Li
   */
  public BroadcastSearchKeywordRequestThread(int taskSize, String dispatcherKey,
MulticastMessageDisposer<SearchKeywordBroadcastRequest> binder)
  {
    super(taskSize, dispatcherKey, binder);
  }

  /*
   * Broadcast the request concurrently. 11/29/2014, Bing Li
   */
  public void run()
  {
    // The instance of the received broadcast request. 11/29/2014, Bing Li
    SearchKeywordBroadcastRequest request;
    // The thread always runs until it is shutdown by the BoundNotificationDispatcher. 11/29/2014, Bing
Li
    while (!this.isShutdown())
    {
      // Check whether the notification queue is empty. 11/29/2014, Bing Li
      while (!this.isEmpty())
      {
        try
        {
          // Dequeue the request. 11/29/2014, Bing Li
          request = this.getNotification();
          // Disseminate the broadcast request to the local node's children. 11/29/2014, Bing Li

  MemoryMulticastor.STORE().disseminateSearchKeywordRequestAmongSubMemServers(request);
          // Notify the binder that the thread's task on the request has done. 11/29/2014, Bing Li
          this.bind(super.getDispatcherKey(), request);
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      try
      {
        // Wait for a moment after all of the existing requests are processed. 11/29/2014, Bing Li
        this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
```

```
    }
  }
}
```

- **BroadcastSearchKeywordRequestThreadCreator**

```
package com.greatfree.testing.memory;

import com.greatfree.concurrency.BoundNotificationThreadCreatable;
import com.greatfree.reuse.MulticastMessageDisposer;
import com.greatfree.testing.message.SearchKeywordBroadcastRequest;

/*
 * The creator initialize the instance of the thread, BroadcastSearchKeywordRequestThread. The
BoundNotificationDispatcher can schedule tasks to the thread. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class BroadcastSearchKeywordRequestThreadCreator implements
BoundNotificationThreadCreatable<SearchKeywordBroadcastRequest,
MulticastMessageDisposer<SearchKeywordBroadcastRequest>,
BroadcastSearchKeywordRequestThread>
{
  @Override
  public BroadcastSearchKeywordRequestThread createNotificationThreadInstance(int taskSize,
String dispatcherKey, MulticastMessageDisposer<SearchKeywordBroadcastRequest> binder)
  {
    return new BroadcastSearchKeywordRequestThread(taskSize, dispatcherKey, binder);
  }
}
```

- **MemoryMulticastor**

```
package com.greatfree.testing.memory;

import java.io.IOException;

import com.greatfree.reuse.ResourcePool;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.IsPublisherExistedAnycastRequest;
import com.greatfree.testing.message.SearchKeywordBroadcastRequest;

/*
 * The multicastor contains all of the multicastors that send requests or notifications to the local node's children memory nodes by an efficient multicast way. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class MemoryMulticastor
{
  // A resource pool to manage the anycastor. 11/29/2014, Bing Li
  private ResourcePool<IsPublisherExistedChildAnycastorSource, IsPublisherExistedChildAnycastor, IsPublisherExistedChildAnycastorCreator, IsPublisherExistedChildAnycastorDisposer> isPublisherExistedAnycastorPool;

  private ResourcePool<SearchKeywordRequestChildBroadcastorSource, SearchKeywordRequestChildBroadcastor, SearchKeywordRequestChildBroadcastorCreator, SearchKeywordRequestChildBroadcastorDisposer> searchKeywordRequestChildBroadcastorPool;

  private MemoryMulticastor()
  {
  }

  /*
   * A singleton definition. 11/29/2014, Bing Li
   */
  private static MemoryMulticastor instance = new MemoryMulticastor();

  public static MemoryMulticastor STORE()
  {
    if (instance == null)
    {
      instance = new MemoryMulticastor();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the multicastors. 11/29/2014, Bing Li
   */
  public void dispose()
  {
    this.isPublisherExistedAnycastorPool.shutdown();
    this.searchKeywordRequestChildBroadcastorPool.shutdown();
  }

  /*
   * Initialize the multicastors. 11/29/2014, Bing Li
   */
  public void init()
  {
    this.isPublisherExistedAnycastorPool = new ResourcePool<IsPublisherExistedChildAnycastorSource, IsPublisherExistedChildAnycastor,
```

```
IsPublisherExistedChildAnycastorCreator,
IsPublisherExistedChildAnycastorDisposer>(ServerConfig.MULTICASTOR_POOL_SIZE, new
IsPublisherExistedChildAnycastorCreator(), new IsPublisherExistedChildAnycastorDisposer(),
ServerConfig.MULTICASTOR_POOL_WAIT_TIME);
    this.searchKeywordRequestChildBroadcastorPool = new
ResourcePool<SearchKeywordRequestChildBroadcastorSource,
SearchKeywordRequestChildBroadcastor, SearchKeywordRequestChildBroadcastorCreator,
SearchKeywordRequestChildBroadcastorDisposer>(ServerConfig.MULTICASTOR_POOL_SIZE, new
SearchKeywordRequestChildBroadcastorCreator(), new
SearchKeywordRequestChildBroadcastorDisposer(),
ServerConfig.MULTICASTOR_POOL_WAIT_TIME);
  }

  /*
   * Disseminate the anycast request to the local node's children. 11/29/2014, Bing Li
   */
  public void
disseminateIsPublisherExistedRequestAmongSubMemServers(IsPublisherExistedAnycastRequest
request)
  {
    try
    {
      // Get an instance of the anycastor. 11/29/2014, Bing Li
      IsPublisherExistedChildAnycastor anycastor = this.isPublisherExistedAnycastorPool.get(new
IsPublisherExistedChildAnycastorSource(SubClientPool.STORE().getPool(),
ServerConfig.MULTICAST_BRANCH_COUNT, ServerConfig.MEMORY_SERVER_PORT, new
IsPublisherExistedRequestCreator()));
      // Check whether the anycastor is valid. 11/29/2014, Bing Li
      if (anycastor != null)
      {
        try
        {
          // Disseminate the anycast request to its children. 11/29/2014, Bing Li
          anycastor.disseminate(request);
          // Collect the instance of anycastor after it is sent out. 11/29/2014, Bing Li
          isPublisherExistedAnycastorPool.collect(anycastor);
        }
        catch (IOException e)
        {
          e.printStackTrace();
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
    }
    catch (InstantiationException e)
    {
      e.printStackTrace();
    }
    catch (IllegalAccessException e)
    {
      e.printStackTrace();
    }
  }

  /*
   * Disseminate the broadcast request to the local node's children. 11/29/2014, Bing Li
   */
  public void
disseminateSearchKeywordRequestAmongSubMemServers(SearchKeywordBroadcastRequest
request)
  {
    try
    {
      // Get an instance of the broadcastor. 11/29/2014, Bing Li
      SearchKeywordRequestChildBroadcastor broadcastor =
```

```
this.searchKeywordRequestChildBroadcastorPool.get(new
SearchKeywordRequestChildBroadcastorSource(SubClientPool.STORE().getPool(),
ServerConfig.MULTICAST_BRANCH_COUNT, ServerConfig.MEMORY_SERVER_PORT, new
SearchKeywordBroadcastRequestCreator()));
        // Check whether the broadcastor is valid. 11/29/2014, Bing Li
        if (broadcastor != null)
        {
          try
          {
            // Disseminate the broadcast request to its children. 11/29/2014, Bing Li
            broadcastor.disseminate(request);
            // Collect the instance of broadcastor after it is sent out. 11/29/2014, Bing Li
            this.searchKeywordRequestChildBroadcastorPool.collect(broadcastor);
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
        }
      }
      catch (InstantiationException e)
      {
        e.printStackTrace();
      }
      catch (IllegalAccessException e)
      {
        e.printStackTrace();
      }
    }
}
```

- **IsPublisherExistedRequestCreator**

```
package com.greatfree.testing.memory;

import java.util.HashMap;

import com.greatfree.multicast.ChildMulticastMessageCreatable;
import com.greatfree.testing.message.IsPublisherExistedAnycastRequest;
import com.greatfree.util.Tools;

/*
 * The creator initializes the multicast requests in the child multicastor to accomplish the goal the anycast requesting. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedRequestCreator implements ChildMulticastMessageCreatable<IsPublisherExistedAnycastRequest>
{
   // Initialize an instance of the anycast request. 11/29/2014, Bing Li
   @Override
   public IsPublisherExistedAnycastRequest createInstanceWithChildren(IsPublisherExistedAnycastRequest msg, HashMap<String, String> children)
   {
      return new IsPublisherExistedAnycastRequest(msg.getURL(), Tools.generateUniqueKey(), msg.getCollaboratorKey(), children);
   }
}
```

- **IsPublisherExistedChildAnycastorSource**

```
package com.greatfree.testing.memory;

import com.greatfree.multicast.ChildMessageCreatorGettable;
import com.greatfree.multicast.ChildMulticastorSource;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.message.IsPublisherExistedAnycastRequest;

/*
 * The source to create the instance of IsPublisherExistedChildAnycastor. It is used by the resource pool
to use the anycastors efficiently. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedChildAnycastorSource extends
ChildMulticastorSource<IsPublisherExistedAnycastRequest, IsPublisherExistedRequestCreator>
implements ChildMessageCreatorGettable<IsPublisherExistedAnycastRequest>
{
  /*
   * Initialize the source. 11/29/2014, Bing Li
   */
  public IsPublisherExistedChildAnycastorSource(FreeClientPool clientPool, int treeBranchCount, int
serverPort, IsPublisherExistedRequestCreator creator)
  {
    super(clientPool, treeBranchCount, serverPort, creator);
  }

  /*
   * Expose the request creator. 11/29/2014, Bing Li
   */
  @Override
  public IsPublisherExistedRequestCreator getMessageCreator()
  {
    return super.getMessageCreator();
  }
}
```

- **IsPublisherExistedChildAnycastor**

```
package com.greatfree.testing.memory;

import com.greatfree.multicast.ChildMulticastor;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.message.IsPublisherExistedAnycastRequest;

/*
 * The anycast is implemented in the same way as an ordinary children multicastor, which transmits the anycast requests to its children. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedChildAnycastor extends ChildMulticastor<IsPublisherExistedAnycastRequest, IsPublisherExistedRequestCreator>
{
  /*
   * Initialize the children anycastor. 11/29/2014, Bing Li
   */
  public IsPublisherExistedChildAnycastor(FreeClientPool clientPool, int treeBranchCount, int clusterServerPort, IsPublisherExistedRequestCreator messageCreator)
  {
    super(clientPool, treeBranchCount, clusterServerPort, messageCreator);
  }
}
```

- **IsPublisherExistedChildAnycastorCreator**

```
package com.greatfree.testing.memory;

import com.greatfree.reuse.HashCreatable;

/*
 * The creator to initialize the instance of IsPublisherExistedChildAnycastor. It is used by the resource
pool to manage instances of IsPublisherExistedChildAnycastor efficiently. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedChildAnycastorCreator implements
HashCreatable<IsPublisherExistedChildAnycastorSource, IsPublisherExistedChildAnycastor>
{
  @Override
  public IsPublisherExistedChildAnycastor
createResourceInstance(IsPublisherExistedChildAnycastorSource source)
  {
    return new IsPublisherExistedChildAnycastor(source.getClientPool(),
source.getTreeBranchCount(), source.getServerPort(), source.getMessageCreator());
  }
}
```

- **IsPublisherExistedChildAnycastorDisposer**

```
package com.greatfree.testing.memory;

import com.greatfree.reuse.HashDisposable;

/*
 * The disposer to collect the resource in the instance of IsPublisherExistedChildAnycastor. It is also
used by the resource pool to utilize the anycastors efficiently. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class IsPublisherExistedChildAnycastorDisposer implements
HashDisposable<IsPublisherExistedChildAnycastor>
{
   @Override
   public void dispose(IsPublisherExistedChildAnycastor t)
   {
      t.dispose();
   }
}
```

- **SearchKeywordBroadcastRequestCreator**

```
package com.greatfree.testing.memory;

import java.util.HashMap;

import com.greatfree.multicast.ChildMulticastMessageCreatable;
import com.greatfree.testing.message.SearchKeywordBroadcastRequest;
import com.greatfree.util.Tools;

/*
 * The creator initiates the instance of SearchKeywordBroadcastRequest that is needed by the
multicastor. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordBroadcastRequestCreator implements
ChildMulticastMessageCreatable<SearchKeywordBroadcastRequest>
{
  @Override
  public SearchKeywordBroadcastRequest
createInstanceWithChildren(SearchKeywordBroadcastRequest msg, HashMap<String, String> children)
  {
    return new SearchKeywordBroadcastRequest(msg.getKeyword(), Tools.generateUniqueKey(),
msg.getCollaboratorKey(), children);
  }
}
```

- **SearchKeywordRequestChildBroadcastorSource**

```
package com.greatfree.testing.memory;

import com.greatfree.multicast.ChildMessageCreatorGettable;
import com.greatfree.multicast.ChildMulticastorSource;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.message.SearchKeywordBroadcastRequest;

/*
 * The source contains the arguments to initialize a new broadcastor to forward the request of
SearchKeywordBroadcastRequest. It is used by the resource pool. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordRequestChildBroadcastorSource extends
ChildMulticastorSource<SearchKeywordBroadcastRequest, SearchKeywordBroadcastRequestCreator>
implements ChildMessageCreatorGettable<SearchKeywordBroadcastRequest>
{
  /*
   * Initialize the source. 11/29/2014, Bing Li
   */
  public SearchKeywordRequestChildBroadcastorSource(FreeClientPool clientPool, int
treeBranchCount, int serverPort, SearchKeywordBroadcastRequestCreator creator)
  {
    super(clientPool, treeBranchCount, serverPort, creator);
  }

  /*
   * Expose the request creator. 11/29/2014, Bing Li
   */
  @Override
  public SearchKeywordBroadcastRequestCreator getMessageCreator()
  {
    return super.getMessageCreator();
  }
}
```

- **SearchKeywordRequestChildBroadcastor**

```
package com.greatfree.testing.memory;

import com.greatfree.multicast.ChildMulticastor;
import com.greatfree.remote.FreeClientPool;
import com.greatfree.testing.message.SearchKeywordBroadcastRequest;

/*
 * The child broadcastor forwards the instance of SearchKeywordBroadcastRequest to the local node's children. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordRequestChildBroadcastor extends ChildMulticastor<SearchKeywordBroadcastRequest, SearchKeywordBroadcastRequestCreator>
{
   public SearchKeywordRequestChildBroadcastor(FreeClientPool clientPool, int treeBranchCount, int clusterServerPort, SearchKeywordBroadcastRequestCreator messageCreator)
   {
      super(clientPool, treeBranchCount, clusterServerPort, messageCreator);
   }
}
```

- **SearchKeywordRequestChildBroadcastorCreator**

```
package com.greatfree.testing.memory;

import com.greatfree.reuse.HashCreatable;

/*
 * The creator initializes an instance of SearchKeywordRequestChildBroadcastor. It works with the
resource pool to utilize the broadcastor efficiently. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordRequestChildBroadcastorCreator implements
HashCreatable<SearchKeywordRequestChildBroadcastorSource,
SearchKeywordRequestChildBroadcastor>
{
   @Override
   public SearchKeywordRequestChildBroadcastor
createResourceInstance(SearchKeywordRequestChildBroadcastorSource source)
   {
      return new SearchKeywordRequestChildBroadcastor(source.getClientPool(),
source.getTreeBranchCount(), source.getServerPort(), source.getMessageCreator());
   }
}
```

- **SearchKeywordRequestChildBroadcastorDisposer**

```
package com.greatfree.testing.memory;

import com.greatfree.reuse.HashDisposable;

/*
 * The disposer collects the resource of a broadcastor. It is used by the resource pool to save resources.
11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchKeywordRequestChildBroadcastorDisposer implements
HashDisposable<SearchKeywordRequestChildBroadcastor>
{
   @Override
   public void dispose(SearchKeywordRequestChildBroadcastor t)
   {
      t.dispose();
   }
}
```

- **LinkRecord**

```
package com.greatfree.testing.memory;

import com.greatfree.util.FreeObject;

/*
 * This is the data that is saved in the distributed memory server. It must derive from FreeObject in order to fit the requirements of ResrouceCache. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class LinkRecord extends FreeObject
{
  private String key;
  private String link;
  private String text;
  private String hubURLKey;

  public LinkRecord(String key, String link, String text, String hubURLKey)
  {
    this.key = key;
    this.link = link;
    this.text = text;
    this.hubURLKey = hubURLKey;
  }

  /*
   * Expose the key. 11/28/2014, Bing Li
   */
  public String getKey()
  {
    return this.key;
  }

  /*
   * Expose the link. 11/28/2014, Bing Li
   */
  public String getLink()
  {
    return this.link;
  }

  /*
   * Expose the text. 11/28/2014, Bing Li
   */
  public String getText()
  {
    return this.text;
  }

  /*
   * Expose the key of the hub URL being crawled. 11/28/2014, Bing Li
   */
  public String getHubURLKey()
  {
    return this.hubURLKey;
  }
}
```

- **LinkPond**

```
package com.greatfree.testing.memory;

import java.util.Map;
import java.util.Set;

import com.google.common.collect.Sets;
import com.greatfree.reuse.ResourceCache;
import com.greatfree.testing.data.CrawledLink;
import com.greatfree.util.Tools;

/*
 * This is a data storage in memory to keep crawled links as one port of the distributed memory system.
11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class LinkPond
{
  // The cache to save link records. 11/28/2014, Bing Li
  private ResourceCache<LinkRecord> cache;

  private LinkPond()
  {
  }

  /*
   * A singleton implementation. 11/28/2014, Bing Li
   */
  private static LinkPond instance = new LinkPond();

  public static LinkPond STORE()
  {
    if (instance == null)
    {
      instance = new LinkPond();
      return instance;
    }
    else
    {
      return instance;
    }
  }

  /*
   * Dispose the cache. 11/28/2014, Bing Li
   */
  public void dispose()
  {
    this.cache.dispose();
  }

  /*
   * Initialize the memory cache. 11/28/2014, Bing Li
   */
  public void init()
  {
    this.cache = new ResourceCache<LinkRecord>(MemConfig.MEMORY_CACHE_SIZE);
  }

  /*
   * Save the data into the memory. 11/28/2014, Bing Li
   */
  public void save(CrawledLink link)
  {
```

```
      this.cache.addResource(new LinkRecord(link.getKey(), link.getLink(), link.getText(),
link.getHubURLKey()));
    }

    /*
     * Check whether the URL is existed. By the way, for the sample code, it is not an efficient way to
retrieve data. 11/29/2014, Bing Li
     */
    public boolean isPublisherExisted(String url)
    {
      // Get the hash key of the URL. 11/29/2014, Bing Li
      String urlKey = Tools.getHash(url);
      // Get all of the links from the cache. 11/29/2014, Bing Li
      Map<String, LinkRecord> records = this.cache.getResources();
      // Scan each link to compare with the URL key. 11/29/2014, Bing Li
      for (LinkRecord record : records.values())
      {
        // Compared the key of the URL with the link's URL key. 11/29/2014, Bing Li
        if (record.getHubURLKey().equals(urlKey))
        {
          // Return true if matched. 11/29/2014, Bing Li
          return true;
        }
      }
      // Return false if not matched. 11/29/2014, Bing Li
      return false;
    }

    /*
     * Retrieve links whose corresponding texts have the keyword. By the way, for the sample code, it is
not an efficient way to retrieve data. 11/29/2014, Bing Li
     */
    public Set<String> getLinksByKeyword(String keyword)
    {
      // Get all of the links from the cache. 11/29/2014, Bing Li
      Map<String, LinkRecord> records = this.cache.getResources();
      // Initialize a set to take the retrieved linnks. 11/29/2014, Bing Li
      Set<String> links = Sets.newHashSet();
      // Scan each link to compare with the URL key. 11/29/2014, Bing Li
      for (LinkRecord record : records.values())
      {
        // Compared the key of the URL with the link's URL key. 11/29/2014, Bing Li
        if (record.getText().indexOf(keyword) >= 0)
        {
          // Add the link. 11/29/2014, Bing Li
          links.add(record.getLink());
        }
      }
      // Return the set of links. 11/29/2014, Bing Li
      return links;
    }
}
```

- **MemConfig**

```
package com.greatfree.testing.memory;

/*
 * The class contains the constants and configurations for the memory server. 11/28/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class MemConfig
{
  public final static int MEMORY_LISTENER_THREAD_POOL_SIZE = 100;
  public final static long MEMORY_LISTENER_THREAD_ALIVE_TIME = 10000;

  public final static int SUB_CLIENT_POOL_SIZE = 500;
  public final static long SUB_CLIENT_IDLE_CHECK_DELAY = 3000;
  public final static long SUB_CLIENT_IDLE_CHECK_PERIOD = 3000;
  public final static long SUB_CLIENT_MAX_IDLE_TIME = 3000;

  public final static int MEMORY_CACHE_SIZE = 10000;
}
```

3.6.5 The Searchers

- **StartSearcher**

```
package com.greatfree.testing.searcher;

import java.io.IOException;

import com.greatfree.exceptions.RemoteReadException;
import com.greatfree.remote.RemoteReader;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.IsPublisherExistedRequest;
import com.greatfree.testing.message.IsPublisherExistedResponse;
import com.greatfree.testing.message.SearchKeywordRequest;
import com.greatfree.testing.message.SearchKeywordResponse;
import com.greatfree.util.NodeID;

/*
 * The process intends to illustrate the sample code for a user end. Through it, a user sends search requests to the coordinator. And then, the coordinator searches within its clusters via anycast or broadcast and responds to the user. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class StartSearcher
{
  /*
   * The unique entry and exit of the searcher process. 11/29/2014, Bing Li
   */
  public static void main(String[] args)
  {
    // Start the search client. 11/29/2014, Bing Li
    Searcher.CLIENT().start(ServerConfig.SEARCH_CLIENT_PORT);

    try
    {
      // Send a search request to the coordinator and wait for the response. This request is processed as an anycast request on the coordinator server. 11/29/2014, Bing Li
      IsPublisherExistedResponse response = (IsPublisherExistedResponse)RemoteReader.REMOTE().read(NodeID.DISTRIBUTED().getKey(), ServerConfig.COORDINATOR_ADDRESS, ServerConfig.COORDINATOR_PORT_FOR_SEARCH, new IsPublisherExistedRequest("http://greatfree.blog.com"));
      // Display the result. 11/29/2014, Bing Li
      System.out.println(response.isExisted());
    }
    catch (RemoteReadException e)
    {
      e.printStackTrace();
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
    catch (ClassNotFoundException e)
    {
      e.printStackTrace();
    }

    try
    {
      // Send a search request to the coordinator and wait for the response. This request is processed as a broadcast request on the coordinator server. 11/29/2014, Bing Li
      SearchKeywordResponse response = (SearchKeywordResponse)RemoteReader.REMOTE().read(NodeID.DISTRIBUTED().getKey(), ServerConfig.COORDINATOR_ADDRESS, ServerConfig.COORDINATOR_PORT_FOR_SEARCH, new SearchKeywordRequest("greatfree labs"));
      // Display the result. 11/29/2014, Bing Li
      for (String link : response.getLinks())
```

```
        {
          System.out.println(link);
        }
      }
      catch (RemoteReadException e)
      {
        e.printStackTrace();
      }
      catch (IOException e)
      {
        e.printStackTrace();
      }
      catch (ClassNotFoundException e)
      {
        e.printStackTrace();
      }

      try
      {
        // Shutdown the search client. 11/29/2014, Bing Li
        Searcher.CLIENT().stop();
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
      catch (IOException e)
      {
        e.printStackTrace();
      }
    }
}
```

- **Searcher**

```
package com.greatfree.testing.searcher;

import java.io.IOException;
import java.net.ServerSocket;
import java.util.ArrayList;
import java.util.List;

import com.greatfree.concurrency.Runner;
import com.greatfree.remote.RemoteReader;
import com.greatfree.testing.client.ClientEventer;
import com.greatfree.testing.client.ClientListener;
import com.greatfree.testing.client.ClientListenerDisposer;
import com.greatfree.testing.client.ClientPool;
import com.greatfree.testing.client.ClientServerIORegistry;
import com.greatfree.testing.client.ClientServerMessageProducer;
import com.greatfree.testing.data.ClientConfig;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.util.UtilConfig;

/*
 * The class intends to illustrate the sample code for a user end. Through it, a user sends search requests to the coordinator. And then, the coordinator searches within its clusters via anycast or broadcast and responds to the user. 11/29/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class Searcher
{
   // A socket that waits for connections from remote nodes. In this case, it is used to wait for connections from the coordinator. 11/29/2014, Bing Li
   private ServerSocket serverSocket;
   // The port that is open to the coordinator. 11/29/2014, Bing Li
   private int port;
   // Multiple threads waiting for remote connections. 11/29/2014, Bing Li
   private List<Runner<ClientListener, ClientListenerDisposer>> listenerRunners;

   private Searcher()
   {
   }
   /*
    * A singleton definition. 11/23/2014, Bing Li
    */
   private static Searcher instance = new Searcher();

   public static Searcher CLIENT()
   {
      if (instance == null)
      {
         instance = new Searcher();
         return instance;
      }
      else
      {
         return instance;
      }
   }

   /*
    * Start the client. 11/23/2014, Bing Li
    */
   public void start(int port)
   {
      // On JD7, the sorting algorithm is replaced with TimSort rather than MargeSort. To run correctly, it is necessary to use the old one. the following line sets that up. 10/03/2014, Bing Li
```

```
    System.setProperty(UtilConfig.MERGE_SORT, UtilConfig.TRUE);

    // Set the port number. 11/23/2014, Bing Li
    this.port = port;
    // Initialize a disposer. 11/23/2014, Bing Li
    ClientListenerDisposer disposer = new ClientListenerDisposer();
    // Initialize a list to take all of the listeners to wait for remote connections concurrently. 11/23/2014,
Bing Li
    this.listenerRunners = new ArrayList<Runner<ClientListener, ClientListenerDisposer>>();
    // The runner is responsible for starting to wait for connections asynchronously. 11/23/2014, Bing Li
    Runner<ClientListener, ClientListenerDisposer> listenerRunner;
    try
    {
      // Initialize the socket to wait for remote connections. 11/23/2014, Bing Li
      this.serverSocket = new ServerSocket(this.port);
      // Start a bunch of threads to listen to connections. 11/23/2014, Bing Li
      for (int i = 0; i < ServerConfig.MAX_CLIENT_LISTEN_THREAD_COUNT; i++)
      {
        // Initialize the runner which contains the listener. 11/23/2014, Bing Li
        listenerRunner = new Runner<ClientListener, ClientListenerDisposer>(new
ClientListener(this.serverSocket), disposer, true);
        // Put the runner into a list for management. 11/23/2014, Bing Li
        this.listenerRunners.add(listenerRunner);
        // Start the runner. 11/23/2014, Bing Li
        listenerRunner.start();
      }
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }

    // Initialize the server IO registry. 11/23/2014, Bing Li
    ClientServerIORegistry.REGISTRY().init();
    // Initialize the client pool. 11/23/2014, Bing Li
    ClientPool.LOCAL().init();

    // Initialize the message producer to dispatcher messages. 11/23/2014, Bing Li
    ClientServerMessageProducer.CLIENT().init();

    // Initialize the eventer to notify the remote server. 11/23/2014, Bing Li
    ClientEventer.NOTIFY().init(ServerConfig.COORDINATOR_ADDRESS,
ServerConfig.SERVER_PORT);
    // Initialize the remote reader to send requests and receive responses from the remote server.
11/23/2014, Bing Li
    RemoteReader.REMOTE().init(ClientConfig.SERVER_CLIENT_POOL_SIZE);

    try
    {
      // The line tries to connect the CServer. Its IP and port number are saved on the server for that.
Then, the RetrievablePool can work on the NodeKey to retrieve the IP and the port number. 10/03/2014,
Bing Li
      ClientEventer.NOTIFY().notifyOnline();
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }

    // Register the client on the remote server. 11/23/2014, Bing Li
    ClientEventer.NOTIFY().register();
  }

  /*
```

```
     * Stop the client. 11/23/2014, Bing Li
     */
    public void stop() throws InterruptedException, IOException
    {
        // Close the listeners. 11/23/2014, Bing Li
        for (Runner<ClientListener, ClientListenerDisposer> runner : this.listenerRunners)
        {
            runner.stop(ClientConfig.TIME_TO_WAIT_FOR_THREAD_TO_DIE);
        }
        // Close the socket. 11/23/2014, Bing Li
        this.serverSocket.close();

        // Dispose the message producer. 11/23/2014, Bing Li
        ClientServerMessageProducer.CLIENT().dispose();

        // Dispose the client pool. 11/23/2014, Bing Li
        ClientPool.LOCAL().dispose();
        // Dispose the eventer. 11/23/2014, Bing Li
        ClientEventer.NOTIFY().dispose();
        // Shutdown the remote reader. 11/23/2014, Bing Li
        RemoteReader.REMOTE().shutdown();
        // Dispose the server IO registry. 11/23/2014, Bing Li
        ClientServerIORegistry.REGISTRY().dispose();
    }
}
```

- **SearchReader**

```
package com.greatfree.testing.searcher;

import java.io.IOException;
import java.util.Set;

import com.greatfree.exceptions.RemoteReadException;
import com.greatfree.remote.RemoteReader;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.testing.message.IsPublisherExistedRequest;
import com.greatfree.testing.message.IsPublisherExistedResponse;
import com.greatfree.testing.message.SearchKeywordRequest;
import com.greatfree.testing.message.SearchKeywordResponse;
import com.greatfree.util.NodeID;

/*
 * The class wraps the class, RemoteReader, to send search requests to the remote server and wait until
search results are received. 11/29/2014, Bing Li
 */

// Created: 11/28/2014, Bing Li
public class SearchReader
{
  public static boolean isPublisherExisted(String url)
  {
    try
    {
      return
((IsPublisherExistedResponse)(RemoteReader.REMOTE().read(NodeID.DISTRIBUTED().getKey(),
ServerConfig.COORDINATOR_ADDRESS, ServerConfig.COORDINATOR_PORT_FOR_SEARCH,
new IsPublisherExistedRequest(url)))).isExisted();
    }
    catch (RemoteReadException e)
    {
      e.printStackTrace();
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
    catch (ClassNotFoundException e)
    {
      e.printStackTrace();
    }
    return false;
  }

  public static Set<String> searchKeyword(String keyword)
  {
    try
    {
      return
((SearchKeywordResponse)(RemoteReader.REMOTE().read(NodeID.DISTRIBUTED().getKey(),
ServerConfig.COORDINATOR_ADDRESS, ServerConfig.COORDINATOR_PORT_FOR_SEARCH,
new SearchKeywordRequest(keyword)))).getLinks();
    }
    catch (RemoteReadException e)
    {
      e.printStackTrace();
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
    catch (ClassNotFoundException e)
```

```
        {
            e.printStackTrace();
        }
        return SearchConfig.NO_LINKS;
    }
}
```

- **SearchConfig**

```
package com.greatfree.testing.searcher;

import java.util.Set;

/*
 * The constants and configurations for the searcher are saved in the class. 12/01/2014, Bing Li
 */

// Created: 11/29/2014, Bing Li
public class SearchConfig
{
   public final static Set<String> NO_LINKS = null;
}
```